目录

00_版本说明篇

百图画鸿蒙 | 百文说内核 | 百万注源码

鸿蒙研究站 | weharmonyos.com | 专注 . 聚焦

进 >> 百篇博客 | 进 >> 官方文档 | 进 >> 参考手册 | 进 >> 注解仓库

get-code | 获取源码/获取工具 南向 - 设备开发	百文说内核 | 脉络渐清晰 双向链表 | 时间管理 | 用栈方式	微信 | QQ群 790015635 | 666
docker | Docker镜像构建 轻内核版 | 标准版 镜像	百篇博客 | 进程管理 进程管理 | Fork | 信号消费	离线文档 | 定期更新 百篇博客分析.zip | 注解文档.chm

关于作者

站长 `turing` ，计算机硕士，某互联网公司技术副总裁 ，计划用 5 - 10 年时间把鸿蒙系统的底层实现整理成档，包括：内核实现、驱动框架、协议栈、应用框架、编译构建、运行时系统 等核心子系统。工程浩大，自不量力，然兴趣所至，义无反顾， 此念不息，坚如磐石。

热爱是所有的理由和答案

- 因大学时阅读 linux 2.6 内核痛并快乐的经历，一直有个心愿，对底层基础技术进行一次系统性的整理，方便自己随时翻看，同时让更多对底层感兴趣的小伙伴减少时间，加速对计算机系统级的理解，而不至于过早的放弃。但因过程种种，多年一直没有行动，基本要放弃这件事了。恰逢 2020/9/10 鸿蒙正式开源，重新激活了多年的心愿，就有那么点如黄河之水一发不可收拾了。
- 包含三部分内容：**注源**，**写博** ，**画图**， 目前对内核源码的注解完成 80% ，博客分析完成80+篇，百图画鸿蒙完成20张，空闲时间几乎被占用，时间不够用，但每天都很充实，连做梦鸿蒙系统都在鱼贯而入。是件很有挑战的事，时间单位以年计，已持续一年半，期间得到众多小伙伴的支持与纠错，在此谢过！:P

如何下载最新PDF版本？

本次版本日期： `2022/04/20` 下载最新版本前往 >> http://weharmonyos.com/history.html

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

基础知识

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

进程管理

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

任务管理

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

内存管理

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么

- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

通讯机制

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析LiteIpc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析LiteIpc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

文件系统

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

硬件架构

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

内核汇编

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()

- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

编译运行

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

调测工具

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

前因后果

- v101.03 鸿蒙内核源码分析(总目录) | 精雕细琢 锤炼精品
- v102.05 鸿蒙内核源码分析(源码注释) | 每天死磕一点点
- v103.05 鸿蒙内核源码分析(静态站点) | 码农都不爱写注释和文档
- v104.01 鸿蒙内核源码分析(参考手册) | 阅读内核源码必备工具

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。

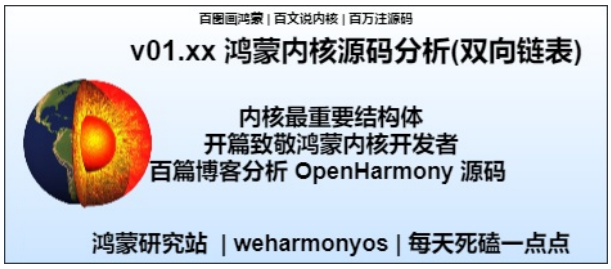
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

01_双向链表篇

本篇关键词：双向链表 、递减满栈、增删改查、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

双向链表是什么？

谁是鸿蒙内核最重要的结构体 ？ 一定是: `LOS_DL_LIST` (双向链表)， 它长这样。

```
typedef struct LOS_DL_LIST {
    struct LOS_DL_LIST *pstPrev; /**< Current node's pointer to the previous node | 前驱节点(左手)*/
    struct LOS_DL_LIST *pstNext; /**< Current node's pointer to the next node | 后继节点(右手)*/
} LOS_DL_LIST;
```

在 `linux` 中是 `list_head` ， 很简单，只有两个指向自己的指针，但因为太简单，所以不简单。站长更愿意将它比喻成人的左右手，其意义是通过寄生在宿主结构体上来体现，可想象成在宿主结构体装上一对对勤劳的双手，它真的很会来事，超级活跃分子，为宿主到处拉朋友，建圈子。

- 基本概念：双向链表是指含有往前和往后两个方向的链表，即每个结点中除存放下一个节点指针外，还增加一个指向前一个节点的指针， 从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点，这种数据结构形式使得双向链表在查找时更加方便，特别是大量数据的遍历。由于双向链表具有对称性，能方便地完成各种插入、删除等操作。
- 使用场景：在内核的各个模块都能看到双向链表的身影，下图是初始化双向链表的操作，因为太多了，只截取了部分:

LOS_ListInit	los_list.h (kernel\include)
VidMapListInit	vid.c (security\vid)
init_file_mapping	fs_file_mapping.c (fs\vfs\operation)
OsVmPageInit	los_vm_page.c (kernel\base\vm)
OsDLnkInitMultiHead	los_multipledlinkhead.c (kernel\base\mem\bestfit)
LOS_EventInit	los_event.c (kernel\base\ipc)
OsSortLinkInit	los_sortlink.c (kernel\base\core)
OsPriQueueInit	los_priqueue.c (kernel\base\sched\sched_sq)
OsSemInit	los_sem.c (kernel\base\ipc)
MtdNorParamAssign	mtd_partition.c (fs\vfs\multi_partition\src)
OsQueueInit	los_queue.c (kernel\base\ipc)
OsVmPhysLruInit	los_vm_phys.c (kernel\base\vm)
OsFutexInit	los_futex.c (kernel\base\ipc)
add_mapping	fs_file_mapping.c (fs\vfs\operation)
OsSemCreate	los_sem.c (kernel\base\ipc)
OsSetMainTask	los_task.c (kernel\base\core)
ShmInit	shm.c (kernel\base\vm)
OsSwtmrInit	los_swtmr.c (kernel\base\core)
OsVmSpaceInitCommon	los_vm_map.c (kernel\base\vm)
LOS_QueueCreate	los_queue.c (kernel\base\ipc)
LOS_QueueCreate	los_queue.c (kernel\base\ipc)
LOS_QueueCreate	los_queue.c (kernel\base\ipc)
OsVmPhysFreeListInit	los_vm_phys.c (kernel\base\vm)
OsCreateProcessGroup	los_process.c (kernel\base\core)
OsCreateProcessGroup	los_process.c (kernel\base\core)
TelnetOpen	telnet_dev.c (net\telnet\src)
LOS_MuxInit	los_mux.c (kernel\base\ipc)

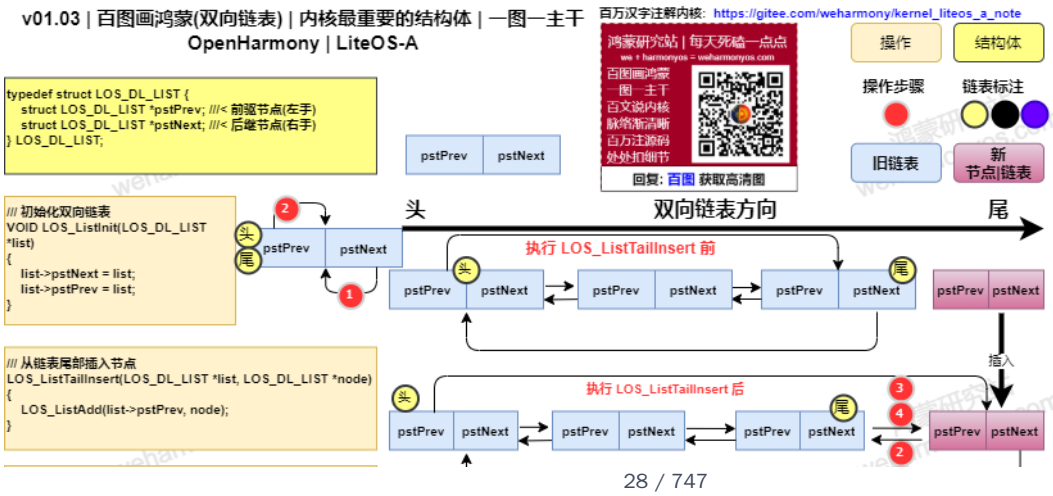
- 可以豪不夸张的说理解 LOS_DL_LIST 及相关函数是读懂鸿蒙内核的关键。前后指针(左右触手)灵活的指挥着系统精准的运行，越是深挖内核代码越是能体会到它在内核举足轻重的地位， 笔者仿佛看到了无数双手前后相连，拉起了一个个双向循环链表，把指针的高效能运用到了极致，这也许就是编程的艺术吧！

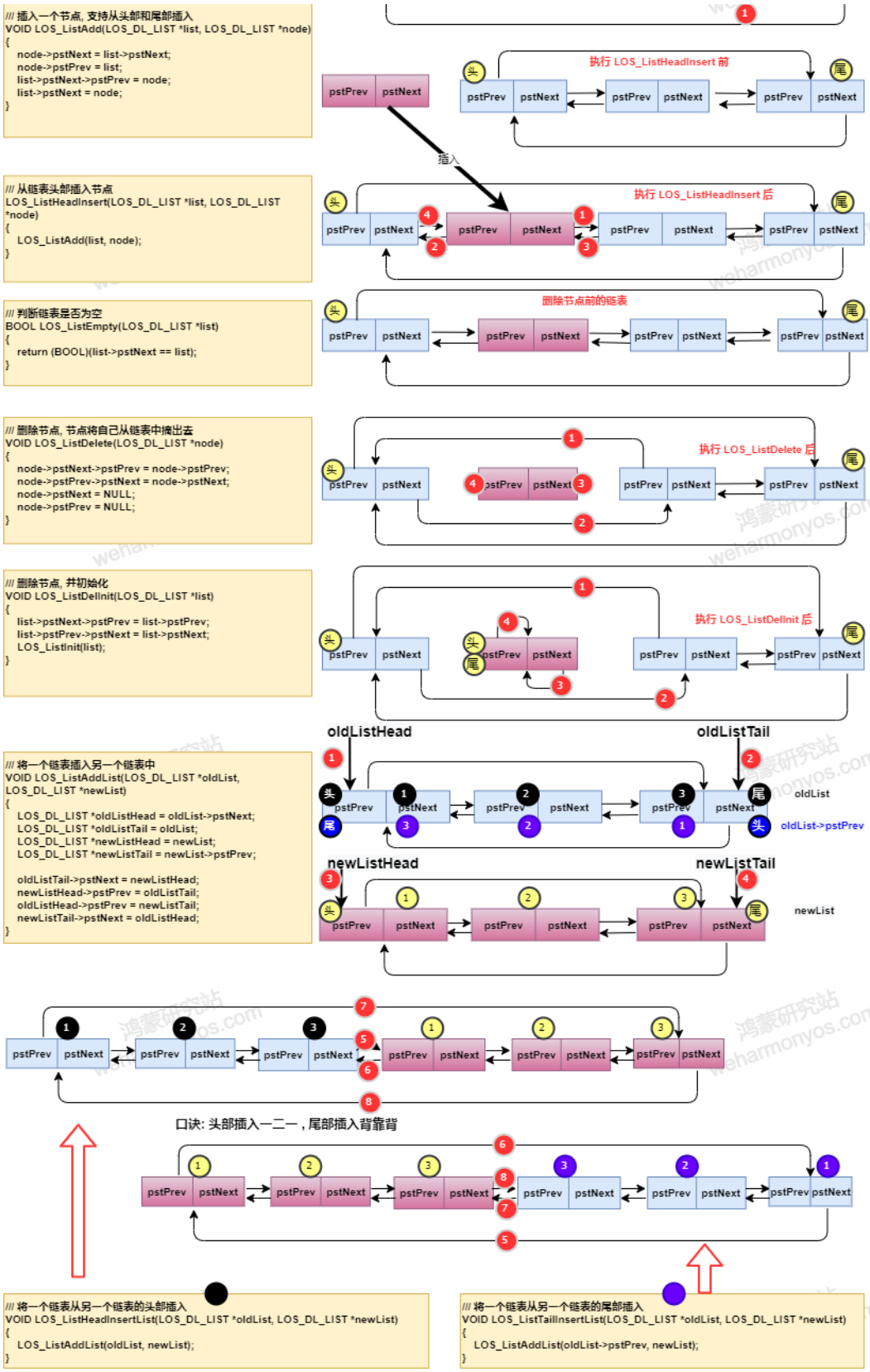
怎么实现 ？

鸿蒙系统中的双向链表模块为用户提供下面几个接口。

```
功能分类          接口名                  描述
初始化链表         LOS_ListInit            对链表进行初始化
增加节点          LOS_ListAdd             将新节点添加到链表中
在链表尾部插入节点   LOS_ListTailInsert       将节点插入到双向链表尾部
在链表头部插入节点   LOS_ListHeadInsert        将节点插入到双向链表头部
删除节点          LOS_ListDelete          将指定的节点从链表中删除
判断双向链表是否为空 LOS_ListEmpty            判断链表是否为空
删除节点并初始化链表 LOS_ListDelInit          将指定的节点从链表中删除使用该节点初始化链表
在链表尾部插入链表   LOS_ListTailInsertList 将链表插入到双向链表尾部
在链表头部插入链表   LOS_ListHeadInsertList 将链表插入到双向链表头部
```

其插入 | 删除 | 遍历操作是它最常用的社交三大件，若不理解透彻在分析源码过程中很容易卡壳。虽在网上能找到很多它的图,但怎么看都不是自己想要的，干脆重画了它的主要操作。



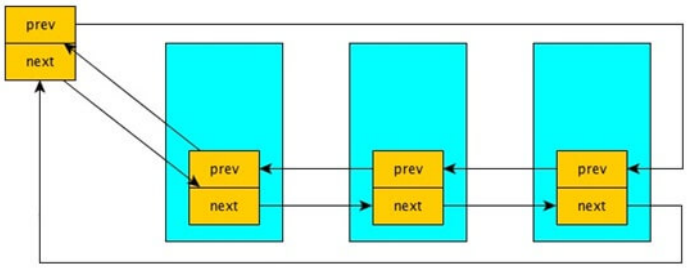



```
//将指定节点初始化为双向链表节点
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListInit(LOS_DL_LIST *list)
{
    list->pstNext = list;
    list->pstPrev = list;
```

```
}

//将指定节点挂到双向链表头部
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListAdd(LOS_DL_LIST *list, LOS_DL_LIST *node)
{
   node->pstNext = list->pstNext;
   node->pstPrev = list;
   list->pstNext->pstPrev = node;
   list->pstNext = node;
}
//将指定节点从链表中删除，自己把自己摘掉
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListDelete(LOS_DL_LIST *node)
{
   node->pstNext->pstPrev = node->pstPrev;
   node->pstPrev->pstNext = node->pstNext;
   node->pstNext = NULL;
   node->pstPrev = NULL;
}
//将指定节点从链表中删除，并使用该节点初始化链表
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListDelInit(LOS_DL_LIST *list)
{
   list->pstNext->pstPrev = list->pstPrev;
   list->pstPrev->pstNext = list->pstNext;
   LOS_ListInit(list);
}
```

数据在哪 ？

有好几个同学问数据在哪？ 确实 LOS_DL_LIST 这个结构看起来怪怪的，它竟没有数据域！所以看到这个结构的人第一反应就是我们怎么访问数据？其实 LOS_DL_LIST 不是拿来单独用的，它是寄生在内容结构体上的，谁用它谁就是它的数据。看图就明白了。



强大的宏

除了内联函数，对双向链表的初始化，偏移定位，遍历 等等操作提供了更强大的宏支持。使内核以极其简洁高效的代码实现复杂逻辑的处理。

```
//定义一个节点并初始化为双向链表节点
#define LOS_DL_LIST_HEAD(list) LOS_DL_LIST list = { &(list), &(list) }

//获取指定结构体内的成员相对于结构体起始地址的偏移量
#define LOS_OFF_SET_OF(type, member) ((UINTPTR)&((type *)0)->member)

//获取包含链表的结构体地址，接口的第一个入参表示的是链表中的某个节点，第二个入参是要获取的结构体名称，第三个入参是链表在该结构体中的名称
#define LOS_DL_LIST_ENTRY(item, type, member) \
   ((type *)(VOID *)((CHAR *)(item) - LOS_OFF_SET_OF(type, member)))

//遍历双向链表
#define LOS_DL_LIST_FOR_EACH(item, list) \
   for (item = (list)->pstNext;        \
        (item) != (list);              \
        item = (item)->pstNext)

//遍历指定双向链表，获取包含该链表节点的结构体地址，并存储包含当前节点的后继节点的结构体地址
#define LOS_DL_LIST_FOR_EACH_ENTRY_SAFE(item, next, list, type, member)              \
   for (item = LOS_DL_LIST_ENTRY((list)->pstNext, type, member),                     \
        next = LOS_DL_LIST_ENTRY((item)->member.pstNext, type, member);              \
        &(item)->member != (list);                                                   \
        item = next, next = LOS_DL_LIST_ENTRY((item)->member.pstNext, type, member))
```

```
//遍历指定双向链表，获取包含该链表节点的结构体地址
#define LOS_DL_LIST_FOR_EACH_ENTRY(item, list, type, member)           \
    for (item = LOS_DL_LIST_ENTRY((list)->pstNext, type, member);      \
        &(item)->member != (list);                                     \
        item = LOS_DL_LIST_ENTRY((item)->member.pstNext, type, member))
```

LOS_OFF_SET_OF 和 LOS_DL_LIST_ENTRY

这里要重点说下 `LOS_OFF_SET_OF` 和 `LOS_DL_LIST_ENTRY` 两个宏，个人认为它们是链表操作中最关键，最重要的宏。在读内核源码的过程会发现 LOS_DL_LIST_ENTRY 高频的出现，它们解决了通过结构体的任意一个成员变量来找到结构体的入口地址。 这个意义重大，因为在运行过程中，往往只能提供成员变量的地址，那它是如何做到通过个人找到组织的呢？

- `LOS_OFF_SET_OF` 用于找到成员变量在结构体中的相对偏移位置， 在系列篇 **(用栈方式篇)** 中 已说过 鸿蒙采用的是递减满栈的方式。而使用 **((type *)0)->member** 是获取 `struct` 成员偏移量的技巧，需要编译器的支持，这种方法背后的想法是让编译器假设结构起始地址位于零并计算成员的地址。 以 `ProcessCB` 结构体举例

```
typedef struct ProcessCB {
    // ...
    LOS_DL_LIST         pendList;                 /**< Block list to which the process belongs | 进程所在的阻塞列表,进程因阻塞挂入相应的链表.*/
    LOS_DL_LIST         childrenList;              /**< Children process list | 孩子进程都挂到这里,形成双循环链表*/
    LOS_DL_LIST         exitChildList;             /**< Exit children process list | 要退出的孩子进程链表，白发人要送黑发人.*/
    LOS_DL_LIST         siblingList;              /**< Linkage in parent's children list | 兄弟进程链表, 56个民族是一家,来自同一个父进程.*/
    LOS_DL_LIST         subordinateGroupList;        /**< Linkage in group list | 进程组员链表*/
    LOS_DL_LIST         threadSiblingList;          /**< List of threads under this process | 进程的线程(任务)列表 */
    LOS_DL_LIST         waitList;     /**< The process holds the waitLits to support wait/waitpid | 父进程通过进程等待的方式，回收子进程资源，获取子进程
} LosProcessCB;
```

`waitList` 因为在结构体的后面，所以它内存地址会比在前面的 `pendList` 高，有了顺序方向就很容易得到 `ProcessCB` 的第一个变量的地址。 `LOS_OFF_SET_OF` 就是干这个的，含义就是相对第一个变量地址，你 `waitList` 偏移了多少。

- 如此，当外面只提供 `waitList` 的地址再减去偏移地址 就可以得到 `ProcessCB` 的起始地址。

```
#define LOS_DL_LIST_ENTRY(item， type， member) \
    ((type *)(VOID *)((CHAR *)(item) - LOS_OFF_SET_OF(type， member)))
```

当然如果提供 `pendList` 或 `exitChildList` 的地址道理一样。 `LOS_DL_LIST_ENTRY` 实现了通过任意成员变量来获取 `ProcessCB` 的起始地址。

OsGetTopTask

有了以上对链表操作的宏，可以使得代码变得简洁易懂，例如在调度算法中获取当前最高优先级的任务时，就需要遍历整个进程和其任务的就绪列表。 `LOS_DL_LIST_FOR_EACH_ENTRY` 高效的解决了层层循环的问题。

```
LITE_OS_SEC_TEXT_MINOR LosTaskCB *OsGetTopTask(VOID)
{
    UINT32 priority， processPriority;
    UINT32 bitmap;
    UINT32 processBitmap;
    LosTaskCB *newTask = NULL;
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 cpuid = ArchCurrCpuid();
#endif
    LosProcessCB *processCB = NULL;
    processBitmap = g_priQueueBitmap;
    while (processBitmap) {
        processPriority = CLZ(processBitmap);
        LOS_DL_LIST_FOR_EACH_ENTRY(processCB， &g_priQueueList[processPriority]， LosProcessCB， pendList) {
            bitmap = processCB->threadScheduleMap;
            while (bitmap) {
                priority = CLZ(bitmap);
                LOS_DL_LIST_FOR_EACH_ENTRY(newTask， &processCB->threadPriQueueList[priority]， LosTaskCB， pendList) {
#if (LOSCFG_KERNEL_SMP == YES)
                    if (newTask->cpuAffiMask & (1U << cpuid)) {
#endif
                        newTask->taskStatus &= ~OS_TASK_STATUS_READY;
```

```
                    OsPriQueueDequeue(processCB->threadPriQueueList，
                                &processCB->threadScheduleMap，
                                &newTask->pendList);
                    OsDequeEmptySchedMap(processCB);
                    goto OUT;
#if (LOSCFG_KERNEL_SMP == YES)
                }
#endif
            }
            bitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - priority - 1));
        }
    }
    processBitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - processPriority - 1));
  }

OUT:
  return newTask;
}
```

结构体的最爱

LOS_DL_LIST 是复杂结构体的最爱，再以 ProcessCB (进程控制块)举例，它是描述一个进程的所有信息，其中用到了 7 个双向链表，这简直比章鱼还牛逼，章鱼也才四双触手，但进程有 7 双(14 只)触手。

```
typedef struct ProcessCB {
  LOS_DL_LIST       pendList;                 /**< Block list to which the process belongs | 进程所在的阻塞列表,进程因阻塞挂入相应的链表.*/
  LOS_DL_LIST       childrenList;             /**< Children process list | 孩子进程都挂到这里,形成双循环链表*/
  LOS_DL_LIST       exitChildList;            /**< Exit children process list | 要退出的孩子进程链表，白发人要送黑发人.*/
  LOS_DL_LIST       siblingList;              /**< Linkage in parent's children list | 兄弟进程链表, 56个民族是一家,来自同一个父进程.*/
  LOS_DL_LIST       subordinateGroupList;     /**< Linkage in group list | 进程组员链表*/
  LOS_DL_LIST       threadSiblingList;        /**< List of threads under this process | 进程的线程(任务)列表 */
  LOS_DL_LIST       waitList;    /**< The process holds the waitLits to support wait/waitpid | 父进程通过进程等待的方式，回收子进程资源，获取子进程
} LosProcessCB;
```

解读

- pendList 个人认为它是鸿蒙内核功能最多的一个链表，它远不止字面意思阻塞链表这么简单，只有深入解读源码后才能体会它真的是太会来事了，一般把它理解为阻塞链表就行。上面挂的是处于阻塞状态的进程。
- childrenList 孩子链表，所有由它fork出来的进程都挂到这个链表上。上面的孩子进程在死亡前会将自己从上面摘出去，转而挂到 exitChildList 链表上。
- exitChildList 退出孩子链表，进入死亡程序的进程要挂到这个链表上，一个进程的死亡是件挺麻烦的事，进程池的数量有限，需要及时回收进程资源，但家族管理关系复杂，要去很多地方消除痕迹。尤其还有其他进程在看你笑话，等你死亡(wait / waitpid)了通知它们一声。
- siblingList 兄弟链表，和你同一个父亲的进程都挂到了这个链表上。
- subordinateGroupList 朋友圈链表，里面是因为兴趣爱好(进程组)而挂在一起的进程，它们可以不是一个父亲，不是一个祖父，但一定是同一个老祖宗(用户态和内核态根进程)。
- threadSiblingList 线程链表，上面挂的是进程ID都是这个进程的线程(任务)，进程和线程的关系是1:N的关系，一个线程只能属于一个进程。这里要注意任务在其生命周期中是不能改所属进程的。
- waitList 是等待子进程消亡的任务链表，注意上面挂的是任务。任务是通过系统调用

  ```
  pid_t wait(int *status);
  pid_t waitpid(pid_t pid, int *status, int options);
  ```

  将任务挂到 waitList 上。鸿蒙 waitpid 系统调用为 SysWait ，具体看进程回收篇。

双向链表是内核最重要的结构体，精读内核的路上它会反复的映入你的眼帘，理解它是理解内核运作的关键所在!

百文说内核 | 抓住主脉络

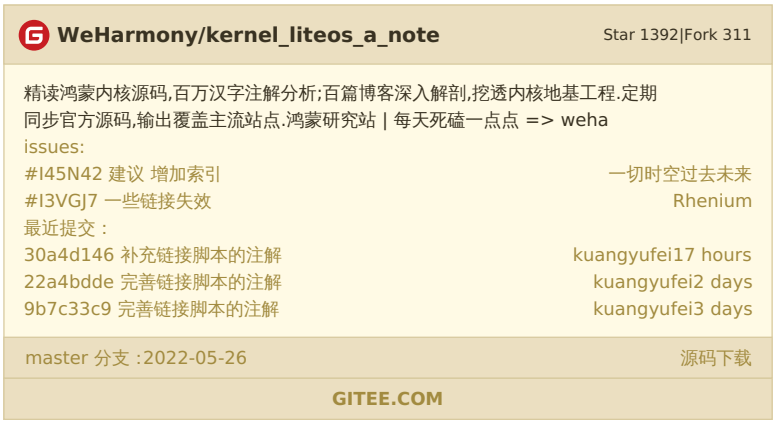
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

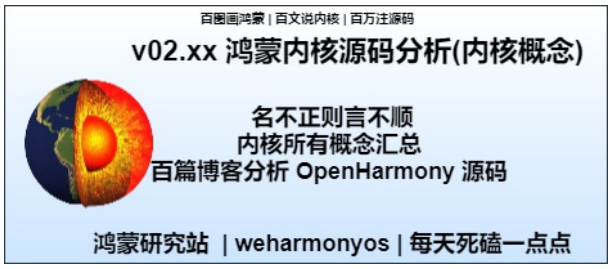


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

02_内核概念篇

本篇关键词：十大模块、概念集、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

进程概念

- 进程：
- 进程空间：
- 线性区：
- 红黑树：
- 堆：
- 栈：

任务概念

- 任务：
- 用户态：
- 内核态：
- 栈：

通讯概念

- 核间通讯：
- 进程通讯：
- 自旋锁：
- 互斥锁：
- 用户快锁：
- 信号量：
- 信号：
- 事件：
- 消息队列：
- LiteIPC：
- 共享内存：

内存概念

- 虚拟地址：
- 物理地址：
- 虚实映射：
- 伙伴算法：

文件概念

处理器概念

网络概念

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

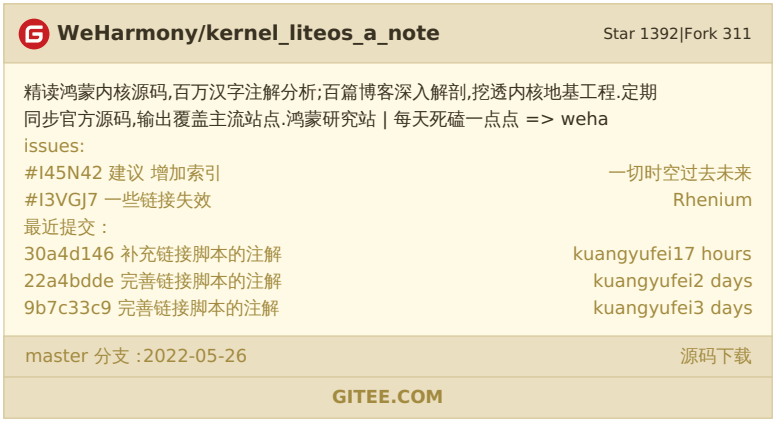
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

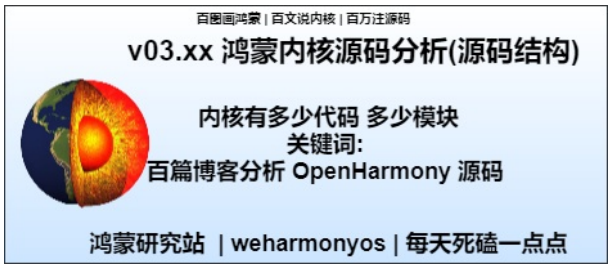


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

03_源码结构篇

本篇关键词：LiteOS-A 、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

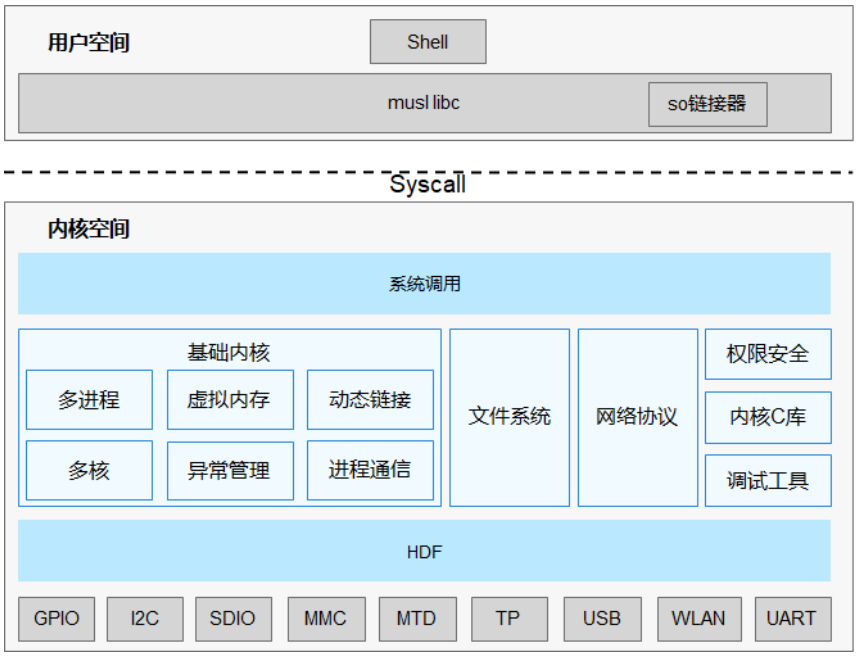
基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

LiteOS-A 内核

OpenHarmony LiteOS-A 内核是基于 Huawei LiteOS 内核演进发展的新一代内核， Huawei LiteOS 是面向 IoT 领域构建的轻量级物联网操作系统。在 IoT 产业高速发展的潮流中， OpenHarmony LiteOS-A 内核能够带给用户小体积、低功耗、高性能的体验以及统一开放的生态系统能力，新增了丰富的内核机制、更加全面的 POSIX 标准接口以及统一驱动框架 HDF （OpenHarmony Driver Foundation）等，为设备厂商提供了更统一的接入方式，为 OpenHarmony 的应用开发者提供了更友好的开发体验。

架构图



(图一)

百篇博客目录

(图二)

源码结构

(图三)

```
/kernel/liteos_a
├── apps                # 用户态的init和shell应用程序
├── arch                # 体系架构的目录，如arm等
│   └── arm             # arm架构代码
├── bsd                 # freebsd相关的驱动和适配层模块代码引入，例如USB等
├── compat              # 内核接口兼容性目录
│   └── posix           # posix相关接口
├── drivers             # 内核驱动
│   └── char            # 字符设备
│       ├── mem         # 访问物理IO设备驱动
│       ├── quickstart  # 系统快速启动接口目录
│       ├── random      # 随机数设备驱动
│       └── video       # framebuffer驱动框架
├── fs                  # 文件系统模块，主要来源于NuttX开源项目
│   ├── fat             # fat文件系统
│   ├── jffs2           # jffs2文件系统
│   ├── include         # 对外暴露头文件存放目录
│   ├── nfs             # nfs文件系统
│   ├── proc            # proc文件系统
│   ├── ramfs           # ramfs文件系统
│   └── vfs             # vfs层
├── kernel              # 进程、内存、IPC等模块
│   ├── base            # 基础内核，包括调度、内存等模块
│   ├── common          # 内核通用组件
│   ├── extended        # 扩展内核，包括动态加载、vdso、liteipc等模块
│   ├── include         # 对外暴露头文件存放目录
│   └── user            # 加载init进程
├── lib                 # 内核的lib库
├── net                 # 网络模块，主要来源于lwip开源项目
├── platform            # 支持不同的芯片平台代码，如Hi3516DV300等
│   ├── hw              # 时钟与中断相关逻辑代码
│   ├── include         # 对外暴露头文件存放目录
│   └── uart            # 串口相关逻辑代码
├── security            # 安全特性相关的代码，包括进程权限管理和虚拟id映射管理
├── syscall             # 系统调用
├── testsuites          # 单元测试用例
├── tools               # 构建工具及相关配置和代码
└── zzz                 # 中文注解版新增目录
```

解读

- **kernel** ：是最重要的模块，包括了百篇博客中的 **进程管理** 、**任务管理** 、**内存管理** 、**通讯机制** 四大核心模块。

- **arch** ：开机代码就在此，包括了所有内核汇编代码，管理处理器，协处理器，中断控制器等核心硬件实现代码，包括了百篇博客中的 **硬件架构**、**内核汇编** 两个大模块。
- **fs** ：文件系统可比作内核的数据库，保存永久/临时性数据，文件系统因技术的更新，各个公司的利益保护等等诸多原因，呈现百花齐放，多达几十种， 通过加入虚拟层 vfs 统一对外部模块提供访问各文件系统服务，这些文件系统可分成以下四种，
  - 磁盘文件系统，包括 `fat` 、 `ntfs` 、 `ext4` 、 `zfs` ...
  - 闪存文件系统，包括 `jffs2` 、 `yaffs` ...
  - 内存文件系统，包括 `proc` 、 `ramfs` 、 `tmpfs`
  - 网络文件系统，包括 `nfs`（Network File System）是一种将远程主机上的分区（目录）经网络挂载到本地系统的一种机制，是一种分布式文件系统，力求客户端主机可以访问服务器端文件，并且其过程与访问本地存储时一样，它由Sun公司开发，于1984年发布。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

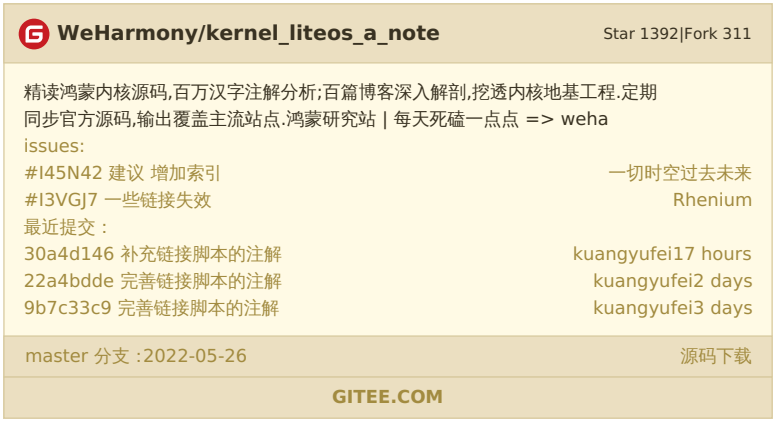
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

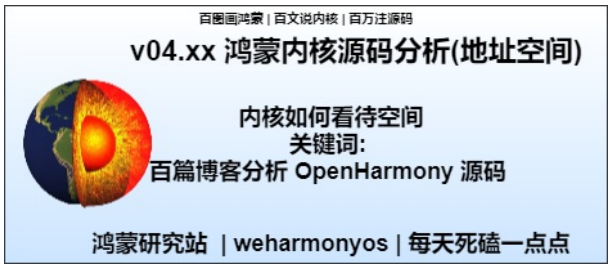


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

04_地址空间篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

距离产生美

计算机怎么理解空间

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

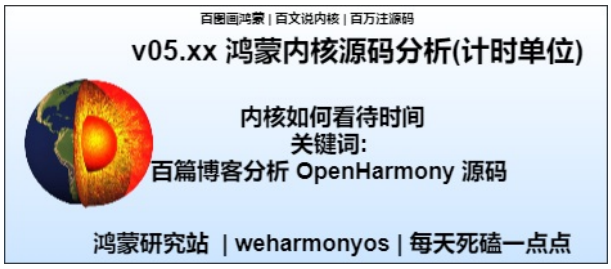
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

05_计时单位篇

本篇关键词：时钟周期、机器周期、指令周期、分频



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

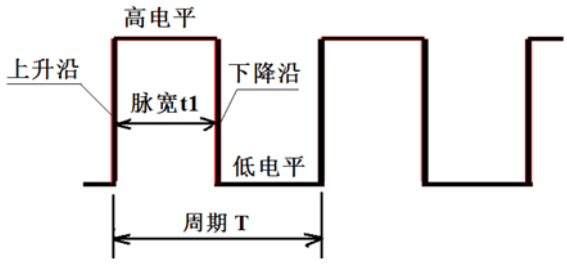
- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

时间都去哪了？

- 施教授说时间就是运动，地球围绕太阳公转一圈叫一年，地球自转一圈叫一天，地球自转了365天才围绕太阳转完了一圈，所以一年等于365天，这些都是运动。
- 我们能感知到时间的流逝，大概也是因为我们身体在运动，细胞一次次的复制，衰老，死亡，直至最后一次再也无法复制代表着生命彻底终结。我们日常总是被时间催促着走，就这样一天又一天，一年再一年。疫情都第三年了，口罩也带习惯了，不敢回头看，常常感叹，时间都去哪了？还没好好感受年轻就老了，还没好好看看你眼睛就花了，转眼就只剩下满脸的皱纹了。哎呀！老衲快不行了，要泪崩了 。。。

计算机怎么理解时间

- 没有时间，我们的生活将是无序的，对计算机更是如此的，而且只会更敏感，否则如何能做到精确到纳秒级的运算，它是如何理解时间的呢？它的敏感源又是什么呢 ？ 答案是：**晶振** 全称是石英晶体振荡器，是一种高精度和高稳定度的振荡器。位置一般在主板上，晶振为系统提供很稳定的脉冲信号，一秒钟内产生的脉冲次数也被称为系统时钟频率。一个标准的脉冲信号如 **(图一)** 所示。



- **时钟周期** 这里的时钟指的就是晶振，也叫晶振周期/振荡周期。它是计算机的最小时间单元，其他周期只能是它的倍数。
- **机器周期** 为什么要搞出来个机器周期，因为到了 `CPU` 操作层面时钟周期不好用，有点类似美元和人民币的关系，美元已经是世界货币了，但每个国家国情不同，操作起来还是得用本国货币方便。一个机器周期等于多个时钟周期。它是 `CPU` 完成一个基本**操作**的时间单元。比如: 取指、译码、存储器读/写、运算等等操作。
- **指令周期** 它是 `CPU` 完成一条**指令**的时间。又称提取－执行周期（fetch-and-execute cycle）是指CPU要执行一条机器指令经过的步骤，由若干机器周期组成。不同的机器分解指令周期的方式也不同，有的处理器对每条指令分解出相同数量的机器周期，另一些处理器根据指令的复杂程度分解出不同数量的机器周期，**(图二)** 为 Cortex-A8处理器架构图，图中清晰的说明了指令从 **取指** - > **译码** -> **执行** -> **储存**四个阶段。

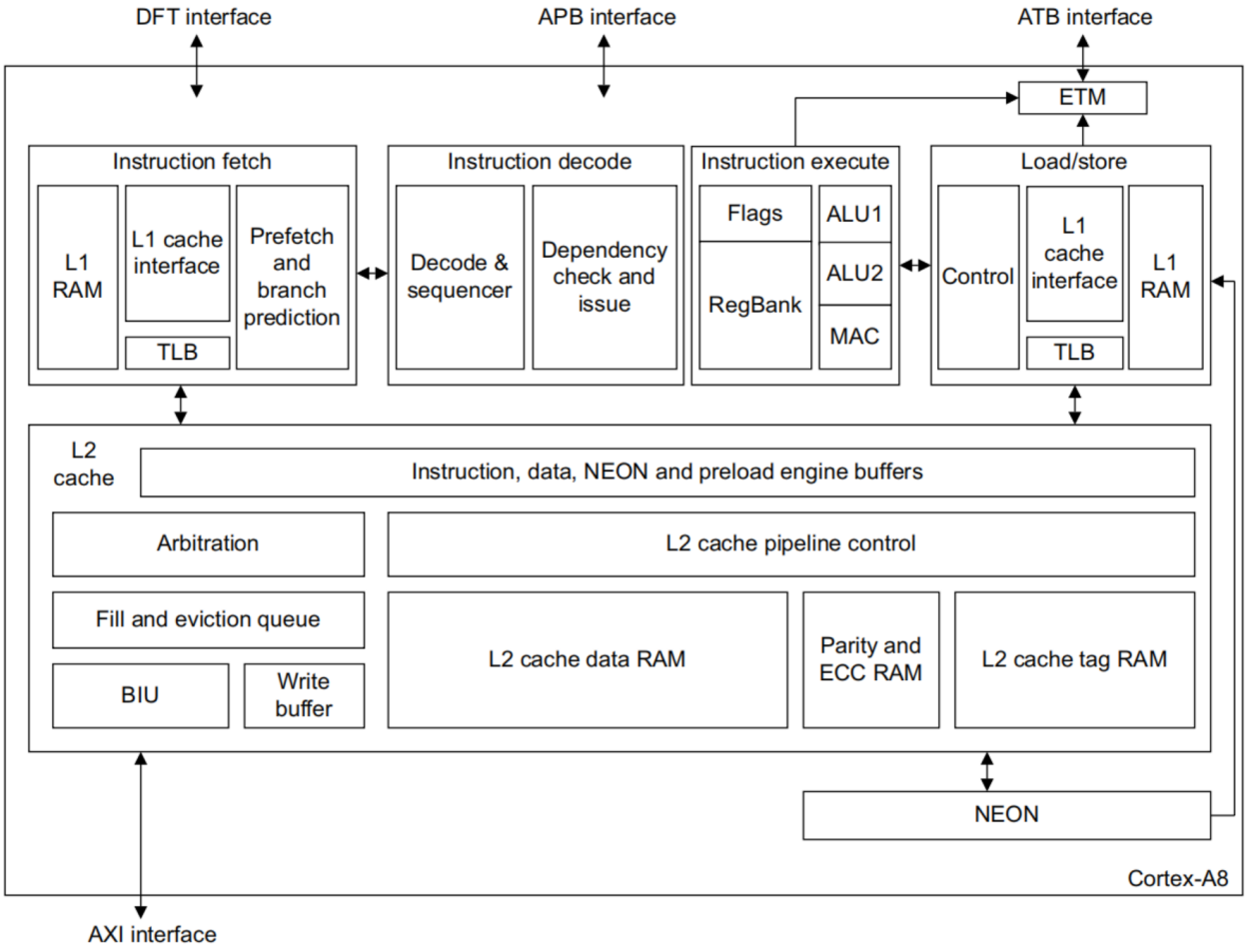
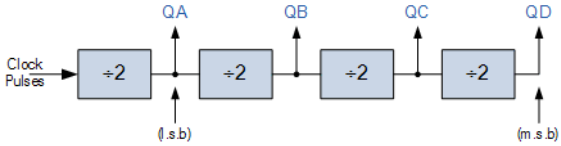
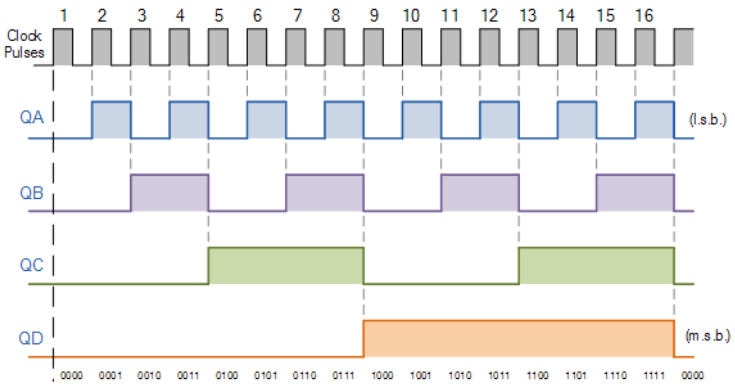


Figure 1-1 Cortex-A8 block diagram

- **总线周期** 它是 CPU 操作总线设备的时间，由于存贮器和 I/O 端口是挂接在总线上的，对它们的访问，是通过总线实现的。通常将进行一次访问所需时间称为一个总线周期。这种访问速度较慢，因硬件发展的原因，各个设备的工作频率无法同步，甚至相差几个数量级， CPU 很快，硬盘很慢，大家又需要协同工作，愉快的玩耍，就出现了**分频**概念，将高频信号变成低频信号，**(图三)** 显示分频器分出来的四个频率代表**二分频、四分频、八分频 、十六分频**





- 从时间长短角度总结下 **时钟周期 < 机器周期 < 指令周期< 总线周期**

周期 、Tick 、秒

- 对于 CPU 来说，它的计时单位是 Cycle (时钟周期)，对于操作系统来说它的计时单位是 Tick (节拍)，而用户的时间单位是秒/毫秒，这就存在一个互逆的换算过程，鸿蒙是如何换算的呢？看几个宏定义就清楚了。

```
#ifndef OS_SYS_CLOCK //HZ:是每秒中的周期性变动重复次数的计量
#define OS_SYS_CLOCK (get_bus_clk()) //系统主时钟频率 例如:50000000 即20纳秒震动一次
#endif
#ifndef LOSCFG_BASE_CORE_TICK_PER_SECOND
#define LOSCFG_BASE_CORE_TICK_PER_SECOND 100 //每秒Tick数，内核定时器触发单位
#endif
#define OS_CYCLE_PER_TICK (g_sysClock / LOSCFG_BASE_CORE_TICK_PER_SECOND) //每个tick时钟周期数
```

  Tick 由系统集成商根据实际情况来配置， Tick 大系统更灵敏，对CPU的要求更高，也会带来更多的性能损耗。 从代码中可以计算出 **OS_CYCLE_PER_TICK = 50万** 个时钟周期，假设指令平均执行周期按10个时钟周期算(实际不能这么计算)，可执行5万条机器指令。

- 在 CP15 协处理器中有个专门的 **CNTFRQ** 寄存器用于存储时钟周期，详见代码：

```
/*
* 见于 << arm 架构参考手册>> B4.1.21 处 CNTFRQ寄存器表示系统计数器的时钟频率。
* 这个寄存器是一个通用的计时器寄存器。
* MRC p15, 0, <Rt>, c14, c0, 0 ; Read CNTFRQ into Rt
* MCR p15, 0, <Rt>, c14, c0, 0 ; Write Rt to CNTFRQ
*/
UINT32 HalClockFreqRead(VOID)
{
    return READ_TIMER_REG32(TIMER_REG_CNTFRQ);
}

VOID HalClockFreqWrite(UINT32 freq)
{
    WRITE_TIMER_REG32(TIMER_REG_CNTFRQ, freq);
}
```

- 鸿蒙内核以全局变量 g_sysClock 表示系统时钟，由硬时钟模块完成对它的初始化，并创建硬定时器 OsTickHandler

```
//硬时钟初始化
LITE_OS_SEC_TEXT_INIT VOID HalClockInit(VOID)
{
    UINT32 ret;

    g_sysClock = HalClockFreqRead();//读取CPU的时钟频率
    ret = LOS_HwiCreate(OS_TICK_INT_NUM, MIN_INTERRUPT_PRIORITY, 0, OsTickHandler, 0);//创建硬中断定时器
    if (ret != LOS_OK) {
        PRINT_ERR("%s, %d create tick irq failed, ret:0x%x\n", __FUNCTION__, __LINE__, ret);
    }
}
```

- 用户与操作系统之间的时间转换函数为

```
 LITE_OS_SEC_TEXT_MINOR UINT32 LOS_MS2Tick(UINT32 millisec)
 {
    if (millisec == OS_MAX_VALUE) {
        return OS_MAX_VALUE;
    }
    return ((UINT64)millisec * LOSCFG_BASE_CORE_TICK_PER_SECOND) / OS_SYS_MS_PER_SECOND;
 }
 LITE_OS_SEC_TEXT_MINOR UINT32 LOS_Tick2MS(UINT32 tick)
 {
    return ((UINT64)tick * OS_SYS_MS_PER_SECOND) / LOSCFG_BASE_CORE_TICK_PER_SECOND;
 }
 LITE_OS_SEC_TEXT_MINOR UINT32 OsNS2Tick(UINT64 nanoseconds)
 {
    const UINT32 nsPerTick = OS_SYS_NS_PER_SECOND / LOSCFG_BASE_CORE_TICK_PER_SECOND;
    UINT64 ticks = (nanoseconds + nsPerTick - 1) / nsPerTick;
    if (ticks > OS_MAX_VALUE) {
        ticks = OS_MAX_VALUE;
    }
    return (UINT32)ticks;
 }
```

Tick硬中断 | OsTickHandler

```
LITE_OS_SEC_TEXT VOID OsTickHandler(VOID)
{
#ifdef LOSCFG_SCHED_TICK_DEBUG
   OsSchedDebugRecordData();
#endif
#ifdef LOSCFG_KERNEL_VDSO
   OsVdsoTimevalUpdate();//更新vdso数据页时间，vdso可以直接在用户进程空间绕过系统调用获取系统时间(例如:gettimeofday)
#endif
#ifdef LOSCFG_BASE_CORE_TICK_HW_TIME
   HalClockIrqClear(); /* diff from every platform */
#endif
   OsSchedTick();//由时钟发起的调度
}
```

- **OsTickHandler** 是控制内核正常运行的发动机，本质是硬中断，在所有硬中断中优先级最低，注意没有硬中断内核将是一片死寂，毫无生气。硬件定时器是有时间规律的重复硬中断，中断处理函数反复的检查任务列表，是否当前任务继续执行，是否时间片到了该切换任务执行了，是否有更高优先级任务需先执行。同时检查软件定时器的任务列表的时间是否到了，可以说软件定时器的时间基础就是硬件定时器，其精确度是一个 `Tick` 。
- 涉及内容具体可翻看系列篇 **中断管理篇**、**调度机制篇**、**软件定时器篇** 、**VDSO篇**

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

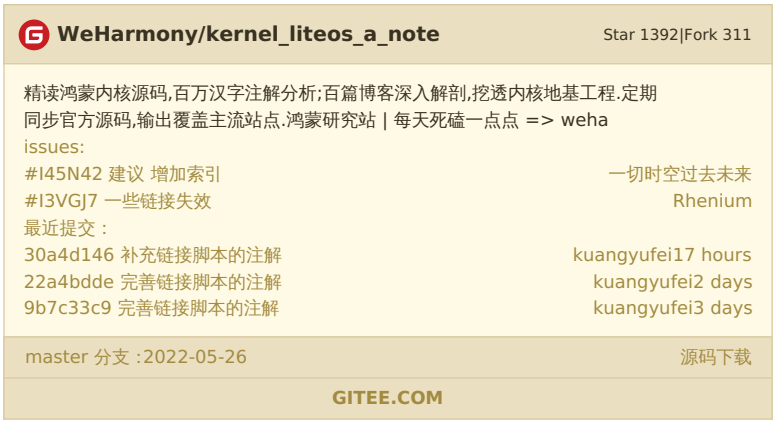
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐

通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

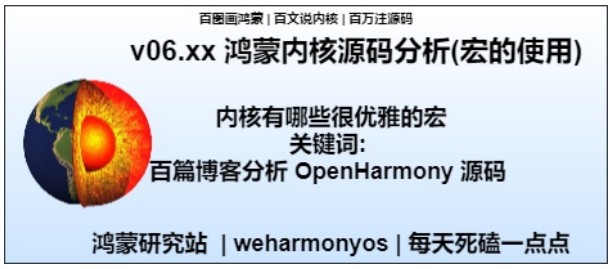


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

06_优雅的宏篇

本篇关键词：__asm__ 、volatile 、DSB



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

什么是宏

在真正的编译开始之前，编译器也需要热热身，俗话说的好："前戏工作不能少，后续推进质量好"。**宏** 就是编译器的前戏部分(预处理)，简单的说就是复制粘贴。大多数人第一次接触 **宏** 多半是因为 **π**

```
#define PI 3.1415926
```

人类进化的结果是不擅长做记忆类操作，但这并非缺点，相比**3.1415926** ，**π** 能减少体能的消耗，这种将复杂信息取个 **别名** 方便记录和传播的做法人类用了几千年。据说在猴子的世界里也有专门负责盯梢的，看到远方有狮子来了，会手舞足蹈描绘狮子的外形和模仿声音以此告知同伴什么动物来了而躲避危险。但人类只需要喊一句 "狮子来了"足以，这才是食物链顶端的做法，单给万千事物取名的这一个能力就能甩动物世界几百条街。更别说以此衍生出来的**畊宏女孩** ，**山东彭于晏** 等等流行网络词汇，其做法都是用极低的成本达到极快的传播。

macro 是 **macro instruction** 的缩写，将某个输入映射到替代输出的一种规则或模式，输入和输出可以是一系列词汇标记或字符，或语法树。为什么中文被翻译成 **宏** 可能是因为单词 **宏观(macro)** 。

若将宏只停留在 **3.1415926** 就简单肤浅了，要知道任何看似简单的背后都可以击鼓传花，熟能生巧，蛤蟆功练到最高境界那也能成为天下第一的。站长真正被宏的强大震撼到的是看了侯俊杰先生的 **<<深入浅出MFC>>** 对消息机制的实现 ，当时不由的惊呼 "靠，原来还能这么玩！" ，从此对宏爱不释手，所以百篇博客必须为此开一篇，更何况鸿蒙内核对宏的讲究也并没有让人失望。想了解 **MFC** 消息映射宏的可 查看 >> 三个奇怪的宏，一张巨大的网

一段难懂的代码

能看懂以下代码中 **for 循环** 的**C语言**功底非同一般，绝对的大神。

```
STATIC VOID HPFPriorityRestore(LosTaskCB *owner, const LOS_DL_LIST *list, const SchedParam *param)
{
    LosTaskCB *pendedTask = NULL;
    // ...
    for (pendedTask = ((LosTaskCB *)(VOID *)((CHAR *)((list)->pstNext) - ((UINTPTR)&((LosTaskCB *)0)->pendList)));
         &(pendedTask)->pendList != (list);
         pendedTask = ((LosTaskCB *)(VOID *)((CHAR *)((pendedTask)->pendList.pstNext) - ((UINTPTR)&((LosTaskCB *)0)->pendList))))
    {
        SchedHPF *pendSp = (SchedHPF *)&pendedTask->sp;
        if ((pendedTask->ops == owner->ops) && (priority != pendSp->priority)) {
```

```
            LOS_BitmapClr(&sp->priBitmap, pendSp->priority);
        }
    }
}
```

天天跟这样的代码打交道容易得脑梗，如果换成以下的样子就简洁了很多

```
STATIC VOID HPFPriorityRestore(LosTaskCB *owner, const LOS_DL_LIST *list, const SchedParam *param)
{
    LosTaskCB *pendedTask = NULL;
    // ...
    LOS_DL_LIST_FOR_EACH_ENTRY(pendedTask, list, LosTaskCB, pendList) {
        SchedHPF *pendSp = (SchedHPF *)&pendedTask->sp;
        if ((pendedTask->ops == owner->ops) && (priority != pendSp->priority)) {
            LOS_BitmapClr(&sp->priBitmap, pendSp->priority);
        }
    }
}
```

LOS_DL_LIST_FOR_EACH_ENTRY(pendedTask, list, LosTaskCB, pendList) 含义如下：

- 遍历一个叫 **list** 的双向链表，链表上挂的是一个个叫 **LosTaskCB** 结构体节点。
- **LosTaskCB** 是通过其成员变量 **pendList** 挂到 **list** 上的。
- 解铃还须系铃人，将节点摘下来也需通过 **pendList** ，并通过地址偏移量找到 **LosTaskCB** 的开始地址后给交给变量 **pendedTask**处理。
- 使用两个嵌套宏 **LOS_DL_LIST_ENTRY** ， **LOS_OFF_SET_OF** 完成了以上操作 ，三个宏完整原型如下

  ```
  #define LOS_DL_LIST_FOR_EACH_ENTRY(item, list, type, member)            \
      for (item = LOS_DL_LIST_ENTRY((list)->pstNext, type, member);       \
          &(item)->member != (list);                                      \
          item = LOS_DL_LIST_ENTRY((item)->member.pstNext, type, member))

  #define LOS_DL_LIST_ENTRY(item, type, member) \
      ((type *)(VOID *)((CHAR *)(item) - LOS_OFF_SET_OF(type, member)))

  #define LOS_OFF_SET_OF(type, member) ((UINTPTR)&((type *)0)->member)
  ```

- 遍历宏很简洁优雅，内核关于双向链表的遍历都可以通过它来完成，其他操作具体可翻看 **双向链表篇**

除了简化对**双向链表**操作还有对 **红黑树**的操作，尝试解读下以下代码的含义

```
RB_WALK(pstTree, pstNode, pstWalk)
{
    OsRbDeleteNode(pstTree, pstNode);
    (VOID)pstTree->pfFree(pstNode);
}
RB_WALK_END(pstTree, pstNode, pstWalk);
```

```
#define RB_WALK(pstTree, pstNode, pstRbWalk) do {                               \
        LosRbWalk *(pstRbWalk) = NULL;      \
        pstRbWalk = LOS_RbCreateWalk(pstTree);  \
        (pstNode) = LOS_RbWalkNext(pstRbWalk);  \
        for (; NULL != (pstNode); (pstNode) = LOS_RbWalkNext(pstRbWalk)) {

#define RB_WALK_END(pstTree, pstNode, pstRbWalk) }                                  \
    LOS_RbDeleteWalk(pstRbWalk);                \
    }                                   \
    while (0);
```

寄存器操作

硬件设备的对外使用接口是 **寄存器**，阅读硬件生产商提供的 **Datasheet**（数据手册）是每个硬件工程师都需具备的基本素养。寄存器分 **专用** 和 **通用** 寄存器，驱动工程师根据数据手册配置的一般是专用寄存器，对这些寄存器不同位的设置对应了不同的功能。通用寄存器的使用一般是由编译器完成，此处不展开讲，后续 **编译器系列篇** 中会详细说明。

以协处理器 **cp15** 举例 ，它是CPU的助手(详见于 **协处理器篇**)，一共有 `16` 个寄存器 `32` 位的寄存器，其编号为 `C0 ~ C15` ，用来控制 `cache` 、 `TCM` 和存储器管理。**cp15** 寄存器都是复合功能寄存器，不同功能对应不同的内存实体，全由访问指令的参数来决定。读写这些寄存器必须使用 **MRC** 和 **MCR** 指令。

```
#define CP15_REG(CRn, Op1, CRm, Op2)    "p15, "#Op1", %0, "#CRn","#CRm","#Op2

#define MIDR              CP15_REG(c0, 0, c0, 0)    /*! Main ID Register | 主ID寄存器 */
#define MPIDR              CP15_REG(c0, 0, c0, 5)    /*! Multiprocessor Affinity Register | 多处理器关联寄存器给每个CPU制定一个逻辑地址*/
#define CCSIDR             CP15_REG(c0, 1, c0, 0)    /*! Cache Size ID Registers | 缓存大小ID寄存器*/
#define CLIDR              CP15_REG(c0, 1, c0, 1)    /*! Cache Level ID Register | 缓存登记ID寄存器*/
#define VPIDR              CP15_REG(c0, 4, c0, 0)    /*! Virtualization Processor ID Register | 虚拟化处理器ID寄存器*/
#define VMPIDR             CP15_REG(c0, 4, c0, 5)    /*! Virtualization Multiprocessor ID Register | 虚拟化多处理器ID寄存器*/

#define ARM_SYSREG_READ(REG)                    \
({                                      \
    UINT32 _val;                            \
    __asm__ volatile("mrc " REG : "=r" (_val)); \
    _val;                                   \
})

#define ARM_SYSREG_WRITE(REG, val)              \
({                                      \
    __asm__ volatile("mcr " REG :: "r" (val));  \
    ISB;                                    \
})

/// 获取当前CPUID
STATIC INLINE UINT32 ArchCurrCpuid(VOID)
{
#ifdef LOSCFG_KERNEL_SMP
    return ARM_SYSREG_READ(MPIDR) & MPIDR_CPUID_MASK;
#else//ARM架构通过MPIDR(Multiprocessor Affinity Register)寄存器给每个CPU指定一个逻辑地址。
    return 0;
#endif
}
```

在单CPU多核的情况下，内核是需要安排并记录各任务运行在哪些核心上，**ArchCurrCpuid** 是获取当前任务运行在具体哪个核上，代码中将宏 **ARM_SYSREG_READ(MPIDR)** 展开后变成

```
({
    UINT32 _val;
    __asm__ volatile("mrc p15, 0, %0, c0, c0,5" : "=r"(_val));
    _val;
})
```

- **__asm__** 或 **asm** 用来声明一个内联汇编表达式。>> 查看内嵌汇编语法
- **__volatile__** 或 **volatile** 是可选的。如果用了它，则是向编译器声明不允许对该内联汇编优化。
- 其中 **%0** 和 **"=r"(_val)** 意思是编译器将选择 **R0** 寄存器来接收指令结果并将 **R0** 的值赋给变量 **_val** ，为什么要这么做呢？因为对协处理器的读写必须通过寄存器，而在C语言层面是不能直接操作寄存器的。
- **_val;** 可理解为代码块的 **return方式** 以便执行接下去的 **& MPIDR_CPUID_MASK** 操作

DSB | DMB | ISB

内核中经常会出现 **DSB** 、**DMB** 、**ISB**、**WFI** ，它们有什么含义和作用呢？具体可翻看 **ARM 体系参考手册 | DSB on page A8-381**

```
#define DSB     __asm__ volatile("dsb" ::: "memory")
#define DMB     __asm__ volatile("dmb" ::: "memory")
#define ISB     __asm__ volatile("isb" ::: "memory")
#define WFI     __asm__ volatile("wfi" ::: "memory")
#define BARRIER __asm__ volatile(""":::"memory") ///< 空指令
#define WFE     __asm__ volatile("wfe" ::: "memory")
#define SEV     __asm__ volatile("sev" ::: "memory")
```

如果没有这些指令的存在会导致系统发生紊乱危象，存在的原因是因为 **流水线** 和 **缓冲区**

- **缓冲区**，写缓冲是为了提高存储器的总体访问效率而设的，但它会带出来一个副作用就是同步问题，会导致写内存的指令被延迟几个周期执行，因此对存储器的设置不能即刻生效，这会导致紧临着的下一条指令仍然使用旧的存储器设置——但程序员的本意显然是使用新的存储器设置。这种紊乱危象是后患无穷的，常会破坏未知地址的数据，有时也会产生非法地址访问。
- **流水线**

指令	全称	功能	
DMB	Data Memory Barrier(DMB) 数据存储器隔离	等待前面访存的指令完成后再执行后面的访存指令	A3.8.3
DSB	Data Synchronization Barrier 数据同步隔离	等待所有前面的指令完成后再执行后面的访存指令	A3.8.3
ISB	Instruction Synchronization Barrier (ISB) 指令同步隔离	等待流水线中所有指令执行完成后再执行后面的指令	A3.8.3
WFI	Wait For Interrupt 等待中断	等待中断，进入休眠模式。	B1.8.14
WFE	Wait For Event 等待事件	等待事件，如果没有之前该事件的记录，进入休眠模式；如果有的话，则清除事件锁存并继续执行；	B1.8.13
SEV	Send Event 发送事件	多处理器环境中向所有的处理器发送事件（包括自身）。	B1.8.13

- 严格程度 DMB < DSB < ISB
- ::: "memory" 强制编译器假设 RAM 所有内存单元均被汇编指令修改，这样 cpu 中的寄存器 和 cache 中已缓存的内存单元中的数据将作废。cpu 将不得不在需要的时候重新读取内存中的数据。这就阻止了 cpu 又将 寄存器, cache 中的数据用于去优化指令，而避免去访问内存。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

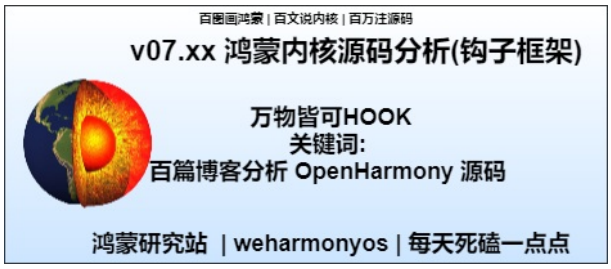
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

07_钩子框架篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

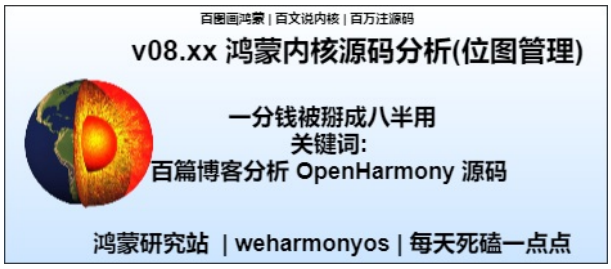
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

08_位图管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

先看四个宏定义，进程和线程（线程就是任务）最高和最低优先级定义，`[0，31]`区间，即`32`级，优先级用于调度，`CPU`根据这个来决定先运行哪个进程和任务。

```
#define OS_PROCESS_PRIORITY_HIGHEST     0 //进程最高优先级
#define OS_PROCESS_PRIORITY_LOWEST      31 //进程最低优先级
#define OS_TASK_PRIORITY_HIGHEST    0 //任务最高优先级，软时钟任务就是最高级任务，见于 OsSwtmrTaskCreate
#define OS_TASK_PRIORITY_LOWEST     31 //任务最低优先级
```

为何进程和线程都是32个优先级？

回答这个问题之前，先回答另一个问题，为什么人类几乎所有的文明都是用十进制的计数方式。答案掰手指就知道了，因为人有十根手指头。玛雅人的二十进制那是把脚指头算上了，但其实也算是十进制的表示。

这是否说明一个问题，认知受环境的影响，方向是怎么简单/方便怎么来。这也可以解释为什么人类语言发音包括各种方言对妈妈这个词都很类似，因为婴儿说mama是最容易的。 注意认识这点很重要！

而计算机的世界是二进制的，是是非非，清清楚楚，特别的简单，二进制已经最简单了，到底啦，不可能有更简单的了。还记得双向链表篇中说过的吗，因为简单所以才不简单啊，大道若简，计算机就靠着这01码，表述万千世界。

但人类的大脑不擅长存储，二进制太长了数到100就撑爆了大脑，记不住，为了记忆和运算方便，编程常用靠近10进制的 16进制来表示，0x9527ABCD 看着比 00110001111001010101001111舒服多了。

应用开发和内核开发有哪些区别？

区别还是很大的，这里只说一点，就是对位的控制能力，内核会出现大量的按位运算(&，|，~，^)， 一个变量的不同位表达不同的含义，但这在应用程序员那是很少看到的，他们用的更多的是逻辑运算（&&，||，！）

```
#define OS_TASK_STATUS_INIT         0x0001U //初始化状态
#define OS_TASK_STATUS_READY        0x0002U //就绪状态的任务都将插入就绪队列
#define OS_TASK_STATUS_RUNNING      0x0004U //运行状态
#define OS_TASK_STATUS_SUSPEND      0x0008U //挂起状态
#define OS_TASK_STATUS_PEND         0x0010U //阻塞状态
```

这是任务各种状态（注者后续将比如成贴标签）表述，将它们还原成二进制就是：

0000000000000001 = 0x0001U

0000000000000010 = 0x0002U

0000000000000100 = 0x0004U

0000000000001000 = 0x0008U

0000000000010000 = 0x0010U

发现二进制这边的区别没有，用每一位来表示一种不同的状态，1表示是，0表示不是。

这样的好处有两点：

1.可以多种标签同时存在 比如 0x07 = 0b00000111，对应以上就是任务有三个标签（初始，就绪，和运行），进程和线程在运行期间是允许多种标签同时存在的。

2.节省了空间，一个变量就搞定了，如果是应用程序员要实现这三个标签同时存在，习惯上要定义三个变量的，因为你的排他性颗粒度是一个变量而不是一个位。

而对位的管理/运算就需要有个专门的管理器：位图管理器 (见源码 los_bitmap.c)

什么是位图管理器？

直接上部分代码，代码关键地方都加了中文注释，简单说就是对位的各种操作，比如如何在某个位上设1？如何找到最高位为1的是哪个位置？这些函数都是有大用途的。

```
//对状态字的某一标志位进行置1操作
VOID LOS_BitmapSet(UINT32 *bitmap， UINT16 pos)
{
    if (bitmap == NULL) {
        return;
    }

    *bitmap |= 1U << (pos & OS_BITMAP_MASK);//在对应位上置1
}
//对状态字的某一标志位进行清0操作
VOID LOS_BitmapClr(UINT32 *bitmap， UINT16 pos)
{
    if (bitmap == NULL) {
        return;
    }

    *bitmap &= ~(1U << (pos & OS_BITMAP_MASK));//在对应位上置0
}
/*****************************************************
杂项算术指令
CLZ 用于计算操作数最高端0的个数，这条指令主要用于一下两个场合
      计算操作数规范化（使其最高位为1）时需要左移的位数
      确定一个优先级掩码中最高优先级
*****************************************************/
//获取状态字中为1的最高位 例如: 00110110 返回 5
UINT16 LOS_HighBitGet(UINT32 bitmap)
{
    if (bitmap == 0) {
        return LOS_INVALID_BIT_INDEX;
    }

    return (OS_BITMAP_MASK - CLZ(bitmap));
}
//获取状态字中为1的最低位， 例如: 00110110 返回 2
UINT16 LOS_LowBitGet(UINT32 bitmap)
{
    if (bitmap == 0) {
        return LOS_INVALID_BIT_INDEX;
    }

    return CTZ(bitmap);//
}
```

位图在哪些地方应用？

内核很多模块在使用位图，这里只说进程和线程模块，还记得开始的问题吗，为何进程和线程都是32个优先级？因为他们的优先级是由位图管理的，管理一个UINT32的变量，所以是32级，一个位一个级别，最高位优先级最低。

```
    UINT32          priBitMap;          /**< BitMap for recording the change of task priority， //任务在执行过程中优先级会经常变化，这个变量用来记录所有曾经
                                            the priority can not be greater than 31 */   //过的优先级，例如 ..01001011 曾经有过 0，1，3，6 优先级
```

这是任务控制块中对调度优先级位图的定义，注意一个任务的优先级在运行过程中可不是一成不变的，内核会根据运行情况而改变它的，这个变量是用来保存这个任务曾经有过的所有优先级历史记录。

比如 任务A的优先级位图是 00000001001011 ，可以看出它曾经有过四个调度等级记录，那如果想知道优先级最低的记录是多少时怎么办呢？

诶，上面的位图管理器函数 UINT16 LOS_HighBitGet(UINT32 bitmap) 就很有用啦 ，它返回的是1在高位出现的位置，可以数一下是 6

因为任务的优先级0最大，所以最终的意思就是A任务曾经有过的最低优先级是6

一定要理解位图的操作，内核中大量存在这类代码，尤其到了汇编层，对寄存器的操作大量的出现。

比如以下这段汇编代码。

```
    MSR     CPSR_c， #(CPSR_INT_DISABLE | CPSR_SVC_MODE)  @禁止中断并切到管理模式
    LDRH    R1，[R0， #4] @将存储器地址为R0+4 的低16位数据读入寄存器R1，并将R1的高16 位清零
    ORR     R1， #OS_TASK_STATUS_RUNNING @或指令 R1=R1|OS_TASK_STATUS_RUNNING
    STRH    R1，[R0， #4] @将寄存器R1中的低16位写入以R0+4为地址的存储器中
```

编程实例

对数据实现位操作，本实例实现如下功能：

- 某一标志位置1。
- 获取标志位为1的最高bit位。
- 某一标志位清0。
- 获取标志位为1的最低bit位。

```
#include "los_bitmap.h"
#include "los_printf.h"

static UINT32 Bit_Sample(VOID)
{
  UINT32 flag = 0x10101010;
  UINT16 pos;

  dprintf("\nBitmap Sample!\n");
  dprintf("The flag is 0x%8x\n"， flag);

  pos = 8;
  LOS_BitmapSet(&flag， pos);
  dprintf("LOS_BitmapSet:\t pos : %d， the flag is 0x%0+8x\n"， pos， flag);

  pos = LOS_HighBitGet(flag);
  dprintf("LOS_HighBitGet:\t The highest one bit is %d， the flag is 0x%0+8x\n"， pos， flag);

  LOS_BitmapClr(&flag， pos);
  dprintf("LOS_BitmapClr:\t pos : %d， the flag is 0x%0+8x\n"， pos， flag);

  pos = LOS_LowBitGet(flag);
  dprintf("LOS_LowBitGet:\t The lowest one bit is %d， the flag is 0x%0+8x\n\n"， pos， flag);
```

```
    return LOS_OK;
}
```

结果验证

```
Bitmap Sample!
The flag is 0x10101010
LOS_BitmapSet: pos : 8， the flag is 0x10101110
LOS_HighBitGet:The highest one bit is 28， the flag is 0x10101110
LOS_BitmapClr: pos : 28， the flag is 0x00101110
LOS_LowBitGet: The lowest one bit is 4， the flag is 0x00101110
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

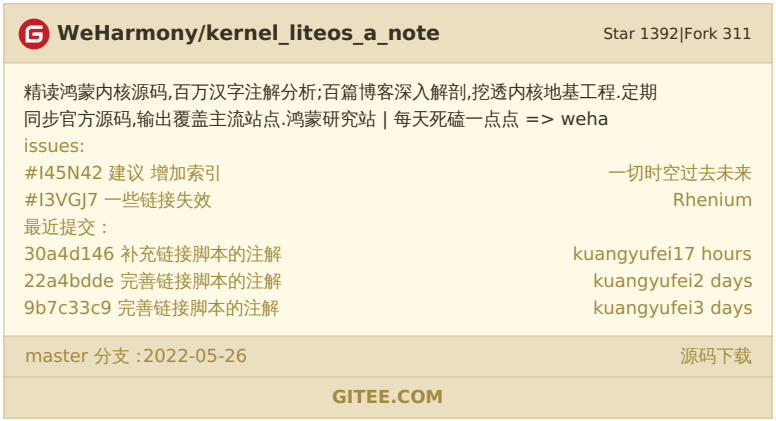
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

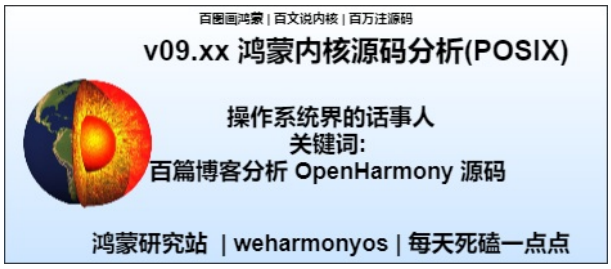


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

09_POSIX篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

几个概念

您肯定听过 `API` ， `Posix` ， `C` 语言库函数，系统调用这些概念，但之间有什么区别和联系，估计一些人没弄明白，本篇重点把它们整明白了。

- `API` ： 搞应用开发的同学不会陌生，应用编程接口的缩写 ，它是对函数的定义，规定了这个函数的功能。任何公司，个人都可以定义自有特色的 `API` 。可以称之为 **私有标准** 。打个比方就相当于是**地方方言**，十里不同音，只有这个地方的人能搞懂，对外会带来极高的沟通成本。
- `Posix` ： `Unix` 开源后很多公司都推出了不同的版本的 Unix 系统。他们的 API 各不相同。这给软件的移植带来了很大的困难。于是 IEEE 制定了基于 Unix 的可移植操作系统接口，目的是为了统一这些 **私有标准** ， 可以称之为 **公共的类Unix接口标准**，所以 posix 标准也是一种 API ， IEEE 这个协会很牛，制定过很多标准，涵盖太空、计算机、电信、生物医学、电力及消费性电子产品等领域，可以说是**科技界标准话事人**。对应**地方方言**，它就相当于是**普通话**，其实呢它也是一种方言。
- C语言库函数： 是基于 `Posix` 标准的具体实现, 相当于**中国人说的普通话**。
- 系统调用： 是内核对外提供的服务总称， 它一般被C语言库函数所调用， 相当于政府对老百姓的办事窗口， 想访问内部资源就需要填表格，走流程。

POSIX简介

当前的POSIX主要分为四个部分：

- **XBD**(Base Definitions)：包含一些通用的术语、概念、接口以及工具函数(cd,mkdir, cp,mv等)和头文件定义(stdio.h, stdlib.h,pthread.h等)。
- **XSH**(System Interfaces)：包含系统服务函数的定义,例如线程、套接字、标准IO、信号处理、错误处理等。
- **XCU**(Shell and Utilities)：包含shell脚本书写的语法、关键字以及工具函数(break,cd,cp,continue,pwd,return)的定义。
- **XRAT**(Rationale)：包含与本标准有关的历史信息以及采用或舍弃某功能的扩展基本原理。 具体查看 >> 官方网站 | opengroup 目前单一UNIX规范第4版中定义了 `1447` 个接口。

  |XBD|XSH|XCU|全部的| |-|-|-|-| |82|1191|174|1447|

鸿蒙支持部分标准 `POSIX` 接口 ,并不是很多 >> 可查看源码 compat/posix

```
compat/posix/src
├── map_error.c //错误码映射 例如: ENOERR EINTR  ESRCH
├── malloc.c //内存分配 例如: malloc zalloc r
├── misc.c //系统信息
```

```
├── mqueue.c //消息队列
├── posix_memalign.c //内存对齐分配
├── pprivate.h //描述 _pthread_data 结构体
├── pthread.c //线程
├── pthread_attr.c //线程属性
├── pthread_cond.c //线程条件
├── pthread_mutex.c //线程互斥
├── sched.c //任务调度
├── semaphore.c //信号量
├── socket.c //网络
└── time.c //时间
```

系统调用

>> 可查看源码 syscall

```
syscall/
├── Makefile
├── fs_syscall.c    //文件模块
├── ipc_syscall.c   //进程通讯模块
├── los_syscall.c   //系统调用主功能函数
├── los_syscall.h
├── misc_syscall.c  //信息配置
├── net_syscall.c   //网络模块
├── process_syscall.c   //进程模块
├── syscall_lookup.h
├── syscall_pub.c   //公有模块
├── syscall_pub.h
├── time_syscall.c  //时间模块
└── vm_syscall.c    //内存模块
```

系统调用的实现函数 `OsArmA32SyscallHandle`,实现过程在 **v95.xx 鸿蒙内核源码分析(系统调用篇)** 中详细说明

进程控制类系统调用，主要有：

- 创建和终止进程的系统调用。
- 获得和设置进程属性的系统调用。
- 等待某事件出现的系统调用。
- `SysFork` ，`SysWait` ，`SysSetProcessGroupID` ，`SysGetUserID` ，`SysSchedYield` ，`SysThreadJoin` ，`SysIoctl`

文件操纵类系统调用，主要有：

- 创建和删除文件。
- 打开和关闭文件的系统调用。
- 读和写文件的系统调用。
- `SysPipe` ，`SysOpen` ，`SysStat` ，`SysRead` ，`SysWrite` ，`SysCreat` ，`SysIoctl`

进程通讯类系统调用，主要有：

- `SysMqOpen` ，`SysSigAction` ，`SysKill` ，`SysMqNotify` ，`SysMqTimedSend`

网络类系统调用，主要有：

- `SysSocket` ，`SysBind` ，`SysConnect` ，`SysListen` ，`SysAccept` ，`SysRecv`

信息配置类系统调用，主要有：

- `SysReboot` ，`SysInfo` ，`SysGetrusage` ，`SysSysconf`

时间类系统调用，主要有：

- `SysTimerCreate` ，`SysTimerDelete` ，`SysClockGettime` ，`SysSetiTimer`

内存类系统调用，主要有：

- `SysMmap` ，`SysMunmap` ，`SysBrk` ，`SysMremap` ，`SysShmGet` ，`SysShmAt` ，`SysShmCtl` ，`SysShmDt`

musl | C标准库函数

musl ，一种C 标准库，官方网站 ，主要使用于以 Linux 内核为主的操作系统上，鸿蒙内核也使用了它，目标为嵌入式系统与移动设备，采用 MIT 许可证发布。作者为瑞奇·费尔克（ Rich Felker ）。开发此库的目的是写一份干净、高效、符合标准的 C 标准库。 Musl 声称与 POSIX 2008 标准和 C11 标准兼容。 musl 是一个非常庞大提供给应用程序使用的工具库，例如: 学习C语言的第一段代码

```
#include <stdio.h>
int main() {
  printf("Hello, World!");
  return 0;
}
```

中的 printf 就是由 stdio.h 提供，由标准库实现的。这是个可变参数实现函数。

```
int printf(const char *restrict fmt, ...)
{
 int ret;
 va_list ap;
 va_start(ap, fmt);
 ret = vfprintf(stdout, fmt, ap);
 va_end(ap);
 return ret;
}
```

如果你一直往下追,你会追到系统调用 write

```
ssize_t write(int fd, const void *buf, size_t count)
{
 return syscall_cp(SYS_write, fd, buf, count);
}
```

```
void SyscallHandleInit(void)
{
#define SYSCALL_HAND_DEF(id, fun, rType, nArg)                                      \
    if ((id) < SYS_CALL_NUM) {                                                      \
        g_syscallHandle[(id)] = (UINTPTR)(fun);                                     \
        g_syscallNArgs[(id) / NARG_PER_BYTE] |= ((id) & 1) ? (nArg) << NARG_BITS : (nArg); \
    }                                                                               \

    #include "syscall_lookup.h"
#undef SYSCALL_HAND_DEF
}
```

\code-v1.1.1-LTS\prebuilts\lite\sysroot\usr\include\arm-liteos\bits\syscall.h

```
#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_creat 8
#define __NR_link 9
```

```
#define SYS_restart_syscall 0
#define SYS_exit 1
#define SYS_fork 2
#define SYS_read 3
#define SYS_write 4
#define SYS_open 5
#define SYS_close 6
```

```
#define SYS_creat 8
#define SYS_link 9

ssize_t read(int fd, void *buf, size_t count)
{
 return syscall_cp(SYS_read, fd, buf, count);
}
#define __syscall_cp(...) __SYSCALL_DISP(__syscall_cp,__VA_ARGS__)
#define syscall_cp(...) __syscall_ret(__syscall_cp(__VA_ARGS__))
```

`\code-v1.1.1-LTS\third_party\musl\src\thread\arm\syscall_cp.s`

```
__syscall_cp_asm:
 mov ip,sp
 stmfd sp!,{r4,r5,r6,r7}
__cp_begin:
 ldr r0,[r0]
 cmp r0,#0
 bne __cp_cancel
 mov r7,r1 //R7寄存器保存软中断号
 mov r0,r2
 mov r1,r3
 ldmfd ip,{r2,r3,r4,r5,r6}
 svc 0 //原 SWI 指令, 软中断指令
__cp_end:
 ldmfd sp!,{r4,r5,r6,r7}
 bx lr
__cp_cancel:
 ldmfd sp!,{r4,r5,r6,r7}
 b __cancel
```

- 其中最重要的命令就是 svc 0，通过这条指令切换到 `svc` 模式（ `svc` 替代了以前的 `swi` 指令，是 `arm` 提供的系统调用指令），进入到软件中断处理函数（ `SWI handler` ）。

问题

那可能有人会问了，操作系统有抽象层，那硬件有没有抽象层，规定处理器内核与外设的接口，统一了内核访问外设寄存器的方法，从而简化了软件的开发，提高重用度，降低软硬件接口开发成本。有的，就是 `CMSIS`（Cortex Microcontroller Software Interface Standard)缩写，中文为Cortex系列微控制器软件接口标准。此标准是ARM公司，芯片供应商以及软件供应商共同制定的，包含以下组件：

- `CMSIS-CORE`：提供与 Cortex-M0、Cortex-M3、Cortex-M4、SC000 和 SC300 处理器与外围寄存器之间的接口
- `CMSIS-DSP`：包含以定点（分数 q7、q15、q31）和单精度浮点（32 位）实现的 60 多种函数的 DSP 库
- `CMSIS-RTOS API`：用于线程控制、资源和时间管理的实时操作系统的标准化编程接口
- `CMSIS-SVD`：包含完整微控制器系统（包括外设）的程序员视图的系统视图描述 XML 文件 此标准可进行全面扩展，以确保适用于所有 Cortex-M 处理器系列微控制器。其中包括所有设备：从最小的 8 KB 设备，直至带有精密通信外设（例如以太网或 USB）的设备。（内核外设功能的内存要求小于 1 KB 代码，低于 10 字节 RAM）。

百文说内核 | 抓住主脉络

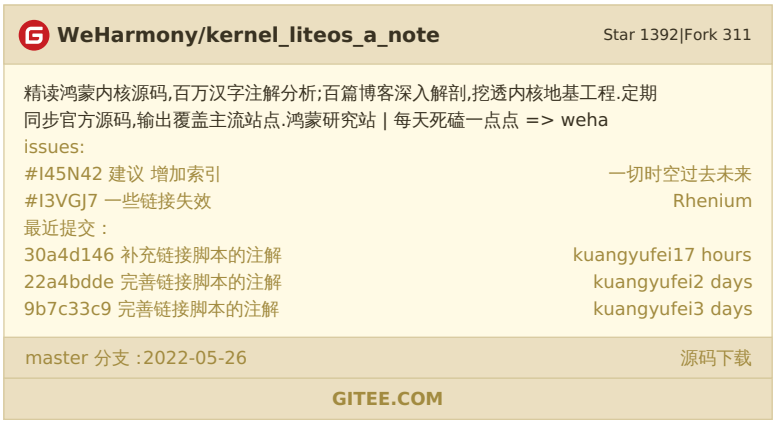
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

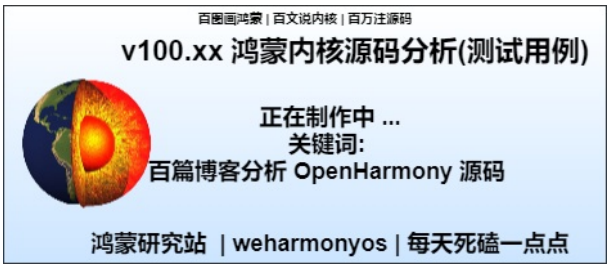


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

100_测试用例篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

调测工具相关篇为:

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

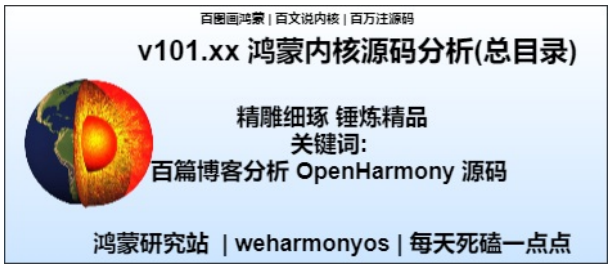
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

101_总目录

本篇关键词：进程管理、任务管理、内存管理、通讯机制、文件管理、硬件架构、内核汇编、编译运行



主流站点覆盖 | 定期同步更新

- 鸿蒙研究站
  - (国内) | http://weharmonyos.com
  - (国外) | https://weharmony.github.io
- oschina | https://my.oschina.net/weharmony
- 博客园 | https://www.cnblogs.com/weharmony/
- 知乎 | https://www.zhihu.com/people/weharmonyos
- csdn | https://blog.csdn.net/kuangyufei
- 51cto | https://harmonyos.51cto.com/column/34
- 掘金 | https://juejin.cn/user/756888642000808

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

基础知识

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴

- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

前因后果

- v101.03 鸿蒙内核源码分析(总目录) | 精雕细琢 锤炼精品
- v102.05 鸿蒙内核源码分析(源码注释) | 每天死磕一点点
- v103.05 鸿蒙内核源码分析(静态站点) | 码农都不爱写注释和文档
- v104.01 鸿蒙内核源码分析(参考手册) | 阅读内核源码必备工具

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。

- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

102_源码注释篇

百图画鸿蒙 | 百文说内核 | 百万注源码

鸿蒙研究站 | weharmonyos.com | 专注 . 聚焦

进 >> 百篇博客 | 进 >> 官方文档 | 进 >> 参考手册 | 进 >> 注解仓库

get-code | 获取源码/获取工具 南向 - 设备开发	百文说内核 | 脉络渐清晰 双向链表 | 时间管理 | 用栈方式	微信 | QQ群 790015635 | 666
docker | Docker镜像构建 轻内核版 | 标准版 镜像	百篇博客 | 进程管理 进程管理 | Fork | 信号消费	离线文档 | 定期更新 百篇博客分析.zip | 注解文档.chm

中文注解鸿蒙内核 | kernel_liteos_a_note 是在 `OpenHarmony` 的 kernel_liteos_a 基础上给内核源码加上中文注解的版本，同步官方代码迭代推进。

为何要精读内核源码？

- 码农的学职生涯，都应精读一遍内核源码。以浇筑好计算机知识大厦的地基，地基纵深的坚固程度，很大程度能决定未来大厦能盖多高。那为何一定要精读细品呢？
- 因为内核代码本身并不太多，都是浓缩的精华，精读是让各个知识点高频出现，不孤立成点状记忆，没有足够连接点的知识点是很容易忘的，点点成线，线面成体，连接越多，记得越牢，如此短时间内容易结成一张高浓度，高密度的系统化知识网，训练大脑肌肉记忆，驻入大脑直觉区，想抹都抹不掉，终生携带，随时调取。跟骑单车一样，一旦学会，即便多年不骑，照样跨上就走，游刃有余。

热爱是所有的理由和答案

- 因大学时阅读 linux 2.6 内核痛并快乐的经历，一直有个心愿，对底层基础技术进行一次系统性的整理，方便自己随时翻看，同时让更多对底层感兴趣的小伙伴减少时间，加速对计算机系统级的理解，而不至于过早的放弃。但因过程种种，多年一直没有行动，基本要放弃这件事了。恰逢 2020/9/10 鸿蒙正式开源，重新激活了多年的心愿，就有那么点如黄河之水一发不可收拾了。

- 包含三部分内容：**注源**，**写博** ，**画图**， 目前对内核源码的注解完成 80% ，博客分析完成80+篇，百图画鸿蒙完成20张，空闲时间几乎被占用，时间不够用，但每天都很充实，连做梦鸿蒙系统都在鱼贯而入。是件很有挑战的事，时间单位以年计，已持续一年半，期间得到众多小伙伴的支持与纠错，在此谢过！:P

(〃·■■_·■■)ゞ鸿蒙内核开发者

- 感谢开放原子开源基金会，致敬鸿蒙内核开发者提供了如此优秀的源码，一了多年的夙愿，津津乐道于此。从内核一行行的代码中能深深感受到开发者各中艰辛与坚持，及鸿蒙生态对未来的价值，这些是张嘴就来的网络喷子们永远不能体会到的。可以毫不夸张的说鸿蒙内核源码可作为大学：C语言，数据结构，操作系统，汇编语言，计算机系统结构，计算机组成原理，微机接口 七门课程的教学项目。如此宝库，不深入研究实在是暴殄天物，于心不忍，坚信鸿蒙大势所趋，未来可期，其必定成功，也必然成功，誓做其坚定的追随者和传播者。

理解内核的三个层级

- 普通概念映射级：这一级不涉及专业知识，用大众所熟知的公共认知就能听明白是个什么概念，也就是说用一个普通人都懂的概念去诠释或者映射一个他们从没听过的概念。让陌生的知识点与大脑中烂熟于心的知识点建立多重链接，加深记忆。说别人能听得懂的话这很重要。一个没学过计算机知识的卖菜大妈就不可能知道内核的基本运作了吗？不一定。在系列篇中试图用故事，打比方，去引导这一层级的认知，希望能卷入更多的人来关注基础软件，人多了场子热起来了创新就来了。
- 专业概念抽象级：对抽象的专业逻辑概念具体化认知， 比如虚拟内存，老百姓是听不懂的，学过计算机的人都懂，具体怎么实现的很多人又都不懂了，但这并不妨碍成为一个优秀的上层应用开发者，因为虚拟内存已经被抽象出来，目的是要屏蔽上层对它具体实现的认知。试图用百篇博客系列篇去拆解那些已经被抽象出来的专业概念， 希望能卷入更多对内核感兴趣的应用软件人才流入基础软硬件生态， 应用软件咱们是无敌宇

宙，但基础软件却很薄弱。

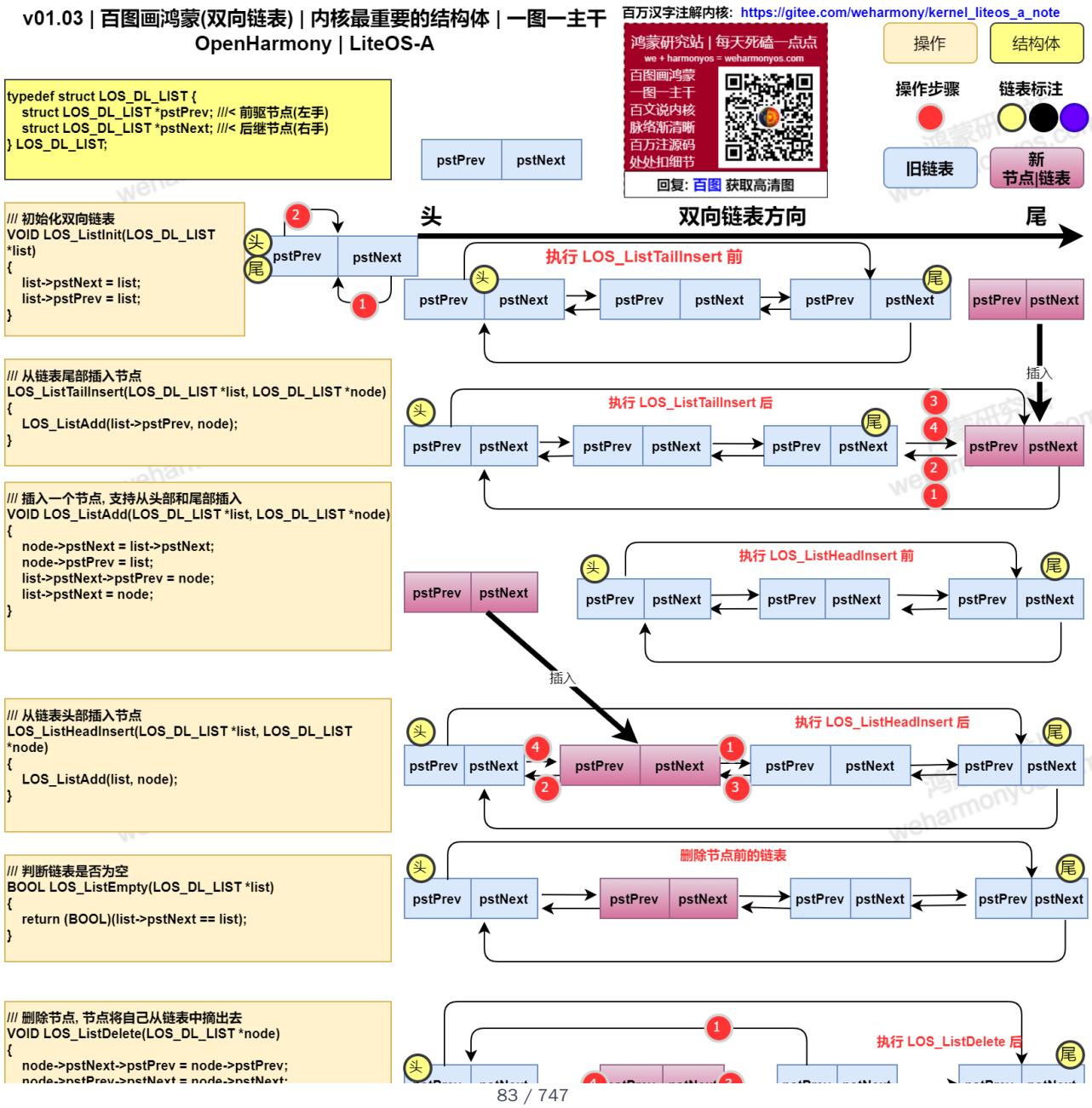
- 具体微观代码级：这一级是具体到每一行代码的实现，到了用代码指令级的地步，这段代码是什么意思？为什么要这么设计？有没有更好的方案？鸿蒙内核源码注解分析 试图从细微处去解释代码实现层，英文真的是天生适合设计成编程语言的人类语言，计算机的01码映射到人类世界的26个字母，诞生了太多的伟大奇迹。但我们的母语注定了很大部分人存在着自然语言层级的理解映射，希望内核注解分析能让更多爱好者节约时间成本，哪怕节约一分钟也是这件事莫大的意义。

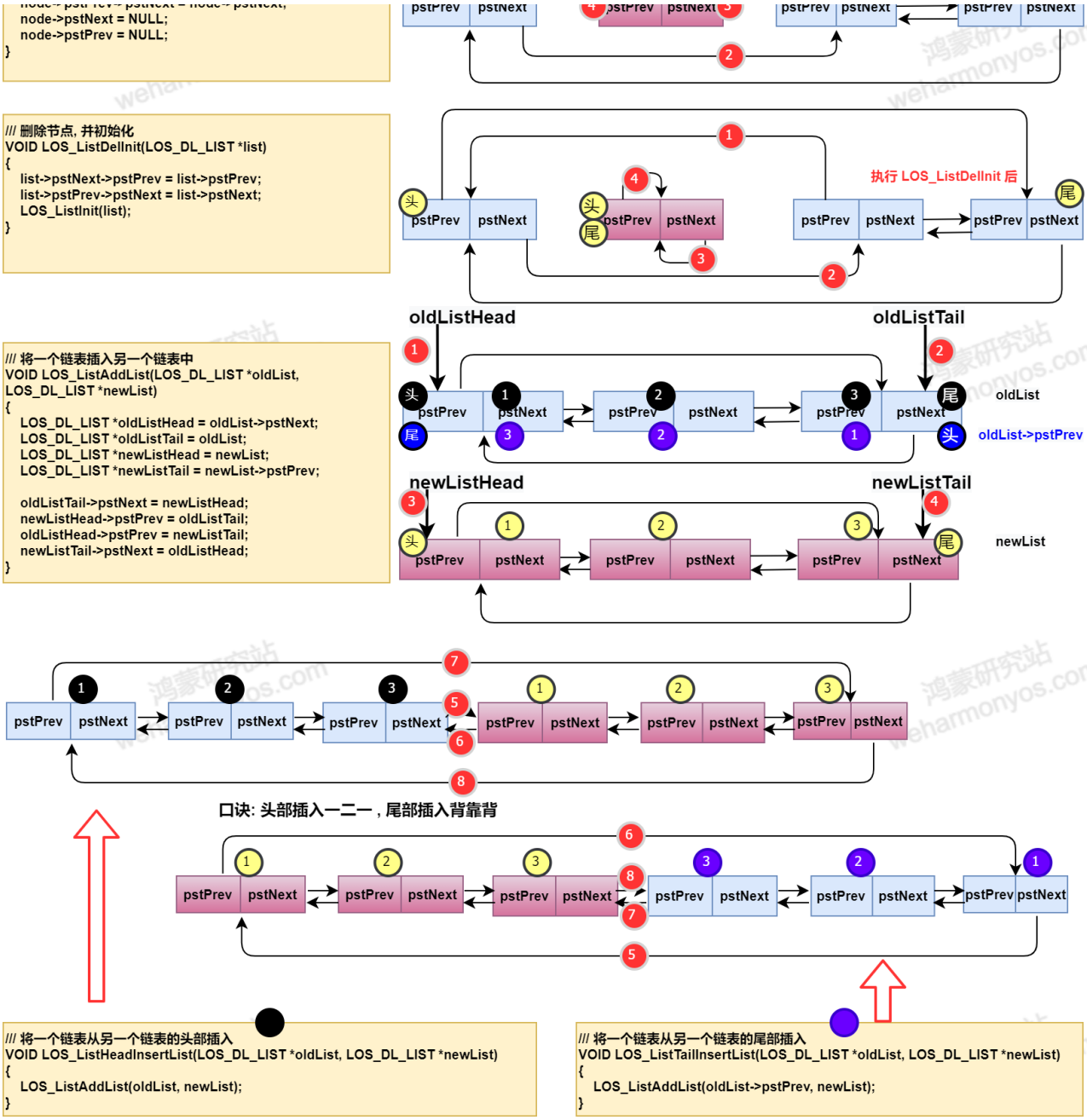
四个维度解剖内核

为了全方位剖析内核，在 **画图**，**写文**，**注源**，**成册** 四个方向做了努力，试图以**讲故事**，**画图表**，**写文档**，**拆源码** 立体的方式表述内核。很喜欢易中天老师的一句话:研究方式不等于表述方式。底层技术并不枯燥，它可以很有意思，它可以是我们生活中的场景。

一： 百图画鸿蒙 | 一图一主干 | 骨骼系统

- 如果把鸿蒙比作人，百图目的是要画出其骨骼系统。
- 百图系列每张图都是心血之作，耗时甚大，能用一张就绝不用两张，所以会画的比较复杂，高清图会很大，可在公众号中回复 **百图** 获取 `3` 倍超高清最新图。 `v**.xx` 代表图的版本，请留意图的更新。
- 例如： **双向链表** 是内核最重要的结构体，站长更愿意将它比喻成人的左右手，其意义是通过寄生在宿主结构体上来体现，可想象成在宿主结构体装上一对对勤劳的双手，它真的很会来事，超级活跃分子，为宿主到处拉朋友，建圈子。其插入 | 删除 | 遍历操作是它最常用的社交三大件，若不理解透彻在分析源码过程中很容易卡壳。虽在网上能找到很多它的图,但怎么看都不是自己想要的，干脆重画了它的主要操作。





二： 百文说内核 | 抓住主脉络 | 肌肉器官

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，公众号回复 **百文** 可方便阅读。

基础知识

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

进程管理

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

任务管理

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

内存管理

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么

- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

通讯机制

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析LiteIpc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析LiteIpc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

文件系统

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

硬件架构

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

内核汇编

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()

- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

编译运行

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

调测工具

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

前因后果

- v101.03 鸿蒙内核源码分析(总目录) | 精雕细琢 锤炼精品
- v102.05 鸿蒙内核源码分析(源码注释) | 每天死磕一点点
- v103.05 鸿蒙内核源码分析(静态站点) | 码农都不爱写注释和文档
- v104.01 鸿蒙内核源码分析(参考手册) | 阅读内核源码必备工具

三： 百万注内核 | 处处扣细节 | 细胞血管

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。

- 因鸿蒙内核6W+代码量，本身只有较少的注释， 中文注解以不对原有代码侵入为前提，源码中所有英文部分都是原有注释，所有中文部分都是中文版的注释，同时为方便同步官方版本的更新，尽量不去增加代码的行数，不破坏文件的结构，注释多类似以下的方式:

    - 在重要模块的 `.c/.h` 文件开始位置先对模块功能做整体的介绍，例如异常接管模块注解如图所示:

```
#endif /* __cplusplus */
#endif /* __cplusplus */
/*****************************************************************************
基本概念
    异常接管是操作系统对运行期间发生的异常情况（芯片硬件异常）进行处理的一系列动作，
    例如打印异常发生时当前函数的调用栈信息、CPU现场信息、任务的堆栈情况等。

    异常接管作为一种调测手段，可以在系统发生异常时给用户提供有用的异常信息，譬如异常类型、
    发生异常时的系统状态等，方便用户定位分析问题。

    异常接管，在系统发生异常时的处理动作为：显示异常发生时正在运行的任务信息
    （包括任务名、任务号、堆栈大小等），以及CPU现场等信息。

运作机制
    每个函数都有自己的栈空间，称为栈帧。调用函数时，会创建子函数的栈帧，
    同时将函数入参、局部变量、寄存器入栈。栈帧从高地址向低地址生长。

    以ARM32 CPU架构为例，每个栈帧中都会保存PC、LR、SP和FP寄存器的历史值。

堆栈分析
    LR寄存器（Link Register），链接寄存器，指向函数的返回地址。
    R11：可以用作通用寄存器，在开启特定编译选项时可以用作帧指针寄存器FP，用来实现栈回溯功能。
    GNU编译器（gcc）默认将R11作为存储变量的通用寄存器，因而默认情况下无法使用FP的栈回溯功能。
        为支持调用栈解析功能，需要在编译参数中添加-fno-omit-frame-pointer选项，提示编译器将R11作为FP使用。
    FP寄存器（Frame Point），帧指针寄存器，指向当前函数的父函数的栈帧起始地址。利用该寄存器可以得到父函数的栈帧，
        从栈帧中获取父函数的FP，就可以得到祖父函数的栈帧，以此类推，可以追溯程序调用栈，得到函数间的调用关系。
    当系统发生异常时，系统打印异常函数的栈帧中保存的寄存器内容，以及父函数、祖父函数的
        栈帧中的LR、FP寄存器内容，用户就可以据此追溯函数间的调用关系，定位异常原因。
    异常接管对系统运行期间发生的芯片硬件异常进行处理，不同芯片的异常类型存在差异，具体异常类型可以查看芯片手册。

异常接管一般的定位步骤如下：
    打开编译后生成的镜像反汇编（asm）文件。
    搜索PC指针（指向当前正在执行的指令）在asm中的位置，找到发生异常的函数。
    根据LR值查找异常函数的父函数。
    重复步骤3，得到函数间的调用关系，找到异常原因。

注意事项
    要查看调用栈信息，必须添加编译选项宏-fno-omit-frame-pointer支持stack frame，否则编译时FP寄存器是关闭的。

参考
    https://gitee.com/LiteOS/LiteOS/blob/master/doc/Huawei_LiteOS_Kernel_Developer_Guide_zh.md
*****************************************************************************/
#define INVALID_CPUID  0xFFFF
#define OS_EXC_VMM_NO_REGION  0x0U
#define OS_EXC_VMM_ALL_REGION 0x1U
```

注解过程中查阅了很多的资料和书籍，在具体代码处都附上了参考链接。

- 绘制字符图帮助理解模块 ，例如 虚拟内存区域分布没有图很难理解。

```
/*!
 * @file    los_vm_zone.h
 * @brief
 * @link
   @verbatim
   虚拟地址空间全景图 从 0x00000000U 至 0xFFFFFFFFU ,外设和主存采用统一编址方式 @note_pic
   鸿蒙源码分析系列篇: http://weharmonyos.com | https://my.oschina.net/weharmony

   +----------------------------+ 0xFFFFFFFFU
   |  IO设备未缓存              |
   |  PERIPH_PMM_SIZE           |
   +----------------------------+ 外围设备未缓存基地址 PERIPH_UNCACHED_BASE
   |  IO设备缓存                |
   |  PERIPH_PMM_SIZE           |
   +----------------------------+ 外围设备缓存基地址 PERIPH_CACHED_BASE
   |  包括 IO设备               |
   |  PERIPH_PMM_SIZE           |
   +----------------------------+ 外围设备基地址 PERIPH_DEVICE_BASE
   |  Vmalloc 空间              |
   |  内核栈 内核堆             |内核动态分配空间
   |  128M                      |
   |  映射区                    |
   +----------------------------+ 内核动态分配开始地址 VMALLOC_START
   |   DDR_MEM_SIZE             |
   |   Uncached段               |
   +----------------------------+ 未缓存虚拟空间基地址 UNCACHED_VMM_BASE = KERNEL_VMM_BASE + KERNEL_VMM_SIZE
   |   内核虚拟空间             |
   |   mmu table 临时页表       |临时页表的作用详见开机阶段汇编代码注释
   |   .bss                     |Block Started by Symbol : 未初始化的全局变量,内核映射页表所在区 g_firstPageTable,这个表在内核启动后更新
   |   .data 可读写数据区       |
   |   .rodata 只读数据区       |
   |   .text 代码区             |
   |  vectors 中断向量表        |
   +----------------------------+ 内核空间开始地址 KERNEL_ASPACE_BASE = KERNEL_VMM_BASE
   |    16M预留区               |
   +----------------------------+ 用户空间栈顶 USER_ASPACE_TOP_MAX = USER_ASPACE_BASE + USER_ASPACE_SIZE
   |                            |
   |    用户空间                |
   |    USER_ASPACE_SIZE        |
   |    用户栈区(stack)         |
   |    映射区(map)             |
   |    堆区 (heap)             |
   |    .bss                    |
   |    .data .text             |
   +----------------------------+ 用户空间开始地址 USER_ASPACE_BASE
   |    16M预留区               |
   +----------------------------+ 0x00000000U
```

- 而函数级注解会详细到重点行，甚至每一行， 例如申请互斥锁的主体函数，不可谓不重要，而官方注释仅有一行，如图所示

```
STATIC UINT32 OsMuxPendOp(LosTaskCB *runTask, LosMux *mutex, UINT32 timeout)
{
    UINT32 ret;
    LOS_DL_LIST *node = NULL;
    LosTaskCB *owner = NULL;

    if ((mutex->muxList.pstPrev == NULL) || (mutex->muxList.pstNext == NULL)) {//列表为空时的处理
        /* This is for mutex macro initialization. */
        mutex->muxCount = 0;//锁计数器清0
        mutex->owner = NULL;//锁没有归属任务
        LOS_ListInit(&mutex->muxList);//初始化锁的任务链表,后续申请这把锁任务都会挂上去
    }

    if (mutex->muxCount == 0) {//无task用锁时,肯定能拿到锁了.在里面返回
        mutex->muxCount++;              //互斥锁计数器加1
        mutex->owner = (VOID *)runTask; //当前任务拿到锁
        LOS_ListTailInsert(&runTask->lockList, &mutex->holdList);//持有锁的任务改变了,节点挂到当前task的锁链表
        if ((runTask->priority > mutex->attr.prioceiling) && (mutex->attr.protocol == LOS_MUX_PRIO_PROTECT)) {//看保护协议的做法是怎样的?
            LOS_BitmapSet(&runTask->priBitMap, runTask->priority);//1.priBitMap是记录任务优先级变化的位图, 这里把任务当前的优先级记录在priBitMap
            OsTaskPriModify(runTask, mutex->attr.prioceiling);//2.把高优先级的mutex->attr.prioceiling设为当前任务的优先级.
        }//注意任务优先级有32个, 是0最高,31最低!!!这里等于提高了任务的优先级,目的是让其在下次调度中继续提高被选中的概率,从而快速的释放锁.
        return LOS_OK;
    }
    //递归锁muxCount>0 如果是递归锁就要处理两种情况 1.runtask持有锁 2.锁被别的任务拿走了
    if (((LosTaskCB *)mutex->owner == runTask) && (mutex->attr.type == LOS_MUX_RECURSIVE)) {//第一种情况 runtask是锁持有方
        mutex->muxCount++;  //递归锁计数器加1,递归锁的目的是防止死锁,鸿蒙默认用的就是递归锁(LOS_MUX_DEFAULT = LOS_MUX_RECURSIVE)
        return LOS_OK;      //成功退出
    }
    //到了这里说明锁在别的任务那里,当前任务只能被阻塞了.
    if (!timeout) {//参数timeout表示等待多久再来拿锁
        return LOS_EINVAL;//timeout = 0表示不等了,没拿到锁就返回不纠结,返回错误.见于LOS_MuxTrylock
    }
    //自己要被阻塞,只能申请调度,让出CPU core 让别的任务上
    if (!OsPreemptableInSched()) {//不能申请调度 (不能调度的原因是因为没有持有调度任务自旋锁)
        return LOS_EDEADLK;//返回错误,自旋锁被别的CPU core 持有
    }

    OsMuxBitmapSet(mutex, runTask, (LosTaskCB *)mutex->owner);//设置锁位图,尽可能的提高锁持有任务的优先级

    owner = (LosTaskCB *)mutex->owner;  //记录持有锁的任务
    runTask->taskMux = (VOID *)mutex;   //记下当前任务在等待这把锁
    node = OsMuxPendFindPos(runTask, mutex);//在等锁链表中找到一个优先级比当前任务更低的任务
    ret = OsTaskWait(node, timeout, TRUE);//task陷入等待状态 TRUE代表需要调度
    if (ret == LOS_ERRNO_TSK_TIMEOUT) {//这行代码虽和OsTaskWait挨在一起,但要过很久才会执行到,因为在OsTaskWait中CPU切换了任务上下文
        runTask->taskMux = NULL;// 所以重新回到这里时可能已经超时了
        ret = LOS_ETIMEDOUT;//返回超时
    }

    if (timeout != LOS_WAIT_FOREVER) {//不是永远等待的情况
        OsMuxBitmapRestore(mutex, runTask, owner);//恢复锁的位图
    }

    return ret;
```

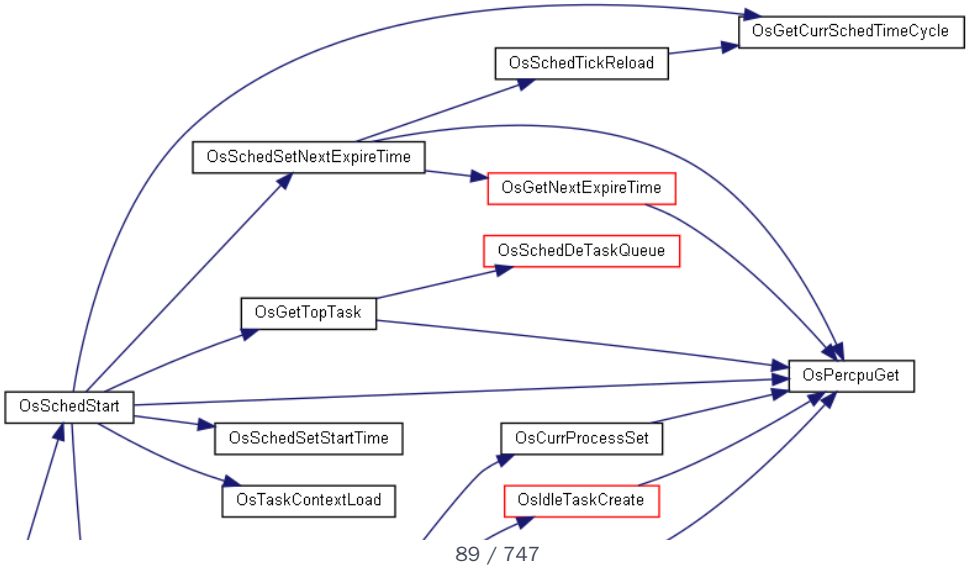
- 注解创建了一些特殊记号，可直接搜索查看

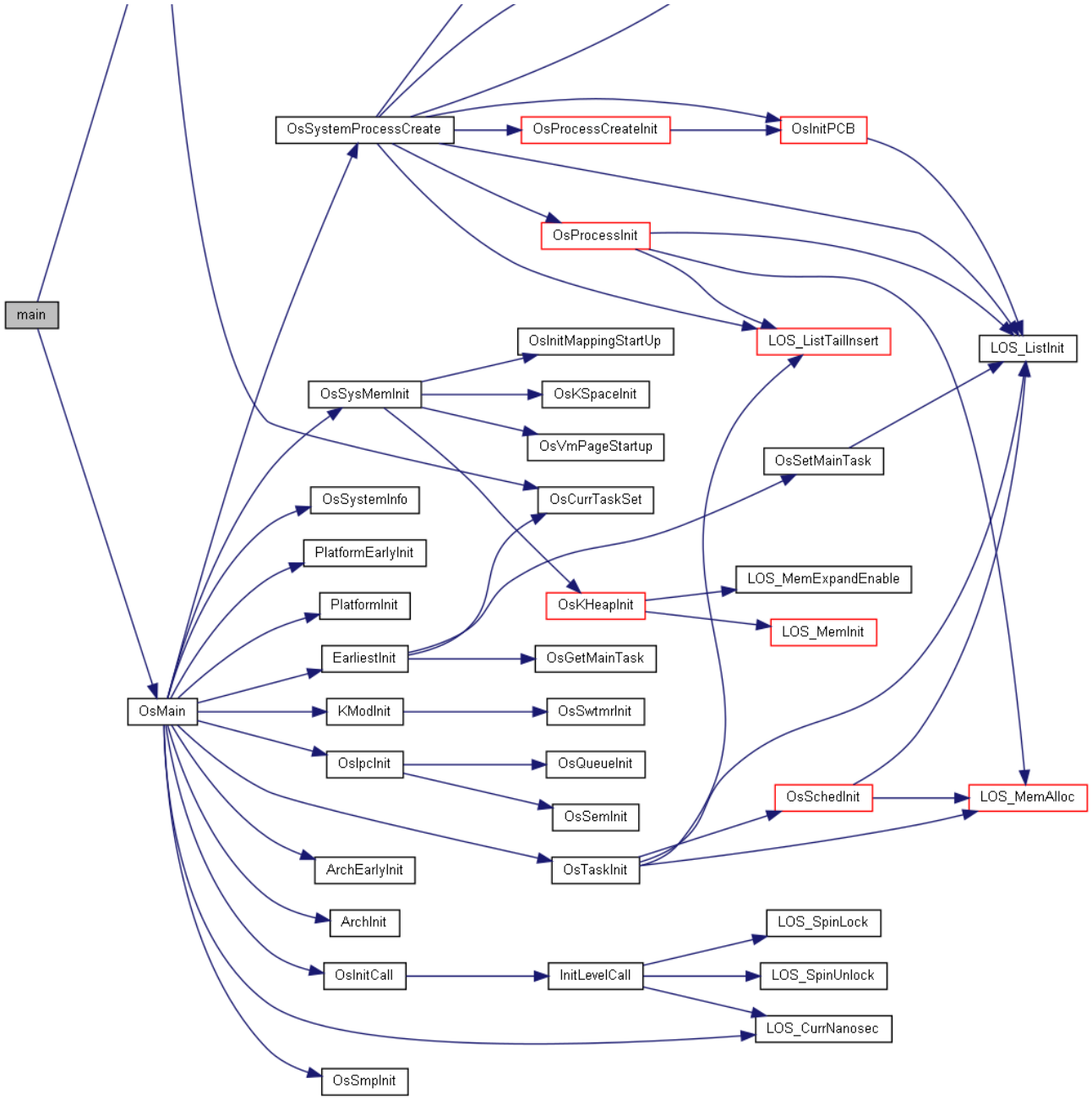
  - ☑ 搜索 `@note_pic` 可查看绘制的全部字符图
  - ☑ 搜索 `@note_why` 是尚未看明白的地方，有看明白的，请新建 Pull Request完善
  - ☑ 搜索 `@note_thinking` 是一些的思考和建议
  - ☑ 搜索 `@note_#if0` 是由第三方项目提供不在内核源码中定义的极为重要结构体，为方便理解而添加的。
  - ☑ 搜索 `@note_link` 是网址链接，方便理解模块信息，来源于官方文档，百篇博客，外部链接
  - ☑ 搜索 `@note_good` 是给源码点赞的地方

四： 参考手册 | Doxygen呈现 | 诊断

在中文加注版基础上构建了参考手册，如此可以看到毛细血管级的网络图，注解支持 doxygen 格式标准。

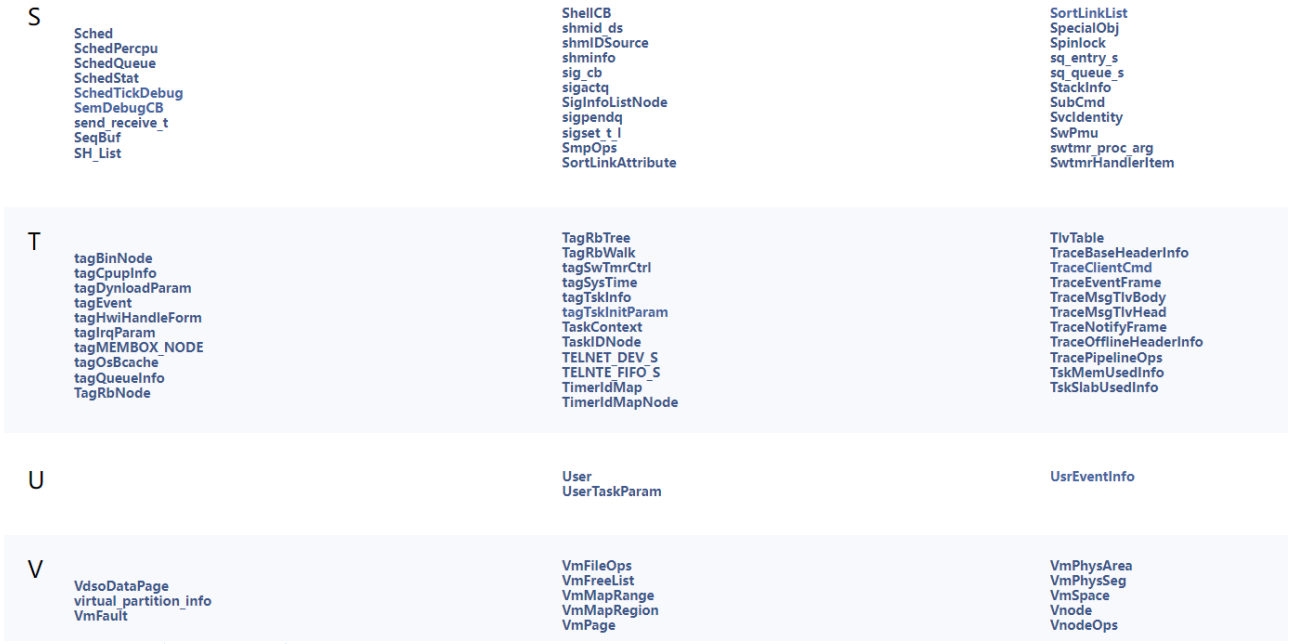
- 图为内核 main 的调用关系直观展现，如果没有这张图，光 main 一个函数就够喝一壶。 main 本身是由汇编指令 bl main 调用





可前往 >> 鸿蒙研究站 | 参考手册 体验

- 图为内核所有结构体索引，点击可查看每个结构变量细节



可前往 >> 鸿蒙研究站 | 结构体索引 体验

四大码仓发布 | 源码同步官方

内核注解同时在 gitee | github | coding | gitcode 发布，并与官方源码按月保持同步，同步历史如下:

- 2022/05/09 -- 标准库(musl，newlib) 目录调整
- 2022/04/16 -- 任务调度模块有很大更新
- 2022/03/23 -- 新增各CPU核自主管理中断, 定时器模块较大调整
- 2022/02/18 -- 官方无代码更新, 只有测试用例的完善
- 2022/01/20 -- 同步官方代码,本次官方对测试用例和MMU做了较大调整
- 2021/12/20 -- 增加 LMS 模块，完善 PM，Fat Cache
- 2021/11/12 -- 加入 epoll 支持，对 shell 模块有较大调整，微调 process ， task ，更正单词拼写错误
- 2021/10/21 -- 增加性能优化模块 perf ，优化了文件映射模块
- 2021/09/14 -- common ， extended 等几个目录结构和Makefile调整
- 2021/08/19 -- 各目录增加了 BUILD。gn 文件，文件系统部分文件调整
- 2021/07/15 -- 改动不大，新增 blackbox ， hidumper ，对一些宏规范化使用
- 2021/06/27 -- 对文件系统/设备驱动改动较大，目录结构进行了重新整理
- 2021/06/08 -- 对编译构建，任务，信号模块有较大的改动
- 2021/05/28 -- 改动不大，主要针对一些错误单词拼写纠正
- 2021/05/13 -- 对系统调用，任务切换，信号处理，异常接管，文件管理， shell 做了较大更新，代码结构更清晰
- 2021/04/21 -- 官方优化了很多之前吐槽的地方，点赞
- 2020/09/16 -- 中文注解版起点

注解子系统仓库

在给鸿蒙内核源码加注过程中发现仅仅注解内核仓库还不够，因为它关联了其他子系统，若对这些子系统不了解是很难完整的注解鸿蒙内核，所以也对这些关联仓库进行了部分注解，这些仓库包括:

- 编译构建子系统 | build_lite
- 协议栈 | lwip
- 文件系统 | NuttX
- 标准库 | musl

关于 zzz 目录

中文加注版比官方版无新增文件，只多了一个 zzz 的目录，里面放了一些加注所需文件，它与内核代码无关，可以忽略它，取名 zzz 是为了排在最后，减少对原有代码目录级的侵入， zzz 的想法源于微信中名称为 AAA 的那帮朋友，你的微信里应该也有他们熟悉的身影吧 :-)

```
/kernel/liteos_a
├── apps                # 用户态的init和shell应用程序
├── arch                # 体系架构的目录，如arm等
│   └── arm             # arm架构代码
├── bsd                 # freebsd相关的驱动和适配层模块代码引入，例如USB等
├── compat              # 内核接口兼容性目录
│   └── posix           # posix相关接口
├── drivers             # 内核驱动
│   └── char            # 字符设备
│       ├── mem         # 访问物理IO设备驱动
│       ├── quickstart  # 系统快速启动接口目录
│       ├── random      # 随机数设备驱动
│       └── video       # framebuffer驱动框架
├── fs                  # 文件系统模块，主要来源于NuttX开源项目
│   ├── fat             # fat文件系统
│   ├── jffs2           # jffs2文件系统
│   ├── include         # 对外暴露头文件存放目录
│   ├── nfs             # nfs文件系统
│   ├── proc            # proc文件系统
│   ├── ramfs           # ramfs文件系统
│   └── vfs             # vfs层
├── kernel              # 进程、内存、IPC等模块
│   ├── base            # 基础内核，包括调度、内存等模块
│   ├── common          # 内核通用组件
│   ├── extended        # 扩展内核，包括动态加载、vdso、liteipc等模块
│   ├── include         # 对外暴露头文件存放目录
│   └── user            # 加载init进程
├── lib                 # 内核的lib库
├── net                 # 网络模块，主要来源于lwip开源项目
├── platform            # 支持不同的芯片平台代码，如Hi3516DV300等
│   ├── hw              # 时钟与中断相关逻辑代码
│   ├── include         # 对外暴露头文件存放目录
│   └── uart            # 串口相关逻辑代码
├── security            # 安全特性相关的代码，包括进程权限管理和虚拟id映射管理
├── syscall             # 系统调用
├── testsuites          # 单元测试用例
├── tools               # 构建工具及相关配置和代码
└── zzz                 # 中文注解版新增目录
```

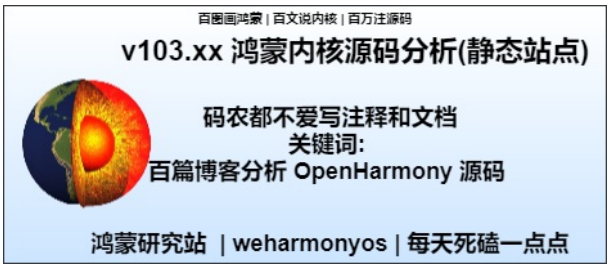
官方文档 | 静态站点呈现

- 研究鸿蒙需不断的翻阅资料，吸取别人的精华，其中官方文档必不可少， 为更好的呈现 **OpenHarmony开发者文档** ， 特意做了静态站点 >> 鸿蒙研究站 | 官方文档 来方便搜索，阅读官方资料。
- 左侧导航栏，右边索引区

- 鸿蒙研究站 定位于做一个专注而靠谱的技术站，没有广告，干净简洁，极佳阅读体验，持续输出，周周更新。同时感谢资助网站建设的各位小伙伴。 >> 我要捐助

103_静态站点篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

前因后果相关篇为:

- v101.03 鸿蒙内核源码分析(总目录) | 精雕细琢 锤炼精品
- v102.05 鸿蒙内核源码分析(源码注释) | 每天死磕一点点
- v103.05 鸿蒙内核源码分析(静态站点) | 码农都不爱写注释和文档
- v104.01 鸿蒙内核源码分析(参考手册) | 阅读内核源码必备工具

一键部署

直接下载或clone本项目到web服务器指定目录即可。

```
git clone https://gitee.com/weharmony/weharmony.git
```

码农都不爱写注释和文档

程序员都不愿意代码写注释和文档，自己也是一路这么过来的，面对海量代码却没有文档，痛苦不堪，追问文档往往换来一句 "代码就是最好的文档"，读源百遍，其意自现。这是一句正确的废话，背后得烧掉了多少时间。等一番寒彻苦，搞懂了之后同样也是不愿写文档，没时间没必要，用同样的话去回应新人对文档要求，何其相似。

点赞鸿蒙的文档

2020/9/10 鸿蒙正式开源后自己用业余时间对鸿蒙内核源码项目kernel_liteos_a_note进行中文注解，高频的接触官方文档，看网上有不少喷官方文档的，往往以偏概全，发现个错误就放大，须知鸿蒙生态何其庞大，有失误和错漏很正常，坦白说鸿蒙文档做的挺不错的，点赞，内容更新也很频繁，因 weharmonyos.com 每周都会同步官方文档，所以很清楚每次都有大的调整，希望多用欣赏的眼光来看鸿蒙，换位思考 u can u up ， 你来未必能做的比现在好。 不要手里拿个锤子看什么都像钉子，干就完了。

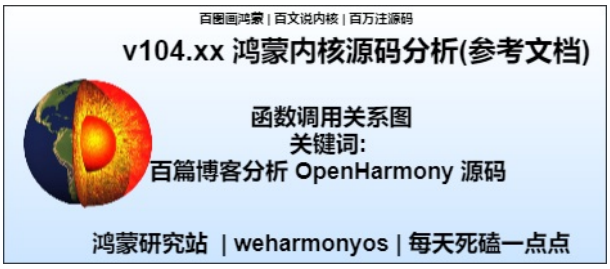
为什么会有 weharmonyos.com

- 在给鸿蒙内核加注和写博客期间需要不断的查找资料，觉得官方目前资料展示方式并不能满足自己的需求，浪费了很多宝贵的时间，所以在想能不能将官方文档(md格式)做个静态站点出来，即方便别人更方便自己，这是一劳永逸，利己利他的事干嘛不做的，刚好五一有成块的时间，本来也想出去走走，结果哪都没去，期间遇到不少问题，但基本都解决了，
- 目前保持每周一次的频率同步官方文档，也想过做成实时更新，但难度很大，因官方文档还不能直接生成静态页面，编译通不过，需要手动去核对和修改。另外导航栏和侧边栏在官方文档的基础上做了索引优化和结构调整，但内容没有做任何的更改，放心使用。
- 鸿蒙研究站的另外一部分是对百篇博客的实时输出，百篇博客内容在其他主流技术站点也有输出，但因各平台的规则不同，更新会较慢，weharmonyos.com上的内容是最新的。其它平台输出如下，感谢这些平台一直以来的支持:
  - oschina | https://my.oschina.net/weharmony
  - 博客园 | https://www.cnblogs.com/weharmony/
  - 知乎 | https://www。zhihu.com/people/weharmonyos
  - csdn | https://blog.csdn。net/kuangyufei
  - 51cto | https://harmonyos。51cto.com/column/34
  - 掘金 | https://juejin.cn/user/756888642000808
- 这里必须要感谢下这套主题的作者 Mr.hope ，人非常的nice， 一直帮着解决问题。再次感谢!!! 主题地址:vuepress-theme-hope 有兴趣的可以去了解下，一个功能强大的 vuepress 主题。

还要啥自行车啊， 赶紧去体验weharmonyos.com吧!

104_参考文档篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

前因后果相关篇为:

- v101.03 鸿蒙内核源码分析(总目录) | 精雕细琢 锤炼精品
- v102.05 鸿蒙内核源码分析(源码注释) | 每天死磕一点点
- v103.05 鸿蒙内核源码分析(静态站点) | 码农都不爱写注释和文档
- v104.01 鸿蒙内核源码分析(参考手册) | 阅读内核源码必备工具

工欲善其事 必先利其器

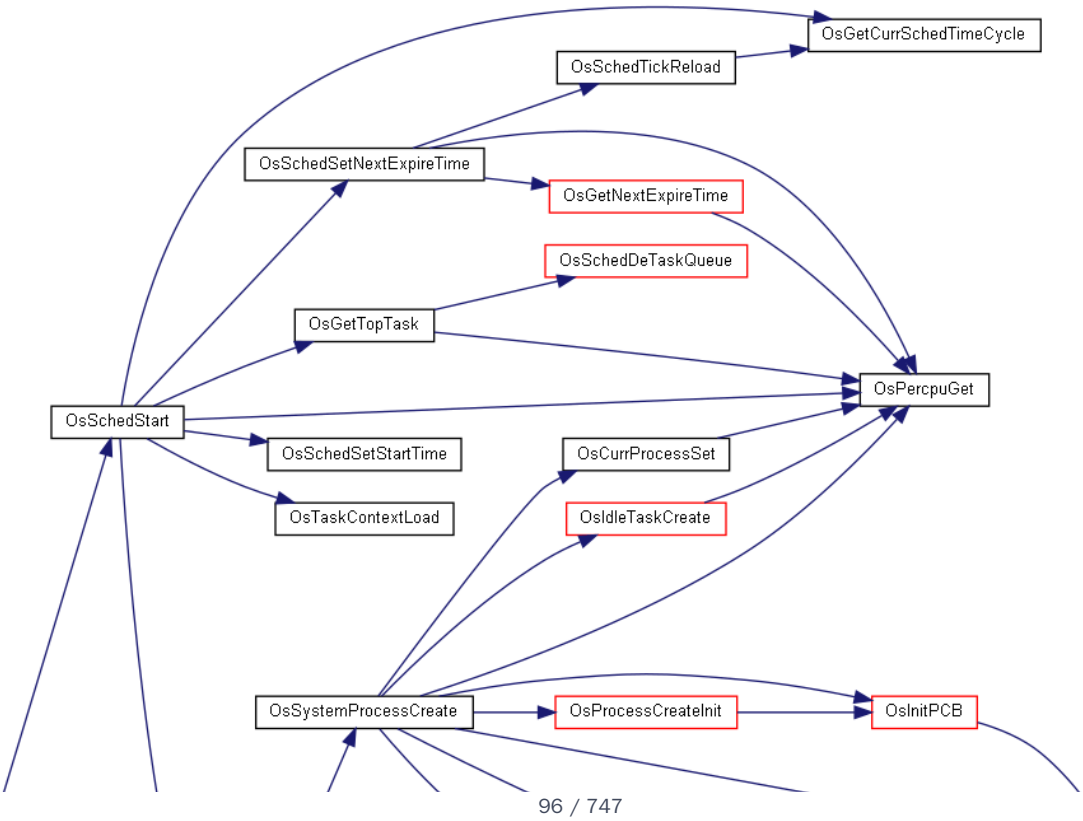
本篇尝试去摸索下鸿蒙内核毛细血管级的脉络，跟踪以下几个问题。

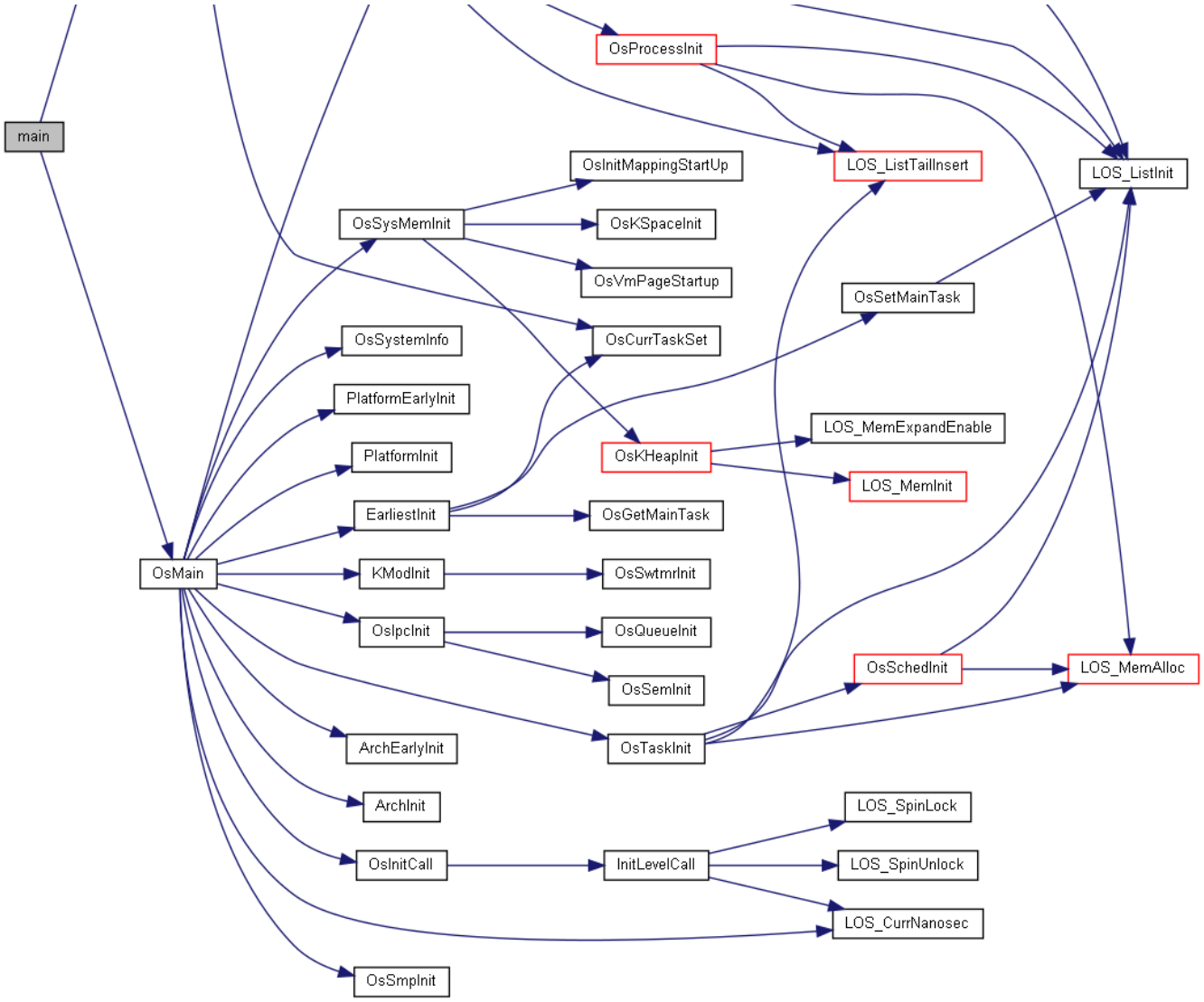
- 鸿蒙有多少个结构体，结构体中每个成员变量的含义是什么？
- 鸿蒙main长啥样，其是如何初始化各个模块的？
- 鸿蒙的任意一个函数的调用和引用关系关系是怎样的？

它已成为众多鸿蒙内核阅读者必不可少的参考手册。

鸿蒙 main 函数长啥样

前往 >> 鸿蒙研究站 | 参考手册版块 点击函数跟踪。






```
/**
 * @brief
 * 内核入口函数，由汇编调用，见于reset_vector_up.S 和 reset_vector_mp.S
 * up指单核CPU， mp指多核CPU bl        main
 * @return LITE_OS_SEC_TEXT_INIT
 */
LITE_OS_SEC_TEXT_INIT INT32 main(VOID)//由主CPU执行，默认0号CPU 为主CPU
{
    UINT32 uwRet;

    uwRet = OsMain();// 内核各模块初始化
    if (uwRet != LOS_OK) {
        return LOS_NOK;
    }

    CPU_MAP_SET(0， OsHwIDGet());//设置CPU映射，参数0 代表0号CPU

    OsSchedStart();//调度开始

    while (1) {
        __asm volatile("wfi");//WFI: wait for Interrupt 等待中断，即下一次中断发生前都在此hold住不干活
    }
}
```

结构体/宏/枚举类型

前往 >> 鸿蒙研究站 | 查看所有结构体索引

S

Sched
SchedPercpu
SchedQueue
SchedStat
SchedTickDebug
SemDebugCB
send_receive_t
SeqBuf
SH_List
ShellCB
shmid_ds
shmIDSource
shminfo
sig_cb
sigactq
SigInfoListNode
sigpendq
sigset_t_l
SmpOps
SortLinkAttribute
SortLinkList
SpecialObj
Spinlock
sq_entry_s
sq_queue_s
StackInfo
SubCmd
SvcIdentity
SwPmu
swtmr_proc_arg
SwtmrHandlerItem

T

tagBinNode
tagCpupInfo
tagDynloadParam
tagEvent
tagHwiHandleForm
tagIrqParam
tagMEMBOX_NODE
tagOsBcache
tagQueueInfo
TagRbNode
TagRbTree
TagRbWalk
tagSwTmrCtrl
tagSysTime
tagTskInfo
tagTskInitParam
TaskContext
TaskIDNode
TELNET_DEV_S
TELNTE_FIFO_S
TimerIdMap
TimerIdMapNode
TlvTable
TraceBaseHeaderInfo
TraceClientCmd
TraceEventFrame
TraceMsgTlvBody
TraceMsgTlvHead
TraceNotifyFrame
TraceOfflineHeaderInfo
TracePipelineOps
TskMemUsedInfo
TskSlabUsedInfo

U

User
UserTaskParam
UsrEventInfo

V

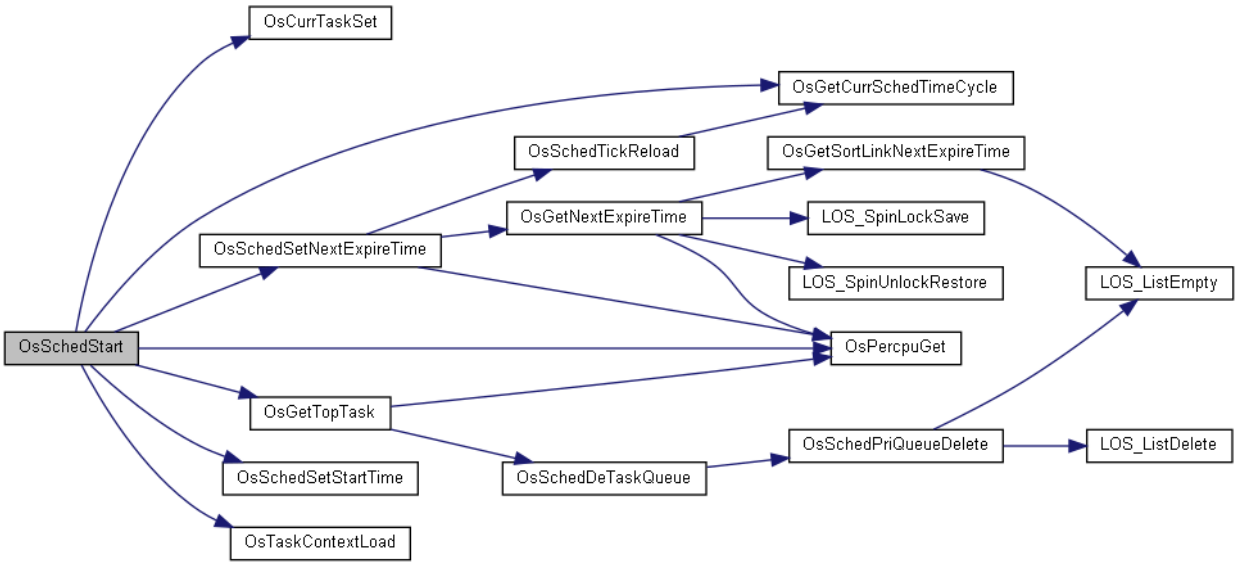
VdsoDataPage
virtual_partition_info
VmFault
VmFileOps
VmFreeList
VmMapRange
VmMapRegion
VmPage
VmPhysArea
VmPhysSeg
VmSpace
Vnode
VnodeOps

```
/*! 进程控制块*/
typedef struct ProcessCB {
    CHAR                processName[OS_PCB_NAME_LEN]; /**< Process name | 进程名称 */
    UINT32              processID;                    /**< Process ID = leader thread ID | 进程ID,由进程池分配,范围[0,64] */
    UINT16              processStatus;                /**< [15:4] Process Status; [3:0] The number of threads currently
                                                           running in the process | 这里设计很巧妙.用一个变量表示了两层逻辑 数量和状态,点赞! @note_good 从这里也可以
    UINT16              priority;                     /**< Process priority | 进程优先级*/
    UINT16              consoleID;                    /**< The console id of task belongs | 任务的控制台id归属 */
    UINT16              processMode;                  /**< Kernel Mode:0; User Mode:1; | 模式指定为内核还是用户进程 */
    UINT16              readyTaskNum;                 /**< The number of ready tasks in the current process */
    UINT32              parentProcessID;              /**< Parent process ID | 父进程ID*/
    UINT32              exitCode;                     /**< Process exit status | 进程退出状态码*/
    LOS_DL_LIST         pendList;                     /**< Block list to which the process belongs | 进程所在的阻塞列表,进程因阻塞挂入相应的链表.*/
    LOS_DL_LIST         childrenList;                 /**< Children process list | 孩子进程都挂到这里,形成双循环链表*/
    LOS_DL_LIST         exitChildList;                /**< Exit children process list | 要退出的孩子进程链表，白发人要送黑发人.*/
    LOS_DL_LIST         siblingList;                  /**< Linkage in parent's children list | 兄弟进程链表，56个民族是一家,来自同一个父进程.*/
    ProcessGroup        *group;                       /**< Process group to which a process belongs | 所属进程组*/
    LOS_DL_LIST         subordinateGroupList;         /**< Linkage in group list | 进程组员链表*/
    UINT32              threadGroupID;                /**< Which thread group , is the main thread ID of the process */
    LOS_DL_LIST         threadSiblingList;            /**< List of threads under this process | 进程的线程(任务)列表 */
    volatile UINT32     threadNumber; /**< Number of threads alive under this process | 此进程下的活动线程数*/
    UINT32              threadCount;  /**< Total number of threads created under this process | 在此进程下创建的线程总数*/ //
    LOS_DL_LIST         waitList;     /**< The process holds the waitLits to support wait/waitpid | 父进程通过进程等待的方式，回收子进程资源，获取子进程退出信息*/
#ifdef LOSCFG_KERNEL_SMP
    UINT32              timerCpu;     /**< CPU core number of this task is delayed or pended | 统计各线程被延期或阻塞的时间*/
#endif
    UINTPTR             sigHandler;   /**< Signal handler | 信号处理函数,处理如 SIGSYS 等信号*/
    sigset_t            sigShare;     /**< Signal share bit | 信号共享位 sigset_t是个64位的变量,对应64种信号*/
#ifdef LOSCFG_KERNEL_LITEIPC
    ProcIpcInfo         *ipcInfo;      /**< Memory pool for lite ipc | 用于进程间通讯的虚拟设备文件系统,设备装载点为 /dev/lite_ipc*/
#endif
#ifdef LOSCFG_KERNEL_VM
    LosVmSpace          *vmSpace;      /**< VMM space for processes | 虚拟空间,描述进程虚拟内存的数据结构，linux称为内存描述符 */
#endif
#ifdef LOSCFG_FS_VFS
    struct files_struct *files;        /**< Files held by the process | 进程所持有的所有文件，注者称之为进程的文件管理器*/
#endif  //每个进程都有属于自己的文件管理器,记录对文件的操作. 注意:一个文件可以被多个进程操作
    timer_t             timerID;       /**< iTimer */

#ifdef LOSCFG_SECURITY_CAPABILITY   //安全能力
    User                *user;      //进程的拥有者
    UINT32              capability; //安全能力范围 对应 CAP_SETGID
#endif
#ifdef LOSCFG_SECURITY_VID  //虚拟ID映射功能
    TimerIdMap          timerIdMap;
#endif
#ifdef LOSCFG_DRIVERS_TZDRIVER
    struct Vnode        *execVnode;    /**< Exec bin of the process | 进程的可执行文件 */
```

宏定义

#define	OS_PCB_NAME_LEN OS_TCB_NAME_LEN
#define	CLONE_VM 0x00000100 子进程与父进程运行于相同的内存空间 更多...
#define	CLONE_FS 0x00000200 子进程与父进程共享相同的文件系统，包括root、当前目录、umask 更多...
#define	CLONE_FILES 0x00000400 子进程与父进程共享相同的文件描述符（file descriptor）表 更多...
#define	CLONE_SIGHAND 0x00000800 子进程与父进程共享相同的信号处理（signal handler）表 更多...
#define	CLONE_PTRACE 0x00002000 若父进程被trace，子进程也被trace 更多...
#define	CLONE_VFORK 0x00004000 父进程被挂起，直至子进程释放虚拟内存资源 更多...
#define	CLONE_PARENT 0x00008000 创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了"兄弟"而不是"父子" 更多...
#define	CLONE_THREAD 0x00010000 Linux 2.4中增加以支持POSIX线程标准，子进程与父进程共享相同的线程群 更多...
#define	OS_PCB_FROM_PID(processID) (((LosProcessCB *)g_processCBArray) + (processID)) 通过数组找到LosProcessCB 更多...
#define	OS_PCB_FROM_SIBLIST(ptr) LOS_DL_LIST_ENTRY((ptr), LosProcessCB, siblingList) 通过siblingList节点找到 LosProcessCB 更多...
#define	OS_PCB_FROM_PENDLIST(ptr) LOS_DL_LIST_ENTRY((ptr), LosProcessCB, pendList) 通过pendlist节点找到 LosProcessCB 更多...

< 任意函数关系图 | 代码实现 | 注解说明 > 三位一体



◆ OsSchedStart()

VOID OsSchedStart (VOID)

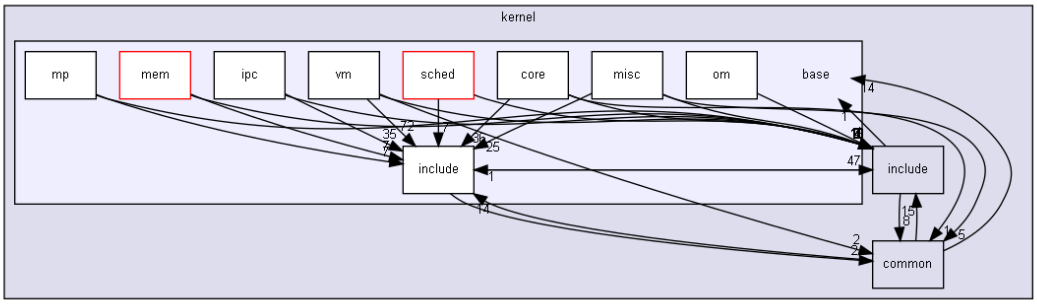
在文件 los_sched.c 第 889 行定义.

```
890 {
891     UINT32 cpuid = ArchCurrCpuid();//从系统寄存器上获取当前执行的CPU核编号
892     UINT32 intSave;
893
894     SCHEDULER_LOCK(intSave);
895
896     if (cpuid == 0) {
897     OsTickStart();//开始了属于本核的tick
898     }
899
900     LosTaskCB *newTask = OsGetTopTask();//拿一个优先级最高的任务
901     LosProcessCB *newProcess = OS_PCB_FROM_PID(newTask->processID);//获取该任务的进程实体
902
903     newTask->taskStatus |= OS_TASK_STATUS_RUNNING;//变成运行状态,注意此时该任务还没真正的运行
904     newProcess->processStatus |= OS_PROCESS_STATUS_RUNNING;
905     newProcess->processStatus = OS_PROCESS_RUNTASK_COUNT_ADD(newProcess->processStatus);//当前任务的数量也增加一个
906
907     OsSchedSetStartTime(HalClockGetCycles());//设置调度开始时间
908     newTask->startTime = OsGetCurrSchedTimeCycle();
909
910 #ifdef LOSCFG_KERNEL_SMP //注意：需要设置当前cpu，以防第一个任务删除可能会失败，因为此标志与实际当前 cpu 不匹配。
911     /*
912      * attention: current cpu needs to be set, in case first task deletion
913      * may fail because this flag mismatch with the real current cpu.
914      */
915     newTask->currCpu = cpuid;//设置当前CPU,确保第一个任务由本CPU核执行
916 #endif
917
918     OsCurrTaskSet((VOID *)newTask);
919
920     /* System start schedule */
921     OS_SCHEDULER_SET(cpuid);
922
923     OsPercpuGet()->responseID = OS_INVALID;
924     OsSchedSetNextExpireTime(newTask->startTime, newTask->taskID, newTask->startTime + newTask->timeSlice, OS_INVALID);
925
926     PRINTK("cpu %d entering scheduler\n", cpuid);
927     OsTaskContextLoad(newTask);
928 }
```

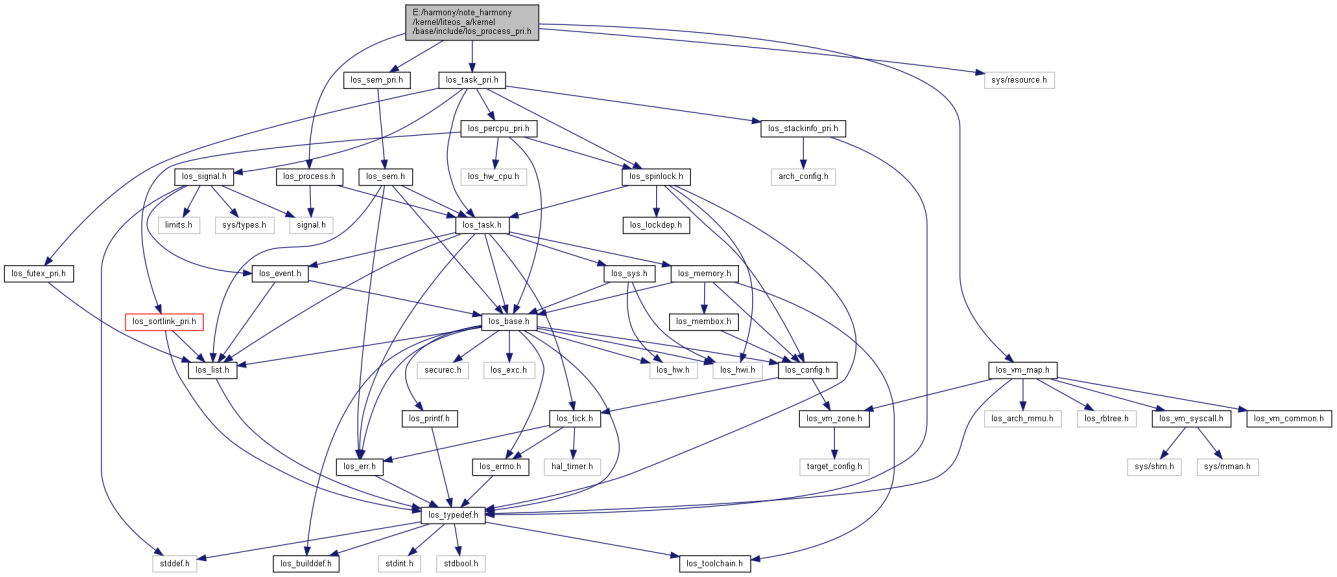
模块之间关系图

base 目录参考

base 的目录依赖关系图



任意头文件的关系图



百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

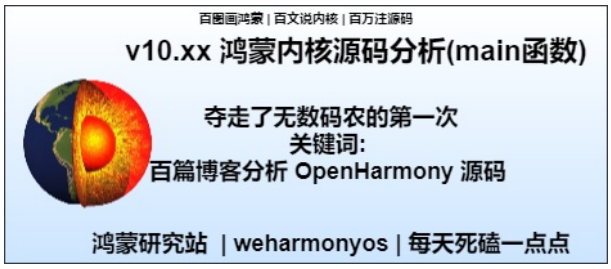
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

10_main函数篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

基础知识相关篇为:

- v01.12 鸿蒙内核源码分析(双向链表) | 谁是内核最重要结构体
- v02.01 鸿蒙内核源码分析(内核概念) | 名不正则言不顺
- v03.02 鸿蒙内核源码分析(源码结构) | 宏观尺度看内核结构
- v04.01 鸿蒙内核源码分析(地址空间) | 内核如何看待空间
- v05.03 鸿蒙内核源码分析(计时单位) | 内核如何看待时间
- v06.01 鸿蒙内核源码分析(优雅的宏) | 编译器也喜欢复制粘贴
- v07.01 鸿蒙内核源码分析(钩子框架) | 万物皆可HOOK
- v08.04 鸿蒙内核源码分析(位图管理) | 一分钱被掰成八半使用
- v09.01 鸿蒙内核源码分析(POSIX) | 操作系统界的话事人
- v10.01 鸿蒙内核源码分析(main函数) | 要走了无数码农的第一次

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

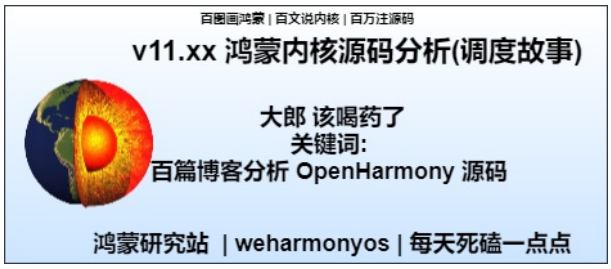
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

11_调度故事篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

本篇用一个故事说清楚鸿蒙进程和线程的调度过程。

有个场馆

某地有一个演出场馆，分成室内馆(400平米)和室外馆(4万平米)，管理处在室内馆，那是工作人员办公的地方，非工作人员不得进入!

场馆的定位是为本地用户提供舞台表演(统称舞台剧)，规定同时只能一个剧上演，但因为生意太好，申请人太多了，所以用馆要先申请->排队->上演。场馆里面有一座永远很准时，不会停的大钟表，每十分钟就自动响一次，场馆里有很多的资源，有篮球，酒馆，小卖部，桌椅，还有演员(人也算资源)，反正就是应有尽有，但是数量有限。

资源由管理处统一管理，这些资源也得先申请才能使用。场地外有个大屏幕，屏幕实时对外公布场馆舞台剧情况，屏幕内容如下:

```
舞台剧名 优先级   状态      进行中节目      就绪节目
-------------------------------------------------------------
管理处    0     正在工作   打扫场地卫生       无
三国演义  19     已就绪     无           骂死王朗
淘宝直播  20     已就绪     无           薇娅9点直播
```

场馆的内部工作也是个剧，只不过它的内部剧，优先级最高。而且注意这里只展示正在和就绪的剧情节目，就绪是指万事俱备，只欠登台表演的意思。

例如上表中有两个剧都准备好了，排成了一个就绪队列，都等着管理处打扫完卫生后表演，但同时只能演一个剧，而三国演义的优先级更高(场馆规定越小的优先级越高)，所以不出意外，下一个表演的节目就是三国演义之骂死王朗。

这里请记住就绪队列，后续会反复的提它，很重要!

表演走什么流程？

用馆者需提交你舞台剧的剧本，剧本可以是玩游戏，拍电视剧，直播电商等等，反正精彩的世界任你书写，场馆内有专人(统称导演)负责跟进你的剧本上演。

剧本由各种各样的场景剧组成(统称节目)，比如要拍个水浒传的剧本。 被分成武松打虎，西门和金莲那点破事等等节目剧。申请流程是去管理处先填一张电子节目表，节目表有固定的格式，填完点提交你的工作就完成了，接下来就是导演的事了。

节目表单格式如下。

```
剧名    节目章回  内容        优先级  所需资源         状态
-------------------------------------------------------------------
水浒传  第18回   武松打虎        12   武松，老虎，        未开始
水浒传  第28回   西门金莲那点破事   2    西门庆，金莲，炕     未开始
水浒传  第36回   武松拳打蒋门神    14   武松，蒋门神，猪肉    未开始
```

故事写到这里，大家脑子里有个画面了吧，记住这两张表，继续走起。

西门大官人什么时候表演？

场馆都会给每个用馆单位发个标号代表你使用场馆的优先级，剧本中每个场景节目也有优先级，都是0级最高，31级最低，这里比如水浒传优先级为8，西门庆和金莲那点破事节目为2，节目资源是需要两位主角(西门，金莲)和王婆，一个炕等资源，这些资源要向场馆负责人申请好，节目资源申请到位了就可以进入就绪队列。

如果你的剧本里没有一个节目的资源申请到了那对不起您连排号的资格都没有。这里假如水浒传审核通过，并只有西门大官人节目资源申请成功，而管理处卫生打扫完了，以上两个表格的内容将做如下更新

```
舞台剧名 优先级   状态      进行中节目      就绪节目
-----------------------------------------------------------
水浒传    8     正在工作   西门金莲那点破事       无
三国演义  19     已就绪     无            骂死王朗
淘宝直播  20     已就绪     无            薇娅9点直播
```

注意虽然三国演义先来，但此时水浒传排在三国的前面，是因为它的优先级高，优先级分32级，0最高，31最低。

```
剧名  节目章回  内容       优先级  所需资源       状态       表演位置
-------------------------------------------------------------------------------
水浒传  第18回 武松打虎       12  武松，老虎，酒18碗 未开始        暂无
水浒传  第28回 西门金莲那点破事 2   西门庆，金莲，炕   正在进行      西门火急火燎的跑进金莲屋内
水浒传  第36回 武松拳打蒋门神   14  武松，蒋门神，猪肉 未开始        暂无
```

注意看表中状态的变化和优先级，一个是剧本的优先级，一个是同一个剧本中节目的优先级。而之前优先级最高的管理处，因为没有其他节目要运行，所以移出了就绪队列。

西门好事被破坏了怎么办了？

场馆会根据节目上的内容把节目演完。每个节目十分钟，时间到了要回去重新排队，如果还是你就可以继续你的表演。但这里经常会有异常情况发生。

比如上级领导给场馆来个电话临时有个更高优先级节目要插进来，没办法西门你的好事要先停止，please stop! 场地要让给别人办事，西门灰溜溜得回就绪队列排队去，但请放心会在你西门退场前会记录下来表演到哪个位置了(比如:西门官人已脱完鞋)，以便回来时继续接着表演。高优先级的事处理完后，如果西门的优先级还是最高的就可以继续用场地，会先还原现场演到哪了再继续办事就完了，绝不重复西门前面的准备工作，否则西门绝不答应!

节目表演完所有资源要回收，这个节目从此消亡，如果你剧本里所有节目都表演完了，那你的整个剧本也可以拜拜了，导演回到导演组，又可以去接下一部戏了。

这里还原下西门被场馆紧急电话打断后表的变化是怎样的，如下:

```
剧本名称 优先级 状态       进行中节目      就绪节目
-----------------------------------------------------------
管理处     0   正在工作    接听上级电话 无
水浒传     8   已就绪      无          西门和金莲那点破事
三国演义    19  已就绪      无          骂死王朗
淘宝直播    20  已就绪      无          薇娅9点直播
```

```
剧名    节目章回  内容        优先级  所需资源         状态    表演位置
-------------------------------------------------------------------------------
水浒传  第18回  武松打虎       12     武松，老虎，酒18碗   未开始     暂无
水浒传  第28回  西门金莲那点破事 2     西门庆，金莲，一个炕  就绪     西门官人脱完鞋
水浒传  第36回  武松拳打蒋门神  14     武松，蒋门神，猪肉    未开始     暂无
```

表演给谁看呢？

外面那些吃瓜观众啊，群众你我他，游戏公司设计了游戏的剧本，电商公司设计了电商剧本，西门大官人被翻拍了这么多次不就是都爱看嘛，场馆会按你的剧本来表演，当然也可以互动，表演的场景需要观众操作时，观众在外面可以操作，发送指令。想想你玩游戏输入名字登录的场景。场馆里面有三个团队，张大爷团队负责导演组演剧本，王场馆负责场地的使用规划的，李后勤负责搞搞后勤。

张大爷团队做什么的？

上面这些工作都是张大爷团队的工作，接待剧本的导演组，管理剧本清单，指派导演跟进，申请节目资源，调整剧本优先级，控制时间，以使舞台能被公平公正的被调度使用等等

王场馆是做什么的？

看名字能知道负责场地用度的，你想想这么多节目，场地只有这么点，同时只能由一个节目上演，怎么合理的规划才能即公平又效率最大化呢，这就是王场馆的工作，但咱王总也有两把刷子，会给用馆公司感觉到整个场馆都是自己在用，具体不在这个故事里说明，后续有专门讲王场馆如何高效的管理内外场地的故事篇。

李后勤是做什么的？

场馆每天的开业，歇业，场地清理，管理处的对外业务，接听电话，有人闹事了怎么处理，收钱开发票 等等也有很多工作统称为后勤工作要有专门的团队来对接，具体不在这里说明，后续也有专门讲这块的故事。

故事想说什么呢？

故事到底想说什么呢？这就是操作系统的调度机制，熟悉了这个故事就熟悉了鸿蒙系统内核任务调度的工作原理！操作系统就是管理场馆和确保工作人员有序工作的系统解决方案商，外面公司只要提供个剧本，就能按剧本把这台戏演好给广大观众观看。有了这个故事垫底，鸿蒙内核源码分析系列就有了一个非常好的开始基础。

内核和故事的关系映射

```
故事概念        内核概念            备注
----------------------------------------------------------------
只能一个剧本演   单CPU            多CPU核指多个剧同时上演
剧本            程序              一个剧本一个负责人跟进，跑起来的程序叫进程
导演            进程              进程负责剧本整个运行过程，是资源管理单元，任务也是一种资源
节目            线程/任务         任务记录节目的整个运行过程，任务是调度的单元
西门被打断      保存现场          本质是保存寄存器(PC，LR，FP，SP)的状态
西门继续来      恢复现场          本质是还原寄存器(PC，LR，FP，SP)的状态
表演场地        用户空间          所有节目都在同一块场地表演
管理处          内核空间          管理处非工作人员不得入内
外部场地        磁盘空间          故事暂未涉及，留在内存故事中讲解
节目内容        代码段            任务涉及的具体代码段
管理处的服务    系统调用          软中断实现，切换至内核栈
场馆大钟        系统时钟          十分钟响一次代表一个节拍(tick)
节目20分钟      时间片            鸿蒙时间片默认 2个tick，20ms
上级电话        中断              硬中断，直接跳到中断处理函数执行
表演顺序        优先级            进程和线程都是32个优先级，[0-31]，从高到低
张大爷          进程/线程管理     抢占式调度，优先级高者运行
王场馆          内存管理          虚拟内存，内存分配，缺页置换 ==
李后勤          异常接管          中断，跟踪，异常接管 ==
```

请牢记这个故事

当然还有很多的细节在故事里没有讲到，比如王场馆和李后勤的工作细节，还有后续故事一一拆解。太细不可能真的在一个故事里全面讲完，笔者想说的是框架，架构思维，要先有整体框架再顺藤摸瓜寻细节，层层深入，否则很容易钻进死胡同里出不来。读着读着就放弃了，其实真没那么难。当你摸清了整个底层的运作机制再看上层的应用，就会有了拨开云雾见阳光，神清气爽的感觉。具体的我们在后续的章节里一一展开，用这个故事去理解鸿蒙系统内核调度过程，没毛病，请务必牢记这个故事。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

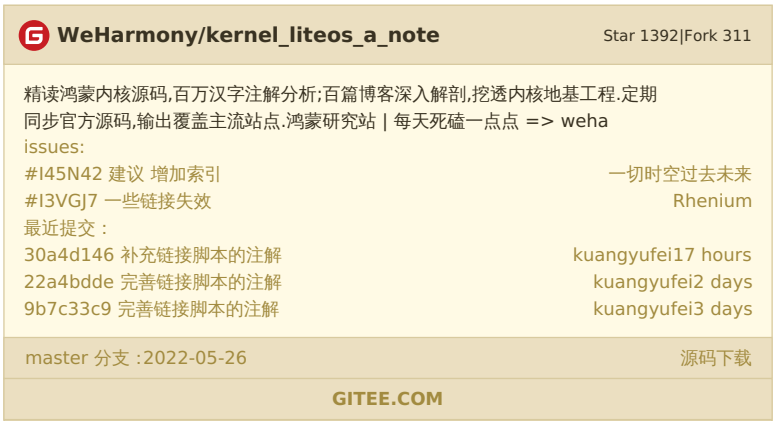
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

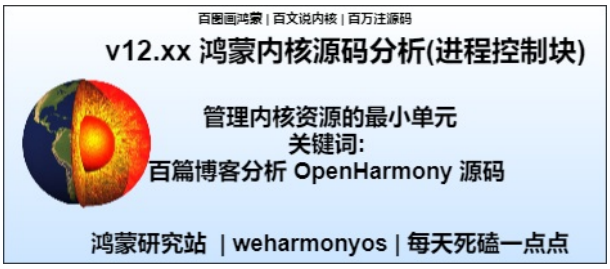


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

12_进程控制块篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

本篇说清楚进程

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 调度故事篇，其中有对进程生活场景式的比喻。

官方基本概念

- 从系统的角度看，进程是资源管理单元。进程可以使用或等待CPU、使用内存空间等系统资源，并独立于其它进程运行。
- 鸿蒙内核的进程模块可以给用户提供多个进程，实现了进程之间的切换和通信，帮助用户管理业务程序流程。这样用户可以将更多的精力投入到业务功能的实现中。
- 鸿蒙内核中的进程采用抢占式调度机制，支持时间片轮转调度方式和FIFO调度机制。
- 鸿蒙内核的进程一共有32个优先级(0-31)，用户进程可配置的优先级有22个(10-31)，最高优先级为10，最低优先级为31。
- 高优先级的进程可抢占低优先级进程，低优先级进程必须在高优先级进程阻塞或结束后才能得到调度。
- 每一个用户态进程均拥有自己独立的进程空间，相互之间不可见，实现进程间隔离。

官方概念解读

官方文档最重要的一句话是进程是资源管理单元，注意是管理资源的， 资源是什么？ 内存，任务，文件，信号量等等都是资源。故事篇中对进程做了一个形象的比喻(导演)，负责节目(任务)的演出，负责协调节目运行时所需的各种资源。让节目能高效顺利的完成。

鸿蒙内核源码分析定位为深挖内核地基，构筑底层网图。就要解剖真身。进程(LosProcessCB)原始真身如下，本篇一一剖析它，看看它到底长啥样。

ProcessCB真身

```
typedef struct ProcessCB {
    CHAR                processName[OS_PCB_NAME_LEN]; /**< Process name */ //进程名称
    UINT32              processID;                  /**< process ID = leader thread ID */ //进程ID，由进程池分配，范围[0，64]
    UINT16              processStatus;              /**< [15:4] process Status; [3:0] The number of threads currently
                                                    running in the process *///这里设计很巧妙。用一个16表示了两层逻辑 数量和状态，点赞!
    UINT16              priority;                   /**< process priority */ //进程优先级
    UINT16              policy;                     /**< process policy */ //进程的调度方式，默认抢占式
    UINT16              timeSlice;                  /**< Remaining time slice *///进程时间片，默认2个tick
    UINT16              consoleID;                  /**< The console id of task belongs  *///任务的控制台id归属
    UINT16              processMode;                /**< Kernel Mode:0; User Mode:1; */ //模式指定为内核还是用户进程
    UINT32              parentProcessID;            /**< Parent process ID */ //父进程ID
    UINT32              exitCode;                   /**< process exit status */ //进程退出状态码
    LOS_DL_LIST         pendList;                   /**< Block list to which the process belongs */ //进程所属的阻塞列表，如果因拿锁失败，就由此节点挂到等锁
    LOS_DL_LIST         childrenList;               /**< my children process list */ //孩子进程都挂到这里，形成双循环链表
    LOS_DL_LIST         exitChildList;              /**< my exit children process list */ //那些要退出孩子进程挂到这里，白发人送黑发人。
    LOS_DL_LIST         siblingList;                /**< linkage in my parent's children list */ //兄弟进程链表， 56个民族是一家，来自同一个父进程。
    ProcessGroup        *group;                     /**< Process group to which a process belongs */ //所属进程组
    LOS_DL_LIST         subordinateGroupList;       /**< linkage in my group list */ //进程是组长时，有哪些组员进程
    UINT32              threadGroupID;              /**< Which thread group ， is the main thread ID of the process */ //哪个线程组是进程的主线程ID
    UINT32              threadScheduleMap;          /**< The scheduling bitmap table for the thread group of the
                                                    process */ //进程的各线程调度位图
    LOS_DL_LIST         threadSiblingList;          /**< List of threads under this process *///进程的线程(任务)列表
    LOS_DL_LIST         threadPriQueueList[OS_PRIORITY_QUEUE_NUM]; /**< The process's thread group schedules the
                                                    priority hash table */ //进程的线程组调度优先级哈希表
    volatile UINT32     threadNumber; /**< Number of threads alive under this process */ //此进程下的活动线程数
    UINT32              threadCount;  /**< Total number of threads created under this process */ //在此进程下创建的线程总数
    LOS_DL_LIST         waitList;     /**< The process holds the waitLits to support wait/waitpid *///进程持有等待链表以支持wait/waitpid
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32              timerCpu;     /**< CPU core number of this task is delayed or pended *///统计各线程被延期或阻塞的时间
#endif
    UINTPTR             sigHandler;   /**< signal handler */ //信号处理函数，处理如 SIGSYS 等信号
    sigset_t            sigShare;     /**< signal share bit */ //信号共享位
#if (LOSCFG_KERNEL_LITEIPC == YES)
    ProcIpcInfo         ipcInfo;      /**< memory pool for lite ipc */ //用于进程间通讯的虚拟设备文件系统，设备装载点为 /dev/lite_ipc
#endif
    LosVmSpace          *vmSpace;     /**< VMM space for processes */ //虚拟空间，描述进程虚拟内存的数据结构，linux称为内存描述符
#ifdef LOSCFG_FS_VFS
    struct files_struct *files;       /**< Files held by the process */ //进程所持有的所有文件，注者称之为进程的文件管理器
#endif //每个进程都有属于自己的文件管理器，记录对文件的操作。 注意:一个文件可以被多个进程操作
    timer_t             timerID;      /**< iTimer */

#ifdef LOSCFG_SECURITY_CAPABILITY //安全能力
    User                *user;  //进程的拥有者
    UINT32              capability; //安全能力范围 对应 CAP_SETGID
#endif
#ifdef LOSCFG_SECURITY_VID
    TimerIdMap          timerIdMap;
#endif
#ifdef LOSCFG_DRIVERS_TZDRIVER
    struct file         *execFile;    /**< Exec bin of the process */
#endif
    mode_t umask;
} LosProcessCB;
```

结构体还是比较复杂，虽一一都做了注解，但还是不够清晰，没有模块化。这里把它分解成以下**六大块**逐一分析:

第一大块:和任务(线程)关系

```
    UINT32              threadGroupID;              /**< Which thread group ， is the main thread ID of the process */ //哪个线程组是进程的主线程ID
    UINT32              threadScheduleMap;          /**< The scheduling bitmap table for the thread group of the
                                                    process */ //进程的各线程调度位图
    LOS_DL_LIST         threadSiblingList;          /**< List of threads under this process *///进程的线程(任务)列表
    LOS_DL_LIST         threadPriQueueList[OS_PRIORITY_QUEUE_NUM]; /**< The process's thread group schedules the
```

```
                                                priority hash table */ //进程的线程组调度优先级哈希表
    volatile UINT32     threadNumber; /**< Number of threads alive under this process */ //此进程下的活动线程数
    UINT32              threadCount;  /**< Total number of threads created under this process */ //在此进程下创建的线程总数
    LOS_DL_LIST         waitList;     /**< The process holds the waitLits to support wait/waitpid *///进程持有等待链表以支持wait/waitpid
```

进程和线程的关系是`1:N`的关系，进程可以有多个任务但一个任务不能同属于多个进程。 任务就是线程，是CPU的调度单元。线程的概念在 v08。xx 鸿蒙内核源码分析(总目录) 中的线程篇中有详细的介绍，可自行翻看。 任务是作为一种资源被进程管理的，进程为任务提供内存支持，提供文件支持，提供设备支持。

进程怎么管理线程的，进程怎么同步线程的状态？

- 1.进程加载时会找到main函数创建第一个线程，一般为主线程，main函数就是入口函数，一切从哪里开始。
- 2.执行过程中根据代码(以java举例 如遇到 new thread)创建新的线程，其本质和main函数创建的线程没有区别，只是入口函数变成了`run()`，统一参与调度.
- 3.线程和线程的关系可以是独立(detached)的，也可以是联结(join)的.联结指的是一个线程可以操作另一个线程(包括回收资源，被对方干掉).
- 4.进程的主线程或所有线程运行结束后，进程转为僵尸态，一般只能由所有线程结束后，进程才能自然消亡.
- 5.进程创建后进入就绪态，发生进程切换时，就绪列表中最高优先级的进程被执行，从而进入运行态.若此时该进程中已无其它线程处于就绪态，则该进程从就绪列表删除，只处于运行态；若此时该进程中还有其它线程处于就绪态，则该进程依旧在就绪队列，此时进程的就绪态和运行态共存.这里要注意的是进程可以允许多种状态并存! 状态并存很自然的会想到位图管理，系列篇中有对位图详细的介绍.
- 6.进程内所有的线程均处于阻塞态时，进程在最后一个线程转为阻塞态时，同步进入阻塞态，然后发生进程切换.
- 7.阻塞进程内的任意线程恢复就绪态时，进程被加入到就绪队列，同步转为就绪态，若此时发生进程切换，则进程状态由就绪态转为运行态
- 8.进程内的最后一个就绪态线程处于阻塞态时，进程从就绪列表中删除，进程由就绪态转为阻塞态.
- 9.进程由运行态转为就绪态的情况有以下两种：
  - 有更高优先级的进程创建或者恢复后，会发生进程调度，此刻就绪列表中最高优先级进程变为运行态，那么原先运行的进程由运行态变为就绪态.
  - 若进程的调度策略为SCHED_RR(抢占式)，且存在同一优先级的另一个进程处于就绪态，则该进程的时间片消耗光之后，该进程由运行态转为就绪态，另一个同优先级的进程由就绪态转为运行态.

第二大块:和其他进程的关系

```
    CHAR                processName[OS_PCB_NAME_LEN]; /**< Process name */ //进程名称
    UINT32              processID;                /**< process ID = leader thread ID */ //进程ID，由进程池分配，范围[0，64]
    UINT16              processStatus;              /**< [15:4] process Status; [3:0] The number of threads currently
                                                 running in the process *///这里设计很巧妙。用一个16表示了两层逻辑 数量和状态，点赞!
    UINT16              priority;                /**< process priority */ //进程优先级
    UINT16              policy;                  /**< process policy */ //进程的调度方式，默认抢占式
    UINT16              timeSlice;                /**< Remaining time slice *///进程时间片，默认2个tick
    UINT16              consoleID;                /**< The console id of task belongs  *///任务的控制台id归属
    UINT16              processMode;              /**< Kernel Mode:0; User Mode:1; */ //模式指定为内核还是用户进程
    UINT32              parentProcessID;           /**< Parent process ID */ //父进程ID
    UINT32              exitCode;                /**< process exit status */ //进程退出状态码
    LOS_DL_LIST         pendList;                 /**< Block list to which the process belongs */ //进程所属的阻塞列表，如果因拿锁失败，就由此节点挂到等锁
    LOS_DL_LIST         childrenList;             /**< my children process list */ //孩子进程都挂到这里，形成双循环链表
    LOS_DL_LIST         exitChildList;            /**< my exit children process list */ //那些要退出孩子进程挂到这里，白发人送黑发人。
    LOS_DL_LIST         siblingList;             /**< linkage in my parent's children list */ //兄弟进程链表， 56个民族是一家，来自同一个父进程。
    #if (LOSCFG_KERNEL_LITEIPC == YES)
    ProcIpcInfo         ipcInfo;       /**< memory pool for lite ipc */ //用于进程间通讯的虚拟设备文件系统，设备装载点为 /dev/lite_ipc
    #endif
```

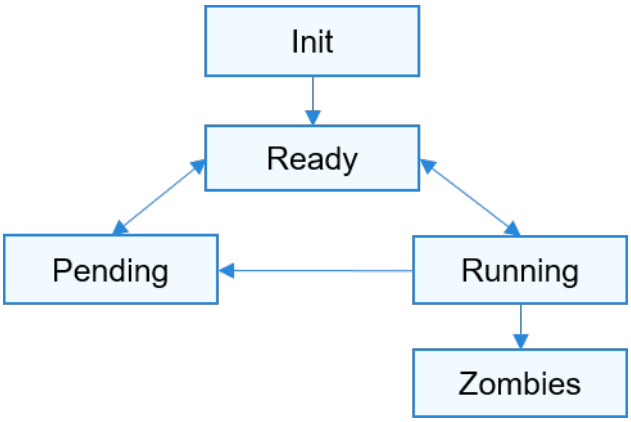
进程是家族式管理的，内核态进程和用户态进程分别有自己的根祖先，祖先进程在内核初始化时就创建好了，分别是1号(用户进程祖先)和2号(内核进程祖先)进程。进程刚生下来就确定了自己的基因，基因决定了你的权限不同， 父亲是谁，兄弟姐妹都有谁都已经安排好了，跟人一样，没法选择出生。 但进程可以有自己的子子孙孙， 从你这一脉繁衍下来的，这很像人类的传承方式。最终会形成树状结构，每个进程都能找到自己的位置。进程的管理遵循以下几点原则:

- 1.进程退出时会主动释放持有的进程资源，但持有的进程pid资源需要父进程通过wait/waitpid或父进程退出时回收.
- 2.一个子进程的消亡要通知父进程，以便父进程在族谱上抹掉它的痕迹，一些异常情况下的坏孩子进程消亡没有告知父进程的，系统也会有定时任务能检测到而回收其资源.

- 3.进程创建后，只能操作自己进程空间的资源，无法操作其它进程的资源（共享资源除外）.
- 4.进程间有多种通讯方式，事件，信号，消息队列，管道等等， liteipc是进程间基于文件的一种通讯方式，它的特点是传递的信息量可以很大.
- 5.高优先级的进程可抢占低优先级进程，低优先级进程必须在高优先级进程阻塞或结束后才能得到调度。

第三大块:进程的五种状态

- 初始化（Init）：该进程正在被创建。
- 就绪（Ready）：该进程在就绪列表中，等待CPU调度。
- 运行（Running）：该进程正在运行。
- 阻塞（Pend）：该进程被阻塞挂起。本进程内所有的线程均被阻塞时，进程被阻塞挂起。



- 僵尸态（Zombies）：该进程运行结束，等待父进程回收其控制块资源。

第四大块:和内存的关系

```
    LosVmSpace        *vmSpace;       /**< VMM space for processes */ //虚拟空间，描述进程虚拟内存的数据结构，linux称为内存描述符
```

- 进程与内存有关的就只有LosVmSpace一个成员变量，叫是进程空间，每一个用户态进程均拥有自己独立的进程空间，相互之间不可见，实现进程间隔离，独立进程空间意味着每个进程都要将自己的虚拟内存和物理内存进行映射。并将映射区保存在自己的进程空间。另外进程的代码区，数据区，堆栈区，映射区都存放在自己的空间中，但内核态进程的空间是共用的，只需一次映射。
- 具体的进入v08。xx 鸿蒙内核源码分析(总目录) 查看内存篇。详细介绍了虚拟内存，物理内存，线性地址，映射关系，共享内存，分配回收，页面置换的概念和实现。

第五大块:和文件的关系

```
#ifdef LOSCFG_FS_VFS
    struct files_struct *files;       /**< Files held by the process */ //进程所持有的所有文件，注者称之为进程的文件管理器
#endif //每个进程都有属于自己的文件管理器，记录对文件的操作。 注意:一个文件可以被多个进程操作
```

进程与文件系统有关的就只有files_struct，可理解为进程的文件管理器，文件也是很复杂的一大块， 后续有系列篇来讲解文件系统的实现。 理解文件系统的主脉络是:

- 1.一个真实的物理文件(`inode`)，可以同时被多个进程打开，并有进程独立的文件描述符， 进程文件描述符(`ProcessFD`)后边映射的是系统文件描述符(`SystemFD`)。
- 2.系统文件描述符(`0-stdin，1-stdout，2-stderr`)默认被内核占用，任何进程的文件描述符前三个都是(`stdin，stdout，stderr`)，默认已经打开，可以直接往里面读写数据。
- 3.文件映射跟内存映射一样，每个进程都需要单独对同一个文件进行映射，page_mapping记录了映射关系，而页高速缓存(`page cache`)提供了文件实际内存存放位置。
- 4.内存<->文件的置换以页为单位(4K)，进程并不能对硬盘文件直接操作，必须通过页高速缓存(`page cache`)完成。其中会涉及到一些经典的概念比如 `COW` (写时拷贝)技术。后续会详细说明。

第六大块:辅助工具

```
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32              timerCpu;     /**< CPU core number of this task is delayed or pended *///统计各线程被延期或阻塞的时间
#endif
#ifdef LOSCFG_SECURITY_CAPABILITY //安全能力
    User                *user;  //进程的拥有者
    UINT32              capability; //安全能力范围 对应 CAP_SETGID
#endif
#ifdef LOSCFG_SECURITY_VID
    TimerIdMap          timerIdMap;
#endif
#ifdef LOSCFG_DRIVERS_TZDRIVER
    struct file         *execFile;     /**< Exec bin of the process */
#endif
```

其余是一些安全性，统计性的能力。

以上就是进程的全貌，看清楚它鸿蒙内核的影像会清晰很多!

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
-

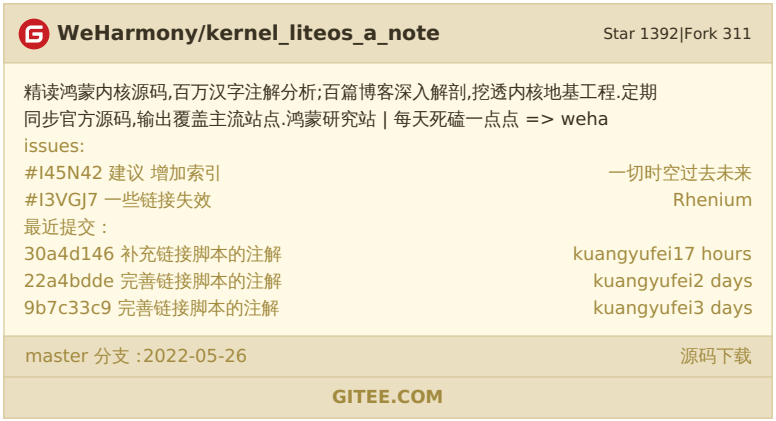
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

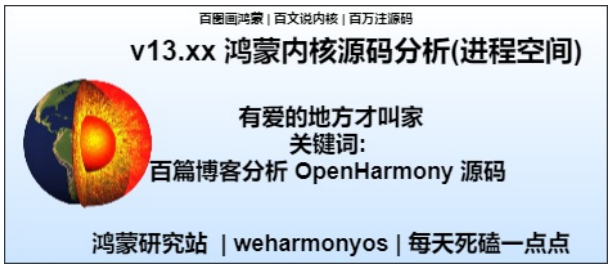


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

13_进程空间篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

虚拟空间

之所以要创建一个虚拟地址空间，目的是为了解决进程地址空间隔离的问题。但程序要想执行，必须运行在真实的内存上，所以，必须在虚拟地址与物理地址间建立一种映射关系。这样，通过映射机制，当程序访问虚拟地址空间上的某个地址值时，就相当于访问了物理地址空间中的值。

```
typedef struct VmSpace {
    LOS_DL_LIST         node;           /**< vm space dl list | 节点,通过它挂到全局虚拟空间 g_vmSpaceList 链表上*/
    LosRbTree           regionRbTree;   /**< region red-black tree root | 采用红黑树方式管理本空间各个线性区*/
    LosMux              regionMux;      /**< region list mutex lock | 虚拟空间操作红黑树互斥锁*/
    VADDR_T             base;           /**< vm space base addr | 虚拟空间的基地址,线性区的分配范围,常用于判断地址是否在内核还是用户空间*/
    UINT32              size;           /**< vm space size | 虚拟空间大小*/
    VADDR_T             heapBase;       /**< vm space heap base address | 堆区基地址，表堆区范围起点*/
    VADDR_T             heapNow;        /**< vm space heap base now | 堆区现地址，表堆区范围终点，do_brk()直接修改堆的大小返回新的堆区结束地址， hea
    LosVmMapRegion      *heap;          /**< heap region | 堆区是个特殊的线性区，用于满足进程的动态内存需求，大家熟知的malloc,realloc,free其实就是在操
    VADDR_T             mapBase;        /**< vm space mapping area base | 虚拟空间映射区基地址,L1，L2表存放在这个区 */
    UINT32              mapSize;        /**< vm space mapping area size | 虚拟空间映射区大小，映射区是个很大的区。*/
    LosArchMmu          archMmu;        /**< vm mapping physical memory | MMU记录<虚拟地址,物理地址>的映射情况 */
#ifdef LOSCFG_DRIVERS_TZDRIVER
    VADDR_T             codeStart;      /**< user process code area start | 代码区开始位置 */
    VADDR_T             codeEnd;        /**< user process code area end | 代码区结束位置 */
#endif
} LosVmSpace;
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

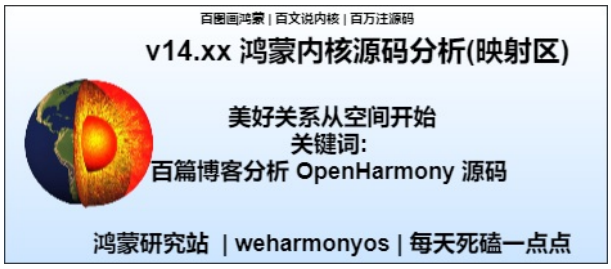
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

14_映射区篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

映射区

映射区，也称为**线性区**，是进程管理空间的单元，是大家耳熟能详的 **堆区**，**栈区**，**数据区** ... 的统称。它当然是很重要的，拿**堆区**举例，应用层被大家所熟知的 `new` 操作是由**堆区**分配的，但具体是怎么分配的很少有人关心，其过程大致分几个步骤：

- 1．在进程虚拟空间上的**堆区中画出**一个映射区，贴上一个堆区标签，它是一段连续的，长度为 `malloc` 的参数大小的虚拟地址范围，也称其为**线性地址**，交给应用说你已经拥有它了。`malloc` 多少次就开多少个这样的映射区。但此时只是记了个账而已，跟咱们商业银行卡里的余额一样，发工资了收到短信通知显示了一个让你面带微笑的数字，没让您看到真正的银子，那咱银子在哪呢？您先甭管，反正现在不用，用的时候自然会给您，而进程虚拟空间相当于商业银行。
- 2．将映射区的虚拟地址和物理地址做映射，并将映射关系保存在进程空间中的**映射区**内。注意这个映射区它也是一个映射区，它是最早被映射的一个区，系列篇之 **页表管理** 中详细说明了它的实现。其具体的位置在**栈区**和**堆区**的中间，**栈区**是由高地址向下生长，**堆区**是由低地址向上生长，都向中央映射区靠拢。
- 3．将映射区的红黑树节点交给红黑树管理，方便后续的查询和销账。这个阶段相当于将余额数字和咱的银子捆绑在一块入库，中央银行得承认这银子属于咱的不是。
- 4．只有应用在真正访问虚拟地址时，会根据进程映射表查询到物理内存是否已经被这段虚拟地址所使用，如果没有，则产生 **缺页中断**。相当于取钱时银行才给您准备好钱，没钱就去中央银行调取，只有中央银行才能发行毛爷爷，银子相当于物理地址，都想要，谁愿意跟它过不去呢？余额显示的数字就相当于虚拟地址，看着也能乐半天，有中央银行背书就行。关于这块系列篇 **缺页中断篇** 中有详细的说明。

回顾下 **进程虚拟空间图**

```
extern "C" {
#endif /* __cplusplus */
#endif /* __cplusplus */
/*****************************************************************************  @note_pic
*   鸿蒙虚拟内存-用户空间图 从 USER_ASPACE_BASE 至 USER_ASPACE_TOP_MAX
*   鸿蒙源码分析系列篇: https://blog.csdn.net/kuangyufei
*                       https://my.oschina.net/u/3751245
*****************************************************************************
//  |            /\              |
//  |            ||              |
//  |----------------------------|内核空间结束位置KERNEL_ASPACE_BASE + KERNEL_ASPACE_SIZE
//  |                            |
//  |        内核空间             |
//  |                            |
//  |                            |
//  |----------------------------|内核空间开始位置 KERNEL_ASPACE_BASE
//  |                            |
//  |        16M 预留             |
//  |----------------------------|用户空间栈顶 USER_ASPACE_TOP_MAX = USER_ASPACE_BASE + USER_ASPACE_SIZE
//  |                            |
//  |        stack区 自上而下     |
//  |                            |
//  |            ||              |
//  |            ||              |
//  |            ||              |
//  |            \/              |
//  |                            |
//  |----------------------------|映射区结束位置 USER_MAP_BASE + USER_MAP_SIZE
//  |    虚拟地址-物理地址映射区   |
//  |                            |
//  |----------------------------|映射区开始位置 USER_MAP_BASE
//  |                            |
//  |                            |
//  |            /\              |
//  |            ||              |
//  |            ||              |
//  |            ||              |
//  |                            |
//  |        heap 自下而上        |
//  |                            |
//  |----------------------------|用户空间堆区开始位置 USER_HEAP_BASE = USER_ASPACE_TOP_MAX >> 2
//  |                            |
//  |        .bss                |
//  |        .data               |
//  |        .text               |
//  |----------------------------|用户空间开始位置 USER_ASPACE_BASE = 0x01000000UL
//  |                            |
//  |        16M预留              |
//  |----------------------------|虚拟内存开始位置 0x00000000

/* user address space, defaults to below kernel space with a 16MB guard gap on either side */
#ifndef USER_ASPACE_BASE //用户地址空间, 默认为低于内核空间, 两侧各有16MB的保护间隙
#define USER_ASPACE_BASE            ((vaddr_t)0x01000000UL) //用户空间基地址 从16M位置开始
#endif
```


结构体

映射区在鸿蒙内核的表达结构体为 `VmMapRegion`

```
struct VmMapRegion {
    LosRbNode          rbNode;        /**< region red-black tree node | 红黑树节点,通过它将本映射区挂在VmSpace.regionRbTree*/
    LosVmSpace         *space;   ///< 所属虚拟空间,虚拟空间由多个映射区组成
    LOS_DL_LIST        node;          /**< region dl list | 链表节点,通过它将本映射区挂在VmSpace.regions上*/
    LosVmMapRange      range;         /**< region address range | 记录映射区的范围*/
    VM_OFFSET_T        pgOff;         /**< region page offset to file | 以文件开始处的偏移量, 必须是分页大小的整数倍, 通常为0, 表示从文件头开始映射。*/
    UINT32             regionFlags;   /**< region flags: cow, user_wired | 映射区标签*/
    UINT32             shmid;         /**< shmid about shared region | shmid为共享映射区id,id背后就是共享映射区*/
    UINT8              forkFlags;     /**< vm space fork flags: COPY, ZERO, | 映射区标记方式*/
    UINT8              regionType;    /**< vm region type: ANON, FILE, DEV | 映射类型是匿名,文件,还是设备,所谓匿名可理解为内存映射*/
  union {
        struct VmRegionFile {// <磁盘文件，物理内存, 用户进程虚拟地址空间 >
            int f_oflags; ///< 读写标签
            struct Vnode *vnode;///< 文件索引节点
            const LosVmFileOps *vmFOps;///< 文件处理各操作接口,open,read,write,close,mmap
        } rf;
   //匿名映射是指那些没有关联到文件页，如进程堆、栈、数据区和任务已修改的共享库等与物理内存的映射
        struct VmRegionAnon {//<swap区，物理内存, 用户进程虚拟地址空间 >
            LOS_DL_LIST  node;         /**< region LosVmPage list | 映射区虚拟页链表*/
        } ra;
        struct VmRegionDev {//设备映射,也是一种文件
            LOS_DL_LIST  node;         /**< region LosVmPage list | 映射区虚拟页链表*/
```

```
        const LosVmFileOps *vmFOps; ///< 操作设备像操作文件一样方便.
    } rd;
  } unTypeData;
};
```

解读

- `rbNode` 红黑树结点，映射区通过它挂到所属进程的红黑树上，每个进程都有一颗红黑树，用于管理本进程空间的映射区，它是第一个成员变量，所以在代码中可以按以下方式使用。

```
ULONG_T LOS_RbAddNode(LosRbTree *pstTree, LosRbNode *pstNew);
BOOL OsInsertRegion(LosRbTree *regionRbTree, LosVmMapRegion *region)
 {
    LOS_RbAddNode(regionRbTree, (LosRbNode *)region)
    // ...
 }
```

- `space` 所属进程空间，一个映射区只属于一个进程空间，共享映射区指的是该映射区与其他进程的某个映射区都映射至同一块已知物理内存(指：地址和大小明确)。
- `node` 从代码历史来看，鸿蒙最早管理映射区使用的是双向链表，后来才使用红黑树，但看代码中没有使用 `node` 的地方，可以把它删除掉了。
- `range` 记录映射区的范围，结构体很简单，只能是一段连续的虚拟地址。

```
typedef struct VmMapRange {//映射区范围结构体
  VADDR_T            base;          /**< vm region base addr | 映射区基地址*/
  UINT32             size;          /**< vm region size | 映射区大小*/
  } LosVmMapRange;
```

- `pgOff` 页偏移 ，与文件映射有关 ，文件是按页(4K)读取进存储空间的，此处记录文件的页偏移
- `regionFlags` 区标识 ，包括 堆区，栈区，数据区，共享区，映射区等等，整个进程虚拟地址空间由它们组成，统称为映射区。

```
  //...
  #define    VM_MAP_REGION_FLAG_STACK             (1<<9)  ///< 映射区的类型:栈区
  #define    VM_MAP_REGION_FLAG_HEAP              (1<<10)  ///< 映射区的类型:堆区
  #define    VM_MAP_REGION_FLAG_DATA              (1<<11)  ///< data数据区 编译在ELF中
  #define    VM_MAP_REGION_FLAG_TEXT              (1<<12)  ///< 代码区
  #define    VM_MAP_REGION_FLAG_BSS               (1<<13)  ///< bbs数据区 由运行时动态分配,bss段（Block Started by Symbol segment）通常是
  #define    VM_MAP_REGION_FLAG_VDSO              (1<<14)  ///< VDSO（Virtual Dynamic Shared Object，虚拟动态共享库）由内核提供的虚拟.s
  #define    VM_MAP_REGION_FLAG_MMAP              (1<<15)  ///< 映射区,虚拟空间内有专门用来存储<虚拟地址-物理地址>映射的区域
  #define    VM_MAP_REGION_FLAG_SHM               (1<<16)  ///< 共享内存区, 被多个进程映射区映射
```

- `shmid` 共享ID， `regionFlags` 为 `VM_MAP_REGION_FLAG_SHM` 时有效，详细内容前往系列篇之**共享内存**了解
- `forkFlags` 表示映射区的两种创建方式 **分配** 和 **共享**
- `regionType` 映射区的映射类型，类型划定的标准是文件， 类型不同决定了 `unTypeData` 不同，它是个联合体，说明映射区只能映射一种类型， 映射区的目的是要处理/计算数据，数据可能来源于普通文件，I/O设备，或者是物理内存 映射有三种类型
  - 匿名映射，那些没有关联到文件页，如进程堆、栈、数据区和任务已修改的共享库，可以理解为与物理内存的直接映射

```
struct VmRegionAnon {
      LOS_DL_LIST  node;         /**< region LosVmPage list | 映射区虚拟页链表*/
} ra;
```

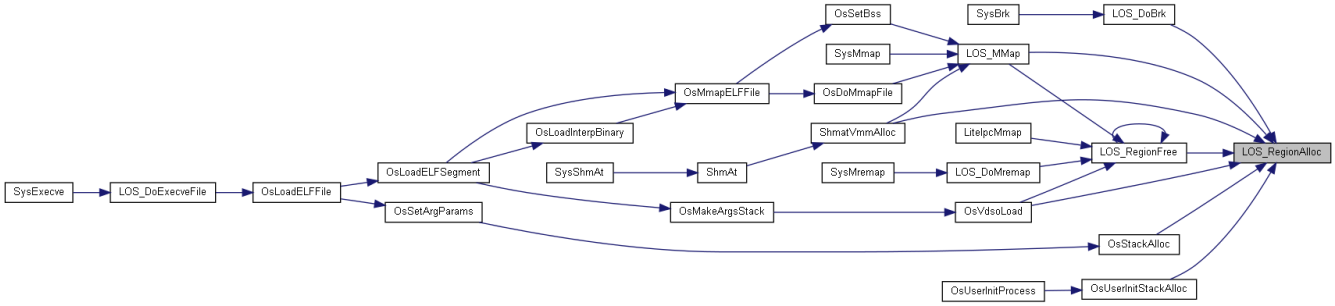
  - 文件映射，跟文件绑定在一块，对外以文件的方式操作映射区，其实背后也需要与物理内存的映射做承载，具体看系列篇之**文件映射**

```
struct VmRegionFile {// <磁盘文件，物理内存, 用户进程虚拟地址空间 >
      int f_oflags; ///< 读写标签
      struct Vnode *vnode;///< 文件索引节点
      const LosVmFileOps *vmFOps;///< 文件处理各操作接口,open,read,write,close,mmap
   } rf;
```

  - 设备映射，设备也是一种文件类型 ，对外同样的是以文件的方式操作映射区，但实现与文件映射完全不同不需要与物理内存映射，而是操作设备的驱动程序，具体看系列篇之**I/O映射**

```
struct VmRegionDev {//设备映射,也是一种文件
      LOS_DL_LIST  node;         /**< region LosVmPage list | 映射区虚拟页链表*/
      const LosVmFileOps *vmFOps; ///< 操作设备像操作文件一样方便.
   } rd;
```

创建映射区



解读 哪些地方会创建映射区呢? 看上面完整的调用图，大概有五个入口：

- 系统调用 **SysBrk --> LOS_DoBrk** ，这个函数看网上很多文章说它，总觉得没有说透，它的作用是创建和修改**堆区**，在堆区初始状态下(堆底地址等于堆顶地址时)，会创建一个映射区作为堆区，从**进程虚拟空间图**中可知，**堆区**是挨着**数据区**的，开始位置是固定的，对于每个进程来说，内核维护着一个 `brk`(break)变量，在鸿蒙内核就是 `heapNow` ，它指向堆区顶部。堆顶可由系统调用**SysBrk** 动态调整，具体看下虚拟空间对堆的描述。

```
typedef struct VmSpace {
 //...堆区描述
 VADDR_T           heapBase;      /**< vm space heap base address | 堆区基地址，表堆区范围起点*/
 VADDR_T           heapNow;       /**< vm space heap base now | 堆顶地址，表示堆区范围终点，do_brk()直接修改堆的大小返回新的堆区结束地址，
 LosVmMapRegion     *heap;         /**< heap region | 堆区是个特殊的映射区，用于满足进程的动态内存需求，大家熟知的malloc,realloc,free其实就是
}

#define USER_HEAP_BASE            ((vaddr_t)(USER_ASPACE_TOP_MAX >> 2))  ///< 堆的开始地址
vmSpace->heapBase = USER_HEAP_BASE;//用户堆区开始地址,只有用户进程需要设置这里，动态内存的开始地址
vmSpace->heapNow = USER_HEAP_BASE;//堆区最新指向地址，用户堆空间大小可通过系统调用 do_brk()扩展
```

为了更好的理解**SysBrk**的实现，此处将代码全部贴出，关键处已添加注释。

```
VOID *LOS_DoBrk(VOID *addr)
{
    LosVmSpace *space = OsCurrProcessGet()->vmSpace;
    size_t size;
    VOID *ret = NULL;
    LosVmMapRegion *region = NULL;
    VOID *alignAddr = NULL;
    VOID *shrinkAddr = NULL;
    if (addr == NULL) {//参数地址未传情况
        return (void *)(UINTPTR)space->heapNow;//以现有指向地址为基础进行扩展
    }

    if ((UINTPTR)addr < (UINTPTR)space->heapBase) {//heapBase是堆区的开始地址，所以参数地址不能低于它
        return (VOID *)-ENOMEM;
    }
    size = (UINTPTR)addr - (UINTPTR)space->heapBase;//算出大小
    size = ROUNDUP(size, PAGE_SIZE); //圆整size
    alignAddr = (CHAR *)(UINTPTR)(space->heapBase) + size;//得到新的映射区的结束地址
    PRINT_INFO("brk addr %p , size 0x%x, alignAddr %p, align %d\n", addr, size, alignAddr, PAGE_SIZE);
    (VOID)LOS_MuxAcquire(&space->regionMux);
    if (addr < (VOID *)(UINTPTR)space->heapNow) {//如果地址小于堆区现地址
        shrinkAddr = OsShrinkHeap(addr, space);//收缩堆区
        (VOID)LOS_MuxRelease(&space->regionMux);
        return shrinkAddr;
    }
    if ((UINTPTR)alignAddr >= space->mapBase) {//参数地址 大于映射区地址
        VM_ERR("Process heap memory space is insufficient");//进程堆空间不足
        ret = (VOID *)-ENOMEM;
        goto REGION_ALLOC_FAILED;
    }
    if (space->heapBase == space->heapNow) {//往往是第一次调用本函数才会出现，因为初始化时 heapBase = heapNow
        region = LOS_RegionAlloc(space, space->heapBase, size,//分配一个可读/可写/可使用的映射区，只需分配一次
                                 VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_WRITE |//映射区的大小由range.size决定
                                 VM_MAP_REGION_FLAG_FIXED | VM_MAP_REGION_FLAG_PERM_USER, 0);
        if (region == NULL) {
            ret = (VOID *)-ENOMEM;
```

```
            VM_ERR("LOS_RegionAlloc failed");
            goto REGION_ALLOC_FAILED;
        }
        region->regionFlags |= VM_MAP_REGION_FLAG_HEAP;//贴上映射区类型为堆区的标签,注意一个映射区可以有多种标签
        space->heap = region;//指定映射区为堆区
    }
    space->heapNow = (VADDR_T)(UINTPTR)alignAddr;//更新堆区顶部位置
    space->heap->range.size = size; //更新堆区大小,经此操作映射区变大或缩小了
    ret = (VOID *)(UINTPTR)space->heapNow;//返回堆顶
REGION_ALLOC_FAILED:
    (VOID)LOS_MuxRelease(&space->regionMux);
    return ret;
}
```

- 系统调用 **SysMmap --> LOS_MMap** ，动态内存一定是从堆区申请的吗？如果 malloc 的请求超过 MMAP_THRESHOLD (默认128KB)，**musl**库则会创建一个匿名映射而不是直接在堆区域分配);具体看下

```
void *malloc(size_t n)
{
  if (n > MMAP_THRESHOLD) { //申请内存大于 128K时,创建一个映射区
    size_t len = n + OVERHEAD + PAGE_SIZE - 1 & -PAGE_SIZE;
    char *base = __mmap(0, len, PROT_READ|PROT_WRITE,
      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    if (base == (void *)-1) return 0;
    c = (void *)(base + SIZE_ALIGN - OVERHEAD);
    c->csize = len - (SIZE_ALIGN - OVERHEAD);
    c->psize = SIZE_ALIGN - OVERHEAD;
    return CHUNK_TO_MEM(c);
  }
  // ...
}
```

  其中的**__mmap**是系统调用，最终会跑到 **LOS_MMap** ，划出一个新的映射区
- 系统调用共享内存 **SysShmAt**，划出一个共享映射区，
- 栈区 **OsStackAlloc**
- 内核内部使用 **OsUserInitProcess**

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

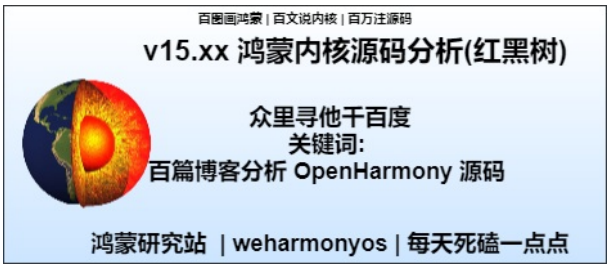
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

15_红黑树篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

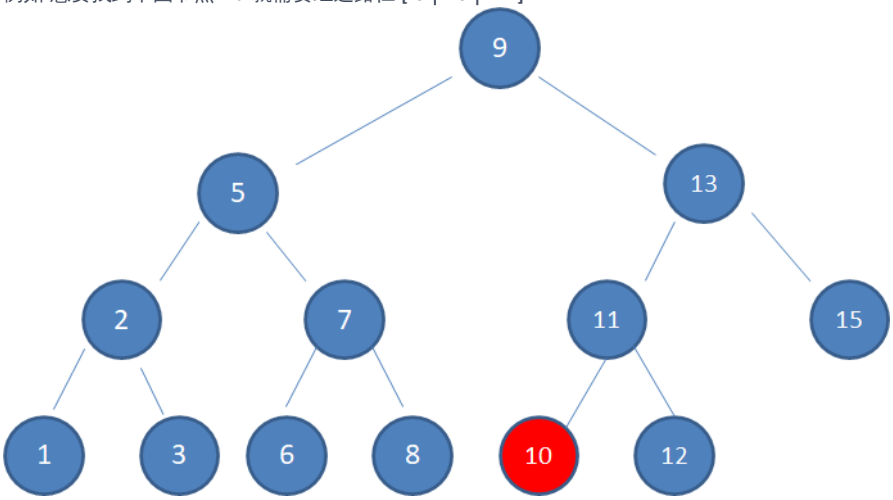
- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

二叉查找树 | BST

要理解红黑树，需要从二叉查找树(Binary Search Tree)开始说起。其特点是：

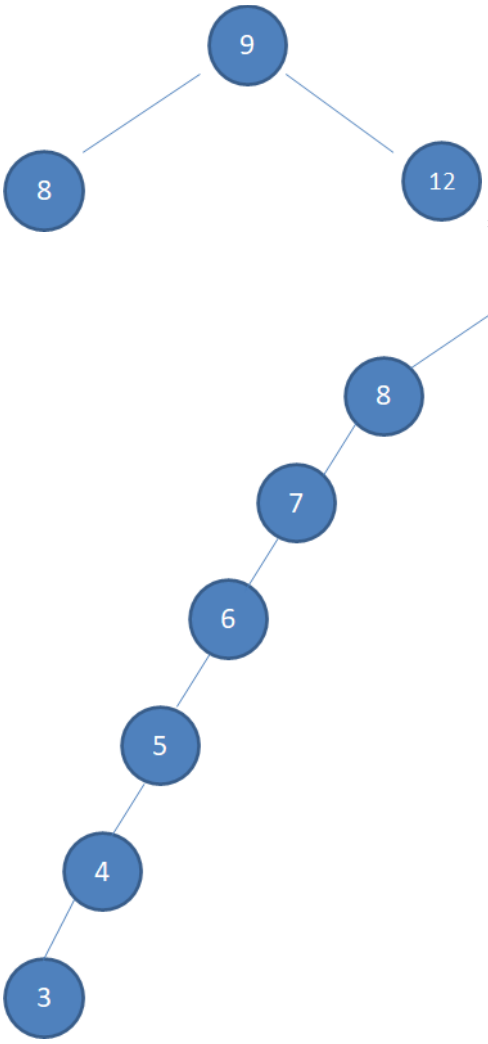
- 1．左子树上所有结点的值均小于或等于它的根结点的值。
- 2．右子树上所有结点的值均大于或等于它的根结点的值。
- 3．左、右子树也分别为二叉排序树。

例如:想要找到下图节点 10 就需要经过路径 [9 | 13 | 11]



在理想的情况下，二叉查找树增删查改的时间复杂度为O(logN)（其中N为节点数），最坏的情况下为O(N)。

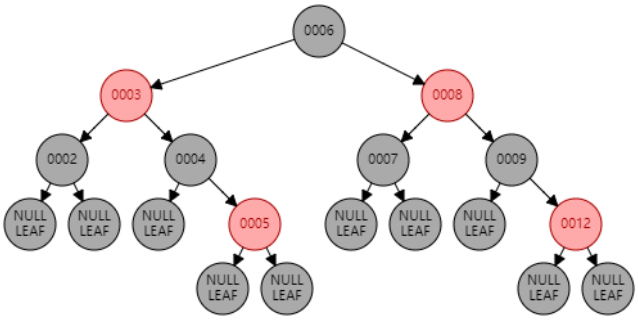
但 BST 的特点是在操作过程中容易失去平衡，数在插入的时候会导致树倾斜，不同的插入顺序会导致树的高度不一样，而树的高度直接的影响了树的查找效率。理想的高度是 logN ，最坏的情况是所有的节点都在一条斜线上，这样的树的高度为 N 。例如:下图为初始的二叉查找树



接下来依次插入如下五个结点：**7,6,5,4,3**，结果将变成简直没法看的

红黑树

基于 BST 存在的问题，一种新的树——平衡二叉查找树即 **红黑树**（Red-Black Tree，以下简称RBTree），它在插入和删除的时候，会通过旋转操作将高度保持在 logN 。通俗的讲就是会自我纠正，跟人睡觉一样，将自己的身体调整到一个让最省力，最舒服的姿势。其实应用非常广泛，比如Linux内核中的完全公平调度器、高精度计时器、ext3文件系统等等，鸿蒙内核中也使用了红黑树来管理进程线性区。上图经过红黑树调整之后的姿



势为：

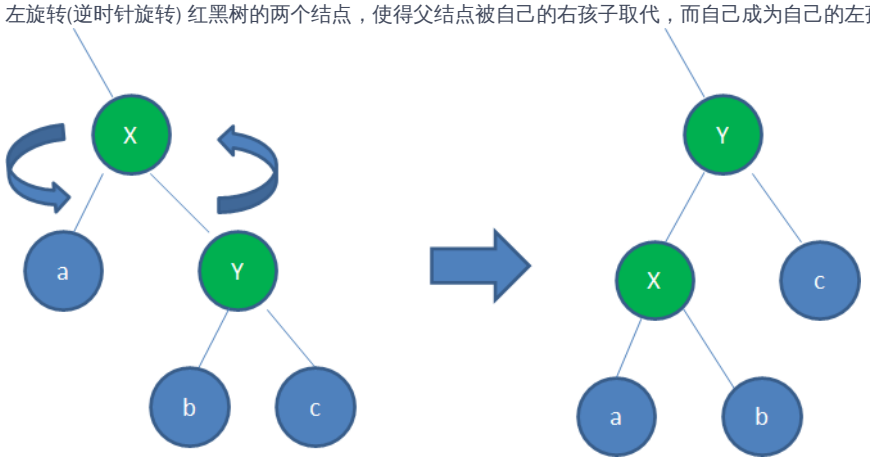
红黑树有以下几个特点：

- 1．任何一个节点都有颜色，黑色或者红色。
- 2．根节点是黑色的。
- 3．父子节点之间不能出现两个连续的红节点。
- 4．任何一个节点向下遍历到其子孙的叶子节点，所经过的黑节点个数必须相等。
- 5．空节点被认为是黑色的。

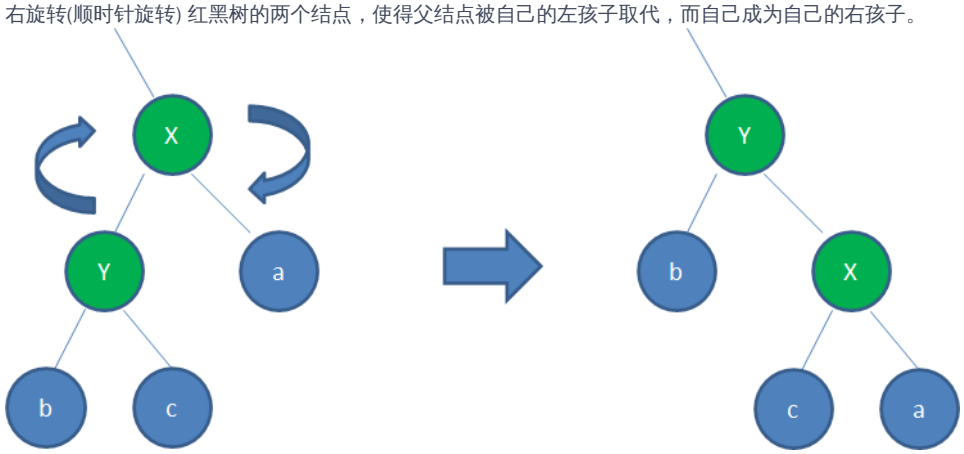
- 6．从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。

红黑树具体的调整姿势的方法有两种：

- **变色** 根据红黑树的规则，尝试把红色结点变为黑色，或者把黑色结点变为红色。
- **旋转**，根据旋转方向又分成 **左旋转** 和 **右旋转**
  - 左旋转(逆时针旋转) 红黑树的两个结点，使得父结点被自己的右孩子取代，而自己成为自己的左孩子。



  - 右旋转(顺时针旋转) 红黑树的两个结点，使得父结点被自己的左孩子取代，而自己成为自己的右孩子。



- 有兴趣的可以前往 红黑树演示网站 玩一下，很有意思

在鸿蒙使用

鸿蒙使用红黑树主要用来管理进程的 **线性区** ，关于线性区不清楚的请自行翻看系列篇，简单说就是一个虚拟地址连续的内存记录。我们动态申请堆内存其实就是申请了一个线性区，释放内存时就需要查找这个记录，这种使用频率很高，而红黑树的查找效率最高。鸿蒙红黑树功能的核心结构体是 **LosRbTree** ，每一个进程都有一颗属于自己的红黑树。

```
typedef struct VmMapRange {//线性区范围结构体
    VADDR_T            base;          /**< vm region base addr | 线性区基地址*/
    UINT32             size;          /**< vm region size | 线性区大小*/
} LosVmMapRange;

typedef struct TagRbNode {//节点
    struct TagRbNode *pstParent; ///< 爸爸是谁？
    struct TagRbNode *pstRight;  ///< 右孩子是谁？
    struct TagRbNode *pstLeft;  ///< 左孩子是谁？
    ULONG_T lColor; //是红还是黑节点
} LosRbNode;

typedef struct TagRbTree {//红黑树控制块
    LosRbNode *pstRoot;//根节点
    LosRbNode stNilT;//叶子节点
    LOS_DL_LIST stWalkHead;
    ULONG_T ulNodes;

    pfRBCmpKeyFn pfCmpKey; //比较两个节点大小 处理函数
```

```
    pfRBFreeFn pfFree; //释放结点占用内存函数
    pfRBGetKeyFn pfGetKey;//获取指定线性区范围 VmMapRange 函数
} LosRbTree;
```

解读

- 红黑树初始化，在进程空间初始化期间会初始化红黑树，并指定三个回调函数(方便红黑树节点的遍历)。

```
//初始化进程的红黑树
VOID LOS_RbInitTree(LosRbTree *pstTree, pfRBCmpKeyFn pfCmpKey, pfRBFreeFn pfFree, pfRBGetKeyFn pfGetKey)
{
    OsRbInitTree(pstTree);
    pstTree->pfCmpKey = pfCmpKey;
    pstTree->pfFree = pfFree;
    pstTree->pfGetKey = pfGetKey;
    return;
}
//初始化进程虚拟空间
STATIC BOOL OsVmSpaceInitCommon(LosVmSpace *vmSpace, VADDR_T *virtTtb){
    //...
    LOS_RbInitTree(&vmSpace->regionRbTree, OsRegionRbCmpKeyFn, OsRegionRbFreeFn, OsRegionRbGetKeyFn);
}
///通过红黑树节点找到对应的线性区
VOID *OsRegionRbGetKeyFn(LosRbNode *pstNode)
{
    LosVmMapRegion *region = (LosVmMapRegion *)LOS_DL_LIST_ENTRY(pstNode, LosVmMapRegion, rbNode);
    return (VOID *)&region->range;
}
///比较两个红黑树节点
ULONG_T OsRegionRbCmpKeyFn(const VOID *pNodeKeyA, const VOID *pNodeKeyB)
{
    LosVmMapRange rangeA = *(LosVmMapRange *)pNodeKeyA;
    LosVmMapRange rangeB = *(LosVmMapRange *)pNodeKeyB;
    UINT32 startA = rangeA.base;
    UINT32 endA = rangeA.base + rangeA.size - 1;
    UINT32 startB = rangeB.base;
    UINT32 endB = rangeB.base + rangeB.size - 1;

    if (startA > endB) {// A基地址大于B的结束地址
        return RB_BIGGER; //说明线性区A更大,在右边
    } else if (startA >= startB) {
        if (endA <= endB) {
            return RB_EQUAL; //相等,说明 A在B中
        } else {
            return RB_BIGGER; //说明 A的结束地址更大
        }
    } else if (startA <= startB) { //A基地址小于等于B的基地址
        if (endA >= endB) {
            return RB_EQUAL; //相等 说明 B在A中
        } else {
            return RB_SMALLER;//说明A的结束地址更小
        }
    } else if (endA < startB) {//A结束地址小于B的开始地址
        return RB_SMALLER;//说明A在
    }
    return RB_EQUAL;
}
```

- 插入结点，注意线性区结构体(`VmMapRegion`)的首个成员便是红黑树节点，对于红黑树来说线性区只是一个红黑树节点。

```
struct VmMapRegion {
  LosRbNode          rbNode;        /**< region red-black tree node | 红黑树节点,通过它将本线性区挂在VmSpace.regionRbTree*/
  // ...
}
//插入线性区,指的是向进程的红黑树 `LosRbTree` 插入 `LosRbNode`
BOOL OsInsertRegion(LosRbTree *regionRbTree, LosVmMapRegion *region)
  {
      if (LOS_RbAddNode(regionRbTree, (LosRbNode *)region) == FALSE) {
          VM_ERR("insert region failed, base: %#x, size: %#x", region->range.base, region->range.size);
          OsDumpAspace(region->space);
```

```
            return FALSE;
        }
        return TRUE;
    }
```

具体插入过程不去说明，已经是很成熟的算法，通过比较线性区的范围来决定是插入进具体位置，插入过程会涉及到红黑树 **颜色翻转** 和 **左右旋转** 两种变化

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

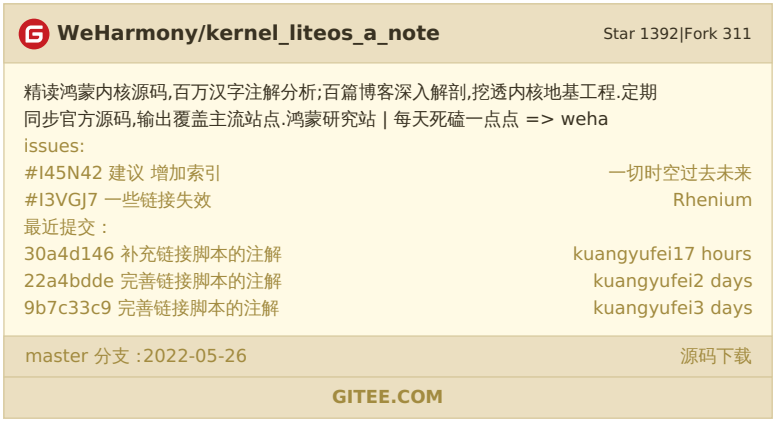
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

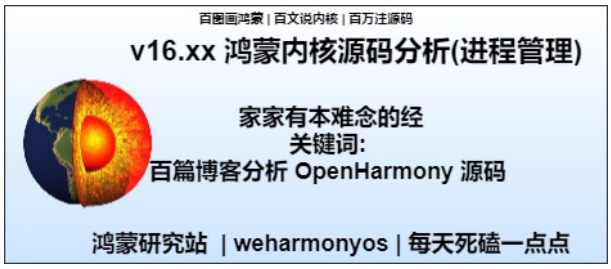


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

16_进程管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

三个进程

鸿蒙有三个特殊的进程，创建顺序如下:

- 2号进程，`KProcess`，为内核态根进程。启动过程中创建。
- 0号进程，`KIdle` 为内核态第二个进程，它是通过 `KProcess` fork 而来的。这有点难理解。
- 1号进程，`init`，为用户态根进程。由任务`SystemInit`创建。

```
OHOS # task

 PID  PPID PGID       UID  Status CPUUSE10s  PName
   1    -1    1         0    Pend       0.0  init
   2    -1    2         0    Pend       0.1  KProcess
   3     1    1         0 Running       0.0  shell

 TID  PID Affi CPU       Status StackSize WaterLine    MEMUSE  TaskName
  20    1  0x3  -1        Delay    0x3000     0xb20    0x8d80  init
   1    2  0x1  -1         Pend    0x6000     0x5c8         0  Swt_Task
   2    2  0x3  -1         Pend    0x6000     0x2ac         0  system_wq
   4    2  0x1  -1        Delay    0x1000     0x268         0  oom_task
   5    2  0x2  -1         Pend    0x6000     0x2b4         0  Swt_Task
   7    2  0x3  -1         Pend    0x6000     0x3e4         0  SendToSer
   8    2  0x3  -1     PendTime    0x6000     0x604         0  tcpip_thread
   9    2  0x1  -1         Pend    0x6000     0x2cc         0  jffs2_gc_thread
  10    2  0x3  -1         Pend    0x3000     0x2bc         0  himci_Task
  11    2  0x3  -1         Pend    0x4000     0x3f0   0x14b8c  eth_irq_Task
  12    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_GIANT_Task
  13    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_NGIAN_ISOC_Task
  14    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_NGIAN_BULK_Task
  15    2  0x3  -1         Pend    0x6000     0x718     0xb34  USB_EXPLR_Task
  16    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_CXFER_Task
  17    2  0x3  -1         Pend    0x6000     0x624         0  TouchEventHandler
  18    2  0x3  -1         Pend    0x6000     0x2c4         0  TouchGetEventTask
  19    2  0x3  -1         Pend    0x6000     0x2e4         0  TouchHandlerEventTask
   3    3  0x3  -1         Pend    0x3000     0xb18    0x5c8  shell
  21    3  0x3   1      Running    0x3000     0xecc   0x46a30  ShellTask
  22    3  0x3  -1         Pend    0x3000     0x924     0x450  ShellEntry
```


- 发现没有在图中看不到0号进程，在看完本篇之后请想想为什么？

家族式管理

- 进程(process)是家族式管理，总体分为两大家族，用户态家族和内核态家族。
- 用户态的进程是平民阶层，屌丝矮挫穷，干着各行各业的活，权利有限，人数众多，活动范围有限(用户空间)。有关单位肯定不能随便进出。这个阶层有个共同的老祖宗g_userInitProcess (1号进程)。

```
g_userInitProcess = 1; /* 1: The root process ID of the user-mode process is fixed at 1 *///用户态的根进程
//获取用户态进程的根进程，所有用户进程都是g_processCBArray[g_userInitProcess] fork来的
LITE_OS_SEC_TEXT UINT32 OsGetUserInitProcessID(VOID)
{
   return g_userInitProcess;
}
```

- 内核态的进程是贵族阶层，管理平民阶层的，维持平民生活秩序的，拥有超级权限，能访问整个空间和所有资源，人数不多。这个阶层老祖宗是g_kernelInitProcess(2号进程)。

```
g_kernelInitProcess = 2; /* 2: The root process ID of the kernel-mode process is fixed at 2 *///内核态的根进程
//获取内核态进程的根进程，所有内核进程都是g_processCBArray[g_kernelInitProcess] fork来的，包括g_processCBArray[g_kernelIdleProcess]进程
LITE_OS_SEC_TEXT UINT32 OsGetKernelInitProcessID(VOID)
{
   return g_kernelInitProcess;
}
```

- 两位老祖宗都不是通过fork来的，而是内核强制规定进程ID号，强制写死基因创建的。
- 这两个阶层可以相互流动吗，有没有可能通过高考改变命运？ 答案是: 绝对冇可能!!! 龙生龙，凤生凤，老鼠生儿会打洞。从老祖宗创建的那一刻起就被刻在基因里了，抹不掉了。 因为后续所有的进程都是由这两位老同志克隆(clone)来的，没得商量的继承这份基因。 `LosProcessCB` 有专门的标签来 `processMode` 区分这两个阶层。整个鸿蒙内核源码并没有提供改变命运机会的 `set` 函数。

```
#define OS_KERNEL_MODE 0x0U //内核态
#define OS_USER_MODE   0x1U //用户态
STATIC INLINE BOOL OsProcessIsUserMode(const LosProcessCB *processCB)///用户模式进程
{
   return (processCB->processMode == OS_USER_MODE);
}
typedef struct ProcessCB {
   // ...
   UINT16            processMode;              /**< Kernel Mode:0; User Mode:1; */ //0位内核态，1为用户态进程
} LosProcessCB;
```

2号进程 KProcess

2号进程为内核态的老祖宗，是内核创建的首个进程，源码过程如下，省略了不相干的代码。

```
bl     main  @带LR的子程序跳转， LR = pc - 4 ，执行C层main函数
/*****************************************************************************
内核入口函数，由汇编调用，见于reset_vector_up.S 和 reset_vector_mp.S
up指单核CPU， mp指多核CPU bl        main
*****************************************************************************/
LITE_OS_SEC_TEXT_INIT INT32 main(VOID)//由主CPU执行，默认0号CPU 为主CPU
{
   // ... 省略
   uwRet = OsMain();// 内核各模块初始化
}
LITE_OS_SEC_TEXT_INIT INT32 OsMain(VOID)
{
   // ...
   ret = OsKernelInitProcess();// 创建内核态根进程
   // ...
   ret = OsSystemInit(); //中间创建了用户态根进程
}
//初始化 2号进程，即内核态进程的老祖宗
LITE_OS_SEC_TEXT_INIT UINT32 OsKernelInitProcess(VOID)
{
   LosProcessCB *processCB = NULL;
   UINT32 ret;
```

```
    ret = OsProcessInit();// 初始化进程模块全部变量，创建各循环双向链表
    if (ret != LOS_OK) {
        return ret;
    }

    processCB = OS_PCB_FROM_PID(g_kernelInitProcess);// 以PID方式得到一个进程
    ret = OsProcessCreateInit(processCB， OS_KERNEL_MODE， "KProcess"， 0);// 初始化进程，最高优先级0，鸿蒙进程一共有32个优先级(0-31) 其中0-9
    if (ret != LOS_OK) {
        return ret;
    }

    processCB->processStatus &= ~OS_PROCESS_STATUS_INIT;// 进程初始化位 置1
    g_processGroup = processCB->group;//全局进程组指向了KProcess所在的进程组
    LOS_ListInit(&g_processGroup->groupList);// 进程组链表初始化
    OsCurrProcessSet(processCB);// 设置为当前进程
    return OsCreateIdleProcess();// 创建一个空闲状态的进程
}
```

解读

- main函数在系列篇中会单独讲，请留意自行翻看，它是在开机之初在SVC模式下创建的。
- 内核态老祖宗的名字叫 `KProcess`，优先级为最高 0 级， `KProcess` 进程是长期活跃的，很多重要的任务都会跑在其之下。例如:
  - `Swt_Task`
  - `oom_task`
  - `system_wq`
  - `tcpip_thread`
  - `SendToSer`
  - `SendToTelnet`
  - `eth_irq_task`
  - `TouchEventHandler`
  - `USB_GIANT_Task` 此处不细讲这些任务，在其他篇幅有介绍，但光看名字也能猜个八九，请自行翻看。
- 紧接着 `KProcess` 以 `CLONE_FILES` 的方式 fork了一个 名为 `KIdle` 的子进程(0号进程)。
- 内核态的所有进程都来自2号进程这位老同志，子子孙孙，代代相传，形成一颗家族树，和人类的传承所不同的是，它们往往是白发人送黑发人，子孙进程往往都是短命鬼，老祖宗最能活，子孙都死绝了它还在，有些收尸的工作要交给它干。

0 号进程 KIdle

0号进程是内核创建的第二个进程，在 `OsKernelInitProcess` 的末尾将 `KProcess` 设为当前进程后，紧接着就 `fork` 了0号进程。为什么一定要先设置当前进程，因为fork需要一个父进程，而此时系统处于启动阶段，并没有当前进程。 是的，您没有看错。进程是操作系统为方便管理资源而衍生出来的概念，系统并不是非要进程，任务才能运行的。 开机阶段就是啥都没有，默认跑在svc模式下，默认起始地址 `reset_vector` 都是由硬件上电后规定的。 进程，线程都是跑起来后慢慢赋予的意义。 `OsCurrProcessSet` 是从软件层面赋予了此为当前进程的这个概念。 `KProcess` 是内核设置的第一个当前进程。有了它，就可以fork， fork， fork！

```
//创建一个名叫"KIdle"的0号进程，给CPU空闲的时候使用
STATIC UINT32 OsCreateIdleProcess(VOID)
{
    UINT32 ret;
    CHAR *idleName = "Idle";
    LosProcessCB *idleProcess = NULL;
    Percpu *perCpu = OsPercpuGet();
    UINT32 *idleTaskID = &perCpu->idleTaskID;//得到CPU的idle task

    ret = OsCreateResourceFreeTask();// 创建一个资源回收任务，优先级为5 用于回收进程退出时的各种资源
    if (ret != LOS_OK) {
        return ret;
    }
 //创建一个名叫"KIdle"的进程，并创建一个idle task，CPU空闲的时候就待在 idle task中等待被唤醒
    ret = LOS_Fork(CLONE_FILES， "KIdle"， (TSK_ENTRY_FUNC)OsIdleTask， LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE);
    if (ret < 0) {//内核进程的fork并不会一次调用，返回两次，此子进程执行的开始位置是参数OsIdleTask
        return LOS_NOK;
    }
    g_kernelIdleProcess = (UINT32)ret;//返回 0号进程

    idleProcess = OS_PCB_FROM_PID(g_kernelIdleProcess);//通过ID拿到进程实体
```

```
    *idleTaskID = idleProcess->threadGroupID;//绑定CPU的IdleTask，或者说改变CPU现有的idle任务
    OS_TCB_FROM_TID(*idleTaskID)->taskStatus |= OS_TASK_FLAG_SYSTEM_TASK;//设定Idle task 为一个系统任务
#if (LOSCFG_KERNEL_SMP == YES)
    OS_TCB_FROM_TID(*idleTaskID)->cpuAffiMask = CPUID_TO_AFFI_MASK(ArchCurrCpuid());//多核CPU的任务指定，防止乱串了，注意多核才会有并行处理
#endif
    (VOID)memset_s(OS_TCB_FROM_TID(*idleTaskID)->taskName， OS_TCB_NAME_LEN， 0， OS_TCB_NAME_LEN);//task 名字先清0
    (VOID)memcpy_s(OS_TCB_FROM_TID(*idleTaskID)->taskName， OS_TCB_NAME_LEN， idleName， strlen(idleName));//task 名字叫 idle
    return LOS_OK;
}
```

解读

- 看过fork篇的可能发现了一个参数， `KIdle` 被创建的方式和通过系统调用创建的方式不一样，一个用的是`CLONE_FILES` ，一个是 `CLONE_SIGHAND` 具体的创建方式如下:

```
#define CLONE_VM       0x00000100 //子进程与父进程运行于相同的内存空间
#define CLONE_FS       0x00000200 //子进程与父进程共享相同的文件系统，包括root、当前目录、umask
#define CLONE_FILES    0x00000400 //子进程与父进程共享相同的文件描述符（file descriptor）表
#define CLONE_SIGHAND  0x00000800 //子进程与父进程共享相同的信号处理（signal handler）表
#define CLONE_PTRACE   0x00002000 //若父进程被trace，子进程也被trace
#define CLONE_VFORK    0x00004000 //父进程被挂起，直至子进程释放虚拟内存资源
#define CLONE_PARENT   0x00008000 //创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
#define CLONE_THREAD   0x00010000 //Linux 2.4中增加以支持POSIX线程标准，子进程与父进程共享相同的线程群
```

- `KIdle` 创建了一个名为`Idle`的任务，任务的入口函数为`OsIdleTask` ，这是个空闲任务，啥也不干的。专门用来给cpu休息的，cpu空闲时就待在这个任务里等活干。

```
LITE_OS_SEC_TEXT WEAK VOID OsIdleTask(VOID)
{
    while (1) {//只有一个死循环
#ifdef LOSCFG_KERNEL_TICKLESS //低功耗模式开关， idle task 中关闭tick
        if (OsTickIrqFlagGet()) {
            OsTickIrqFlagSet(0);
            OsTicklessStart();
        }
#endif
        Wfi();//WFI指令:arm core 立即进入low-power standby state，进入休眠模式，等待中断。
    }
}
```

- fork 内核态进程和fork用户态进程有个地方会不一样，就是SP寄存器的值。fork用户态的进程一次调用两次返回(父子进程各一次)，返回的位置一样(是因为拷贝了父进程陷入内核时的上下文)。所以可以通过返回值来判断是父还是子返回。这个在fork篇中有详细的描述。请自行翻看。 但fork内核态进程虽也有两次返回，但是返回的位置却不一样，子进程的返回位置是由内核指定的，例如: `Idle` 任务的入口函数为 `OsIdleTask` 。详见代码:

```
//任务初始化时拷贝任务信息
 STATIC VOID OsInitCopyTaskParam(LosProcessCB *childProcessCB， const CHAR *name， UINTPTR entry， UINT32 size，
                                 TSK_INIT_PARAM_S *childPara)
 {
    LosTaskCB *mainThread = NULL;
    UINT32 intSave;

    SCHEDULER_LOCK(intSave);
    mainThread = OsCurrTaskGet();//获取当前task，注意变量名从这里也可以看出 thread 和 task 是一个概念，只是内核常说task，上层应用说thread ，

    if (OsProcessIsUserMode(childProcessCB)) {//用户态进程
        childPara->pfnTaskEntry = mainThread->taskEntry;//拷贝当前任务入口地址
        childPara->uwStackSize = mainThread->stackSize; //栈空间大小
        childPara->userParam.userArea = mainThread->userArea;  //用户态栈区栈顶位置
        childPara->userParam.userMapBase = mainThread->userMapBase; //用户态栈底
        childPara->userParam.userMapSize = mainThread->userMapSize; //用户态栈大小
    } else {//注意内核态进程创建任务的入口由外界指定，例如 OsCreateIdleProcess 指定了OsIdleTask
        childPara->pfnTaskEntry = (TSK_ENTRY_FUNC)entry;//参数(sp)为内核态入口地址
        childPara->uwStackSize = size;//参数(size)为内核态栈大小
    }
    childPara->pcName = (CHAR *)name;     //拷贝进程名字
```

```
    childPara->policy = mainThread->policy;    //拷贝调度模式
    childPara->usTaskPrio = mainThread->priority;  //拷贝优先级
    childPara->processID = childProcessCB->processID; //拷贝进程ID
    if (mainThread->taskStatus & OS_TASK_FLAG_PTHREAD_JOIN) {
        childPara->uwResved = OS_TASK_FLAG_PTHREAD_JOIN;
    } else if (mainThread->taskStatus & OS_TASK_FLAG_DETACHED) {
        childPara->uwResved = OS_TASK_FLAG_DETACHED;
    }

    SCHEDULER_UNLOCK(intSave);
}
```

- 结论是创建0号进程中的 `OsCreateIdleProcess` 调用 `LOS_Fork` 后只会有一次返回。而且返回值为0，因为 `g_freeProcess` 中0号进程还没有被分配。详见代码，注意看最后的注释:

```
//进程模块初始化，被编译放在代码段 .init 中
 LITE_OS_SEC_TEXT_INIT UINT32 OsProcessInit(VOID)
 {
    UINT32 index;
    UINT32 size;

    g_processMaxNum = LOSCFG_BASE_CORE_PROCESS_LIMIT;//默认支持64个进程
    size = g_processMaxNum * sizeof(LosProcessCB);//算出总大小

    g_processCBArray = (LosProcessCB *)LOS_MemAlloc(m_aucSysMem1， size);// 进程池，占用内核堆，内存池分配
    if (g_processCBArray == NULL) {
        return LOS_NOK;
    }
    (VOID)memset_s(g_processCBArray， size， 0， size);//安全方式重置清0

    LOS_ListInit(&g_freeProcess);//进程空闲链表初始化，创建一个进程时从g_freeProcess中申请一个进程描述符使用
    LOS_ListInit(&g_processRecyleList);//进程回收链表初始化，回收完成后进入g_freeProcess等待再次被申请使用

    for (index = 0; index < g_processMaxNum; index++) {//进程池循环创建
        g_processCBArray[index].processID = index;//进程ID[0-g_processMaxNum-1]赋值
        g_processCBArray[index].processStatus = OS_PROCESS_FLAG_UNUSED;// 默认都是白纸一张，贴上未使用标签
        LOS_ListTailInsert(&g_freeProcess， &g_processCBArray[index].pendList);//注意g_freeProcess挂的是pendList节点，所以使用要通过OS_PCB
    }

    g_userInitProcess = 1; /* 1: The root process ID of the user-mode process is fixed at 1 *///用户态的根进程
    LOS_ListDelete(&g_processCBArray[g_userInitProcess].pendList);// 将1号进程从空闲链表上摘出去

    g_kernelInitProcess = 2; /* 2: The root process ID of the kernel-mode process is fixed at 2 *///内核态的根进程
    LOS_ListDelete(&g_processCBArray[g_kernelInitProcess].pendList);// 将2号进程从空闲链表上摘出去

    //注意:这波骚操作之后，g_freeProcess链表上还有，0，3，4，...g_processMaxNum-1号进程。创建进程是从g_freeProcess上申请
    //即下次申请到的将是0号进程，而 OsCreateIdleProcess 将占有0号进程。

    return LOS_OK;
 }
```

1号进程 init

1号进程为用户态的老祖宗。创建过程如下， 省略了不相干的代码。

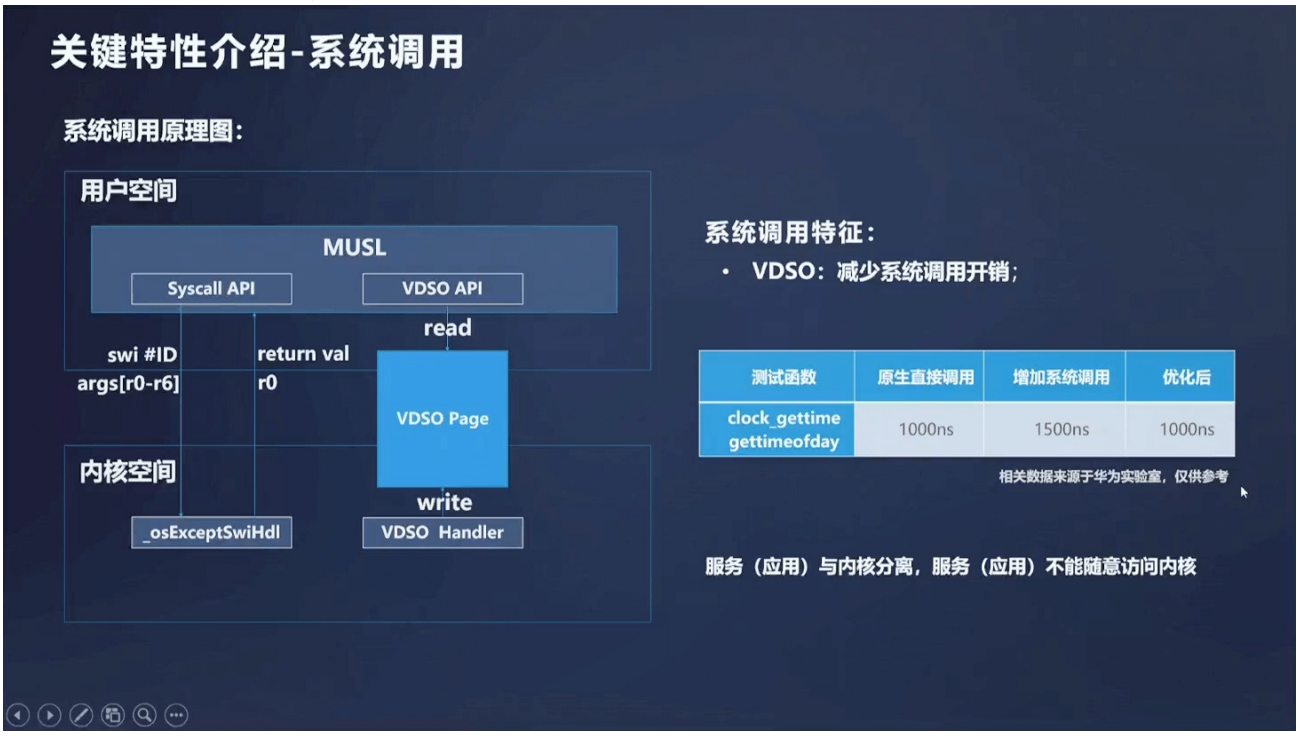
```
LITE_OS_SEC_TEXT_INIT INT32 OsMain(VOID)
{
   // ...
   ret = OsKernelInitProcess();// 创建内核态根进程
   // ...
   ret = OsSystemInit(); //中间创建了用户态根进程
}
UINT32 OsSystemInit(VOID)
{
   //..
   ret = OsSystemInitTaskCreate();//创建了一个系统任务，
}
```

```
STATIC UINT32 OsSystemInitTaskCreate(VOID)
{
    UINT32 taskID;
    TSK_INIT_PARAM_S sysTask;

    (VOID)memset_s(&sysTask， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));
    sysTask.pfnTaskEntry = (TSK_ENTRY_FUNC)SystemInit;//任务的入口函数，这个函数实现由外部提供
    sysTask.uwStackSize = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;//16K
    sysTask.pcName = "SystemInit";//任务的名称
    sysTask.usTaskPrio = LOSCFG_BASE_CORE_TSK_DEFAULT_PRIO;// 内核默认优先级为10
    sysTask.uwResved = LOS_TASK_STATUS_DETACHED;//任务分离模式
#if (LOSCFG_KERNEL_SMP == YES)
    sysTask.usCpuAffiMask = CPUID_TO_AFFI_MASK(ArchCurrCpuid());//cpu 亲和性设置，记录执行过任务的CPU，尽量确保由同一个CPU完成任务周期
#endif
    return LOS_TaskCreate(&taskID， &sysTask);//创建任务并加入就绪队列，并立即参与调度
}
//SystemInit的实现由由外部提供 比如..\vendor\hi3516dv300\module_init\src\system_init.c
void SystemInit(void)
{
    // ...
    if (OsUserInitProcess()) {//创建用户态进程的老祖宗
        PRINT_ERR("Create user init process faialed!\n");
        return;
    }
}
//用户态根进程的创建过程
LITE_OS_SEC_TEXT_INIT UINT32 OsUserInitProcess(VOID)
{
    INT32 ret;
    UINT32 size;
    TSK_INIT_PARAM_S param = { 0 };
    VOID *stack = NULL;
    VOID *userText = NULL;
    CHAR *userInitTextStart = (CHAR *)&__user_init_entry;//代码区开始位置 ，对应 LITE_USER_SEC_ENTRY
    CHAR *userInitBssStart = (CHAR *)&__user_init_bss;// 未初始化数据区（BSS）。在运行时改变其值 对应 LITE_USER_SEC_BSS
    CHAR *userInitEnd = (CHAR *)&__user_init_end;// 结束地址
    UINT32 initBssSize = userInitEnd - userInitBssStart;
    UINT32 initSize = userInitEnd - userInitTextStart;

    LosProcessCB *processCB = OS_PCB_FROM_PID(g_userInitProcess);//"Init进程的优先级是 28"
    ret = OsProcessCreateInit(processCB， OS_USER_MODE， "Init"， OS_PROCESS_USERINIT_PRIORITY);// 初始化用户进程，它将是所有应用程序的父进
    if (ret != LOS_OK) {
        return ret;
    }

    userText = LOS_PhysPagesAllocContiguous(initSize >> PAGE_SHIFT);// 分配连续的物理页
    if (userText == NULL) {
        ret = LOS_NOK;
        goto ERROR;
    }

    (VOID)memcpy_s(userText， initSize， (VOID *)&__user_init_load_addr， initSize);// 安全copy 经加载器load的结果 __user_init_load_addr -> userTe
    ret = LOS_VaddrToPaddrMmap(processCB->vmSpace， (VADDR_T)(UINTPTR)userInitTextStart， LOS_PaddrQuery(userText)，
                               initSize， VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_WRITE |
                               VM_MAP_REGION_FLAG_PERM_EXECUTE | VM_MAP_REGION_FLAG_PERM_USER);// 虚拟地址与物理地址的映射
    if (ret < 0) {
        goto ERROR;
    }

    (VOID)memset_s((VOID *)((UINTPTR)userText + userInitBssStart - userInitTextStart)， initBssSize， 0， initBssSize);// 除了代码段，其余都清0

    stack = OsUserInitStackAlloc(g_userInitProcess， &size);//分配任务在用户态下的运行栈，大小为1M
    if (stack == NULL) {
        PRINTK("user init process malloc user stack failed!\n");
        ret = LOS_NOK;
        goto ERROR;
    }

    param.pfnTaskEntry = (TSK_ENTRY_FUNC)userInitTextStart;// 从代码区开始执行，也就是应用程序main 函数的位置
    param.userParam.userSP = (UINTPTR)stack + size;// 用户态栈底
    param.userParam.userMapBase = (UINTPTR)stack;// 用户态栈顶
```

```
    param.userParam.userMapSize = size;// 用户态栈大小
    param.uwResved = OS_TASK_FLAG_PTHREAD_JOIN;// 可结合的（joinable）能够被其他线程收回其资源和杀死
    ret = OsUserInitProcessStart(g_userInitProcess， &param);// 创建一个任务，来运行main函数
    if (ret != LOS_OK) {
        (VOID)OsUnMMap(processCB->vmSpace， param.userParam.userMapBase， param.userParam.userMapSize);
        goto ERROR;
    }

    return LOS_OK;

ERROR:
    (VOID)LOS_PhysPagesFreeContiguous(userText， initSize >> PAGE_SHIFT);//释放物理内存块
    OsDeInitPCB(processCB);//删除PCB块
    return ret;
}
```

解读

- 从代码中可以看出用户态的老祖宗创建过程有点意思，首先它的源头和内核态老祖宗一样都在`OsMain`。
- 通过创建一个分离模式，优先级为10的系统任务`SystemInit`，来完成。任务的入口函数`SystemInit()`的实现由平台集成商来指定。 本篇采用了`hi3516dv300`的实现。也就是说用户态祖宗的创建是在`sysTask。uwStackSize = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;//16K`栈中完成的。这个任务归属于内核进程`KProcess`。
- 用户态老祖宗的名字叫`Init`，优先级为28级。
- 用户态的每个进程有独立的虚拟进程空间`vmSpace`，拥有独立的内存映射表(L1，L2表)，申请的内存需要重新映射，映射过程在内存系列篇中有详细的说明。
- `init`创建了一个任务，任务的入口地址为`__user_init_entry`，由编译器指定。
- 用户态进程是指应有程序运行的进程，通过动态加载ELF文件的方式启动。具体加载流程系列篇有讲解，不细说。用户态进程运行在用户空间，但通过系统调用可陷入内核空间。具体看这张图:



百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

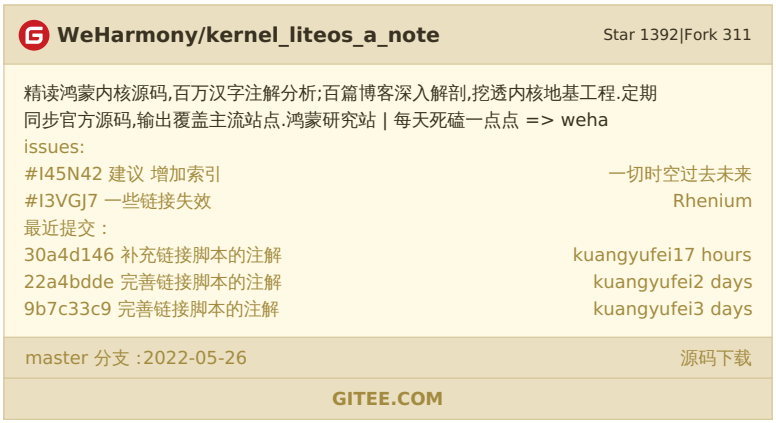
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

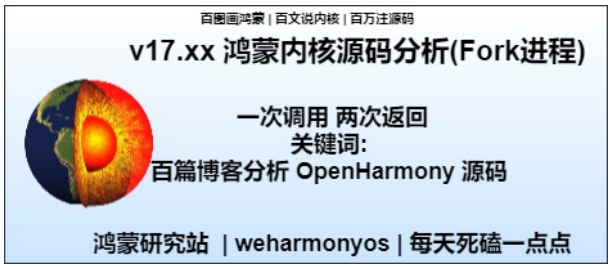


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

17_Fork进程篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

笔者第一次看到fork时，说是一次调用，两次返回，当时就懵圈了，多新鲜，真的很难理解。因为这足以颠覆了以往对函数的认知， 函数调用还能这么玩，父进程调用一次，父子进程各返回一次。而且只能通过返回值来判断是哪个进程的返回。所以一直有几个问题缠绕在脑海中。

- fork是什么？ 外部如何正确使用它。
- 为什么要用fork这种设计？ fork的本质和好处是什么？
- 怎么做到的？ 调用fork()使得父子进程各返回一次，怎么做到返回两次的，其中到底发生了什么？
- 为什么 `pid = 0` 代表了是子进程的返回？ 为什么父进程不需要返回 0 ？

直到看了linux内核源码后才搞明白，但系列篇的定位是挖透鸿蒙的内核源码，所以本篇将深入fork函数，用鸿蒙内核源码去说明白这些问题。在看本篇之前建议要先看系列篇的其他篇幅。如(任务切换篇，寄存器篇，工作模式篇，系统调用篇 等)，有了这些基础，会很好理解fork的实现过程。

fork是什么

先看一个网上经常拿来说fork的一个代码片段。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 pid_t pid;
 char *message;
 int n;
 pid = fork();
 if (pid < 0) {
  perror("fork failed");
  exit(1);
 }
 if (pid == 0) {
  message = "This is the child\n";
  n = 6;
 } else {
  message = "This is the parent\n";
  n = 3;
```

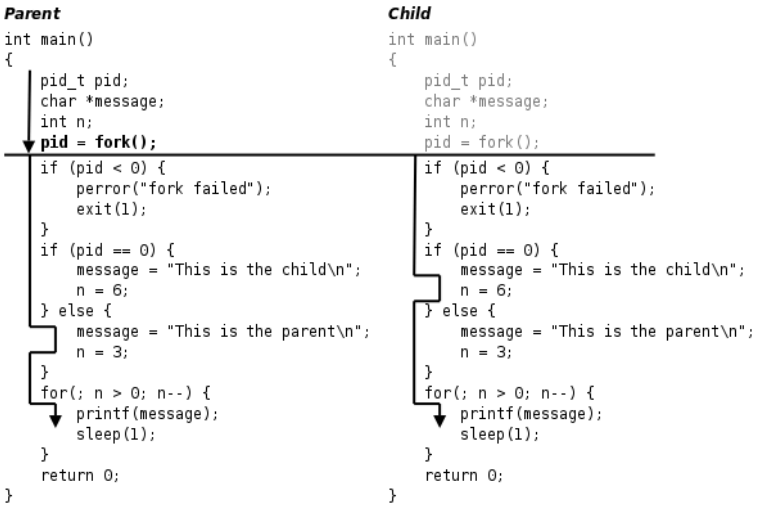
```
 }
 for(; n > 0; n--) {
  printf(message);
  sleep(1);
 }
 return 0;
}
```

- `pid < 0` fork 失败
- `pid == 0` fork成功，是子进程的返回
- `pid > 0` fork成功，是父进程的返回
- `fork` 的返回值这样规定是有道理的。`fork` 在子进程中返回0，子进程仍可以调用 `getpid` 函数得到自己的进程id，也可以调用 `getppid` 函数得到父进程的id。在父进程中用 `getpid` 可以得到自己的进程id，然而要想得到子进程的id，只有将 `fork` 的返回值记录下来，别无它法。
- 子进程并没有真正执行 `fork()`，而是内核用了一个很巧妙的方法获得了返回值，并且将返回值硬生生的改写成了0，这是笔者认为 `fork` 的实现最精彩的部分。

运行结果

```
$ ./a.out
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

这个程序的运行过程如下图所示。



解读

- `fork()` 是一个系统调用，因此会切换到SVC模式运行。在SVC栈中父进程复制出一个子进程，父进程和子进程的PCB信息相同，用户态代码和数据也相同。
- 从案例的执行上可以看出，fork 之后的代码父子进程都会执行，即代码段指向(PC寄存器)是一样的。实际上fork只被父进程调用了一次，子进程并没有执行 `fork` 函数，但是却获得了一个返回值，`pid == 0`，这个非常重要。这是本篇说明的重点。
- 从执行结果上看，父进程打印了三次(This is the parent)，因为 n = 3。 子进程打印了六次(This is the child)，因为 n = 6。 而子程序并没有执行以下代码:

```
pid_t pid;
char *message;
int n;
```

子进程是从 `pid = fork()` 后开始执行的，按理它不会在新任务栈中出现这些变量，而实际上后面又能顺利的使用这些变量，说明父进程当前任务

的用户态的数据也复制了一份给子进程的新任务栈中。

- 被fork成功的子进程跑的首条代码指令是 `pid = 0` ，这里的0是返回值，存放在 `R0` 寄存器中。说明父进程的任务上下文也进行了一次拷贝，父进程从内核态回到用户态时恢复的上下文和子进程的任务上下文是一样的，即 PC寄存器指向是一样的，如此才能确保在代码段相同的位置执行。
- 执行 `./a.out` 后 第一条打印的是 `This is the child` 说明 `fork()` 中发生了一次调度，CPU切到了子进程的任务执行， `sleep(1)` 的本质在系列篇中多次说过是任务主动放弃CPU的使用权，将自己挂入任务等待链表，由此发生一次任务调度，CPU切到父进程执行，才有了打印第二条的 `This is the parent` ，父进程的 `sleep(1)` 又切到子进程如此往返，直到 n = 0， 结束父子进程。
- 但这个例子和笔者的解读只解释了fork是什么的使用说明书，并猜测其中做了些什么，并没有说明为什么要这样做和代码是怎么实现的。 正式结合鸿蒙的源码说清楚为什么和怎么做这两个问题？

为什么是fork

fork函数的特点概括起来就是“调用一次，返回两次”，在父进程中调用一次，在父进程和子进程中各返回一次。从上图可以看出，一开始是一个控制流程，调用fork之后发生了分叉，变成两个控制流程，这也就是“fork”（分叉）这个名字的由来了。 系列篇已经写了40+多篇，已经很容易理解一个程序运行起来就需要各种资源(内存，文件，ipc，监控信息等等)，资源就需要管理，进程就是管理资源的容器。这些资源相当于干活需要各种工具一样，干活的工具都差不多，实在没必再走流程一一申请，而且申请下来会发现和别人手里已有的工具都一样， 别人有直接拿过来使用它不香吗？ 所以最简单的办法就是认个干爹，让干爹拷贝一份干活工具给你。这样只需要专心的干好活(任务)就行了。 fork的本质就是copy，具体看代码。

fork怎么实现的？

```
//系统调用之fork ，建议去 https://gitee.com/weharmony/kernel_liteos_a_note fork 一下？ :P
int SysFork(void)
{
    return OsClone(CLONE_SIGHAND， 0， 0);//本质就是克隆
}
LITE_OS_SEC_TEXT INT32 OsClone(UINT32 flags， UINTPTR sp， UINT32 size)
{
    UINT32 cloneFlag = CLONE_PARENT | CLONE_THREAD | CLONE_VFORK | CLONE_VM;

    if (flags & (~cloneFlag)) {
        PRINT_WARN("Clone dont support some flags!\n");
    }

    return OsCopyProcess(cloneFlag & flags， NULL， sp， size);
}
STATIC INT32 OsCopyProcess(UINT32 flags， const CHAR *name， UINTPTR sp， UINT32 size)
{
    UINT32 intSave， ret， processID;
    LosProcessCB *run = OsCurrProcessGet();//获取当前进程

    LosProcessCB *child = OsGetFreePCB();//从进程池中申请一个进程控制块，鸿蒙进程池默认64
    if (child == NULL) {
        return -LOS_EAGAIN;
    }
    processID = child->processID;

    ret = OsForkInitPCB(flags， child， name， sp， size);//初始化进程控制块
    if (ret != LOS_OK) {
        goto ERROR_INIT;
    }

    ret = OsCopyProcessResources(flags， child， run);//拷贝进程的资源，包括虚拟空间，文件，安全，IPC ==
    if (ret != LOS_OK) {
        goto ERROR_TASK;
    }

    ret = OsChildSetProcessGroupAndSched(child， run);//设置进程组和加入进程调度就绪队列
    if (ret != LOS_OK) {
        goto ERROR_TASK;
    }

    LOS_MpSchedule(OS_MP_CPU_ALL);//给各CPU发送准备接受调度信号
    if (OS_SCHEDULER_ACTIVE) {//当前CPU core处于活动状态
        LOS_Schedule();// 申请调度
    }
```

```
    return processID;

ERROR_TASK:
    SCHEDULER_LOCK(intSave);
    (VOID)OsTaskDeleteUnsafe(OS_TCB_FROM_TID(child->threadGroupID)， OS_PRO_EXIT_OK， intSave);
ERROR_INIT:
    OsDeInitPCB(child);
    return -ret;
}
### OsForkInitPCB
STATIC UINT32 (UINT32 flags， LosProcessCB *child， const CHAR *name， UINTPTR sp， UINT32 size)
{
    UINT32 ret;
    LosProcessCB *run = OsCurrProcessGet();//获取当前进程

    ret = OsInitPCB(child， run->processMode， OS_PROCESS_PRIORITY_LOWEST， LOS_SCHED_RR， name);//初始化PCB信息，进程模式，优先级，调
    if (ret != LOS_OK) {
        return ret;
    }

    ret = OsCopyParent(flags， child， run);//拷贝父亲大人的基因信息
    if (ret != LOS_OK) {
        return ret;
    }

    return OsCopyTask(flags， child， name， sp， size);//拷贝任务，设置任务入口函数，栈大小
}
//初始化PCB块
STATIC UINT32 OsInitPCB(LosProcessCB *processCB， UINT32 mode， UINT16 priority， UINT16 policy， const CHAR *name)
{
    UINT32 count;
    LosVmSpace *space = NULL;
    LosVmPage *vmPage = NULL;
    status_t status;
    BOOL retVal = FALSE;

    processCB->processMode = mode;      //用户态进程还是内核态进程
    processCB->processStatus = OS_PROCESS_STATUS_INIT; //进程初始状态
    processCB->parentProcessID = OS_INVALID_VALUE;  //爸爸进程，外面指定
    processCB->threadGroupID = OS_INVALID_VALUE;  //所属线程组
    processCB->priority = priority;      //进程优先级
    processCB->policy = policy;       //调度算法 LOS_SCHED_RR
    processCB->umask = OS_PROCESS_DEFAULT_UMASK;  //掩码
    processCB->timerID = (timer_t)(UINTPTR)MAX_INVALID_TIMER_VID;

    LOS_ListInit(&processCB->threadSiblingList);//初始化孩子任务/线程链表，上面挂的都是由此fork的孩子线程 见于 OsTaskCBInit LOS_ListTailInsert(&(pro
    LOS_ListInit(&processCB->childrenList);  //初始化孩子进程链表，上面挂的都是由此fork的孩子进程 见于 OsCopyParent LOS_ListTailInsert(&parentProc
    LOS_ListInit(&processCB->exitChildList); //初始化记录退出孩子进程链表，上面挂的是哪些exit 见于 OsProcessNaturalExit LOS_ListTailInsert(&parentC
    LOS_ListInit(&(processCB->waitList));  //初始化等待任务链表 上面挂的是处于等待的 见于 OsWaitInsertWaitLIstInOrder LOS_ListHeadInsert(&processC

    for (count = 0; count < OS_PRIORITY_QUEUE_NUM; ++count) { //根据 priority数 创建对应个数的队列
        LOS_ListInit(&processCB->threadPriQueueList[count]); //初始化一个个线程队列，队列中存放就绪状态的线程/task
    }//在鸿蒙内核中 task就是thread，在鸿蒙源码分析系列篇中有详细阐释 见于 https://my.oschina.net/u/3751245

    if (OsProcessIsUserMode(processCB)) {// 是否为用户模式进程
        space = LOS_MemAlloc(m_aucSysMem0， sizeof(LosVmSpace));//分配一个虚拟空间
        if (space == NULL) {
            PRINT_ERR("%s %d， alloc space failed\n"， __FUNCTION__， __LINE__);
            return LOS_ENOMEM;
        }
        VADDR_T *ttb = LOS_PhysPagesAllocContiguous(1);//分配一个物理页用于存储L1页表 4G虚拟内存分成 （4096*1M）
        if (ttb == NULL) {//这里直接获取物理页ttb
            PRINT_ERR("%s %d， alloc ttb or space failed\n"， __FUNCTION__， __LINE__);
            (VOID)LOS_MemFree(m_aucSysMem0， space);
            return LOS_ENOMEM;
        }
        (VOID)memset_s(ttb， PAGE_SIZE， 0， PAGE_SIZE);//内存清0
        retVal = OsUserVmSpaceInit(space， ttb);//初始化虚拟空间和进程mmu
        vmPage = OsVmVaddrToPage(ttb);//通过虚拟地址拿到page
        if ((retVal == FALSE) || (vmPage == NULL)) {//异常处理
            PRINT_ERR("create space failed! ret: %d， vmPage: %#x\n"， retVal， vmPage);
```

```
            processCB->processStatus = OS_PROCESS_FLAG_UNUSED;//进程未使用，干净
            (VOID)LOS_MemFree(m_aucSysMem0， space);//释放虚拟空间
            LOS_PhysPagesFreeContiguous(ttb， 1);//释放物理页，4K
            return LOS_EAGAIN;
        }
        processCB->vmSpace = space;//设为进程虚拟空间
        LOS_ListAdd(&processCB->vmSpace->archMmu.ptList， &(vmPage->node));//将空间映射页表挂在 空间的mmu L1页表， L1为表头
    } else {
        processCB->vmSpace = LOS_GetKVmSpace();//内核共用一个虚拟空间，内核进程 常驻内存
    }

#ifdef LOSCFG_SECURITY_VID
    status = VidMapListInit(processCB);
    if (status != LOS_OK) {
        PRINT_ERR("VidMapListInit failed!\n");
        return LOS_ENOMEM;
    }
#endif
#ifdef LOSCFG_SECURITY_CAPABILITY
    OsInitCapability(processCB);
#endif

    if (OsSetProcessName(processCB， name) != LOS_OK) {
        return LOS_ENOMEM;
    }

    return LOS_OK;
}
```

```
//拷贝一个Task过程
STATIC UINT32 OsCopyTask(UINT32 flags， LosProcessCB *childProcessCB， const CHAR *name， UINTPTR entry， UINT32 size)
{
    LosTaskCB *childTaskCB = NULL;
    TSK_INIT_PARAM_S childPara = { 0 };
    UINT32 ret;
    UINT32 intSave;
    UINT32 taskID;

    OsInitCopyTaskParam(childProcessCB， name， entry， size， &childPara);//初始化Task参数

    ret = LOS_TaskCreateOnly(&taskID， &childPara);//只创建任务，不调度
    if (ret != LOS_OK) {
        if (ret == LOS_ERRNO_TSK_TCB_UNAVAILABLE) {
            return LOS_EAGAIN;
        }
        return LOS_ENOMEM;
    }

    childTaskCB = OS_TCB_FROM_TID(taskID);//通过taskId获取task实体
    childTaskCB->taskStatus = OsCurrTaskGet()->taskStatus;//任务状态先同步，注意这里是赋值操作. ...01101001
    if (childTaskCB->taskStatus & OS_TASK_STATUS_RUNNING) {//因只能有一个运行的task，所以如果一样要改4号位
        childTaskCB->taskStatus &= ~OS_TASK_STATUS_RUNNING;//将四号位清0 ，变成 ...01100001
    } else {//非运行状态下会发生什么？
        if (OS_SCHEDULER_ACTIVE) {//克隆线程发生错误未运行
            LOS_Panic("Clone thread status not running error status: 0x%x\n"， childTaskCB->taskStatus);
        }
        childTaskCB->taskStatus &= ~OS_TASK_STATUS_UNUSED;//干净的Task
        childProcessCB->priority = OS_PROCESS_PRIORITY_LOWEST;//进程设为最低优先级
    }

    if (OsProcessIsUserMode(childProcessCB)) {//是否是用户进程
        SCHEDULER_LOCK(intSave);
        OsUserCloneParentStack(childTaskCB， OsCurrTaskGet());//拷贝当前任务上下文给新的任务
        SCHEDULER_UNLOCK(intSave);
    }
    OS_TASK_PRI_QUEUE_ENQUEUE(childProcessCB， childTaskCB);//将task加入子进程的就绪队列
    childTaskCB->taskStatus |= OS_TASK_STATUS_READY;//任务状态贴上就绪标签
    return LOS_OK;
}
```

```
//把父任务上下文克隆给子任务
LITE_OS_SEC_TEXT VOID OsUserCloneParentStack(LosTaskCB *childTaskCB， LosTaskCB *parentTaskCB)
{
   TaskContext *context = (TaskContext *)childTaskCB->stackPointer;
   VOID *cloneStack = (VOID *)(((UINTPTR)parentTaskCB->topOfStack + parentTaskCB->stackSize) - sizeof(TaskContext));
 //cloneStack指向 TaskContext
   LOS_ASSERT(parentTaskCB->taskStatus & OS_TASK_STATUS_RUNNING);//当前任务一定是正在运行的task

   (VOID)memcpy_s(childTaskCB->stackPointer， sizeof(TaskContext)， cloneStack， sizeof(TaskContext));//直接把任务上下文拷贝了一份
   context->R[0] = 0;//R0寄存器为0，这个很重要， pid = fork()  pid == 0 是子进程返回。
}
```

解读

- 系统调用是通过 `CLONE_SIGHAND` 的方式创建子进程的。具体有哪些创建方式如下:

  ```
  #define CLONE_VM      0x00000100 //子进程与父进程运行于相同的内存空间
  #define CLONE_FS      0x00000200 //子进程与父进程共享相同的文件系统，包括root、当前目录、umask
  #define CLONE_FILES   0x00000400 //子进程与父进程共享相同的文件描述符（file descriptor）表
  #define CLONE_SIGHAND  0x00000800 //子进程与父进程共享相同的信号处理（signal handler）表
  #define CLONE_PTRACE   0x00002000 //若父进程被trace，子进程也被trace
  #define CLONE_VFORK    0x00004000 //父进程被挂起，直至子进程释放虚拟内存资源
  #define CLONE_PARENT   0x00008000 //创建的子进程的父进程是调用者的父进程，新进程与创建它的进程成了“兄弟”而不是“父子”
  #define CLONE_THREAD   0x00010000 //Linux 2.4中增加以支持POSIX线程标准，子进程与父进程共享相同的线程群
  ```

  此处不展开细说，进程之间发送信号用于异步通讯，系列篇有专门的篇幅说信号(signal)，请自行翻看。

- 可以看出fork的主体函数是 `OsCopyProcess` ，先申请一个干净的PCB，相当于申请一个容器装资源。

- 初始化这个容器 `OsForkInitPCB` ， `OsInitPCB` 先把容器打扫干净，虚拟空间，地址映射表(L1表)，各种链表初始化好，为接下来的内容拷贝做好准备。

- `OsCopyParent` 把家族基因/关系传递给子进程，谁是你的老祖宗，你的七大姑八大姨是谁都得告诉你知道，这些都将挂到你已经初始化好的链表上。

- `OsCopyTask` 这个很重要，拷贝父进程当前执行的任务数据给子进程的新任务，系列篇中已经说过，真正让CPU干活的是任务(线程)，所以子进程需要创建一个新任务 `LOS_TaskCreateOnly` 来接受当前任务的数据，这个数据包括栈的数据，运行代码段指向， `OsUserCloneParentStack` 将用户态的上下文数据 TaskContext 拷贝到子进程新任务的栈底位置， 也就是说新任务运行栈中此时只有上下文的数据。而且有最最最重要的一句代码 `context->R[0] = 0;` 强制性的将未来恢复上下文 `R0` 寄存器的数据改成了0， 这意味着调度算法切到子进程的任务后， 任务干的第一件事是恢复上下文，届时 `R0` 寄存器的值变成0，而 `R0=0` 意味着什么？ 同时 `LR/SP` 寄存器的值也和父进程的一样。这又意味着什么？

- 系列篇寄存器篇中以说过返回值就是存在R0寄存器中， `A()->B()` ，A拿B的返回值只认 `R0` 的数据，读到什么就是什么返回值，而R0寄存器值等于0，等同于获得返回值为0， 而LR寄存器所指向的指令是 `pid=返回值` ， sp寄存器记录了栈中的开始计算的位置，如此完全还原了父进程调用 `fork()` 前的运行场景，唯一的区别是改变了 `R0` 寄存器的值，所以才有了

  ```
  pid = 0;//fork()的返回值，注意子进程并没有执行fork()，它只是通过恢复上下文获得了一个返回值。
  if (pid == 0) {
    message = "This is the child\n";
    n = 6;
   }
  ```

  由此确保了这是子进程的返回。这是 `fork()` 最精彩的部分。一定要好好理解。 `OsCopyTask``OsUserCloneParentStack` 的代码细节。会让你醍醐灌顶，永生难忘。

- 父进程的返回是 `processID = child->processID;` 是子进程的ID，任何子进程的ID是不可能等于0的，成功了只能是大于0。 失败了就是负数 `return -ret;`

- `OsCopyProcessResources` 用于赋值各种资源，包括拷贝虚拟空间内存，拷贝打开的文件列表，IPC等等。

- `OsChildSetProcessGroupAndSched` 设置子进程组和调度的准备工作，加入调度队列，准备调度。

- `LOS_MpSchedule` 是个核间中断，给所有CPU发送调度信号，让所有CPU发生一次调度。由此父进程让出CPU使用权，因为子进程的调度优先级和父进程是平级，而同级情况下子进程的任务已经插到就绪队列的头部位置 `OS_PROCESS_PRI_QUEUE_ENQUEUE` 排在了父进程任务的前面，所以在没有比他们更高优先级的进程和任务出现之前，下一次被调度到的任务就是子进程的任务。也就是在本篇开头看到的

  ```
  $ ./a.out
  This is the child
  ```

```
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

- 以上为fork在鸿蒙内核的整个实现过程，务必结合系列篇其他篇理解，一次理解透彻，终生不忘。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

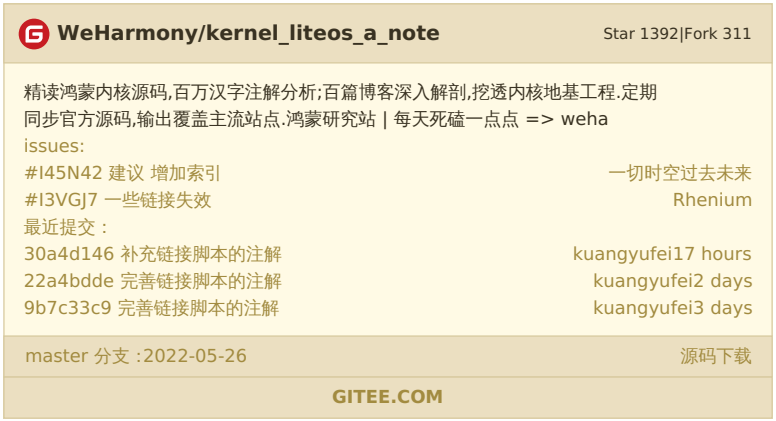
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

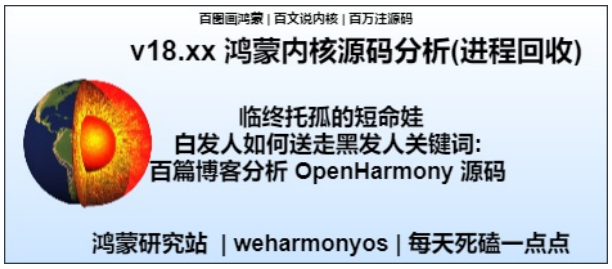


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

18_进程回收篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

进程关系链

进程是家族式管理的，父子关系，兄弟关系，朋友关系，子女关系，甚至陌生人关系(等待你消亡)在一个进程的生命周期中都会记录下来。用什么来记录呢？当然是内核最重要的胶水结构体`LOS_DL_LIST`，进程控制块(以下简称`PCB`)用了8个双向链表来记录进程家族的基因关系和运行时关系。如下:

```
typedef struct ProcessCB {
    //...此处省略其他变量
    LOS_DL_LIST         pendList;                 /**< Block list to which the process belongs */ //进程所属的阻塞列表，如果因拿锁失败，就由此节点挂到等锁
    LOS_DL_LIST         childrenList;             /**< my children process list */ //孩子进程都挂到这里，形成双循环链表
    LOS_DL_LIST         exitChildList;            /**< my exit children process list */ //那些要退出孩子进程挂到这里，白发人送黑发人。
    LOS_DL_LIST         siblingList;              /**< linkage in my parent's children list */ //兄弟进程链表， 56个民族是一家，来自同一个父进程。
    LOS_DL_LIST         subordinateGroupList;     /**< linkage in my group list */ //进程是组长时，有哪些组员进程
    LOS_DL_LIST         threadSiblingList;        /**< List of threads under this process *///进程的线程(任务)列表
    LOS_DL_LIST         threadPriQueueList[OS_PRIORITY_QUEUE_NUM]; /**< The process's thread group schedules thepriority hash table */ //进程的
    LOS_DL_LIST         waitList;     /**< The process holds the waitLits to support wait/waitpid *///进程持有等待链表以支持wait/waitpid
} LosProcessCB;
```

解读

- `pendList` 个人认为它是鸿蒙内核功能最多的一个链表，它远不止字面意思阻塞链表这么简单，只有深入解读源码后才能体会它真的是太会来事了，一般把它理解为阻塞链表就行。上面挂的是处于阻塞状态的进程。
- `childrenList` 孩子链表，所有由它fork出来的进程都挂到这个链表上。上面的孩子进程在死亡前会将自己从上面摘出去，转而挂到`exitChildList`链表上。
- `exitChildList` 退出孩子链表，进入死亡程序的进程要挂到这个链表上，一个进程的死亡是件挺麻烦的事，进程池的数量有限，需要及时回收进程资源，但家族管理关系复杂，要去很多地方消除痕迹。尤其还有其他进程在看你笑话，等你死亡(`wait`/`waitpid`)了通知它们一声。
- `siblingList` 兄弟链表，和你同一个父亲的进程都挂到了这个链表上。
- `subordinateGroupList` 朋友圈链表，里面是因为兴趣爱好(进程组)而挂在一起的进程，它们可以不是一个父亲，不是一个祖父，但一定是同一个老祖宗(用户态和内核态根进程)。
- `threadSiblingList` 线程链表，上面挂的是进程ID都是这个进程的线程(任务)，进程和线程的关系是1:N的关系，一个线程只能属于一个进程。这里要注意任务在其生命周期中是不能改所属进程的。

- threadPriQueueList 线程的调度队列数组，一共32个，任务和进程一样有32个优先级，调度算法的过程是先找到优先级最高的进程，在从该进程的任务队列里去最高的优先级任务运行。
- waitList 是等待子进程消亡的任务链表，注意上面挂的是任务。任务是通过系统调用

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid， int *status， int options);
```

将任务挂到 waitList 上。鸿蒙waitpid系统调用为 SysWait ，稍后会讲。

进程正常死亡过程

一个进程的自然消亡过程如下

```
//一个进程的自然消亡过程，参数是当前运行的任务
STATIC VOID OsProcessNaturalExit(LosTaskCB *runTask， UINT32 status)
{
    LosProcessCB *processCB = OS_PCB_FROM_PID(runTask->processID);//通过task找到所属PCB
    LosProcessCB *parentCB = NULL;
    LOS_ASSERT(!(processCB->threadScheduleMap != 0));//断言没有任务需要调度了，当前task是最后一个了
    LOS_ASSERT(processCB->processStatus & OS_PROCESS_STATUS_RUNNING);//断言必须为正在运行的进程
    OsChildProcessResourcesFree(processCB);//释放孩子进程的资源
#ifdef LOSCFG_KERNEL_CPUP
    OsCpupClean(processCB->processID);
#endif
    /* is a child process */
    if (processCB->parentProcessID != OS_INVALID_VALUE) {//判断是否有父进程
        parentCB = OS_PCB_FROM_PID(processCB->parentProcessID);//获取父进程实体
        LOS_ListDelete(&processCB->siblingList);//将自己从兄弟链表中摘除，家人们，永别了!
        if (!OsProcessExitCodeSignalIsSet(processCB)) {//是否设置了退出码？
            OsProcessExitCodeSet(processCB， status);//将进程状态设为退出码
        }
        LOS_ListTailInsert(&parentCB->exitChildList， &processCB->siblingList);//挂到父进程的孩子消亡链表，家人中，永别的可不止我一个。
        LOS_ListDelete(&processCB->subordinateGroupList);//和志同道合的朋友们永别了，注意家里可不一定是朋友的，所有各有链表。
        LOS_ListTailInsert(&processCB->group->exitProcessList， &processCB->subordinateGroupList);//挂到进程组消亡链表，朋友中，永别的可不止我一
        OsWaitCheckAndWakeParentProcess(parentCB， processCB);//检查父进程的等待任务并唤醒任务，此处将会切换到其他任务运行。
        OsDealAliveChildProcess(processCB);//老父亲临终向各自的祖宗托孤
        processCB->processStatus |= OS_PROCESS_STATUS_ZOMBIES;//贴上僵死进程的标签
        (VOID)OsKill(processCB->parentProcessID， SIGCHLD， OS_KERNEL_KILL_PERMISSION);//以内核权限发送SIGCHLD(子进程退出)信号。
        LOS_ListHeadInsert(&g_processRecyleList， &processCB->pendList);//将进程通过其阻塞节点挂入全局进程回收链表
        OsRunTaskToDelete(runTask);//删除正在运行的任务
        return;
    }
    LOS_Panic("pid : %u is the root process exit!\n"， processCB->processID);
    return;
}
```

解读

- 退群，向兄弟姐妹 siblingList 告别，向朋友圈(进程组)告别 subordinateGroupList 。
- 留下你的死亡记录，老父亲记录到 exitChildList ，朋友圈记录到 exitProcessList 中。
- 告诉后人死亡原因 OsProcessExitCodeSet ，因为 waitList 上挂的任务在等待你的死亡信息。
- 向老祖宗托孤，用户态和内核态进程都有自己的祖宗进程(1和2号进程)，老祖宗身子硬朗，最后死。所有的短命鬼进程都可以把自己的孩子委托给老祖宗照顾，老祖宗会一视同仁。
- 将自己变成了 OS_PROCESS_STATUS_ZOMBIES 僵尸进程。
- 老父亲跑到村口广播这个孩子已经死亡的信号 OsKill 。
- 将自己挂入进程回收链表，等待回收任务 ResourcesTask 回收资源。
- 最后删除这个正在运行的任务，很明显其中一定会发生一次调度 OsSchedResched 。

```
//删除一个正在运行的任务
LITE_OS_SEC_TEXT VOID OsRunTaskToDelete(LosTaskCB *taskCB)
```

```
    {
        LosProcessCB *processCB = OS_PCB_FROM_PID(taskCB->processID);//拿到task所属进程
        OsTaskReleaseHoldLock(processCB， taskCB);//task还锁
        OsTaskStatusUnusedSet(taskCB);//task重置为未使用状态，等待回收
        LOS_ListDelete(&taskCB->threadList);//从进程的线程链表中将自己摘除
        processCB->threadNumber--;//进程的活动task --，注意进程还有一个记录总task的变量 processCB->threadCount
        LOS_ListTailInsert(&g_taskRecyleList， &taskCB->pendList);//将task插入回收链表，等待回收资源再利用
        OsEventWriteUnsafe(&g_resourceEvent， OS_RESOURCE_EVENT_FREE， FALSE， NULL);//发送释放资源的事件，事件由 OsResourceRecovery
        OsSchedResched();//申请调度
        return;
    }
```

- 但这是一个自然死亡的进程，还有很多非正常死亡在其他篇幅中已有说明。请自行翻看。非正常死亡的会产生僵尸进程。这种进程需要别的进程通过 `waitpid` 来回收。

孤儿进程

一般情况下往往是白发人送黑发人，子进程的生命周期是要短于父进程。但因为fork之后，进程之间相互独立，调度算法一视同仁，父子之间是弱的关系力，就什么情况都可能发生了。内核是允许老父亲先走的，如果父进程退出而它的一个或多个子进程还在运行，那么这些子进程就被称为孤儿进程，孤儿进程最终将被两位老祖宗(用户态和内核态)所收养，并由老祖宗完成对它们的状态收集工作。

```
//当一个进程自然退出的时候，它的孩子进程由两位老祖宗收养
STATIC VOID OsDealAliveChildProcess(LosProcessCB *processCB)
{
    UINT32 parentID;
    LosProcessCB *childCB = NULL;
    LosProcessCB *parentCB = NULL;
    LOS_DL_LIST *nextList = NULL;
    LOS_DL_LIST *childHead = NULL;
    if (!LOS_ListEmpty(&processCB->childrenList)) {//如果存在孩子进程
        childHead = processCB->childrenList.pstNext;//获取孩子链表
        LOS_ListDelete(&(processCB->childrenList));//清空自己的孩子链表
        if (OsProcessIsUserMode(processCB)) {//是用户态进程
            parentID = g_userInitProcess;//用户态进程老祖宗
        } else {
            parentID = g_kernelInitProcess;//内核态进程老祖宗
        }
        for (nextList = childHead; ;) {//遍历孩子链表
            childCB = OS_PCB_FROM_SIBLIST(nextList);//找到孩子的真身
            childCB->parentProcessID = parentID;//孩子磕头认老祖宗为爸爸
            nextList = nextList->pstNext;//找下一个孩子进程
            if (nextList == childHead) {//一圈下来，孩子们都磕完头了
                break;
            }
        }
        parentCB = OS_PCB_FROM_PID(parentID);//找个老祖宗的真身
        LOS_ListTailInsertList(&parentCB->childrenList， childHead);//挂到老祖宗的孩子链表上
    }

    return;
}
```

解读

- 函数很简单，都一一注释了，老父亲临终托付后事，请各自的老祖宗照顾孩子。
- 从这里也可以看出进程的家族管理模式，两个家族从进程的出生到死亡负责到底。

僵尸进程

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的 `PCB` 还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。

如果一个进程已经终止，但是它的父进程尚未调用wait或waitpid对它进行清理，这时的进程状态称为僵尸（Zombie）进程，即 Z 进程。任何进程在刚终止时都是僵尸进程，正常情况下，僵尸进程都立刻被父进程清理了。 不正常情况下就需要手动 `waitpid` 清理了。

waitpid

在鸿蒙系统中，一个进程结束了，但是它的父进程没有等待（调用 wait waitpid ）它，那么它将变成一个僵尸进程。通过系统调用 waitpid 可以彻底的清理掉子进程。归还 pcb 。最终调用到 SysWait

```
#include <sys/wait.h>
#include "syscall.h"
pid_t waitpid(pid_t pid， int *status， int options)
{
 return syscall_cp(SYS_wait4， pid， status， options， 0);
}
```

```
//等待子进程结束
int SysWait(int pid， USER int *status， int options， void *rusage)
{
    (void)rusage;
    return LOS_Wait(pid， status， (unsigned int)options， NULL);
}
//返回已经终止的子进程的进程ID号，并清除僵死进程.
LITE_OS_SEC_TEXT INT32 LOS_Wait(INT32 pid， USER INT32 *status， UINT32 options， VOID *rusage)
{
    (VOID)rusage;
    UINT32 ret;
    UINT32 intSave;
    LosProcessCB *childCB = NULL;
    LosProcessCB *processCB = NULL;
    LosTaskCB *runTask = NULL;
    ret = OsWaitOptionsCheck(options);//参数检查，只支持LOS_WAIT_WNOHANG
    if (ret != LOS_OK) {
        return -ret;
    }
    SCHEDULER_LOCK(intSave);
    processCB = OsCurrProcessGet(); //获取当前进程
    runTask = OsCurrTaskGet();  //获取当前任务
    ret = OsWaitChildProcessCheck(processCB， pid， &childCB);//先检查下看能不能找到参数要求的退出子进程
    if (ret != LOS_OK) {
        pid = -ret;
        goto ERROR;
    }
    if (childCB != NULL) {//找到了进程
        return OsWaitRecycleChildPorcess(childCB， intSave， status);//回收进程
    }
 //没有找到，看是否要返回还是去做个登记
    if ((options & LOS_WAIT_WNOHANG) != 0) {//有LOS_WAIT_WNOHANG标签
        runTask->waitFlag = 0;//等待标识置0
        pid = 0;//这里置0，是为了 return 0
        goto ERROR;
    }
 //等待孩子进程退出
    OsWaitInsertWaitListInOrder(runTask， processCB);//将当前任务挂入进程waitList链表
 //发起调度的目的是为了让出CPU，让其他进程/任务运行
    OsSchedResched();//发起调度
    runTask->waitFlag = 0;
    if (runTask->waitID == OS_INVALID_VALUE) {
        pid = -LOS_ECHILD;//没有此子进程
        goto ERROR;
    }
    childCB = OS_PCB_FROM_PID(runTask->waitID);//获取当前任务的等待子进程ID
    if (!(childCB->processStatus & OS_PROCESS_STATUS_ZOMBIES)) {//子进程非僵死进程
        pid = -LOS_ESRCH;//没有此进程
        goto ERROR;
    }
 //回收僵死进程
    return OsWaitRecycleChildPorcess(childCB， intSave， status);
ERROR:
    SCHEDULER_UNLOCK(intSave);
    return pid;
}
```

解读

- `pid`是数据参数，根据不同的参数代表不同的含义，含义如下:

```
|参数值|说明|
|pid<-1|等待进程组号为pid绝对值的任何子进程。|
|pid=-1|等待任何子进程，此时的waitpid()函数就退化成了普通的wait()函数。|
|pid=0|等待进程组号与目前进程相同的任何子进程，也就是说任何和调用waitpid()函数的进程在同一个进程组的进程。|
|pid>0|等待进程号为pid的子进程。|
```

`pid`不同值代表的真正含义可以看这个函数`OsWaitSetFlag`。

```
//设置等待子进程退出方式方法
    STATIC UINT32 OsWaitSetFlag(const LosProcessCB *processCB， INT32 pid， LosProcessCB **child)
    {
        LosProcessCB *childCB = NULL;
        ProcessGroup *group = NULL;
        LosTaskCB *runTask = OsCurrTaskGet();
        UINT32 ret;
        if (pid > 0) {//等待进程号为pid的子进程结束
            /* Wait for the child process whose process number is pid。 */
            childCB = OsFindExitChildProcess(processCB， pid);//看能否从退出的孩子链表中找到PID
            if (childCB != NULL) {//找到了，确实有一个已经退出的PID，注意一个进程退出时会挂到父进程的exitChildList上
                goto WAIT_BACK;//直接成功返回
            }
            ret = OsFindChildProcess(processCB， pid);//看能否从现有的孩子链表中找到PID
            if (ret != LOS_OK) {
                return LOS_ECHILD;//参数进程并没有这个PID孩子，返回孩子进程失败。
            }
            runTask->waitFlag = OS_PROCESS_WAIT_PRO;//设置当前任务的等待类型
            runTask->waitID = pid; //当前任务要等待进程ID结束
        } else if (pid == 0) {//等待同一进程组中的任何子进程
            /* Wait for any child process in the same process group */
            childCB = OsFindGroupExitProcess(processCB->group， OS_INVALID_VALUE);//看能否从退出的孩子链表中找到PID
            if (childCB != NULL) {//找到了，确实有一个已经退出的PID
                goto WAIT_BACK;//直接成功返回
            }
            runTask->waitID = processCB->group->groupID;//等待进程组的任意一个子进程结束
            runTask->waitFlag = OS_PROCESS_WAIT_GID;//设置当前任务的等待类型
        } else if (pid == -1) {//等待任意子进程
            /* Wait for any child process */
            childCB = OsFindExitChildProcess(processCB， OS_INVALID_VALUE);//看能否从退出的孩子链表中找到PID
            if (childCB != NULL) {//找到了，确实有一个已经退出的PID
                goto WAIT_BACK;
            }
            runTask->waitID = pid;//等待PID，这个PID可以和当前进程没有任何关系
            runTask->waitFlag = OS_PROCESS_WAIT_ANY;//设置当前任务的等待类型
        } else { /* pid < -1 */ //等待指定进程组内为|pid|的所有子进程
            /* Wait for any child process whose group number is the pid absolute value。 */
            group = OsFindProcessGroup(-pid);//先通过PID找到进程组
            if (group == NULL) {
                return LOS_ECHILD;
            }
            childCB = OsFindGroupExitProcess(group， OS_INVALID_VALUE);//在进程组里任意一个已经退出的子进程
            if (childCB != NULL) {
                goto WAIT_BACK;
            }
            runTask->waitID = -pid;//此处用负数是为了和(pid == 0)以示区别，因为二者的waitFlag都一样。
            runTask->waitFlag = OS_PROCESS_WAIT_GID;//设置当前任务的等待类型
        }
    WAIT_BACK:
        *child = childCB;
        return LOS_OK;
    }
```

- `status`带走进程退出码，`exitCode`分成了三个部分格式如下

```
/*
* Process exit code
```

```
* 31    15          8          7       0
* |     | exit code  | core dump | signal |
*/
#define OS_PRO_EXIT_OK 0 //进程正常退出
//置进程退出码第七位为1
STATIC INLINE VOID OsProcessExitCodeCoreDumpSet(LosProcessCB *processCB)
{
    processCB->exitCode |= 0x80U;// 0b10000000
}
//设置进程退出信号(0 ~ 7)
STATIC INLINE VOID OsProcessExitCodeSignalSet(LosProcessCB *processCB， UINT32 signal)
{
    processCB->exitCode |= signal & 0x7FU;//0b01111111
}
//清除进程退出信号(0 ~ 7)
STATIC INLINE VOID OsProcessExitCodeSignalClear(LosProcessCB *processCB)
{
    processCB->exitCode &= (~0x7FU);//低7位全部清0
}
//进程退出码是否被设置过，默认是 0 ，如果 & 0x7FU 还是 0 ，说明没有被设置过。
STATIC INLINE BOOL OsProcessExitCodeSignalIsSet(LosProcessCB *processCB)
{
    return (processCB->exitCode) & 0x7FU;
}
//设置进程退出号(8 ~ 15)
STATIC INLINE VOID OsProcessExitCodeSet(LosProcessCB *processCB， UINT32 code)
{
    processCB->exitCode |= ((code & 0x000000FFU) << 8U) & 0x0000FF00U; /* 8: Move 8 bits to the left， exitCode */
}
```

`0 - 7`为信号位，信号处理有专门的篇幅，此处不做详细介绍，请自行翻看，这里仅列出部分信号含义。

```
#define SIGHUP    1 //终端挂起或者控制进程终止
#define SIGINT    2 //键盘中断（如break键被按下）
#define SIGQUIT   3 //键盘的退出键被按下
#define SIGILL    4 //非法指令
#define SIGTRAP   5 //跟踪陷阱（trace trap），启动进程，跟踪代码的执行
#define SIGABRT   6 //由abort(3)发出的退出指令
#define SIGIOT    SIGABRT //abort发出的信号
#define SIGBUS    7 //总线错误
#define SIGFPE    8 //浮点异常
#define SIGKILL   9  //常用的命令 kill 9 123 | 不能被忽略、处理和阻塞
#define SIGUSR1   10 //用户自定义信号1
#define SIGSEGV   11 //无效的内存引用， 段违例（segmentation     violation），进程试图去访问其虚地址空间以外的位置
#define SIGUSR2   12 //用户自定义信号2
#define SIGPIPE   13 //向某个非读管道中写入数据
#define SIGALRM   14 //由alarm(2)发出的信号，默认行为为进程终止
#define SIGTERM   15 //终止信号
#define SIGSTKFLT 16 //栈溢出
#define SIGCHLD   17 //子进程结束信号
#define SIGCONT   18 //进程继续（曾被停止的进程）
#define SIGSTOP   19 //终止进程    | 不能被忽略、处理和阻塞
#define SIGTSTP   20 //控制终端（tty）上 按下停止键
#define SIGTTIN   21 //进程停止，后台进程企图从控制终端读
#define SIGTTOU   22 //进程停止，后台进程企图从控制终端写
#define SIGURG    23 //I/O有紧急数据到达当前进程
#define SIGXCPU   24 //进程的CPU时间片到期
#define SIGXFSZ   25 //文件大小的超出上限
#define SIGVTALRM 26 //虚拟时钟超时
#define SIGPROF   27 //profile时钟超时
#define SIGWINCH  28 //窗口大小改变
#define SIGIO     29 //I/O相关
#define SIGPOLL   29 //
#define SIGPWR    30 //电源故障，关机
#define SIGSYS    31 //系统调用中参数错，如系统调用号非法
#define SIGUNUSED SIGSYS  //系统调用异常
```

- `options`是行为参数，提供了一些另外的选项来控制waitpid()函数的行为。

参数值	鸿蒙支持	说明
LOS_WAIT_WNOHANG	支持	如果没有孩子进程退出，则立即返回，而不是阻塞在这个函数上等待；如果结束了，则返回该子进程的进程号。
LOS_WAIT_WUNTRACED	不支持	报告终止或停止的子进程的状态
LOS_WAIT_WCONTINUED	不支持	

鸿蒙目前只支持了LOS_WAIT_WNOHANG模式，内核源码中虽有`LOS_WAIT_WUNTRACED`和`LOS_WAIT_WCONTINUED`的实现痕迹，但是整体阅读下来比较乱，应该是没有写好。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

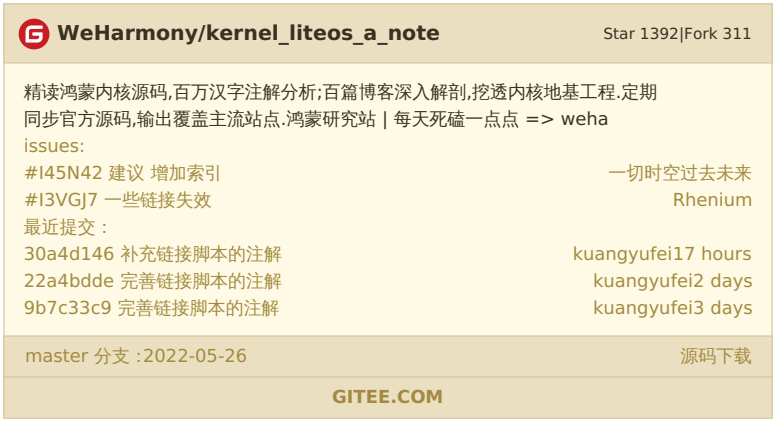
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

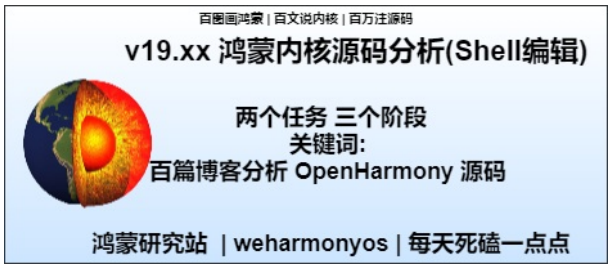


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

19_Shell编辑篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

系列篇从内核视角用一句话概括`shell`的底层实现为：**两个任务，三个阶段**。其本质是独立进程，因而划到进程管理模块。每次创建`shell`进程都会再创建两个任务。

- **客户端任务(ShellEntry)：** 负责接受来自终端(控制台)敲入的一个个字符，字符按`VT`规范组装成一句句的命令。
- **服务端任务(ShellTask)：** 对命令进行解析并执行，将结果输出到控制台。

而按命令生命周期可分三个阶段。

- **编辑：** 鸿蒙在这个部分实现了一个简单的编辑器功能，处理控制台输入的每个字符，主要包括了对控制字符 例如`<ESC>`，`\t`，`\b`，`\n`，`\r`，四个方向键`0x41 ~ 0x44`的处理。
- **解析：** 对编辑后的字符串进行解析，解析出命令项和参数项，找到对应的命令项执行函数。
- **执行：** 命令可通过静态和动态两种方式注册到内核，解析出具体命令后在注册表中找到对应函数回调。将结果输出到控制台。

编辑部分由客户端任务完成，后两个部分由服务端任务完成，命令全局注册由内核完成。

- 本篇主要说 **客户端任务** 和 **编辑过程**
- **服务端任务** 和 **解析/执行过程** 已在(Shell解析篇)中说明，请自行翻看。

什么是 Shell

从用户视角看，`shell`是用户窥视和操作内核的一个窗口，内核并非铁板一块，对应用层开了两个窗口，一个是系统调用，一个就是`shell`，由内核提供实现函数，由用户提供参数执行。区别是`shell`是由独立的任务去完成，可通过将`shell`命令序列化编写成独立的，简单的`shell`程序，所以`shell`也是一门脚本语言，系统调用是依附于应用程序的任务去完成，能做的有限。通过`shell`窗口能看到`cpu`的运行情况，内存的消耗情况，网络的链接状态等等。

鸿蒙 Shell 代码在哪

与`shell`对应的概念是`kernel`，在鸿蒙内核，这两部分代码是分开放的，`shell`代码在 查看 shell 代码 ，目录结构如下。

```
├─include
│      dmesg.h
│      dmesg_pri.h
│      shcmd.h
│      shcmdparse.h
│      shell.h
```

```
|      shell_lk.h
|      shell_pri.h
|      shmsg.h
|      show.h
|
└─src
   ├─base
   |      shcmd.c
   |      shcmdparse.c
   |      shell_lk.c
   |      shmsg.c
   |      show.c
   |
   └─cmds
          date_shellcmd.c
          dmesg.c
          hwi_shellcmd.c
          shell_shellcmd.c
          watch_shellcmd.c
```

Shell 控制块

跟进程，任务一样，每个概念的背后需要一个主结构体来的支撑，`shell`的主结构体就是`ShellCB`，掌握它就可以将`shell`拿捏的死死的，搞不懂这个结构体就读不懂`shell`的内核实现。所以在上面花再多功夫也不为过。

```
typedef struct {
   UINT32   consoleID; //控制台ID
   UINT32   shellTaskHandle; //shell服务端任务ID
   UINT32   shellEntryHandle; //shell客户端任务ID
   VOID     *cmdKeyLink;  //待处理的shell命令链表
   VOID     *cmdHistoryKeyLink;//已处理的历史记录链表，去重，10个
   VOID     *cmdMaskKeyLink; //主要用于方向键上下遍历历史命令
   UINT32   shellBufOffset; //buf偏移量
   UINT32   shellKeyType; //按键类型
   EVENT_CB_S shellEvent; //事件类型触发
   pthread_mutex_t keyMutex; //按键互斥量
   pthread_mutex_t historyMutex; //历史记录互斥量
   CHAR     shellBuf[SHOW_MAX_LEN]; //shell命令buf，接受键盘的输入，需要对输入字符解析。
   CHAR     shellWorkingDirectory[PATH_MAX];//shell的工作目录
} ShellCB;
//一个shell命令的结构体，命令有长有短，鸿蒙采用了可变数组的方式实现
typedef struct {
   UINT32 count; //字符数量
   LOS_DL_LIST list; //双向链表
   CHAR cmdString[0]; //字符串，可变数组的一种实现方式。
} CmdKeyLink;

enum {
   STAT_NOMAL_KEY， //普通的按键
   STAT_ESC_KEY， //<ESC>键在VT控制规范中时控制的起始键
   STAT_MULTI_KEY //组合键
};
```

解读

- 鸿蒙支持两种方式在控制台输入`Shell`命令，关于控制台请自行翻看控制台篇。
  - 在串口工具中直接输入`Shell`命令`CONSOLE_SERIAL`。
  - 在`telnet`工具中输入`Shell`命令`CONSOLE_TELNET`。
- `shellTaskHandle`和`shellEntryHandle`编辑/处理`shell`命令的两个任务ID，本篇重点说后一个。
- `cmdKeyLink`，`cmdHistoryKeyLink`，`cmdMaskKeyLink`是三个类型为`CmdKeyLink`的结构体，本质是双向链表，对应编辑`shell`命令过程中的三个功能。
  - `cmdKeyLink`待执行的命令链表
  - `cmdHistoryKeyLink`存储命令历史记录的，即：`history`命令显示的内容
  - `cmdMaskKeyLink`记录按上下方向键输出的内容，这个有点难理解，自行在`shell`中按上下方向键自行体验
- `shellBufOffset`和`shellBuf`是成对出现的，其中存放的就是用户敲入处理后的字符。
- `keyMutex`和`historyMutex`为操作链表所需的互斥锁，内核用的最多的就是这类锁。

- `shellEvent` 用于任务之间的通讯，比如。
  - `SHELL_CMD_PARSE_EVENT` :编辑完成了通知解析任务开始执行
  - `CONSOLE_SHELL_KEY_EVENT` :收到来自控制台的 `CTRL + C` 信号产生的事件。
- `shellKeyType` 按键的类型，分三种 普通，键，组合键
- `shellWorkingDirectory` 工作区就不用说了，从哪个目录进入 `shell` 的

创建 Shell

```
//shell进程的入口函数
int main(int argc， char **argv)
{
    //...
    g_shellCB = shellCB;//全局变量，说明鸿蒙同时只支持一个shell进程
    return OsShellCreateTask(shellCB);//初始化两个任务
}
//创建shell任务
STATIC UINT32 OsShellCreateTask(ShellCB *shellCB)
{
    UINT32 ret = ShellTaskInit(shellCB);//执行shell命令的任务初始化
    if (ret != LOS_OK) {
        return ret;
    }
    return ShellEntryInit(shellCB);//通过控制台接收shell命令的任务初始化
}
//进入shell客户端任务初始化，这个任务负责编辑命令，处理命令产生的过程，例如如何处理方向键，退格键，回车键等
LITE_OS_SEC_TEXT_MINOR UINT32 ShellEntryInit(ShellCB *shellCB)
{
    UINT32 ret;
    CHAR *name = NULL;
    TSK_INIT_PARAM_S initParam = {0};

    if (shellCB->consoleID == CONSOLE_SERIAL) {
        name = SERIAL_ENTRY_TASK_NAME;
    } else if (shellCB->consoleID == CONSOLE_TELNET) {
        name = TELNET_ENTRY_TASK_NAME;
    } else {
        return LOS_NOK;
    }

    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ShellEntry;//任务入口函数
    initParam.usTaskPrio   = 9; /* 9:shell task priority */
    initParam.auwArgs[0]   = (UINTPTR)shellCB;
    initParam.uwStackSize  = 0x1000;
    initParam.pcName       = name;
    initParam.uwResved     = LOS_TASK_STATUS_DETACHED;

    ret = LOS_TaskCreate(&shellCB->shellEntryHandle， &initParam);//创建任务
#ifdef LOSCFG_PLATFORM_CONSOLE
    (VOID)ConsoleTaskReg((INT32)shellCB->consoleID， shellCB->shellEntryHandle);//将任务注册到控制台
#endif

    return ret;
}
```

解读

- `main` 为 `shell` 进程的主任务，每个进程都会创建一个默认的线程(任务)，这个任务的入口函数就是大家熟知的 `main` 函数，不清楚的自行翻看任务管理各篇有详细的说明。
- 由 `main` 任务再创建两个任务，即本篇开头说的两个任务，本篇重点说其中的一个 `ShellEntry` ，任务优先级为 `9` ，算是较高优先级。
- 指定内核栈大小为 `0x1000 = 4K` ，因任务只负责编辑处理控制台输入的字符，命令的执行在其他任务，所以4K的内核空间足够使用。
- `ShellEntry` 为入口函数，这个函数的实现为本篇的重点

ShellEntry | 编辑过程

```
LITE_OS_SEC_TEXT_MINOR UINT32 ShellEntry(UINTPTR param)
{
```

```
    CHAR ch;
    INT32 n = 0;
    ShellCB *shellCB = (ShellCB *)param;

    CONSOLE_CB *consoleCB = OsGetConsoleByID((INT32)shellCB->consoleID);//获取控制台
    if (consoleCB == NULL) {
        PRINT_ERR("Shell task init error!\n");
        return 1;
    }

    (VOID)memset_s(shellCB->shellBuf， SHOW_MAX_LEN， 0， SHOW_MAX_LEN);//重置shell命令buf

    while (1) {
#ifdef LOSCFG_PLATFORM_CONSOLE
        if (!IsConsoleOccupied(consoleCB)) {//控制台是否被占用
#endif
            /* is console ready for shell ？ */
            n = read(consoleCB->fd， &ch， 1);//从控制台读取一个字符内容，字符一个个处理
            if (n == 1) {//如果能读到一个字符
                ShellCmdLineParse(ch， (pf_OUTPUT)dprintf， shellCB);
            }
            if (is_nonblock(consoleCB)) {//在非阻塞模式下暂停 50ms
                LOS_Msleep(50); /* 50: 50MS for sleep */
            }
#ifdef LOSCFG_PLATFORM_CONSOLE
        }
#endif
    }
}
//对命令行内容解析
LITE_OS_SEC_TEXT_MINOR VOID ShellCmdLineParse(CHAR c， pf_OUTPUT outputFunc， ShellCB *shellCB)
{
    const CHAR ch = c;
    INT32 ret;
 //不是回车键和字符串结束，且偏移量为0
    if ((shellCB->shellBufOffset == 0) && (ch != '\n') && (ch != '\0')) {
        (VOID)memset_s(shellCB->shellBuf， SHOW_MAX_LEN， 0， SHOW_MAX_LEN);//重置buf
    }
 //遇到回车或换行
    if ((ch == '\r') || (ch == '\n')) {
        if (shellCB->shellBufOffset < (SHOW_MAX_LEN - 1)) {
            shellCB->shellBuf[shellCB->shellBufOffset] = '\0';//字符串结束
        }
        shellCB->shellBufOffset = 0;
        (VOID)pthread_mutex_lock(&shellCB->keyMutex);
        OsShellCmdPush(shellCB->shellBuf， shellCB->cmdKeyLink);//解析回车或换行
        (VOID)pthread_mutex_unlock(&shellCB->keyMutex);
        ShellNotify(shellCB);//通知任务解析shell命令
        return;
    } else if ((ch == '\b') || (ch == 0x7F)) { /* backspace or delete(0x7F) */ //遇到删除键
        if ((shellCB->shellBufOffset > 0) && (shellCB->shellBufOffset < (SHOW_MAX_LEN - 1))) {
            shellCB->shellBuf[shellCB->shellBufOffset - 1] = '\0';//填充`\0`
            shellCB->shellBufOffset--;//buf减少
            outputFunc("\b \b");//回调入参函数
        }
        return;
    } else if (ch == 0x09) { /* 0x09: tab *///遇到tab键
        if ((shellCB->shellBufOffset > 0) && (shellCB->shellBufOffset < (SHOW_MAX_LEN - 1))) {
            ret = OsTabCompletion(shellCB->shellBuf， &shellCB->shellBufOffset);//解析tab键
            if (ret > 1) {
                outputFunc("OHOS # %s"， shellCB->shellBuf);//回调入参函数
            }
        }
        return;
    }
    /* parse the up/down/right/left key */
    ret = ShellCmdLineCheckUDRL(ch， shellCB);//解析上下左右键
    if (ret == LOS_OK) {
        return;
    }
```

```
    if ((ch != '\n') && (ch != '\0')) {//普通的字符的处理
        if (shellCB->shellBufOffset < (SHOW_MAX_LEN - 1)) {//buf范围
            shellCB->shellBuf[shellCB->shellBufOffset] = ch;//直接加入
        } else {
            shellCB->shellBuf[SHOW_MAX_LEN - 1] = '\0';//加入字符串结束符
        }
        shellCB->shellBufOffset++;//偏移量增加
        outputFunc("%c" ， ch);//向终端输出字符
    }

    shellCB->shellKeyType = STAT_NOMAL_KEY;//普通字符
}
```

解读

- `ShellEntry`内部是个死循环，不断的读取控制台输入的每个字符，注意是按字符处理。
- 处理四个方向，换行回车，`tab`，`backspace`，`delete`，`esc`等控制键，相当于重新认识了下`Ascii`表。可以把`shell`终端理解为一个简单的编辑器。
    - 按回车键 表示完成前面的输入，进入解析执行阶段。
    - 按方向键 要显示上/下一个命令的内容，一直按就一直显示上上/下下命令。
    - 按`tab`键 是要补齐命令的内容，目前鸿蒙支持如下命令:

      ```
      arp         cat         cd          chgrp       chmod       chown       cp          cpup
      date        dhclient    dmesg       dns         format      free        help        hwi
      ifconfig    ipdebug     kill        log         ls          lsfd        memcheck    mkdir
      mount       netstat     oom         partinfo    partition   ping        ping6       pwd
      reset       rm          rmdir       sem         statfs      su          swtmr       sync
      systeminfo  task        telnet      test        tftp        touch       umount      uname
      watch       writeproc
      ```

      例如:当在控制台按下`ch`和`tab`键后会输出以下三个

      ```
      chgrp       chmod       chown
      ```

      内容，这些功能对使用者而已看似再平常不过，但都需要内核一一实现。
- `shellBuf`存储编辑结果，当按下回车键时，将结果保存并交付给下一个阶段使用。

百文说内核 | 抓住主脉络

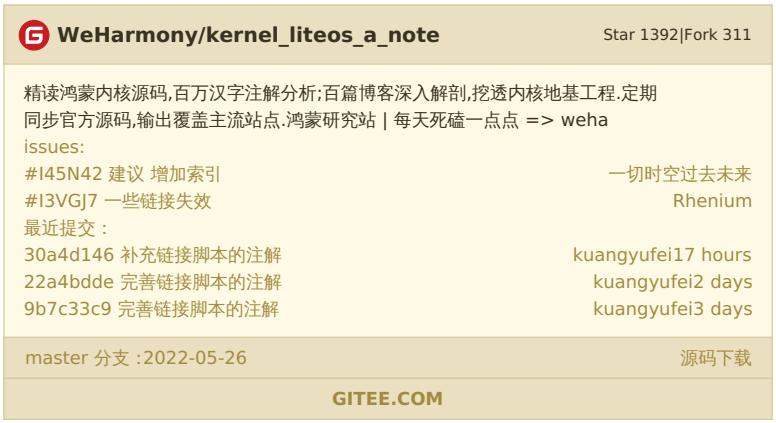
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

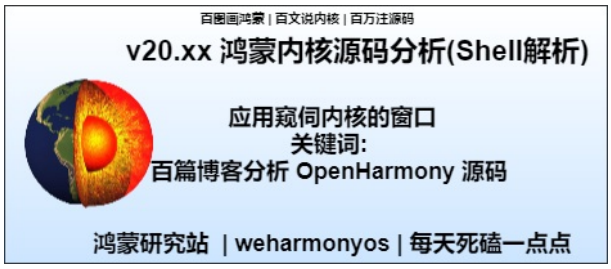


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

20_Shell解析篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

进程管理相关篇为:

- v11.04 鸿蒙内核源码分析(调度故事) | 大郎，该喝药了
- v12.03 鸿蒙内核源码分析(进程控制块) | 可怜天下父母心
- v13.01 鸿蒙内核源码分析(进程空间) | 有爱的地方才叫家
- v14.01 鸿蒙内核源码分析(线性区) | 人要有空间才能好好相处
- v15.01 鸿蒙内核源码分析(红黑树) | 众里寻他千百度
- v16.06 鸿蒙内核源码分析(进程管理) | 家家有本难念的经
- v17.05 鸿蒙内核源码分析(Fork进程) | 一次调用 两次返回
- v18.02 鸿蒙内核源码分析(进程回收) | 临终托孤的短命娃
- v19.03 鸿蒙内核源码分析(Shell编辑) | 两个任务 三个阶段
- v20.01 鸿蒙内核源码分析(Shell解析) | 应用窥伺内核的窗口

系列篇从内核视角用一句话概括`shell`的底层实现为：**两个任务，三个阶段**。其本质是独立进程，因而划到进程管理模块。每次创建`shell`进程都会再创建两个任务。

- **客户端任务(ShellEntry)：** 负责接受来自终端(控制台)敲入的一个个字符，字符按`VT`规范组装成一句句的命令。
- **服务端任务(ShellTask)：** 对命令进行解析并执行，将结果输出到控制台。 而按命令生命周期可分三个阶段。
- **编辑：** 鸿蒙在这个部分实现了一个简单的编辑器功能，处理控制台输入的每个字符，主要包括了对控制字符 例如`<ESC>`，`\t`，`\b`，`\n`，`\r`，四个方向键`0x41 ~ 0x44`的处理。
- **解析：** 对编辑后的字符串进行解析，解析出命令项和参数项，找到对应的命令项执行函数。
- **执行：** 命令可通过静态和动态两种方式注册到内核，解析出具体命令后在注册表中找到对应函数回调。将结果输出到控制台。

编辑部分由客户端任务完成，后两个部分由服务端任务完成，命令全局注册由内核完成。

- 本篇主要说 **服务端任务** 和 **解析/执行过程**。
- **客户端任务** 和 **编辑过程** 已在(Shell编辑篇)中说明，请自行翻看。

总体过程

- 第一步： 将支持的`shell`命令注册进全局链表，支持静态和动态两种方式，内容包括命令项，参数信息和回调函数。
- 第二步： 由独立任务解析出用户输入的命令行，拆分出命令项和参数内容
- 第三步： 通过命令项在全局链表中遍历找到已注册的回调函数，并执行。

结构体

鸿蒙对命令的注册用了三个结构体，个人感觉前两个可以合成一个，降低代码阅读难度。

```
STATIC CmdModInfo g_cmdInfo;//shell 命令模块信息，上面挂了所有的命令项(ls，cd ，cp ==)

typedef struct {//命令项
    CmdType cmdType; //命令类型
  //CMD_TYPE_EX：不支持标准命令参数输入，会把用户填写的命令关键字屏蔽掉，例如：输入ls /ramfs，传入给注册函数的参数只有/ramfs，而ls命令关键字并不会
  //CMD_TYPE_STD：支持的标准命令参数输入，所有输入的字符都会通过命令解析后被传入。
    const CHAR *cmdKey; //命令关键字，例如:ls 函数在Shell中访问的名称。
    UINT32 paraNum;  //调用的执行函数的入参最大个数，暂不支持。
    CmdCallBackFunc cmdHook;//命令执行函数地址，即命令实际执行函数。
```

```
} CmdItem;
typedef struct { //命令节点
    LOS_DL_LIST list; //双向链表
    CmdItem *cmd; //命令项
} CmdItemNode;

/* global info for shell module */
typedef struct {//shell 模块的全局信息
    CmdItemNode cmdList; //命令项节点
    UINT32 listNum;//节点数量
    UINT32 initMagicFlag;//初始魔法标签 0xABABABAB
    LosMux muxLock; //操作链表互斥锁
    CmdVerifyTransID transIdHook;//暂不知何意
} CmdModInfo;
```

解读

- `CmdItem` 为注册的内容载体结构体， `cmdHook` 为回调函数，是命令的真正执行体。
- 通过双向链表 `CmdItemNode.list` 将所有命令穿起来
- `CmdModInfo` 记录命令数量和操作的互斥锁， `shell` 的魔法数字为 `0xABABABAB`

第一步 | Shell 注册

- **静态宏方式注册，链接时处理** 静态注册命令方式一般用在系统常用命令注册，鸿蒙已支持以下命令。

```
    arp         cat         cd          chgrp       chmod       chown       cp          cpup
    date        dhclient    dmesg       dns         format      free        help        hwi
    ifconfig    ipdebug     kill        log         ls          lsfd        memcheck    mkdir
    mount       netstat     oom         partinfo    partition   ping        ping6       pwd
    reset       rm          rmdir       sem         statfs      su          swtmr       sync
    systeminfo  task        telnet      test        tftp        touch       umount      uname
    watch       writeproc
```

例如注册 `ls` 命令

```
SHELLCMD_ENTRY(ls_shellcmd， CMD_TYPE_EX， "ls"， XARGS， (CMD_CBK_FUNC)osShellCmdLs)
```

需在链接选项中添加链接该新增命令项参数，具体在 `liteos_tables_ldflags.mk` 文件的 `LITEOS_TABLES_LDFLAGS` 项下添加 `-uls_shellcmd` 。至于 `SHELLCMD_ENTRY` 是如何实现的在链接阶段的注册，请自行翻看(内联汇编篇)，有详细说明实现细节。

- **动态命令方式，运行时处理** 动态注册命令方式一般用在用户命令注册，具体实现代码如下:

```
osCmdReg(CMD_TYPE_EX， "ls"， XARGS， (CMD_CBK_FUNC)osShellCmdLs)
{
    // ....
    //5.正式创建命令，挂入链表
    return OsCmdItemCreate(cmdType， cmdKey， paraNum， cmdProc);//不存在就注册命令
}
//创建一个命令项，例如 chmod
STATIC UINT32 OsCmdItemCreate(CmdType cmdType， const CHAR *cmdKey， UINT32 paraNum， CmdCallBackFunc cmdProc)
{
    CmdItem *cmdItem = NULL;
    CmdItemNode *cmdItemNode = NULL;
    //1.构造命令节点过程
    cmdItem = (CmdItem *)LOS_MemAlloc(m_aucSysMem0， sizeof(CmdItem));
    if (cmdItem == NULL) {
        return OS_ERRNO_SHELL_CMDREG_MEMALLOC_ERROR;
    }
    (VOID)memset_s(cmdItem， sizeof(CmdItem)， '\0'， sizeof(CmdItem));

    cmdItemNode = (CmdItemNode *)LOS_MemAlloc(m_aucSysMem0， sizeof(CmdItemNode));
    if (cmdItemNode == NULL) {
        (VOID)LOS_MemFree(m_aucSysMem0， cmdItem);
        return OS_ERRNO_SHELL_CMDREG_MEMALLOC_ERROR;
    }
    (VOID)memset_s(cmdItemNode， sizeof(CmdItemNode)， '\0'， sizeof(CmdItemNode));
```

```
    cmdItemNode->cmd = cmdItem;   //命令项
    cmdItemNode->cmd->cmdHook = cmdProc;//回调函数 osShellCmdLs
    cmdItemNode->cmd->paraNum = paraNum;//`777`，'/home'
    cmdItemNode->cmd->cmdType = cmdType;//关键字类型
    cmdItemNode->cmd->cmdKey = cmdKey; //`chmod`
    //2.完成构造后挂入全局链表
    (VOID)LOS_MuxLock(&g_cmdInfo.muxLock， LOS_WAIT_FOREVER);
    OsCmdAscendingInsert(cmdItemNode);//按升序方式插入
    g_cmdInfo.listNum++;//命令总数增加
    (VOID)LOS_MuxUnlock(&g_cmdInfo.muxLock);

    return LOS_OK;
  }
```

第二步 解析 | ShellTask

```
//shell 服务端任务初始化，这个任务负责解析和执行命令
LITE_OS_SEC_TEXT_MINOR UINT32 ShellTaskInit(ShellCB *shellCB)
{
   CHAR *name = NULL;
   TSK_INIT_PARAM_S initParam = {0};
 //输入Shell命令的两种方式
   if (shellCB->consoleID == CONSOLE_SERIAL) { //通过串口工具
      name = SERIAL_SHELL_TASK_NAME;
   } else if (shellCB->consoleID == CONSOLE_TELNET) {//通过远程工具
      name = TELNET_SHELL_TASK_NAME;
   } else {
      return LOS_NOK;
   }

   initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ShellTask;//任务入口函数，主要是解析shell命令
   initParam.usTaskPrio   = 9; /* 9:shell task priority */
   initParam.auwArgs[0]   = (UINTPTR)shellCB;
   initParam.uwStackSize  = 0x3000;
   initParam.pcName       = name;
   initParam.uwResved     = LOS_TASK_STATUS_DETACHED;

   (VOID)LOS_EventInit(&shellCB->shellEvent);//初始化事件，以事件方式通知任务解析命令

   return LOS_TaskCreate(&shellCB->shellTaskHandle， &initParam);//创建任务
}
LITE_OS_SEC_TEXT_MINOR UINT32 ShellTask(UINTPTR param1，
                                UINTPTR param2，
                                UINTPTR param3，
                                UINTPTR param4)
{
   UINT32 ret;
   ShellCB *shellCB = (ShellCB *)param1;
   (VOID)param2;
   (VOID)param3;
   (VOID)param4;

   while (1) {
      PRINTK("\nOHOS # ");//读取shell 输入事件 例如: cat weharmony.net 命令
      ret = LOS_EventRead(&shellCB->shellEvent，
                          0xFFF， LOS_WAITMODE_OR | LOS_WAITMODE_CLR， LOS_WAIT_FOREVER);
      if (ret == SHELL_CMD_PARSE_EVENT) {//获得解析命令事件
         ShellCmdProcess(shellCB);//处理命令
      } else if (ret == CONSOLE_SHELL_KEY_EVENT) {//退出shell事件
         break;
      }
   }
   OsShellKeyDeInit((CmdKeyLink *)shellCB->cmdKeyLink);//
   OsShellKeyDeInit((CmdKeyLink *)shellCB->cmdHistoryKeyLink);
   (VOID)LOS_EventDestroy(&shellCB->shellEvent);//注销事件
   (VOID)LOS_MemFree((VOID *)m_aucSysMem0， shellCB);//释放shell控制块
   return 0;
}
```

解读

- 任务优先级和 客户端任务 一样同为 `9`
- 指定内核栈大小为 `0x3000 = 12K` ，因任务负责命令的解析和执行，所以需要更大的内核空间。
- 任务的入口函数 `ShellTask` ，一个死循环在以 `LOS_WAIT_FOREVER` 方式死等事件发生。
  - `SHELL_CMD_PARSE_EVENT` 通知开始解析事件，该事件由 客户端任务 `ShellEntry` 检测到回车键时发出。

```
STATIC VOID ShellNotify(ShellCB *shellCB)
{
    (VOID)LOS_EventWrite(&shellCB->shellEvent， SHELL_CMD_PARSE_EVENT);
}
```

  - `CONSOLE_SHELL_KEY_EVENT` 收到 `exit` 命令时将发出该事件，退出 `shell` 回收资源 鸿蒙内核是如何管理和使用事件的请自行翻看(事件控制篇)
- 层层跟进 `ShellCmdProcess` ，解析出命令项和参数内容，最终跑到 `OsCmdExec` 中遍历 已注册的命令表，找出命令对应的函数完成回调。

```
LITE_OS_SEC_TEXT_MINOR UINT32 OsCmdExec(CmdParsed *cmdParsed， CHAR *cmdStr)
{
    UINT32 ret;
    CmdCallBackFunc cmdHook = NULL;
    CmdItemNode *curCmdItem = NULL;
    UINT32 i;
    const CHAR *cmdKey = NULL;

    if ((cmdParsed == NULL) || (cmdStr == NULL) || (strlen(cmdStr) == 0)) {
        return (UINT32)OS_ERROR;
    }

    ret = OsCmdParse(cmdStr， cmdParsed);//解析出命令关键字，参数
    if (ret != LOS_OK) {
        goto OUT;
    }
    //遍历命令注册全局链表
    LOS_DL_LIST_FOR_EACH_ENTRY(curCmdItem， &(g_cmdInfo.cmdList.list)， CmdItemNode， list) {
        cmdKey = curCmdItem->cmd->cmdKey;
        if ((cmdParsed->cmdType == curCmdItem->cmd->cmdType) &&
            (strlen(cmdKey) == strlen(cmdParsed->cmdKeyword)) &&
            (strncmp(cmdKey， (CHAR *)(cmdParsed->cmdKeyword)， strlen(cmdKey)) == 0)) {//找到命令的回调函数 例如: ls <-> osShellCmdLs
            cmdHook = curCmdItem->cmd->cmdHook;
            break;
        }
    }

    ret = OS_ERROR;
    if (cmdHook != NULL) {//执行命令，即回调函数
        ret = (cmdHook)(cmdParsed->paramCnt， (const CHAR **)cmdParsed->paramArray);
    }

OUT:
    for (i = 0; i < cmdParsed->paramCnt; i++) {//无效的命令要释放掉保存参数的内存
        if (cmdParsed->paramArray[i] != NULL) {
            (VOID)LOS_MemFree(m_aucSysMem0， cmdParsed->paramArray[i]);
            cmdParsed->paramArray[i] = NULL;
        }
    }

    return (UINT32)ret;
}
```

第三步 | 执行

想知道有哪些系统 `shell` 命令，可以搜索关键词 `SHELLCMD_ENTRY` 拿到所有通过静态方式注册的命令。

```
---- SHELLCMD_ENTRY Matches (70 in 30 files) ----
api_shell.c (net\lwip-2.1\enhancement\src) line 1144 : SHELLCMD_ENTRY(ifconfig_shellcmd, CMD_TYPE_EX, "ifconfig", XARGS, (CmdCallBackFunc)lwip_ifco
api_shell.c (net\lwip-2.1\enhancement\src) line 1481 : SHELLCMD_ENTRY(arp_shellcmd, CMD_TYPE_EX, "arp", 1, (CmdCallBackFunc)lwip_arp);
api_shell.c (net\lwip-2.1\enhancement\src) line 1924 : SHELLCMD_ENTRY(ping_shellcmd, CMD_TYPE_EX, "ping", XARGS, (CmdCallBackFunc)osShellPing);
api_shell.c (net\lwip-2.1\enhancement\src) line 2077 : SHELLCMD_ENTRY(ping_shellcmd, CMD_TYPE_EX, "ping", XARGS, (CmdCallBackFunc)osShellPing);
api_shell.c (net\lwip-2.1\enhancement\src) line 2586 : SHELLCMD_ENTRY(ping6_shellcmd, CMD_TYPE_EX, "ping6", XARGS, (CmdCallBackFunc)osShellPing6);
api_shell.c (net\lwip-2.1\enhancement\src) line 2630 : SHELLCMD_ENTRY(ntpdate_shellcmd, CMD_TYPE_EX, "ntpdate", XARGS, (CmdCallBackFunc)osShellNtpd
api_shell.c (net\lwip-2.1\enhancement\src) line 2704 : SHELLCMD_ENTRY(dns_shellcmd, CMD_TYPE_EX, "dns", XARGS, (CmdCallBackFunc)osShellDns);
api_shell.c (net\lwip-2.1\enhancement\src) line 3379 : SHELLCMD_ENTRY(netstat_shellcmd, CMD_TYPE_EX, "netstat", XARGS, (CmdCallBackFunc)osShellNets
api_shell.c (net\lwip-2.1\enhancement\src) line 3429 : SHELLCMD_ENTRY(dhclient_shellcmd, CMD_TYPE_EX, "dhclient", XARGS, (CmdCallBackFunc)OsShellDh
api_shell.c (net\lwip-2.1\enhancement\src) line 3547 : SHELLCMD_ENTRY(tcpserver_shellcmd, CMD_TYPE_EX, "tcpserver", XARGS, (CmdCallBackFunc)osTcpse
api_shell.c (net\lwip-2.1\enhancement\src) line 3622 : SHELLCMD_ENTRY(udpserver_shellcmd, CMD_TYPE_EX, "udpserver", XARGS, (CmdCallBackFunc)udpserv
api_shell.c (net\lwip-2.1\enhancement\src) line 3743 : SHELLCMD_ENTRY(netdebug_shellcmd, CMD_TYPE_EX, "netdebug", XARGS, (CmdCallBackFunc)osShellNe
api_shell.c (net\lwip-2.1\enhancement\src) line 3873 : SHELLCMD_ENTRY(ipdebug_shellcmd, CMD_TYPE_EX, "ipdebug", XARGS, (CmdCallBackFunc)osShellIpDe
api_shell.c (net\lwip-2.1\enhancement\src) line 3884 : SHELLCMD_ENTRY(reboot_shellcmd, CMD_TYPE_EX, "reboot", XARGS, (CmdCallBackFunc)osShellReboot
cpup_shellcmd.c (kernel\extended\cpup) line 157 : SHELLCMD_ENTRY(cpup_shellcmd, CMD_TYPE_EX, "cpup", XARGS, (CmdCallBackFunc)OsShellCmdCpup);//采用
date_shellcmd.c (shell\full\src\cmds) line 301 : SHELLCMD_ENTRY(date_shellcmd, CMD_TYPE_STD, "date", XARGS, (CmdCallBackFunc)OsShellCmdDate);
disk_shellcmd.c (drivers\block\disk\src) line 92 : SHELLCMD_ENTRY(partinfo_shellcmd, CMD_TYPE_EX, "partinfo", XARGS, (CmdCallBackFunc)osShellCmdPar
dmesg.c (shell\full\src\cmds) line 786 : SHELLCMD_ENTRY(dmesg_shellcmd, CMD_TYPE_STD, "dmesg", XARGS, (CmdCallBackFunc)OsShellCmdDmesg);
fat_shellcmd.c (fs\fat\os_adapt) line 79 : SHELLCMD_ENTRY(format_shellcmd, CMD_TYPE_EX, "format", XARGS, (CmdCallBackFunc)osShellCmdFormat);
hwi_shellcmd.c (shell\full\src\cmds) line 116 : SHELLCMD_ENTRY(hwi_shellcmd, CMD_TYPE_EX, "hwi", 0, (CmdCallBackFunc)OsShellCmdHwi);
kill_shellcmd.c (kernel\base\misc) line 114 : SHELLCMD_ENTRY(kill_shellcmd, CMD_TYPE_EX, "kill", 2, (CmdCallBackFunc)OsShellCmdKill);
los_excinfo.c (kernel\common) line 193 : SHELLCMD_ENTRY(readExcInfo_shellcmd, CMD_TYPE_EX, "excInfo", 0, (CmdCallBackFunc)OsShellCmdReadExcInfo);
los_queue_debug.c (kernel\base\ipc) line 195 : SHELLCMD_ENTRY(queue_shellcmd, CMD_TYPE_EX, "queue", 0, (CmdCallBackFunc)OsShellCmdQueueInfoGet);
los_sem_debug.c (kernel\base\ipc) line 299 : SHELLCMD_ENTRY(sem_shellcmd, CMD_TYPE_EX, "sem", 1, (CmdCallBackFunc)OsShellCmdSemInfoGet);//采用shell
los_stackinfo.c (kernel\base\misc) line 146 : SHELLCMD_ENTRY(stack_shellcmd, CMD_TYPE_EX, "stack", 1, (CmdCallBackFunc)OsExcStackInfo);//采用shell
los_trace.c (kernel\extended\trace) line 416 : SHELLCMD_ENTRY(tracestart_shellcmd,   CMD_TYPE_EX, "trace_start", 0, (CmdCallBackFunc)LOS_TraceStart
los_trace.c (kernel\extended\trace) line 417 : SHELLCMD_ENTRY(tracestop_shellcmd,    CMD_TYPE_EX, "trace_stop",  0, (CmdCallBackFunc)LOS_TraceStop)
los_trace.c (kernel\extended\trace) line 418 : SHELLCMD_ENTRY(tracesetmask_shellcmd, CMD_TYPE_EX, "trace_mask",  1, (CmdCallBackFunc)OsShellCmdTrac
los_trace.c (kernel\extended\trace) line 419 : SHELLCMD_ENTRY(tracereset_shellcmd,   CMD_TYPE_EX, "trace_reset", 0, (CmdCallBackFunc)LOS_TraceReset
los_trace.c (kernel\extended\trace) line 420 : SHELLCMD_ENTRY(tracedump_shellcmd,    CMD_TYPE_EX, "trace_dump", 1, (CmdCallBackFunc)OsShellCmdTrace
main.c (apps\tftp\src) line 158 : SHELLCMD_ENTRY(tftp_shellcmd, CMD_TYPE_EX, "tftp", XARGS, (CmdCallBackFunc)(uintptr_t)osShellTftp);
mempt_shellcmd.c (kernel\base\misc) line 237 : SHELLCMD_ENTRY(memused_shellcmd, CMD_TYPE_EX, "memused", 0, (CmdCallBackFunc)OsShellCmdMemUsed);
mempt_shellcmd.c (kernel\base\misc) line 241 : SHELLCMD_ENTRY(memcheck_shellcmd, CMD_TYPE_EX, "memcheck", 0, (CmdCallBackFunc)OsShellCmdMemCheck);
mempt_shellcmd.c (kernel\base\misc) line 243 : SHELLCMD_ENTRY(free_shellcmd, CMD_TYPE_EX, "free", XARGS, (CmdCallBackFunc)OsShellCmdFree);
mempt_shellcmd.c (kernel\base\misc) line 244 : SHELLCMD_ENTRY(uname_shellcmd, CMD_TYPE_EX, "uname", XARGS, (CmdCallBackFunc)OsShellCmdUname);
mtd_shellcmd.c (drivers\mtd\multi_partition\src) line 91 : SHELLCMD_ENTRY(partition_shellcmd, CMD_TYPE_EX, "partition", XARGS, (CmdCallBackFunc)osS
panic_shellcmd.c (kernel\base\misc) line 120 : SHELLCMD_ENTRY(panic_reset_shellcmd, CMD_TYPE_EX, "panicreset", 1, (CmdCallBackFunc)OsShellCmdSystem
proc_shellcmd.c (fs\proc\src) line 123 : SHELLCMD_ENTRY(writeproc_shellcmd, CMD_TYPE_EX, "writeproc", XARGS, (CmdCallBackFunc)OsShellCmdWriteProc);
shcmd.h (shell\full\include) line 82 : #define SHELLCMD_ENTRY(1, cmdType, cmdKey, paraNum, cmdHook)    \
shell_lk.c (shell\full\src\base) line 250 : SHELLCMD_ENTRY(log_shellcmd, CMD_TYPE_EX, "log", 1, (CmdCallBackFunc)CmdLog);
shell_shellcmd.c (shell\full\src\cmds) line 74 : SHELLCMD_ENTRY(help_shellcmd, CMD_TYPE_EX, "help", 0, (CmdCallBackFunc)OsShellCmdHelp);
shm.c (kernel\base\vm) line 923 : SHELLCMD_ENTRY(shm_shellcmd, CMD_TYPE_SHOW, "shm", 2, (CmdCallBackFunc)OsShellCmdShm);
swtmr_shellcmd.c (kernel\base\misc) line 140 : SHELLCMD_ENTRY(swtmr_shellcmd, CMD_TYPE_EX, "swtmr", 1, (CmdCallBackFunc)OsShellCmdSwtmrInfoGet);//
sysinfo_shellcmd.c (kernel\base\misc) line 171 : SHELLCMD_ENTRY(systeminfo_shellcmd, CMD_TYPE_EX, "systeminfo", 1, (CmdCallBackFunc)OsShellCmdSyste
task_shellcmd.c (kernel\base\misc) line 610 : SHELLCMD_ENTRY(task_shellcmd, CMD_TYPE_EX, "task", 1, (CmdCallBackFunc)OsShellCmdDumpTask);
```

其中有网络的，进程的，任务的，内存的 等等，此处列出几个常用的 shell 命令的实现。

ls 命令

```
SHELLCMD_ENTRY(ls_shellcmd， CMD_TYPE_EX， "ls"， XARGS， (CmdCallBackFunc)osShellCmdLs);

/*******************************************************
命令功能
ls命令用来显示当前目录的内容。

命令格式
ls [path]

path为空时，显示当前目录的内容。
path为无效文件名时，显示失败，提示：
ls error: No such directory。
path为有效目录路径时，会显示对应目录下的内容。

使用指南
ls命令显示当前目录的内容。
ls可以显示文件的大小。
proc下ls无法统计文件大小，显示为0。

*******************************************************/
int osShellCmdLs(int argc， const char **argv)
{
  char *fullpath = NULL;
  const char *filename = NULL;
  int ret;
  char *shell_working_directory = OsShellGetWorkingDirtectory();//获取当前工作目录
  if (shell_working_directory == NULL)
    {
      return -1;
    }

  ERROR_OUT_IF(argc > 1， PRINTK("ls or ls [DIRECTORY]\n")， return -1);

  if (argc == 0)//木有参数时 -> #ls
    {
      ls(shell_working_directory);//执行ls 当前工作目录
      return 0;
```

```
    }

  filename = argv[0];//有参数时 -> #ls ../harmony  or #ls /no such file or directory
  ret = vfs_normalize_path(shell_working_directory， filename， &fullpath);//获取全路径，注意这里带出来fullpath，而fullpath已经在内核空间
  ERROR_OUT_IF(ret < 0， set_err(-ret， "ls error")， return -1);

  ls(fullpath);//执行 ls 全路径
  free(fullpath);//释放全路径，为啥要释放，因为fullpath已经由内核空间分配

  return 0;
}
```

task 命令

```
SHELLCMD_ENTRY(task_shellcmd， CMD_TYPE_EX， "task"， 1， (CmdCallBackFunc)OsShellCmdDumpTask);

LITE_OS_SEC_TEXT_MINOR UINT32 OsShellCmdDumpTask(INT32 argc， const CHAR **argv)
{
    UINT32 flag = 0;
#ifdef LOSCFG_KERNEL_VM
    flag |= OS_PROCESS_MEM_INFO;
#endif

    if (argc >= 2) { /* 2: The task shell name restricts the parameters */
        goto TASK_HELP;
    }

    if (argc == 1) {
        if (strcmp("-a"， argv[0]) == 0) {
            flag |= OS_PROCESS_INFO_ALL;
        } else if (strcmp("-i"， argv[0]) == 0) {
            if (!OsShellShowTickRespo()) {
                return LOS_OK;
            }
            goto TASK_HELP;
        } else if (strcmp("-t"， argv[0]) == 0) {
            if (!OsShellShowSchedParam()) {
                return LOS_OK;
            }
            goto TASK_HELP;
        } else {
            goto TASK_HELP;
        }
    }

    return OsShellCmdTskInfoGet(OS_ALL_TASK_MASK， NULL， flag);

TASK_HELP:
    PRINTK("Unknown option: %s\n"， argv[0]);
    PRINTK("usage: task or task -a\n");
    return LOS_NOK;
}
```

cat 命令

```
SHELLCMD_ENTRY(cat_shellcmd， CMD_TYPE_EX， "cat"， XARGS， (CmdCallBackFunc)osShellCmdCat);

/*****************************************************************
cat用于显示文本文件的内容.cat [pathname]
cat weharmony.txt
*****************************************************************/
int osShellCmdCat(int argc， const char **argv)
{
  char *fullpath = NULL;
  int ret;
  unsigned int ca_task;
  struct Vnode *vnode = NULL;
```

```
  TSK_INIT_PARAM_S init_param;
  char *shell_working_directory = OsShellGetWorkingDirtectory();//显示当前目录 pwd
  if (shell_working_directory == NULL)
    {
      return -1;
    }

  ERROR_OUT_IF(argc != 1 , PRINTK("cat [FILE]\n") , return -1);

  ret = vfs_normalize_path(shell_working_directory , argv[0] , &fullpath);//由相对路径获取绝对路径
  ERROR_OUT_IF(ret < 0 , set_err(-ret , "cat error") , return -1);

  VnodeHold();
  ret = VnodeLookup(fullpath , &vnode , O_RDONLY);
    if (ret != LOS_OK)
      {
        set_errno(-ret);
        perror("cat error");
        VnodeDrop();
        free(fullpath);
        return -1;
      }
    if (vnode->type != VNODE_TYPE_REG)
      {
        set_errno(EINVAL);
        perror("cat error");
        VnodeDrop();
        free(fullpath);
        return -1;
      }
  VnodeDrop();
  (void)memset_s(&init_param , sizeof(init_param) , 0 , sizeof(TSK_INIT_PARAM_S));
  init_param.pfnTaskEntry = (TSK_ENTRY_FUNC)osShellCmdDoCatShow;
  init_param.usTaskPrio   = CAT_TASK_PRIORITY; //优先级10
  init_param.auwArgs[0]   = (UINTPTR)fullpath; //入口参数
  init_param.uwStackSize  = CAT_TASK_STACK_SIZE;//内核栈大小
  init_param.pcName       = "shellcmd_cat"; //任务名称
  init_param.uwResved     = LOS_TASK_STATUS_DETACHED | OS_TASK_FLAG_SPECIFIES_PROCESS;
  init_param.processID    = 2; /* 2: kProcess */ //内核任务

  ret = (int)LOS_TaskCreate(&ca_task , &init_param);//创建任务显示cat内容

  if (ret != LOS_OK)
    {
      free(fullpath);
    }

  return ret;
}
```

你能看明白这些命令的底层实现吗？ 如果看明白了，可能会不由得发出 **原来如此** 的感叹!

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

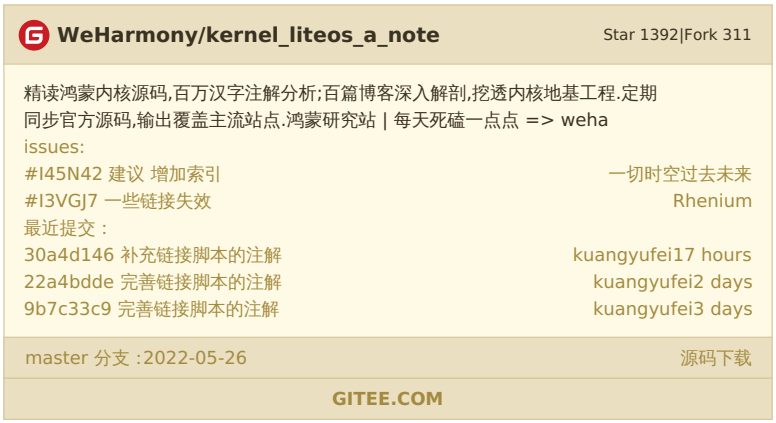
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

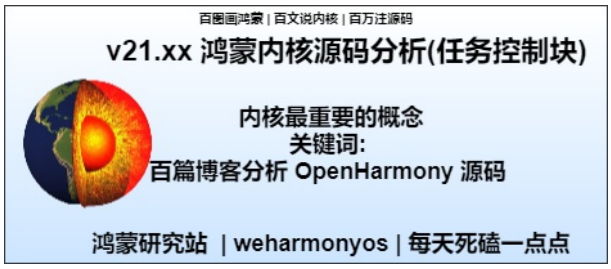


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

21_任务控制块篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

本篇说清楚任务的问题

在鸿蒙内核线程(thread)就是任务(task)，也可以叫作业。线程是对外的说法，对内就叫任务。跟王二毛一样， 在公司叫你王董，回到家里还有领导，就叫二毛啊。这多亲切。在鸿蒙内核是大量的task，很少看到thread，只出现在posix层。当一个东西理解就行。

读本篇之前建议先阅读

- v08.xx 鸿蒙内核源码分析(总目录) | 百万汉字注解 百篇博客分析 进程线程部分。 鸿蒙内核源码分析定位为深挖内核地基，构筑底层网图。就要见真身，剖真人。任务(LosTaskCB)原始真身如下，本篇一一剖析它，看看它的五脏六腑里到底是个啥。

```
typedef struct {
    VOID          *stackPointer;      /**< Task stack pointer */ //内核态栈指针，SP位置，切换任务时先保存上下文并指向TaskContext位置
    UINT16         taskStatus;        /**< Task status */   //各种状态标签，可以拥有多种标签，按位标识
    UINT16         priority;          /**< Task priority */  //任务优先级[0:31]，默认是31级
    UINT16         policy;    //任务的调度方式(三种 .. LOS_SCHED_RR )
    UINT16         timeSlice;         /**< Remaining time slice *///剩余时间片
    UINT32         stackSize;         /**< Task stack size */  //非用户模式下栈大小
    UINTPTR        topOfStack;        /**< Task stack top */  //非用户模式下的栈顶 bottom = top + size
    UINT32         taskID;            /**< Task ID */    //任务ID，任务池本质是一个大数组，ID就是数组的索引，默认 < 128
    TSK_ENTRY_FUNC  taskEntry;        /**< Task entrance function */ //任务执行入口函数
    VOID          *joinRetval;        /**< pthread adaption */ //用来存储join线程的返回值
    VOID          *taskSem;           /**< Task-held semaphore */ //task在等哪个信号量
    VOID          *taskMux;           /**< Task-held mutex */  //task在等哪把锁
    VOID          *taskEvent;         /**< Task-held event */  //task在等哪个事件
    UINTPTR        args[4];           /**< Parameter ,  of which the maximum number is 4 */ //入口函数的参数 例如 main (int argc，char *argv[])
    CHAR           taskName[OS_TCB_NAME_LEN]; /**< Task name */ //任务的名称
    LOS_DL_LIST    pendList;          /**< Task pend node */  //如果任务阻塞时就通过它挂到各种阻塞情况的链表上，比如OsTaskWait时
    LOS_DL_LIST    threadList;        /**< thread list */   //挂到所属进程的线程链表上
    SortLinkList   sortList;          /**< Task sortlink node */ //挂到cpu core 的任务执行链表上
    UINT32         eventMask;         /**< Event mask */   //事件屏蔽
    UINT32         eventMode;         /**< Event mode */   //事件模式
    UINT32         priBitMap;         /**< BitMap for recording the change of task priority ,  //任务在执行过程中优先级会经常变化，这个变量用来记录所有曾经
                                           the priority can not be greater than 31 */   //过的优先级，例如 ..01001011 曾经有过 0，1，3，6 优先级
    INT32          errorNo;           /**< Error Num */
    UINT32         signal;            /**< Task signal */ //任务信号类型，(SIGNAL_NONE，SIGNAL_KILL，SIGNAL_SUSPEND，SIGNAL_AFFI)
    sig_cb         sig;    //信号控制块，这里用于进程间通讯的信号，类似于 linux singal模块
```

```
#if (LOSCFG_KERNEL_SMP == YES)
    UINT16          currCpu;           /**< CPU core number of this task is running on */ //正在运行此任务的CPU内核号
    UINT16          lastCpu;           /**< CPU core number of this task is running on last time */ //上次运行此任务的CPU内核号
    UINT16          cpuAffiMask;       /**< CPU affinity mask ， support up to 16 cores */ //CPU亲和力掩码，最多支持16核，亲和力很重要，多核情况下尽量一
    UINT32          timerCpu;          /**< CPU core number of this task is delayed or pended */ //此任务的CPU内核号被延迟或挂起
#if (LOSCFG_KERNEL_SMP_TASK_SYNC == YES)
    UINT32          syncSignal;        /**< Synchronization for signal handling */ //用于CPU之间 同步信号
#endif
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;         /**< Schedule statistics */ //调度统计
#endif
#endif
    UINTPTR         userArea;   //使用区域，由运行时划定，根据运行态不同而不同
    UINTPTR         userMapBase;  //用户模式下的栈底位置
    UINT32          userMapSize;       /**< user thread stack size ，real size : userMapSize + USER_STACK_MIN_SIZE */
    UINT32          processID;         /**< Which belong process *///所属进程ID
    FutexNode       futex;    //实现快锁功能
    LOS_DL_LIST     joinList;          /**< join list */ //联结链表，允许任务之间相互释放彼此
    LOS_DL_LIST     lockList;          /**< Hold the lock list */ //拿到了哪些锁链表
    UINT32          waitID;            /**< Wait for the PID or GID of the child process */ //等待孩子的PID或GID进程
    UINT16          waitFlag;          /**< The type of child process that is waiting ， belonging to a group or parent ，
                                    a specific child process ， or any child process */
#if (LOSCFG_KERNEL_LITEIPC == YES)
    UINT32          ipcStatus;   //IPC状态
    LOS_DL_LIST     msgListHead;  //消息队列头结点，上面挂的都是任务要读的消息
    BOOL            accessMap[LOSCFG_BASE_CORE_TSK_LIMIT];//访问图，指的是task之间是否能访问的标识，LOSCFG_BASE_CORE_TSK_LIMIT 为任务池总数
#endif
} LosTaskCB;
```

结构体还是比较复杂，虽一一都做了注解，但还是不够清晰，没有模块化。这里把它分解成以下六大块逐一分析:

第一大块:多核CPU相关块

```
#if (LOSCFG_KERNEL_SMP == YES) //多CPU核支持
    UINT16          currCpu;           /**< CPU core number of this task is running on */ //正在运行此任务的CPU内核号
    UINT16          lastCpu;           /**< CPU core number of this task is running on last time */ //上次运行此任务的CPU内核号
    UINT16          cpuAffiMask;       /**< CPU affinity mask ， support up to 16 cores */ //CPU亲和力掩码，最多支持16核，亲和力很重要，多核情况下尽量一
    UINT32          timerCpu;          /**< CPU core number of this task is delayed or pended */ //此任务的CPU内核号被延迟或挂起
#if (LOSCFG_KERNEL_SMP_TASK_SYNC == YES)
    UINT32          syncSignal;        /**< Synchronization for signal handling */ //用于CPU之间 同步信号
#endif
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;         /**< Schedule statistics */ //调度统计
#endif
#endif
```

鸿蒙内核支持多CPU，谁都知道多CPU当然好，效率高，快嘛，但凡事有两面性，在享受一个东西带来好处的同时，也得承担伴随它一起带来的麻烦和风险。多核有哪些的好处和麻烦，这里不展开说，后续有专门的文章和视频说明。任务可叫线程，或叫作业。CPU就是做作业的，多个CPU就是有多个能做作业的，一个作业能一鼓作气做完吗？

答案是:往往不行，因为现实不允许，作业可以有N多，而CPU数量非常有限，所以经常做着A作业被老板打断让去做B作业。这老板就是调度算法。A作业被打断回来接着做的还会是原来那个CPU吗？

答案是:不一定。 变量cpuAffiMask叫CPU亲和力，它的作用是可以指定A的作业始终是同一个CPU来完成， 也可以随便，交给调度算法，分到谁就谁来，这方面可以不挑。

第二大块:栈空间

```
    VOID            *stackPointer;     /**< Task stack pointer */  //内核态栈指针，SP位置，切换任务时先保存上下文并指向TaskContext位置。
    UINT32          stackSize;         /**< Task stack size */     //内核态栈大小
    UINTPTR         topOfStack;        /**< Task stack top */      //内核态栈顶 bottom = top + size
```

```
    UINTPTR         userArea;       //使用区域，由运行时划定，根据运行态不同而不同
    UINTPTR         userMapBase;    //用户态下的栈底位置
    UINT32          userMapSize;    /**< user thread stack size ，real size : userMapSize + USER_STACK_MIN_SIZE */
```

进程分内核态进程和用户态进程，这个区别表现在线程(任务)层面上就是

- 内核态进程下创建的任务只有内核态的栈空间，`OsTaskStackAlloc` 负责内核态栈空间的分配。`OsTaskStackInit` 负责对内核态栈的初始化。

```
//任务栈初始化，非常重要的函数，返回任务上下文
LITE_OS_SEC_TEXT_INIT VOID *OsTaskStackInit(UINT32 taskID， UINT32 stackSize， VOID *topStack， BOOL initFlag)
{
    UINT32 index = 1;
    TaskContext *taskContext = NULL;

    if (initFlag == TRUE) {
        OsStackInit(topStack， stackSize);
    }
    taskContext = (TaskContext *)(((UINTPTR)topStack + stackSize) - sizeof(TaskContext));//上下文存放在栈的底部

    /* initialize the task context */ //初始化任务上下文
#ifdef LOSCFG_GDB
    taskContext->PC = (UINTPTR)OsTaskEntrySetupLoopFrame;
#else
    taskContext->PC = (UINTPTR)OsTaskEntry;//程序计数器，CPU首次执行task时跑的第一条指令位置
#endif
    taskContext->LR = (UINTPTR)OsTaskExit;  /* LR should be kept， to distinguish it's THUMB or ARM instruction */
    taskContext->resved = 0x0;
    taskContext->R[0] = taskID;             /* R0 */
    taskContext->R[index++] = 0x01010101;   /* R1， 0x01010101 : reg initialed magic word */ //0x55
    for (; index < GEN_REGS_NUM; index++) {//R2 - R12的初始化很有意思，为什么要这么做？
        taskContext->R[index] = taskContext->R[index - 1] + taskContext->R[1]; /* R2 - R12 */
    }//R[2]=R[2]<<1=0xAA

#ifdef LOSCFG_INTERWORK_THUMB // 16位模式
    taskContext->regPSR = PSR_MODE_SVC_THUMB; /* CPSR (Enable IRQ and FIQ interrupts， THUMNB-mode) */
#else //用于设置CPSR寄存器
    taskContext->regPSR = PSR_MODE_SVC_ARM;   /* CPSR (Enable IRQ and FIQ interrupts， ARM-mode) */
#endif

#if !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (index = 0; index < FP_REGS_NUM; index++) {
        taskContext->D[index] = 0xAAA0000000000000LL + index; /* D0 - D31 */
    }
    taskContext->regFPSCR = 0;
    taskContext->regFPEXC = FP_EN;
#endif

    return (VOID *)taskContext;
}
```

可以看到，初始化了任务上下文(TaskContext)，并将任务上下文放在了栈底，初始化任务上下文目的是为了在运行阶段先初始化R0~R15，CPSR寄存器的值。 保存上下文和恢复上下文都是针对寄存器值而言的。这个工作是在内核态的栈中完成的，也就是说一个任务的上下文就是保存在任务的内核态栈中。 `OsTaskStackInit` 的返回值将赋给 `stackPointer` ，即寄存器SP

- 用户态进程下创建的任务除了有内核态的栈空间外，还有用户态栈空间。

```
//用户任务使用栈初始化
LITE_OS_SEC_TEXT_INIT VOID OsUserTaskStackInit(TaskContext *context， TSK_ENTRY_FUNC taskEntry， UINTPTR stack)
{
    LOS_ASSERT(context != NULL);

#ifdef LOSCFG_INTERWORK_THUMB
    context->regPSR = PSR_MODE_USR_THUMB;
#else
    context->regPSR = PSR_MODE_USR_ARM;//工作模式:用户模式 + 工作状态:arm
#endif
```

```
    context->R[0] = stack;//栈指针给r0寄存器
    context->SP = TRUNCATE(stack， LOSCFG_STACK_POINT_ALIGN_SIZE);//异常模式所专用的堆栈 segment fault 输出回溯信息
    context->LR = 0;//保存子程序返回地址 例如 a call b ，在b中保存 a地址
    context->PC = (UINTPTR)taskEntry;//入口函数
}
```

注意看里面的内容用户栈的初始化时修改了任务的上下文内容，任务的上下文内容是始终保存在内核栈中，注意这个不要搞混了。 `OsUserTaskStackInit` 只是修改上下文地址中的内容。 `context->SP` 的值被修改了，这个修改意味着任务被调度后首先是恢复上下文，即要重置SP寄存器的值，SP的值将被变成context->SP，由此就指向了用户栈空间运行 `context->PC` 也被改变了，这意味着入口地址(代码段位置)也改变了。 `context->LR` 默认是为0，不跳转到任务地方。 在后续每次调度上下文切换过程中，context的内容将不断的变化。

第三大块:资源竞争/同步

```
    VOID          *taskSem;          /**< Task-held semaphore */ //task在等哪个信号量
    VOID          *taskMux;          /**< Task-held mutex */  //task在等哪把锁
    VOID          *taskEvent;        /**< Task-held event */  //task在等哪个事件
    UINT32         eventMask;         /**< Event mask */   //事件屏蔽
    UINT32         eventMode;         /**< Event mode */   //事件模式
    FutexNode      futex;    //实现快锁功能
    LOS_DL_LIST    joinList;          /**< join list */ //联结链表，允许任务之间相互释放彼此
    LOS_DL_LIST    lockList;          /**< Hold the lock list */ //拿到了哪些锁链表
    UINT32         signal;           /**< Task signal */ //任务信号类型，(SIGNAL_NONE，SIGNAL_KILL，SIGNAL_SUSPEND，SIGNAL_AFFI)
    sig_cb         sig;
```

公司的资源是有限的，CPU自己也是公司的资源，除了它还有其他的设备，比如做作业用的黑板，用户A，B，C都可能用到，狼多肉少，咋搞？

互斥量(taskMux，futex)能解决这个问题，办事前先拿锁，拿到了锁的爽了，没有拿到的就需要排队，在lockList上排队，注意lockList是个双向链表，它是内核最重要的结构体，开篇就提过，没印象的看 (双向链表篇)，上面挂都是等锁进房间的西门大官人。这是互斥量的原理，解决任务间资源紧张的竞争性问题。

另外一个是用于任务的同步的信号量(sig_cb)，任务和任务之间是会有关联的，现实生活中公司的A，B用户之间本身有业务往来的正常，CPU在帮B做作业的时候发现前置条件是需要A完成某项作业才能进行，这时B就需要主动让出CPU先办完A的事。这就是信号量的原理，解决的是任务间的同步问题。

第四大块:任务调度

前面说过了作业N多，做作业的只有几个人，单核CPU等于只有一个人干活。那要怎么分配CPU，就需要调度算法。

```
    UINT16         taskStatus;        /**< Task status */   //各种状态标签，可以拥有多种标签，按位标识
    UINT16         priority;         /**< Task priority */  //任务优先级[0:31]，默认是31级
    UINT16         policy;    //任务的调度方式(三种 .. LOS_SCHED_RR )
    UINT16         timeSlice;         /**< Remaining time slice *///剩余时间片
    CHAR           taskName[OS_TCB_NAME_LEN]; /**< Task name */ //任务的名称
    LOS_DL_LIST    pendList;          /**< Task pend node */  //如果任务阻塞时就通过它挂到各种阻塞情况的链表上，比如OsTaskWait时
    LOS_DL_LIST    threadList;        /**< thread list */   //挂到所属进程的线程链表上
    SortLinkList   sortList;          /**< Task sortlink node */ //挂到cpu core 的任务执行链表上
```

是简单的先来后到(FIFO)吗？ 当然也支持这个方式。鸿蒙内核用的是抢占式调度(policy)，就是可以插队，比优先级(priority)大小，[0，31]级，数字越大的优先级越低，跟考试一样，排第一才是最牛的。

鸿蒙排0的最牛! 想也想得到内核的任务优先级都是很高的，比如资源回收任务排第5，定时器任务排第0。够牛了吧。普通老百姓排多少呢？默认28级，惨!!!

另外任务有时间限制timeSlice，叫时间片，默认20ms，用完了会给你重置，发起重新调度，找出优先级高的执行，阻塞的任务(比如没拿到锁的，等信号量同步的，等读写消息队列的)都挂到pendList上，方便管理。

第五大块:任务间通讯

```
#if (LOSCFG_KERNEL_LITEIPC == YES)
    UINT32         ipcStatus;   //IPC状态
    LOS_DL_LIST    msgListHead;  //消息队列头结点，上面挂的都是任务要读的消息
    BOOL           accessMap[LOSCFG_BASE_CORE_TSK_LIMIT];//访问图，指的是task之间是否能访问的标识，LOSCFG_BASE_CORE_TSK_LIMIT 为任务池总数
#endif
```

这个很重要，解决任务间通讯问题，要知道进程负责的是资源的管理功能，什么意思？就是它不并负责内容的生产和消费，它只负责管理确保你的内容到达率和完整性。生产者和消费者始终是任务。进程管了哪些东西系列篇有专门的文章，请自行翻看。

liteipc是鸿蒙专有的通讯消息队列实现。简单说它是基于文件的，而传统的ipc消息队列是基于内存的。有什么区别也不在这里讨论，已有专门的文章分析。

第六大块:辅助工具

要知道任务对内核来说太重要了，是任务让CPU忙里忙外的，那中间出差错了怎么办，怎么诊断你问题出哪里了，就需要一些工具，比如死锁检测，比如占用CPU，内存监控 如下:

```
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) //死锁检测开关
    LockDep         lockDep;
#endif
#if (LOSCFG_KERNEL_SCHED_STATISTICS == YES) //调度统计开关，显然打开这个开关性能会受到影响，鸿蒙默认是关闭的
    SchedStat       schedStat;          /**< Schedule statistics */ //调度统计
#endif
```

以上就是任务的五脏六腑，看清楚它鸿蒙内核的影像会清晰很多!

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

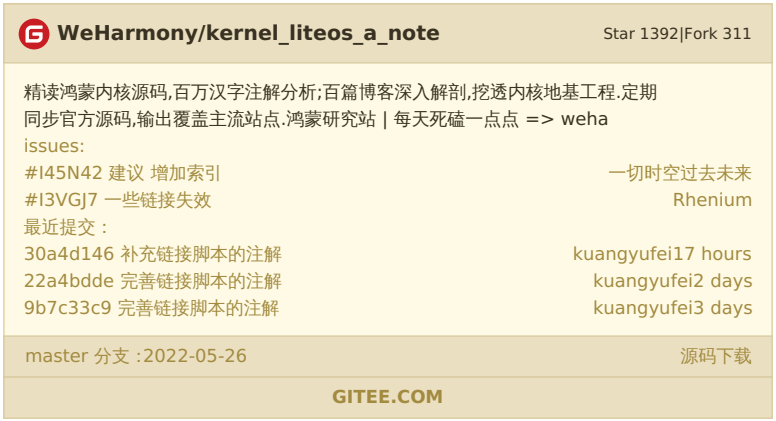
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

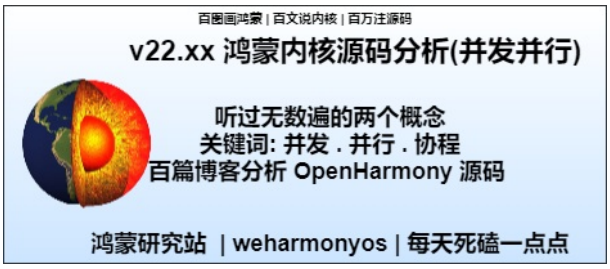


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

22_并发并行篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

本篇说清楚并发并行

读本篇之前建议先读系列进程/线程篇，会对并行并发更深的理解。

理解并发概念

- 并发（Concurrent）:多个线程在单个核心运行，同一时间只能一个线程运行，内核不停切换线程，看起来像同时运行，实际上是线程被高速的切换。
- 通俗好理解的比喻就是高速单行道，单行道指的是CPU的核数，跑的车就是线程(任务)，进程就是管理车的公司，一个公司可以有很多台车。并发和并行跟CPU的核数有关。车道上同时只能跑一辆车，但因为指挥系统很牛，够快，在毫秒级内就能换车跑，人根本感知不到切换。所以外部的感知会是同时在进行，实现了微观上的串行，宏观上的并行。
- 线程切换的本质是CPU要换场地上班，去哪里上班由哪里提供场地，那个场地就是任务栈，每个任务栈中保存了上班的各种材料，来了就行立马干活。那些材料就是任务上下文。简单的说就是上次活干到那里了，回来继续接着干。上下文由任务栈自己保存，CPU不管的，它来了只负责任务交过来的材料，材料显示去哪里搬砖它就去哪里搬砖。

记住一个单词就能记住并行并发的区别， 发单，发单(并发单行)。

理解并行概念

并行（Parallel）每个线程分配给独立的CPU核心，线程真正的同时运行。

通俗好理解的比喻就是高速多行道，实现了微观和宏观上同时进行。 并行当然是快，人多了干活就不那么累，但干活人多了必然会带来人多的管理问题，会把问题变复杂，请想想会出现哪些问题？

理解协程概念

这里说下协程，例如go语言是有协程支持的，其实协程跟内核层没有关系，是应用层的概念。是在线程之上更高层的封装，用通俗的比喻来说就是在车内另外搞了几条车道玩。其对内核来说没有新东西，内核只负责车的调度，至于车内你想怎么弄那是应用程序自己的事。本质的区别是CPU根本没有换地方上班(没有被调度)，而并发/并行都是换地方上班了。

内核如何描述CPU

```
typedef struct {
    SortLinkAttribute taskSortLink;            /* task sort link */ //每个CPU core 都有一个task排序链表
    SortLinkAttribute swtmrSortLink;           /* swtmr sort link */ //每个CPU core 都有一个定时器排序链表

    UINT32 idleTaskID;                      /* idle task id */  //空闲任务ID 见于 OsIdleTaskCreate
    UINT32 taskLockCnt;                     /* task lock flag */ //任务锁的数量，当 > 0 的时候，需要重新调度了
    UINT32 swtmrHandlerQueue;                   /* software timer timeout queue id */ //软时钟超时队列句柄
    UINT32 swtmrTaskID;                     /* software timer task id */ //软时钟任务ID

    UINT32 schedFlag;                       /* pending scheduler flag */ //调度标识 INT_NO_RESCH INT_PEND_RESCH
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 excFlag;                         /* cpu halt or exc flag */ //CPU处于停止或运行的标识
#endif
} Percpu;

Percpu g_percpu[LOSCFG_KERNEL_CORE_NUM];//全局CPU数组
```

这是内核对CPU的描述，主要是两个排序链表，一个是任务的排序，一个是定时器的排序。什么意思？ 在系列篇中多次提过，任务是内核的调度单元，注意可不是进程，虽然调度也需要进程参与，也需要切换进程，切换用户空间。但调度的核心是切换任务，每个任务的代码指令才是CPU的粮食，它吃的是一条条的指令。每个任务都必须指定取粮地址(即入口函数)。

另外还有一个东西能提供入口函数，就是定时任务。很重要也很常用，没它某宝每晚9点的准时秒杀实现不了。在内核每个CPU都有自己独立的任务和定时器链表。

每次Tick的到来，处理函数会去扫描这两个链表，看有没有定时器超时的任务需要执行，有则立即执行定时任务，定时任务是所有任务中优先级最高的，0号优先级，在系列篇中有专门讲定时器任务，可自行翻看。

LOSCFG_KERNEL_SMP

```
# if (LOSCFG_KERNEL_SMP == YES)
# define LOSCFG_KERNEL_CORE_NUM                 LOSCFG_KERNEL_SMP_CORE_NUM //多核情况下支持的CPU核数
# else
# define LOSCFG_KERNEL_CORE_NUM                 1 //单核配置
# endif
```

多CPU核的操作系统有3种处理模式(SMP+AMP+BMP) 鸿蒙实现的是 SMP 的方式

- 非对称多处理（Asymmetric multiprocessing，AMP）每个CPU内核运行一个独立的操作系统或同一操作系统的独立实例（instantiation）。
- 对称多处理（Symmetric multiprocessing，SMP）一个操作系统的实例可以同时管理所有CPU内核，且应用并不绑定某一个内核。
- 混合多处理（Bound multiprocessing，BMP）一个操作系统的实例可以同时管理所有CPU内核，但每个应用被锁定于某个指定的核心。

宏LOSCFG_KERNEL_SMP表示对多CPU核的支持，鸿蒙默认是打开LOSCFG_KERNEL_SMP的。

多CPU核支持

鸿蒙内核对CPU的操作见于 los_mp.c ，因文件不大，这里把代码都贴出来了。

```
#if (LOSCFG_KERNEL_SMP == YES)
//给参数CPU发送调度信号
VOID LOS_MpSchedule(UINT32 target)//target每位对应CPU core
{
    UINT32 cpuid = ArchCurrCpuid();
    target &= ~(1U << cpuid);//获取除了自身之外的其他CPU
    HalIrqSendIpi(target， LOS_MP_IPI_SCHEDULE);//向目标CPU发送调度信号，核间中断(Inter-Processor Interrupts)，IPI
}
//硬中断唤醒处理函数
VOID OsMpWakeHandler(VOID)
{
    /* generic wakeup ipi， do nothing */
}
//硬中断调度处理函数
VOID OsMpScheduleHandler(VOID)
{//将调度标志设置为与唤醒功能不同，这样就可以在硬中断结束时触发调度程序。
    /*
    * set schedule flag to differ from wake function，
```

```
    * so that the scheduler can be triggered at the end of irq。
    */
    OsPercpuGet()->schedFlag = INT_PEND_RESCH;//给当前Cpu贴上调度标签
}
//硬中断暂停处理函数
VOID OsMpHaltHandler(VOID)
{
    (VOID)LOS_IntLock();
    OsPercpuGet()->excFlag = CPU_HALT;//让当前Cpu停止工作

    while (1) {}//陷入空循环，也就是空闲状态
}
//MP定时器处理函数， 递归检查所有可用任务
VOID OsMpCollectTasks(VOID)
{
    LosTaskCB *taskCB = NULL;
    UINT32 taskID = 0;
    UINT32 ret;

    /* recursive checking all the available task */
    for (; taskID <= g_taskMaxNum; taskID++) { //递归检查所有可用任务
        taskCB = &g_taskCBArray[taskID];

        if (OsTaskIsUnused(taskCB) || OsTaskIsRunning(taskCB)) {
            continue;
        }

        /* 虽然任务状态不是原子的，但此检查可能成功，但无法完成删除，此删除将在下次运行之前处理
        * though task status is not atomic， this check may success but not accomplish
        * the deletion; this deletion will be handled until the next run。
        */
        if (taskCB->signal & SIGNAL_KILL) {//任务收到被干掉信号
            ret = LOS_TaskDelete(taskID);//干掉任务，回归任务池
            if (ret != LOS_OK) {
                PRINT_WARN("GC collect task failed err:0x%x\n"， ret);
            }
        }
    }
}
//MP(multiprocessing) 多核处理器初始化
UINT32 OsMpInit(VOID)
{
    UINT16 swtmrId;

    (VOID)LOS_SwtmrCreate(OS_MP_GC_PERIOD， LOS_SWTMR_MODE_PERIOD， //创建一个周期性，持续时间为 100个tick的定时器
                          (SWTMR_PROC_FUNC)OsMpCollectTasks， &swtmrId， 0);//OsMpCollectTasks为超时回调函数
    (VOID)LOS_SwtmrStart(swtmrId);//开始定时任务

    return LOS_OK;
}
#endif
```

代码一一都加上了注解，这里再一一说明下:

1.OsMpInit

多CPU核的初始化， 多核情况下每个CPU都有各自的编号， 内核有分成主次CPU， 0号默认为主CPU， OsMain()由主CPU执行，被汇编代码调用。 初始化只开了个定时任务，只干一件事就是回收不用的任务。回收的条件是任务是否收到了被干掉的信号。 例如shell命令 kill 9 14 ，意思是干掉14号线程的信号，这个信号会被线程保存起来。 可以选择自杀也可以等着被杀。 这里要注意，鸿蒙有两种情况下任务不能被干掉， 一种是系统任务不能被干掉的， 第二种是正在运行状态的任务。

2.次级CPU的初始化

同样由汇编代码调用，通过以下函数执行，完成每个CPU核的初始化

```
//次级CPU初始化，本函数执行的次数由次级CPU的个数决定。 例如:在四核情况下，会被执行3次， 0号通常被定义为主CPU 执行main
LITE_OS_SEC_TEXT_INIT VOID secondary_cpu_start(VOID)
{
```

```
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 cpuid = ArchCurrCpuid();

    OsArchMmuInitPerCPU();//每个CPU都需要初始化MMU

    OsCurrTaskSet(OsGetMainTask());//设置CPU的当前任务

    /* increase cpu counter */
    LOS_AtomicInc(&g_ncpu); //统计CPU的数量

    /* store each core's hwid */
    CPU_MAP_SET(cpuid， OsHwIDGet());//存储每个CPU的 hwid
    HalIrqInitPercpu(); //CPU硬件中断初始化

    OsCurrProcessSet(OS_PCB_FROM_PID(OsGetKernelInitProcessID())); //设置内核进程为CPU进程
    OsSwtmrInit();  //定时任务初始化，每个CPU维护自己的定时器队列
    OsIdleTaskCreate(); //创建空闲任务，每个CPU维护自己的任务队列
    OsStart(); //本CPU正式启动在内核层的工作
    while (1) {
        __asm volatile("wfi");//wait for Interrupt 等待中断，即下一次中断发生前都在此hold住不干活
    }//类似的还有 WFE: wait for Events 等待事件，即下一次事件发生前都在此hold住不干活
#endif
}
```

可以看出次级CPU有哪些初始化步骤:

- 初始化MMU，OsArchMmuInitPerCPU
- 设置当前任务 OsCurrTaskSet
- 初始化硬件中断 HalIrqInitPercpu
- 初始化定时器队列 OsSwtmrInit
- 创建空任务 OsIdleTaskCreate， 外面没有任务的时CPU就待在这个空任务里自己转圈圈。
- 开始自己的工作流程 OsStart，正式开始工作，跑任务

多CPU核还有哪些问题？

- CPU之间抢资源的情况要怎么处理？
- CPU之间通讯(也叫核间通讯)怎么解决？
- 如果确保两个CPU不会同时执行同一个任务？
- 汇编代码如何实现对各CPU的调动

请前往系列篇或直接前往内核注解代码查看。这里不再做说明。

百文说内核 | 抓住主脉络

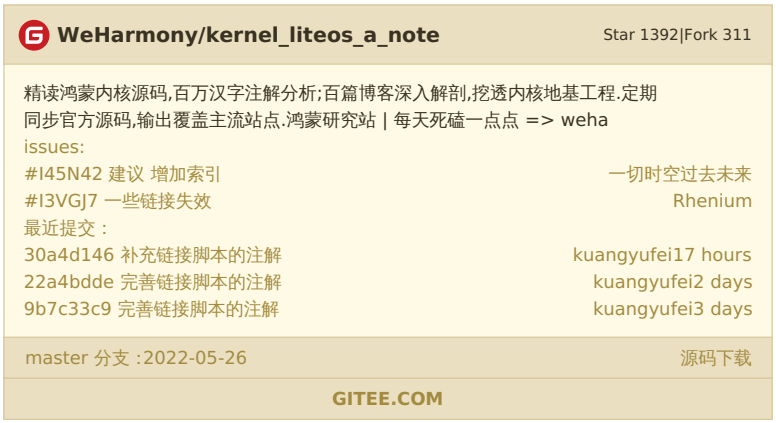
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

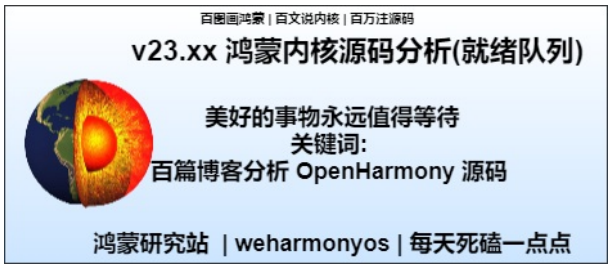


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

23_就绪队列篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

为何单独讲调度队列？

鸿蒙内核代码中有两个源文件是关于队列的，一个是用于调度的队列，另一个是用于线程间通讯的IPC队列。

IPC队列后续有专门的博文讲述，这两个队列的数据结构实现采用的都是双向循环链表，再说一遍LOS_DL_LIST实在是太重要了，是理解鸿蒙内核的关键，说是最重要的代码一点也不为过，源码出现在 sched_sq模块，说明是用于任务的调度的，sched_sq模块只有两个文件，另一个los_sched。c就是调度代码。

涉及函数

功能分类	接口名	描述
创建队列	OsPriQueueInit	创建了32个就绪队列
获取最高优先级队列	OsPriQueueTop	查最高优先级任务
从头部入队列	OsPriQueueEnqueueHead	从头部插入某个就绪队列
从尾部入队列	OsPriQueueEnqueue	默认是从尾部插入某个就绪队列
出队列	OsPriQueueDequeue	从最高优先级的就绪队列中删除
	OsPriQueueProcessDequeue	从进程队列中删除
	OsPriQueueProcessSize	用进程查队列中元素个数
	OsPriQueueSize	用任务查队列中元素个数
	OsTaskPriQueueTop	查最高优先级任务
	OsDequeEmptySchedMap	进程出列
	OsGetTopTask	获取被调度选择的task

鸿蒙内核进程和线程各有32个就绪队列，进程队列用全局变量存放， 创建进程时入队， 任务队列放在进程的threadPriQueueList中。

映射张大爷的故事：就绪队列就是在外面排队的32个通道，按优先级0-31依次排好，张大爷的办公室有个牌子，类似打篮球的记分牌，一共32个，一字排开，队列里有人时对应的牌就是1，没有就是0 ，这样张大爷每次从0位开始看，看到的第一个1那就是最高优先级的那个人。办公室里的记分牌就是位图调度器。

位图调度器

```
//* 0x80000000U = 10000000000000000000000000000000(32位，1是用于移位的，设计之精妙，点赞)
#define PRIQUEUE_PRIOR0_BIT   0x80000000U
LITE_OS_SEC_BSS LOS_DL_LIST *g_priQueueList = NULL; //所有的队列 原始指针
LITE_OS_SEC_BSS UINT32 g_priQueueBitmap; // 位图调度
```

整个los_priqueue。c就只有两个全部变量，一个是 LOS_DL_LIST *g_priQueueList 是32个进程就绪队列的头指针，在就绪队列中会讲另一个UINT32 g_priQueueBitmap 估计很多人会陌生，是一个32位的变量，叫位图调度器。怎么理解它呢？

鸿蒙系统的调度是抢占式的，task分成32个优先级，如何快速的知道哪个队列是空的，哪个队列里有任务需要一个标识，而且要极高效的实现？答案是:位图调度器。 系列篇已有专门讲位图管理的文章，自行翻看。简单说就是一个变量的位来标记对应队列中是否有任务，在位图调度下，任务优先级的值越小则代表具有越高的优先级，每当需要进行调度时，从最低位向最高位查找出第一个置 1 的位的所在位置，即为当前最高优先级，然后从对应优先级就绪队列获得相应的任务控制块，整个调度器的实现复杂度是 O(1)，即无论任务多少，其调度时间是固定的。

进程就绪队列机制

CPU执行速度是很快的，其运算速度和内存的读写速度是数量级的差异，与硬盘的读写更是指数级。 鸿蒙内核默认一个时间片是 10ms， 资源很宝贵，它不断在众多任务中来回的切换，所以绝不能让CPU等待任务，CPU时间很宝贵，没准备好的任务不要放进来。这就是进程和线程就绪队列的机制，一共有32个任务就绪队列，因为线程的优先级是默认32个， 每个队列中放同等优先级的task。 队列初始化做了哪些工作？详细看代码

```
#define OS_PRIORITY_QUEUE_NUM 32

//内部队列初始化
UINT32 OsPriQueueInit(VOID)
{
    UINT32 priority;

    /* system resident resource *///常驻内存
    g_priQueueList = (LOS_DL_LIST *)LOS_MemAlloc(m_aucSysMem0， (OS_PRIORITY_QUEUE_NUM * sizeof(LOS_DL_LIST)));//分配32个队列头节点
```

```
    if (g_priQueueList == NULL) {
        return LOS_NOK;
    }

    for (priority = 0; priority < OS_PRIORITY_QUEUE_NUM; ++priority) {
        LOS_ListInit(&g_priQueueList[priority]);//队列初始化，前后指针指向自己
    }
    return LOS_OK;
}
```

因TASK 有32个优先级，在初始化时内核一次性创建了32个双向循环链表，每种优先级都有一个队列来记录就绪状态的tasks的位置，g_priQueueList分配的是一个连续的内存块，存放了32个双向链表

几个常用函数

还是看入队和出队的源码吧，注意bitmap的变化！

从代码中可以知道，调用了LOS_ListTailInsert，注意是从循环链表的尾部插入的，也就是同等优先级的TASK被排在了最后一个执行，只要每次都是从尾部插入，就形成了一个按顺序执行的队列。鸿蒙内核的设计可谓非常巧妙，用极少的代码，极高的效率实现了队列功能。

```
VOID OsPriQueueEnqueue(LOS_DL_LIST *priQueueList， UINT32 *bitMap， LOS_DL_LIST *priqueueItem， UINT32 priority)
{
    /*
     * Task control blocks are inited as zero。 And when task is deleted，
     * and at the same time would be deleted from priority queue or
     * other lists， task pend node will restored as zero。
     */
    LOS_ASSERT(priqueueItem->pstNext == NULL);

    if (LOS_ListEmpty(&priQueueList[priority])) {
        *bitMap |= PRIQUEUE_PRIOR0_BIT >> priority;//对应优先级位 置1
    }

    LOS_ListTailInsert(&priQueueList[priority]， priqueueItem);
}

VOID OsPriQueueEnqueueHead(LOS_DL_LIST *priQueueList， UINT32 *bitMap， LOS_DL_LIST *priqueueItem， UINT32 priority)
{
    /*
     * Task control blocks are inited as zero。 And when task is deleted，
     * and at the same time would be deleted from priority queue or
     * other lists， task pend node will restored as zero。
     */
    LOS_ASSERT(priqueueItem->pstNext == NULL);

    if (LOS_ListEmpty(&priQueueList[priority])) {
        *bitMap |= PRIQUEUE_PRIOR0_BIT >> priority;//对应优先级位 置1
    }

    LOS_ListHeadInsert(&priQueueList[priority]， priqueueItem);
}

VOID OsPriQueueDequeue(LOS_DL_LIST *priQueueList， UINT32 *bitMap， LOS_DL_LIST *priqueueItem)
{
    LosTaskCB *task = NULL;
    LOS_ListDelete(priqueueItem);

    task = LOS_DL_LIST_ENTRY(priqueueItem， LosTaskCB， pendList);
    if (LOS_ListEmpty(&priQueueList[task->priority])) {
        *bitMap &= ~(PRIQUEUE_PRIOR0_BIT >> task->priority);//队列空了，对应优先级位 置0
    }
}
```

同一个进程下的线程的优先级可以不一样吗？

请先想一下这个问题。

进程和线程是一对多的父子关系，内核调度的单元是任务(线程)，鸿蒙内核中任务和线程是一个东西，只是不同的身份。一个进程可以有多个线程，

线程又有各自独立的状态，那进程状态该怎么界定？例如：ProcessA 有 TaskA(阻塞状态) ，TaskB(就绪状态) 两个线程，ProcessA是属于阻塞状态还是就绪状态呢？

先看官方文档的说明后再看源码。

进程状态迁移说明：

- Init→Ready：

  进程创建或fork时，拿到该进程控制块后进入Init状态，处于进程初始化阶段，当进程初始化完成将进程插入调度队列，此时进程进入就绪状态。

- Ready→Running：

  进程创建后进入就绪态，发生进程切换时，就绪列表中最高优先级的进程被执行，从而进入运行态。若此时该进程中已无其它线程处于就绪态，则该进程从就绪列表删除，只处于运行态；若此时该进程中还有其它线程处于就绪态，则该进程依旧在就绪队列，此时进程的就绪态和运行态共存。

- Running→Pend：

  进程内所有的线程均处于阻塞态时，进程在最后一个线程转为阻塞态时，同步进入阻塞态，然后发生进程切换。

- Pend→Ready / Pend→Running：

  阻塞进程内的任意线程恢复就绪态时，进程被加入到就绪队列，同步转为就绪态，若此时发生进程切换，则进程状态由就绪态转为运行态。

- Ready→Pend：

  进程内的最后一个就绪态线程处于阻塞态时，进程从就绪列表中删除，进程由就绪态转为阻塞态。

- Running→Ready：

  进程由运行态转为就绪态的情况有以下两种：

  1。 有更高优先级的进程创建或者恢复后，会发生进程调度，此刻就绪列表中最高优先级进程变为运行态，那么原先运行的进程由运行态变为就绪态。 2。 若进程的调度策略为SCHED_RR，且存在同一优先级的另一个进程处于就绪态，则该进程的时间片消耗光之后，该进程由运行态转为就绪态，另一个同优先级的进程由就绪态转为运行态。

- Running→Zombies：

  当进程的主线程或所有线程运行结束后，进程由运行态转为僵尸态，等待父进程回收资源。

从文档中可知，一个进程是可以两种状态共存的。

```
UINT16          processStatus;              /**< [15:4] process Status; [3:0] The number of threads currently
                                            running in the process */

processCB->processStatus &= ~(status | OS_PROCESS_STATUS_PEND);//取反后的与位运算
processCB->processStatus |= OS_PROCESS_STATUS_READY;//或位运算
```

一个变量存两种状态，怎么做到的？答案还是 按位保存啊。还记得上面的位图调度 g_priQueueBitmap吗，那可是存了32种状态的。其实这在任何一个系统的内核源码中都很常见，类似的还有 左移 <<，右移 >>等等

继续说进程和线程的关系，线程的优先级必须和进程一样吗？他们可以不一样吗？答案是：当然不一样，否则怎么会有设置task优先级的函数。其实task有专门的bitmap来记录它曾经有过的优先级记录， 比如在调度过程中如果遇到阻塞，内核往往会提高持有锁的task的优先级，让它能以最大概率被下一轮调度选中而快速释放锁资源。

task调度器

真正让CPU工作的是task，进程只是个装task的容器，task有任务栈空间，进程结构体LosProcessCB 有一个这样的定义。看名字就知道了，那是跟调度相关的。

```
UINT32          threadScheduleMap;          /**< The scheduling bitmap table for the thread group of the
                                            process */
LOS_DL_LIST     threadPriQueueList[OS_PRIORITY_QUEUE_NUM]; /**< The process's thread group schedules the
                                                priority hash table */
```

咋一看怎么进程的结构体里也有32个队列，其实这就是task的就绪状态队列。threadScheduleMap就是进程自己的位图调度器。具体看进程入队和出

队的源码。调度过程是先去进程就绪队列里找最高优先级的进程，然后去该进程找最高优先级的线程来调度。具体看笔者认为的内核最美函数OsGetTopTask，能欣赏到他的美就读懂了就绪队列是怎么管理的。

```
LITE_OS_SEC_TEXT_MINOR LosTaskCB *OsGetTopTask(VOID)
{
    UINT32 priority， processPriority;
    UINT32 bitmap;
    UINT32 processBitmap;
    LosTaskCB *newTask = NULL;
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 cpuid = ArchCurrCpuid();
#endif
    LosProcessCB *processCB = NULL;
    processBitmap = g_priQueueBitmap;
    while (processBitmap) {
        processPriority = CLZ(processBitmap);
        LOS_DL_LIST_FOR_EACH_ENTRY(processCB， &g_priQueueList[processPriority]， LosProcessCB， pendList) {
            bitmap = processCB->threadScheduleMap;
            while (bitmap) {
                priority = CLZ(bitmap);
                LOS_DL_LIST_FOR_EACH_ENTRY(newTask， &processCB->threadPriQueueList[priority]， LosTaskCB， pendList) {
#if (LOSCFG_KERNEL_SMP == YES)
                    if (newTask->cpuAffiMask & (1U << cpuid)) {
#endif
                        newTask->taskStatus &= ~OS_TASK_STATUS_READY;
                        OsPriQueueDequeue(processCB->threadPriQueueList，
                                          &processCB->threadScheduleMap，
                                          &newTask->pendList);
                        OsDequeEmptySchedMap(processCB);
                        goto OUT;
#if (LOSCFG_KERNEL_SMP == YES)
                    }
#endif
                }
                bitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - priority - 1));
            }
        }
        processBitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - processPriority - 1));
    }

OUT:
    return newTask;
}
```

映射张大爷的故事：张大爷喊到张全蛋时进场时表演时，张全蛋要决定自己的哪个节目先表演，也要查下他的清单上优先级，它同样也有个张大爷同款记分牌，就这么简单。

百文说内核 | 抓住主脉络

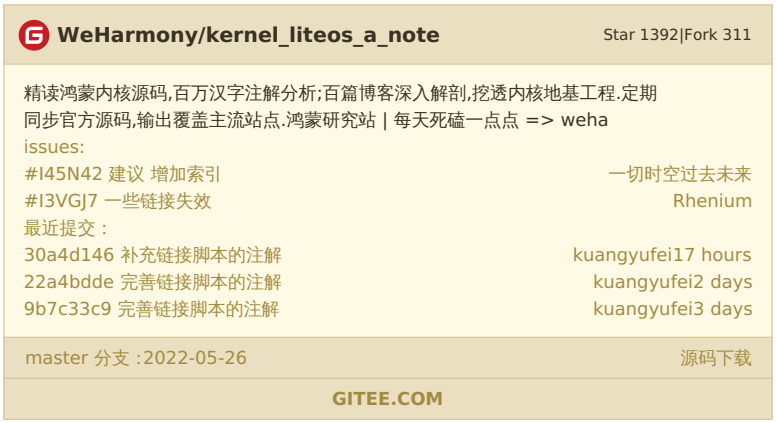
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

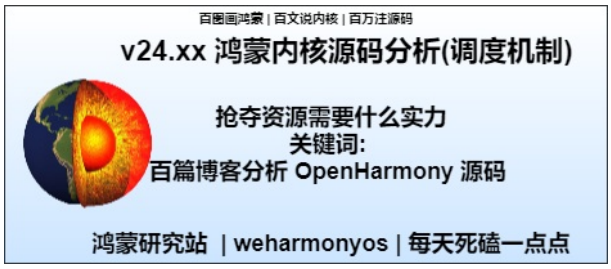


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

24_调度机制篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

为什么学个东西要学那么多的概念？

鸿蒙的内核中 Task 和 线程 在广义上可以理解为是一个东西，但狭义上肯定会有区别，区别在于管理体系的不同，Task是调度层面的概念，线程是进程层面概念。比如 main() 函数中首个函数 OsSetMainTask(); 就是设置启动任务，但此时啥都还没开始呢，Kprocess 进程都没创建，怎么会有大家一般意义上所理解的线程呢。狭义上的后续有 鸿蒙内核源码分析(启动过程篇) 来说明。不知道大家有没有这种体会，学一个东西的过程中要接触很多新概念，尤其像 Java/android 的生态，概念贼多，很多同学都被绕在概念中出不来，痛苦不堪。那问题是为什么需要这么多的概念呢？

举个例子就明白了：

假如您去深圳参加一个面试老板问你哪里人？你会说是 江西人，湖南人。。。 而不会说是张家村二组的张全蛋，这样还谁敢要你。但如果你参加同乡会别人问你同样问题，你不会说是来自东北那旮旯的，却反而要说张家村二组的张全蛋。明白了吗？张全蛋还是那个张全蛋，但因为场景变了，您的说法就得必须跟着变，否则没法愉快的聊天。程序设计就是源于生活，归于生活，大家对程序的理解就是要用生活中的场景去打比方，更好的理解概念。

那在内核的调度层面，咱们只说 `task`， `task` 是内核调度的单元，调度就是围着它转。

进程和线程的状态迁移图

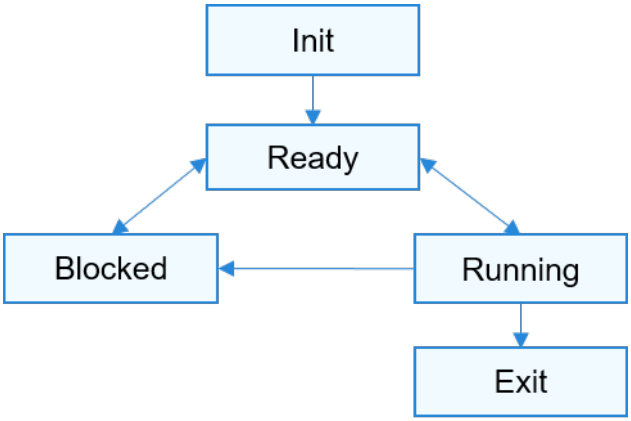
先看看task从哪些渠道产生：

LOS_TaskCreate	los_task.c (kernel\base\core)
osShellPing	api_shell.c (net\lwip-2.1\enhancement\src)
BcacheSyncThreadInit	bcache.c (fs\vfs\bcache\src)
BcacheAsyncPrereadInit	bcache.c (fs\vfs\bcache\src)
OsConsoleBufInit	console.c (kernel\common)
OsSystemInitTaskCreate	los_config.c (kernel\common)
OsMemShowTaskCreate	los_config.c (kernel\common)
OsSwtmrTaskCreate	los_swtmr.c (kernel\base\core)
OsIdleTaskCreate	los_task.c (kernel\base\core)
OsCreateResourceFreeTask	los_task.c (kernel\base\core)
LOS_TaskCreate	los_task.h (kernel\include)
ShellTaskInit	shmsg.c (shell\full\src\base)
ShellEntryInit	shmsg.c (shell\full\src\base)
sys_thread_new	sys_arch.c (net\lwip-2.1\porting\src)
TelnetdTaskInit	telnet_loop.c (net\telnet\src)
osShellCmdCat	vfs_shellcmd.c (fs\vfs\vfs_cmd)
OsWatchTaskCreate	watch_shell.c (shell\full\src\cmds)

渠道很多，可能是shell 的一个命令，也可能由内核创建，更多的是大家编写应用程序new出来的一个线程。

调度的内容task已经有了，那他们是如何被有序调度的呢？答案：是32个进程和线程就绪队列，各32个哈，为什么是32个，鸿蒙系统源码分析(总目录) 文章里有详细说明，自行去翻。这张进程状态迁移示意图一定要看明白。

注意:进程和线程的队列内的内容只针对就绪状态，其他状态内核并没有用队列去描述它，(线程的阻塞状态用的是pendlist链表)，因为就绪就意味着工作都准备好了就等着被调度到CPU来执行了。所以理解就绪队列很关键，有三种情况会加入就绪队列。



- Init→Ready：

  进程创建或fork时，拿到该进程控制块后进入Init状态，处于进程初始化阶段，当进程初始化完成将进程插入调度队列，此时进程进入就绪状态。

- Pend→Ready / Pend→Running：

  阻塞进程内的任意线程恢复就绪态时，进程被加入到就绪队列，同步转为就绪态，若此时发生进程切换，则进程状态由就绪态转为运行态。

- Running→Ready：

  进程由运行态转为就绪态的情况有以下两种：

- 有更高优先级的进程创建或者恢复后，会发生进程调度，此刻就绪列表中最高优先级进程变为运行态，那么原先运行的进程由运行态变为就绪态。
- 若进程的调度策略为SCHED_RR，且存在同一优先级的另一个进程处于就绪态，则该进程的时间片消耗光之后，该进程由运行态转为就绪态，另一个同优先级的进程由就绪态转为运行态。

谁来触发调度工作？

就绪队列让task各就各位，在其生命周期内不停的进行状态流转，调度是让task交给CPU处理，那又是什么让调度去工作的呢？它是如何被触发的？

笔者能想到的触发方式是以下四个：

- Tick(时钟管理)，类似于JAVA的定时任务，时间到了就触发。系统定时器是内核时间机制中最重要的一部分，它提供了一种周期性触发中断机制，即系统定时器以HZ（时钟节拍率）为频率自行触发时钟中断。当时钟中断发生时，内核就通过时钟中断处理程序OsTickHandler对其进行处理。鸿蒙内核默认是10ms触发一次，执行以下中断函数：

```
/*
 * Description : Tick interruption handler
 */
LITE_OS_SEC_TEXT VOID OsTickHandler(VOID)
{
    UINT32 intSave;

    TICK_LOCK(intSave);
    g_tickCount[ArchCurrCpuid()]++;
    TICK_UNLOCK(intSave);

#ifdef LOSCFG_KERNEL_VDSO
    OsUpdateVdsoTimeval();
#endif

#ifdef LOSCFG_KERNEL_TICKLESS
    OsTickIrqFlagSet(OsTicklessFlagGet());
#endif

#if (LOSCFG_BASE_CORE_TICK_HW_TIME == YES)
    HalClockIrqClear(); /* diff from every platform */
#endif

    OsTimesliceCheck();//时间片检查

    OsTaskScan(); /* task timeout scan *///任务扫描，发起调度

#if (LOSCFG_BASE_CORE_SWTMR == YES)
    OsSwtmrScan();//软时钟扫描检查
#endif
}
```

里面对任务进行了扫描，时间片到了或就绪队列有高或同级task， 会执行调度。

- 第二个是各种软硬中断，如何USB插拔，键盘，鼠标这些外设引起的中断，需要去执行中断处理函数。
- 第三个是程序主动中断，比如运行过程中需要申请其他资源，而主动让出控制权，重新调度。
- 最后一个是创建一个新进程或新任务后主动发起的抢占式调度，新进程会默认创建一个main task， task的首条指令(入口函数)就是我们上层程序的main函数，它被放在代码段的第一的位置。
- 哪些地方会申请调度？看一张图。

LOS_Schedule	los_sched_pri.h (kernel\base\include)
OsTimesliceCheck	los_timeslice.c (kernel\base\core)
OsTimesliceCheck	los_timeslice.c (kernel\base\core)
OsEventWrite	los_event.c (kernel\base\ipc)
LOS_SemPost	los_sem.c (kernel\base\ipc)
OsQueueOperate	los_queue.c (kernel\base\ipc)
OsTaskScan	los_task.c (kernel\base\core)
LOS_MuxUnlock	los_mux.c (kernel\base\ipc)
OsFutexWake	los_futex.c (kernel\base\ipc)
LOS_TaskCreate	los_task.c (kernel\base\core)
LOS_TaskResume	los_task.c (kernel\base\core)
OsFutexRequeue	los_futex.c (kernel\base\ipc)
OsSetProcessScheduler	los_process.c (kernel\base\core)
LiteIpcWrite	hm_liteipc.c (kernel\extended\liteipc)
LOS_TaskPriSet	los_task.c (kernel\base\core)
LOS_TaskUnlock	los_task.c (kernel\base\core)
LOS_TaskCpuAffiSet	los_task.c (kernel\base\core)
OsCopyProcess	los_process.c (kernel\base\core)
OsTaskSchedulerSetUnsafe	los_task.c (kernel\base\core)

这里提下图中的 OsCopyProcess()， 这是fork进程的主体函数，可以看出fork之后立即申请了一次调度。

```
LITE_OS_SEC_TEXT INT32 LOS_Fork(UINT32 flags， const CHAR *name， const TSK_ENTRY_FUNC entry， UINT32 stackSize)
{
    UINT32 cloneFlag = CLONE_PARENT | CLONE_THREAD | CLONE_VFORK | CLONE_FILES;
```

```
    if (flags & (~cloneFlag)) {
        PRINT_WARN("Clone dont support some flags!\n");
    }

    flags |= CLONE_FILES;
    return OsCopyProcess(cloneFlag & flags， name， (UINTPTR)entry， stackSize);
}

STATIC INT32 OsCopyProcess(UINT32 flags， const CHAR *name， UINTPTR sp， UINT32 size)
{
    UINT32 intSave， ret， processID;
    LosProcessCB *run = OsCurrProcessGet();

    LosProcessCB *child = OsGetFreePCB();
    if (child == NULL) {
        return -LOS_EAGAIN;
    }
    processID = child->processID;

    ret = OsForkInitPCB(flags， child， name， sp， size);
    if (ret != LOS_OK) {
        goto ERROR_INIT;
    }

    ret = OsCopyProcessResources(flags， child， run);
    if (ret != LOS_OK) {
        goto ERROR_TASK;
    }

    ret = OsChildSetProcessGroupAndSched(child， run);
    if (ret != LOS_OK) {
        goto ERROR_TASK;
    }

    LOS_MpSchedule(OS_MP_CPU_ALL);
    if (OS_SCHEDULER_ACTIVE) {
        LOS_Schedule();// 申请调度
    }

    return processID;

ERROR_TASK:
    SCHEDULER_LOCK(intSave);
    (VOID)OsTaskDeleteUnsafe(OS_TCB_FROM_TID(child->threadGroupID)， OS_PRO_EXIT_OK， intSave);
ERROR_INIT:
    OsDeInitPCB(child);
    return -ret;
}
```

原来创建一个进程这么简单，真的就是在COPY！

源码告诉你调度过程是怎样的

以上是需要提前了解的信息，接下来直接上源码看调度过程吧，文件就三个函数，主要就是这个了：

```
VOID OsSchedResched(VOID)
{
    LOS_ASSERT(LOS_SpinHeld(&g_taskSpin));//调度过程要上锁
    newTask = OsGetTopTask(); //获取最高优先级任务
    OsSchedSwitchProcess(runProcess， newProcess);//切换进程
    (VOID)OsTaskSwitchCheck(runTask， newTask);//任务检查
    OsCurrTaskSet((VOID*)newTask);//*设置当前任务
    if (OsProcessIsUserMode(newProcess)) {//判断是否为用户态，使用用户空间
        OsCurrUserTaskSet(newTask->userArea);//设置任务空间
    }
    /* do the task context switch */
    OsTaskSchedule(newTask， runTask); //切换CPU任务上下文，汇编代码实现
}
```

函数有点长，笔者留了最重要的几行，看这几行就够了，流程如下:

- 调度过程要自旋锁，多核情况下只能被一个CPU core 执行。 不允许任何中断发生， 没错，说的是任何事是不能去打断它，否则后果太严重了，这可是内核在切换进程和线程的操作啊。
- 在就绪队列里找个最高优先级的task
- 切换进程，就是task归属的那个进程设为运行进程，这里要注意，老的task和老进程只是让出了CPU指令执行权，其他都还在内存，资源也都没有释放。
- 设置新任务为当前任务
- 用户模式下需要设置task运行空间，因为每个task栈是不一样的。空间部分具体在系列篇内存中查看
- 是最重要的，切换任务上下文，参数是新老两个任务，一个要保存现场，一个要恢复现场。

什么是任务上下文？v08。xx 鸿蒙内核源码分析(总目录) 任务切换篇已有详细的描述，请自行翻看。

请读懂OsGetTopTask()

读懂OsGetTopTask()，就明白了就绪队列是怎么回事了。这里提下goto语句，几乎所有内核代码都会大量的使用goto语句，鸿蒙内核有617个goto远大于264个break，还有人说要废掉goto，你知道内核开发者青睐goto的真正原因吗？

```
LITE_OS_SEC_TEXT_MINOR LosTaskCB *OsGetTopTask(VOID)
{
    UINT32 priority， processPriority;
    UINT32 bitmap;
    UINT32 processBitmap;
    LosTaskCB *newTask = NULL;
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 cpuid = ArchCurrCpuid();
#endif
    LosProcessCB *processCB = NULL;
    processBitmap = g_priQueueBitmap;
    while (processBitmap) {
        processPriority = CLZ(processBitmap);
        LOS_DL_LIST_FOR_EACH_ENTRY(processCB， &g_priQueueList[processPriority]， LosProcessCB， pendList) {
            bitmap = processCB->threadScheduleMap;
            while (bitmap) {
                priority = CLZ(bitmap);
                LOS_DL_LIST_FOR_EACH_ENTRY(newTask， &processCB->threadPriQueueList[priority]， LosTaskCB， pendList) {
#if (LOSCFG_KERNEL_SMP == YES)
                    if (newTask->cpuAffiMask & (1U << cpuid)) {
#endif
                        newTask->taskStatus &= ~OS_TASK_STATUS_READY;
                        OsPriQueueDequeue(processCB->threadPriQueueList，
                                          &processCB->threadScheduleMap，
                                          &newTask->pendList);
                        OsDequeEmptySchedMap(processCB);
                        goto OUT;
#if (LOSCFG_KERNEL_SMP == YES)
                    }
#endif
                }
                bitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - priority - 1));
            }
        }
        processBitmap &= ~(1U << (OS_PRIORITY_QUEUE_NUM - processPriority - 1));
    }

OUT:
    return newTask;
}

#ifdef __cplusplus
#if __cplusplus
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很

重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。

- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

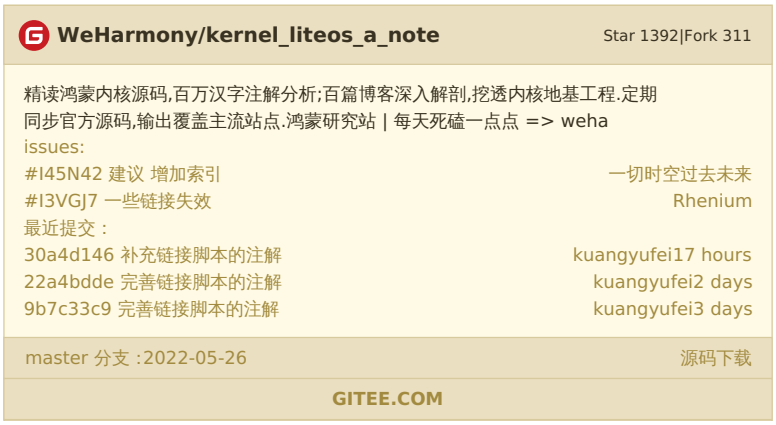
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

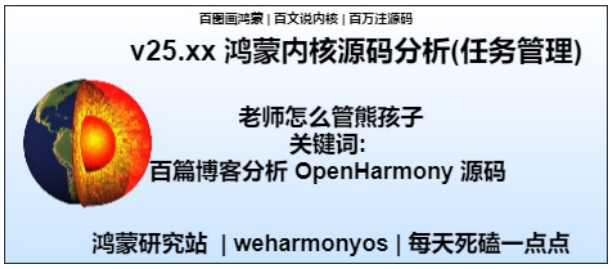


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

25_任务管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

任务即线程

在鸿蒙内核中，广义上可理解为一个任务就是一个线程

官方是怎么描述线程的

基本概念
从系统的角度看，线程是竞争系统资源的最小运行单元。线程可以使用或等待CPU、使用内存空间等系统资源，并独立于其它线程运行。

鸿蒙内核每个进程内的线程独立运行、独立调度，当前进程内线程的调度不受其它进程内线程的影响。

鸿蒙内核中的线程采用抢占式调度机制，同时支持时间片轮转调度和FIFO调度方式。

鸿蒙内核的线程一共有32个优先级(0-31)，最高优先级为0，最低优先级为31。

当前进程内高优先级的线程可抢占当前进程内低优先级线程，当前进程内低优先级线程必须在当前进程内高优先级线程阻塞或结束后才能得到调度。

线程状态说明：

初始化（Init）：该线程正在被创建。

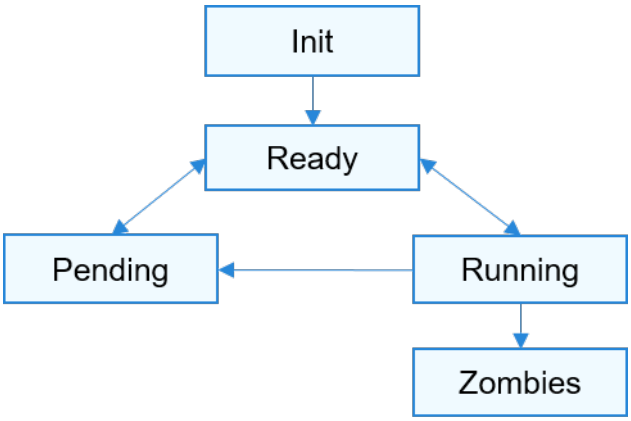
就绪（Ready）：该线程在就绪列表中，等待CPU调度。

运行（Running）：该线程正在运行。

阻塞（Blocked）：该线程被阻塞挂起。Blocked状态包括：pend(因为锁、事件、信号量等阻塞)、suspend（主动pend）、delay(延时阻塞)、pendtime(因为锁、事件、信号量时间等超时等待)。

退出（Exit）：该线程运行结束，等待父线程回收其控制块资源。

图 1 线程状态迁移示意图



注意官方文档说的是线程，没有提到task(任务)，但内核源码中却有大量 task代码，很少有线程(thread)代码 ，这是怎么回事？
其实在鸿蒙内核中， task就是线程， 初学者完全可以这么理解，但二者还是有区别，否则干嘛要分两个词描述。
会有什么区别？是管理上的区别，task是调度层面的概念，线程是进程层面的概念。 就像同一个人在不同的管理体系中会有不同的身份一样，一个男人既可以是 孩子，爸爸，丈夫，或者程序员，视角不同功能也会不同。

如何证明是一个东西，继续再往下看。

执行task命令

看shell task 命令的执行结果:

```
OHOS # task

  PID  PPID PGID       UID  Status CPUUSE10s  PName
    1    -1    1         0    Pend       0.0  init
    2    -1    2         0    Pend       0.1  KProcess
    3     1    1         0 Running       0.0  shell

  TID  PID Affi CPU       Status StackSize WaterLine    MEMUSE  TaskName
   20    1  0x3  -1        Delay    0x3000     0xb20    0x8d80  init
    1    2  0x1  -1         Pend    0x6000     0x5c8         0  Swt_Task
    2    2  0x3  -1         Pend    0x6000     0x2ac         0  system_wq
    4    2  0x1  -1        Delay    0x1000     0x268         0  oom_task
    5    2  0x2  -1         Pend    0x6000     0x2b4         0  Swt_Task
    7    2  0x3  -1         Pend    0x6000     0x3e4         0  SendToSer
    8    2  0x3  -1     PendTime    0x6000     0x604         0  tcpip_thread
    9    2  0x1  -1         Pend    0x6000     0x2cc         0  jffs2_gc_thread
   10    2  0x3  -1         Pend    0x3000     0x2bc         0  himci_Task
   11    2  0x3  -1         Pend    0x4000     0x3f0   0x14b8c  eth_irq_Task
   12    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_GIANT_Task
   13    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_NGIAN_ISOC_Task
   14    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_NGIAN_BULK_Task
   15    2  0x3  -1         Pend    0x6000     0x718     0xb34  USB_EXPLR_Task
   16    2  0x3  -1         Pend    0x6000     0x2dc         0  USB_CXFER_Task
   17    2  0x3  -1         Pend    0x6000     0x624         0  TouchEventHandler
   18    2  0x3  -1         Pend    0x6000     0x2c4         0  TouchGetEventTask
   19    2  0x3  -1         Pend    0x6000     0x2e4         0  TouchHandlerEventTask
    3    3  0x3  -1         Pend    0x3000     0xb18    0x5c8  shell
   21    3  0x3   1      Running    0x3000     0xecc   0x46e38  ShellTask
   22    3  0x3  -1         Pend    0x3000     0x924     0x450  ShellEntry
```


task命令 查出每个任务在生命周期内的运行情况，它运行的内存空间，优先级，时间片，入口执行函数，进程ID，状态等等信息，非常的复杂。这么复杂的信息就需要一个结构体来承载。而这个结构体就是 LosTaskCB(任务控制块)

对应张大爷的故事：task就是一个用户的节目清单里的一个节目，用户总清单就是一个进程，所以上面会有很多的节目。

task长得什么样子

说LosTaskCB之前先说下官方文档任务状态对应的 define，可以看出task和线程是一个东西。

```
#define OS_TASK_STATUS_INIT       0x0001U
#define OS_TASK_STATUS_READY      0x0002U
#define OS_TASK_STATUS_RUNNING    0x0004U
#define OS_TASK_STATUS_SUSPEND    0x0008U
#define OS_TASK_STATUS_PEND       0x0010U
#define OS_TASK_STATUS_DELAY      0x0020U
#define OS_TASK_STATUS_TIMEOUT    0x0040U
#define OS_TASK_STATUS_PEND_TIME  0x0080U
#define OS_TASK_STATUS_EXIT       0x0100U
```

LosTaskCB长什么样？抱歉，它确实有点长，但还是要全部贴出全貌。

```
typedef struct {
    VOID            *stackPointer;      /**< Task stack pointer | 内核栈指针位置(SP)  */
    UINT16          taskStatus;         /**< Task status | 各种状态标签，可以拥有多种标签，按位标识 */
    UINT16          priority;           /**< Task priority | 任务优先级[0:31],默认是31级  */
    UINT16          policy;    ///< 任务的调度方式(三种 .. LOS_SCHED_RR       LOS_SCHED_FIFO .. )
    UINT64          startTime;          /**< The start time of each phase of task | 任务开始时间  */
    UINT64          irqStartTime;       /**< Interrupt start time | 任务中断开始时间  */
    UINT32          irqUsedTime;        /**< Interrupt consumption time | 任务中断消耗时间  */
    UINT32          initTimeSlice;      /**< Task init time slice | 任务初始的时间片  */
    INT32           timeSlice;          /**< Task remaining time slice | 任务剩余时间片  */
    UINT32          waitTimes;          /**< Task delay time, tick number | 设置任务调度延期时间  */
    SortLinkList    sortList;           /**< Task sortlink node | 跟CPU捆绑的任务排序链表节点,上面挂的是就绪队列的下一个阶段,进入CPU要执行的任务队列  */
    UINT32          stackSize;          /**< Task stack size | 内核态栈大小,内存来自内核空间  */
    UINTPTR         topOfStack;         /**< Task stack top | 内核态栈顶 bottom = top + size */
    UINT32          taskID;             /**< Task ID | 任务ID，任务池本质是一个大数组，ID就是数组的索引，默认 < 128 */
    TSK_ENTRY_FUNC  taskEntry;          /**< Task entrance function | 任务执行入口地址 */
    VOID            *joinRetval;        /**< pthread adaption | 用来存储join线程的入口地址 */
    VOID            *taskMux;           /**< Task-held mutex | task在等哪把锁 */
    VOID            *taskEvent;         /**< Task-held event | task在等哪个事件 */
    UINTPTR         args[4];            /**< Parameter, of which the maximum number is 4 | 入口函数的参数 例如 main (int argc,char *argv[]) */
    CHAR            taskName[OS_TCB_NAME_LEN]; /**< Task name | 任务的名称 */
    LOS_DL_LIST     pendList;           /**< Task pend node | 如果任务阻塞时就通过它挂到各种阻塞情况的链表上,比如OsTaskWait时 */
    LOS_DL_LIST     threadList;         /**< thread list | 挂到所属进程的线程链表上 */
    UINT32          eventMask;          /**< Event mask | 任务对哪些事件进行屏蔽 */
    UINT32          eventMode;          /**< Event mode | 事件三种模式(LOS_WAITMODE_AND,LOS_WAITMODE_OR,LOS_WAITMODE_CLR) */
    UINT32          priBitMap;          /**< BitMap for recording the change of task priority,the priority can not be greater than 31
            | 任务在执行过程中优先级会经常变化，这个变量用来记录所有曾经变化过的优先级，例如 ..01001011 曾经有过 0,1,3,6 优先级 */
#ifdef LOSCFG_KERNEL_CPUP
    OsCpupBase      taskCpup;           /**< task cpu usage | CPU 使用统计 */
#endif
    INT32           errorNo;            /**< Error Num | 错误序号 */
    UINT32          signal;             /**< Task signal | 任务信号类型,(SIGNAL_NONE,SIGNAL_KILL,SIGNAL_SUSPEND,SIGNAL_AFFI) */
    sig_cb          sig;    ///< 信号控制块，用于异步通信,类似于 linux singal模块
#ifdef LOSCFG_KERNEL_SMP
    UINT16          currCpu;            /**< CPU core number of this task is running on | 正在运行此任务的CPU内核号 */
    UINT16          lastCpu;            /**< CPU core number of this task is running on last time | 上次运行此任务的CPU内核号 */
    UINT16          cpuAffiMask;        /**< CPU affinity mask, support up to 16 cores | CPU亲和力掩码，最多支持16核，亲和力很重要，多核情况下尽量一个任
#ifdef LOSCFG_KERNEL_SMP_TASK_SYNC //多核情况下的任务同步开关,采用信号量实现
    UINT32          syncSignal;         /**< Synchronization for signal handling | 用于CPU之间同步信号量 */
#endif
#ifdef LOSCFG_KERNEL_SMP_LOCKDEP //SMP死锁检测开关
    LockDep         lockDep; ///< 死锁依赖检测
#endif
#endif
#ifdef LOSCFG_SCHED_DEBUG //调试调度开关
    SchedStat       schedStat;          /**< Schedule statistics | 调度统计 */
#endif
    UINTPTR         userArea;   ///< 用户空间的堆区开始位置
    UINTPTR         userMapBase;  ///< 用户空间的栈顶位置,内存来自用户空间,和topOfStack有本质的区别.
    UINT32          userMapSize;        /**< user thread stack size ,real size : userMapSize + USER_STACK_MIN_SIZE | 用户栈大小 */
    UINT32          processID;          /**< Which belong process | 所属进程ID */
    FutexNode       futex;    ///< 实现快锁功能
    LOS_DL_LIST     joinList;           /**< join list | 联结链表,允许任务之间相互释放彼此 */
    LOS_DL_LIST     lockList;           /**< Hold the lock list | 该链表上挂的都是已持有的锁 */
    UINTPTR         waitID;             /**< Wait for the PID or GID of the child process | 等待子进程的PID或GID */
    UINT16          waitFlag;           /**< The type of child process that is waiting, belonging to a group or parent,
                                        a specific child process, or any child process | 等待的子进程以什么样的方式结束(OS_TASK_WAIT_PROCESS | OS_TASK_WA
#ifdef LOSCFG_KERNEL_LITEIPC //轻量级进程间通信开关
    IpcTaskInfo     *ipcTaskInfo; ///< 任务间通讯信息结构体
#endif
#ifdef LOSCFG_KERNEL_PERF
    UINTPTR         pc; ///< pc寄存器
    UINTPTR         fp; ///< fp寄存器
#endif
} LosTaskCB;
```

结构体LosTaskCB内容很多，各代表什么含义？

LosTaskCB相当于任务在内核中的身份证，它反映出每个任务在生命周期内的运行情况。既然是周期就会有状态，要运行就需要内存空间，就需要被内核算法调度，被选中CPU就去执行代码段指令，CPU要执行就需要告诉它从哪里开始执行，因为是多线程，但只有一个CPU就需要不断的切换任务，那执行会被中断，也需要再恢复后继续执行，又如何保证恢复的任务执行不会出错，这些问题都需要说明白。

Task怎么管理

什么是任务池？

前面已经说了任务是内核调度层面的概念，调度算法保证了task有序的执行，调度机制详见其他姊妹篇的介绍。
如此多的任务怎么管理和执行？管理靠任务池和就绪队列，执行靠调度算法。
代码如下（OsTaskInit）：

```
LITE_OS_SEC_TEXT_INIT UINT32 OsTaskInit(VOID)
{
    UINT32 index;
    UINT32 ret;
    UINT32 size;

    g_taskMaxNum = LOSCFG_BASE_CORE_TSK_LIMIT;//任务池中最多默认128个，可谓铁打的任务池流水的线程
    size = (g_taskMaxNum + 1) * sizeof(LosTaskCB);//计算需分配内存总大小
    /*
     * This memory is resident memory and is used to save the system resources
     * of task control block and will not be freed。
     */
    g_taskCBArray = (LosTaskCB *)LOS_MemAlloc(m_aucSysMem0， size);//任务池 常驻内存，不被释放
    if (g_taskCBArray == NULL) {
        return LOS_ERRNO_TSK_NO_MEMORY;
    }
    (VOID)memset_s(g_taskCBArray， size， 0， size);

    LOS_ListInit(&g_losFreeTask);//空闲任务链表
    LOS_ListInit(&g_taskRecyleList);//需回收任务链表
    for (index = 0; index < g_taskMaxNum; index++) {
        g_taskCBArray[index]。taskStatus = OS_TASK_STATUS_UNUSED;
        g_taskCBArray[index]。taskID = index;//任务ID最大默认127
        LOS_ListTailInsert(&g_losFreeTask， &g_taskCBArray[index]。pendList);//都插入空闲任务列表
    }//注意:这里挂的是pendList节点，所以取TCB要通过 OS_TCB_FROM_PENDLIST 取。

    ret = OsPriQueueInit();//创建32个任务优先级队列，即32个双向循环链表
    if (ret != LOS_OK) {
        return LOS_ERRNO_TSK_NO_MEMORY;
    }

    /* init sortlink for each core */
    for (index = 0; index < LOSCFG_KERNEL_CORE_NUM; index++) {
        ret = OsSortLinkInit(&g_percpu[index]。taskSortLink);//每个CPU内核都有一个执行任务链表
        if (ret != LOS_OK) {
            return LOS_ERRNO_TSK_NO_MEMORY;
        }
    }
    return LOS_OK;
}
```

g_taskCBArray 就是个任务池，默认创建128个任务，常驻内存，不被释放。
g_losFreeTask是空闲任务链表，想创建任务时来这里申请一个空闲任务，用完了就回收掉，继续给后面的申请使用。
g_taskRecyleList是回收任务链表，专用来回收exit 任务，任务所占资源被确认归还后被彻底删除，就像员工离职一样，得有个离职队列和流程，要归还电脑，邮箱，有没有借钱要还的 等操作。

对应张大爷的故事：用户要来场馆领取表格填节目单，场馆只准备了128张表格，领完就没有了，但是节目表演完了会回收表格，这样多了一张表格就可以给其他人领取了，这128张表格对应鸿蒙内核这就是任务池，简单吧。

就绪队列是怎么回事

CPU执行速度是很快的，鸿蒙内核默认一个时间片是 10ms， 资源有限，需要在众多任务中来回的切换，所以绝不能让CPU等待任务，CPU就像公司最大的领导，下面很多的部门等领导来审批，吃饭。只有大家等领导，哪有领导等你们的道理，所以工作要提前准备好，每个部门的优先级又不一样，所以每个部门都要有个任务队列，里面放的是领导能直接处理的任务，没准备好的不要放进来，因为这是给CPU提前准备好的粮食！
这就是就绪队列的原理，一共有32个就绪队列，进程和线程都有，因为线程的优先级是默认32个， 每个队列中放同等优先级的task。

还是看源码吧

```
#define OS_PRIORITY_QUEUE_NUM 32
LITE_OS_SEC_BSS LOS_DL_LIST *g_priQueueList = NULL;//队列链表
LITE_OS_SEC_BSS UINT32 g_priQueueBitmap;//队列位图 UINT32每位代表一个优先级，共32个优先级
//内部队列初始化
UINT32 OsPriQueueInit(VOID)
{
    UINT32 priority;

    /* system resident resource *///常驻内存
    g_priQueueList = (LOS_DL_LIST *)LOS_MemAlloc(m_aucSysMem0， (OS_PRIORITY_QUEUE_NUM * sizeof(LOS_DL_LIST)));//分配32个队列头节点
    if (g_priQueueList == NULL) {
        return LOS_NOK;
    }

    for (priority = 0; priority < OS_PRIORITY_QUEUE_NUM; ++priority) {
        LOS_ListInit(&g_priQueueList[priority]);//队列初始化，前后指针指向自己
    }
    return LOS_OK;
}
```

注意看 `g_priQueueList` 的内存分配，就是 `32` 个 `LOS_DL_LIST` ，还记得 `LOS_DL_LIST` 的妙用吗，不清楚去翻双向链表篇。

对应张大爷的故事：就是门口那些排队的都是至少有一个节目单是符合表演标准的，资源都到位了，没有的连排队的资格都木有，就慢慢等吧。

任务栈是怎么回事

每个任务都是独立开的，任务之间也相互独立，之间通讯通过IPC，这里的“独立”指的是每个任务都有自己的运行环境 —— 栈空间，称为任务栈，栈空间里保存的信息包含局部变量、寄存器、函数参数、函数返回地址等等
但系统中只有一个CPU，任务又是独立的，调度的本质就是CPU执行一个新task，老task在什么地方被中断谁也不清楚，是随机的。那如何保证老任务被再次调度选中时还能从上次被中断的地方继续玩下去呢？

答案是：任务上下文，CPU内有一堆的寄存器，CPU运行本质的就是这些寄存器的值不断的变化，只要切换时把这些值保存起来，再还原回去就能保证task的连续执行，让用户毫无感知。鸿蒙内核给一个任务执行的时间是 20ms ，也就是说有多任务竞争的情况下，一秒钟内最多要来回切换50次。

对应张大爷的故事：就是碰到节目没有表演完就必须打断的情况下，需要把当时的情况记录下来，比如小朋友在演躲猫猫的游戏，一半不演了，张三正在树上，李四正在厕所躲，都记录下来，下次再回来你们上次在哪就会哪呆着去，就位了继续表演。这样就接上了，观众就木有感觉了。
任务上下文(TaskContext)是怎样的呢？还是直接看源码

```
/* The size of this structure must be smaller than or equal to the size specified by OS_TSK_STACK_ALIGN (16 bytes). */
typedef struct { //参考OsTaskSchedule来理解
#if !defined(LOSCFG_ARCH_FPU_DISABLE) //支持浮点运算
    UINT64 D[FP_REGS_NUM]; /* D0-D31 */
    UINT32 regFPSCR;       /* FPSCR */
    UINT32 regFPEXC;       /* FPEXC */
#endif
    UINT32 R4;
    UINT32 R5;
    UINT32 R6;
    UINT32 R7;
    UINT32 R8;
    UINT32 R9;
    UINT32 R10;
    UINT32 R11;

    /* It has the same structure as IrqContext */
    UINT32 reserved2; /**< Multiplexing registers, used in interrupts and system calls but with different meanings */
    UINT32 reserved1; /**< Multiplexing registers, used in interrupts and system calls but with different meanings */
    UINT32 USP;       /**< User mode sp register */
    UINT32 ULR;       /**< User mode lr register */
    UINT32 R0;
    UINT32 R1;
    UINT32 R2;
    UINT32 R3;
```

```
    UINT32 R12;
    UINT32 LR;
    UINT32 PC;
    UINT32 regCPSR;
} TaskContext;
```

发现基本都是CPU寄存器的恢复现场值， 具体各寄存器有什么作用大家可以去网上详查，后续也有专门的文章来介绍。这里说其中的三个寄存器SP， LR， PC

LR
用途有二，一是保存子程序返回地址，当调用BL、 BX、 BLX等跳转指令时会自动保存返回地址到LR；二是保存异常发生的异常返回地址。

PC（Program Counter）
为程序计数器，用于保存程序的执行地址，在ARM的三级流水线架构中，程序流水线包括取址、译码和执行三个阶段， PC指向的是当前取址的程序地址，所以32位ARM中，译码地址（正在解析还未执行的程序）为PC-4，执行地址（当前正在执行的程序地址）为PC-8， 当突然发生中断的时候，保存的是PC的地址。

SP
每一种异常模式都有其自己独立的r13，它通常指向异常模式所专用的堆栈，当ARM进入异常模式的时候，程序就可以把一般通用寄存器压入堆栈，返回时再出栈，保证了各种模式下程序的状态的完整性。

任务栈初始化

任务栈的初始化就是任务上下文的初始化，因为任务没开始执行，里面除了上下文不会有其他内容，注意上下文存放的位置在栈的底部。初始状态下sp就是指向的栈底， 栈顶内容永远是0xCCCCCCCC"烫烫烫烫"，这几个字应该很熟悉吗？ 如果不是那几个字了，那说明栈溢出了， 后续篇会详细说明这块，大家也可以自行去看代码，很有意思。

Task函数集

```
/// 内核态任务运行栈初始化
LITE_OS_SEC_TEXT_INIT VOID *OsTaskStackInit(UINT32 taskID, UINT32 stackSize, VOID *topStack, BOOL initFlag)
{
    if (initFlag == TRUE) {
        OsStackInit(topStack, stackSize);
    }
    TaskContext *taskContext = (TaskContext *)(((UINTPTR)topStack + stackSize) - sizeof(TaskContext));//上下文存放在栈的底部
    /* initialize the task context */ //初始化任务上下文
#ifdef LOSCFG_GDB
    taskContext->PC = (UINTPTR)OsTaskEntrySetupLoopFrame;
#else
    taskContext->PC = (UINTPTR)OsTaskEntry;//内核态任务有统一的入口地址.
#endif
    taskContext->LR = (UINTPTR)OsTaskExit;  /* LR should be kept, to distinguish it's THUMB or ARM instruction */
    taskContext->R0 = taskID;               /* R0 */
#ifdef LOSCFG_THUMB
    taskContext->regCPSR = PSR_MODE_SVC_THUMB; /* CPSR (Enable IRQ and FIQ interrupts, THUMNB-mode) */
#else //用于设置CPSR寄存器
    taskContext->regCPSR = PSR_MODE_SVC_ARM;   /* CPSR (Enable IRQ and FIQ interrupts, ARM-mode) */
#endif
#if !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (UINT32 index = 0; index < FP_REGS_NUM; index++) {
        taskContext->D[index] = 0xAAA0000000000000LL + index; /* D0 - D31 */
    }
    taskContext->regFPSCR = 0;
    taskContext->regFPEXC = FP_EN;
#endif
    return (VOID *)taskContext;
}
```

使用场景和功能

任务创建后，内核可以执行锁任务调度，解锁任务调度，挂起，恢复，延时等操作，同时也可以设置任务优先级，获取任务优先级。任务结束的时候，则进行当前任务自删除操作。
Huawei LiteOS 系统中的任务管理模块为用户提供下面几种功能。

```
接口名 | 描述
```

```
LOS_TaskCreateOnly | 创建任务，并使该任务进入suspend状态，并不调度。
LOS_TaskCreate | 创建任务，并使该任务进入ready状态，并调度。
LOS_TaskDelete | 删除指定的任务。
LOS_TaskResume | 恢复挂起的任务。
LOS_TaskSuspend | 挂起指定的任务。
LOS_TaskDelay | 任务延时等待。
LOS_TaskYield | 显式放权，调整指定优先级的任务调度顺序。
LOS_TaskLock | 锁任务调度。
LOS_TaskUnlock | 解锁任务调度。
LOS_CurTaskPriSet | 设置当前任务的优先级。
LOS_TaskPriSet | 设置指定任务的优先级。
LOS_TaskPriGet | 获取指定任务的优先级。
LOS_CurTaskIDGet | 获取当前任务的ID。
LOS_TaskInfoGet | 设置指定任务的优先级。
LOS_TaskPriGet | 获取指定任务的信息。
LOS_TaskStatusGet | 获取指定任务的状态。
LOS_TaskNameGet | 获取指定任务的名称。
LOS_TaskInfoMonitor | 监控所有任务，获取所有任务的信息。
LOS_NextTaskIDGet | 获取即将被调度的任务的ID。
```

创建任务的过程

创建任务之前先了解另一个结构体 tagTskInitParam

```
typedef struct tagTskInitParam {//Task的初始化参数
    TSK_ENTRY_FUNC  pfnTaskEntry;  /**< Task entrance function */ //任务的入口函数
    UINT16          usTaskPrio;    /**< Task priority */ //任务优先级
    UINT16          policy;        /**< Task policy */  //任务调度方式
    UINTPTR         auwArgs[4];    /**< Task parameters， of which the maximum number is four */ //入口函数的参数，最多四个
    UINT32          uwStackSize;   /**< Task stack size */ //任务栈大小
    CHAR           *pcName;        /**< Task name */  //任务名称
#if (LOSCFG_KERNEL_SMP == YES)
    UINT16          usCpuAffiMask; /**< Task cpu affinity mask         */ //任务cpu亲和力掩码
#endif
    UINT32          uwResved;      /**< It is automatically deleted if set to LOS_TASK_STATUS_DETACHED。
                                        It is unable to be deleted if set to 0。 */ //如果设置为LOS_TASK_STATUS_DETACHED，则自动删除。如果设置为0，则无法删除
    UINT16          consoleID;     /**< The console id of task belongs  */ //任务的控制台id所属
    UINT32          processID; //进程ID
    UserTaskParam   userParam; //在用户态运行时栈参数
} TSK_INIT_PARAM_S;
```

这些初始化参数是外露的任务初始参数，`pfnTaskEntry 对java来说就是你new进程的run()，`需要上层使用者提供。 看个例子吧:shell中敲 ping 命令看下它创建的过程

```
u32_t osShellPing(int argc， const char **argv)
{

    int ret;
    u32_t i = 0;
    u32_t count = 0;
    int count_set = 0;
    u32_t interval = 1000; /* default ping interval */
    u32_t data_len = 48; /* default data length */
    ip4_addr_t dst_ipaddr;
    TSK_INIT_PARAM_S stPingTask;
    // 。。。省去一些中间代码
    /* start one task if ping forever or ping count greater than 60 */
    if (count == 0 || count > LWIP_SHELL_CMD_PING_RETRY_TIMES) {

        if (ping_taskid > 0) {

            PRINTK("Ping task already running and only support one now\n");
            return LOS_NOK;
        }
        stPingTask.pfnTaskEntry = (TSK_ENTRY_FUNC)ping_cmd;//线程的执行函数
        stPingTask.uwStackSize = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;//0x4000 = 16K
        stPingTask.pcName = "ping_task";
        stPingTask.usTaskPrio = 8; /* higher than shell 优先级高于10，属于内核态线程*/
```

```
        stPingTask.uwResved = LOS_TASK_STATUS_DETACHED;
        stPingTask.auwArgs[0] = dst_ipaddr。addr; /* network order */
        stPingTask.auwArgs[1] = count;
        stPingTask.auwArgs[2] = interval;
        stPingTask.auwArgs[3] = data_len;
        ret = LOS_TaskCreate((UINT32 *)(&ping_taskid)， &stPingTask);
    }
 // ...
    return LOS_OK;
ping_error:
    lwip_ping_usage();
    return LOS_NOK;
}
```

发现 ping 的调度优先级是 8 ，比 shell 还高，那 shell 的是多少？答案是:看源码是 9

```
LITE_OS_SEC_TEXT_MINOR UINT32 ShellTaskInit(ShellCB *shellCB)
{

    CHAR *name = NULL;
    TSK_INIT_PARAM_S initParam = {
   0};
    if (shellCB->consoleID == CONSOLE_SERIAL) {

        name = SERIAL_SHELL_TASK_NAME;
    } else if (shellCB->consoleID == CONSOLE_TELNET) {

        name = TELNET_SHELL_TASK_NAME;
    } else {

        return LOS_NOK;
    }
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ShellTask;
    initParam.usTaskPrio   = 9; /* 9:shell task priority */
    initParam.auwArgs[0]   = (UINTPTR)shellCB;
    initParam.uwStackSize  = 0x3000;
    initParam.pcName       = name;
    initParam.uwResved     = LOS_TASK_STATUS_DETACHED;
    (VOID)LOS_EventInit(&shellCB->shellEvent);
    return LOS_TaskCreate(&shellCB->shellTaskHandle， &initParam);
}
```

关于shell后续会详细介绍，请持续关注。
前置条件了解清楚后，具体看任务是如何一步步创建的，如何和进程绑定，加入调度就绪队列，还是继续看源码

```
//创建Task
LITE_OS_SEC_TEXT_INIT UINT32 LOS_TaskCreate(UINT32 *taskID， TSK_INIT_PARAM_S *initParam)
{
    UINT32 ret;
    UINT32 intSave;
    LosTaskCB *taskCB = NULL;

    if (initParam == NULL) {
        return LOS_ERRNO_TSK_PTR_NULL;
    }

    if (OS_INT_ACTIVE) {
        return LOS_ERRNO_TSK_YIELD_IN_INT;
    }

    if (initParam->uwResved & OS_TASK_FLAG_IDLEFLAG) {//OS_TASK_FLAG_IDLEFLAG 是属于内核 idle进程专用的
        initParam->processID = OsGetIdleProcessID();//获取空闲进程
    } else if (OsProcessIsUserMode(OsCurrProcessGet())) {//当前进程是否为用户模式
        initParam->processID = OsGetKernelInitProcessID();//不是就取"Kernel"进程
    } else {
        initParam->processID = OsCurrProcessGet()->processID;//获取当前进程 ID赋值
    }
    initParam->uwResved &= ~OS_TASK_FLAG_IDLEFLAG;//不能是 OS_TASK_FLAG_IDLEFLAG
```

```
    initParam->uwResved &= ~OS_TASK_FLAG_PTHREAD_JOIN;//不能是 OS_TASK_FLAG_PTHREAD_JOIN
    if (initParam->uwResved & LOS_TASK_STATUS_DETACHED) {//是否设置了自动删除
        initParam->uwResved = OS_TASK_FLAG_DETACHED;//自动删除，注意这里是 = ，也就是说只有 OS_TASK_FLAG_DETACHED 一个标签了
    }

    ret = LOS_TaskCreateOnly(taskID， initParam);//创建一个任务，这是任务创建的实体，前面都只是前期准备工作
    if (ret != LOS_OK) {
        return ret;
    }
    taskCB = OS_TCB_FROM_TID(*taskID);//通过ID拿到task实体

    SCHEDULER_LOCK(intSave);
    taskCB->taskStatus &= ~OS_TASK_STATUS_INIT;//任务不再是初始化
    OS_TASK_SCHED_QUEUE_ENQUEUE(taskCB， 0);//进入调度就绪队列，新任务是直接进入就绪队列的
    SCHEDULER_UNLOCK(intSave);

    /* in case created task not running on this core，
       schedule or not depends on other schedulers status。 */
    LOS_MpSchedule(OS_MP_CPU_ALL);//如果创建的任务没有在这个核心上运行，是否调度取决于其他调度程序的状态。
    if (OS_SCHEDULER_ACTIVE) {//当前CPU核处于可调度状态
        LOS_Schedule();//发起调度
    }

    return LOS_OK;
}
```

对应张大爷的故事：就是节目单要怎么填，按格式来，从哪里开始演，要多大的空间，王场馆好协调好现场的环境。这里注意 在同一个节目单只要节目没演完，王场馆申请场地的空间就不能给别人用，这个场地空间对应的就是鸿蒙任务的栈空间，除非整个节目单都完了，就回收了。把整个场地干干净净的留给下一个人的节目单来表演。

至此的创建已经完成，已各就各位，源码最后还申请了一次`LOS_Schedule()`;因为鸿蒙的调度方式是抢占式的，如何本次`task`的任务优先级高于其他就绪队列，那么接下来要执行的任务就是它了！

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

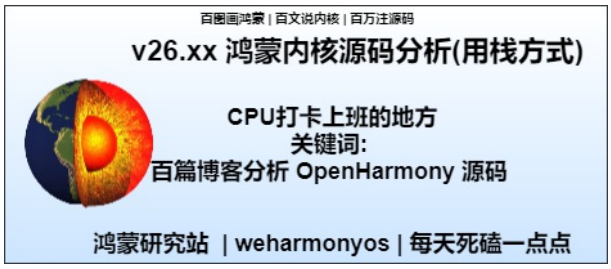
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

26_用栈方式篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

精读内核源码就绕不过汇编语言，鸿蒙内核有6个汇编文件，读不懂它们就真的很难理解以下问题。

1.系统调用是如何实现的？

2.CPU是如何切换任务和进程上下文的？

3.硬件中断是如何处理的？

4.main函数到底是怎么来的？

5.开机最开始发生了什么？

6.关机最后的最后又发生了什么？

以下是一个很简单的C文件编译成汇编代码后的注解。 读懂这些注解会发现汇编很可爱，甚至还会上瘾，并没有想象中的那么恐怖，读懂它会颠覆你对汇编和栈的认知。

```
#include <stdio.h>
#include <math.h>

int square(int a , int b){
   return a*b;
}

int fp(int b)
{
   int a = 1;
   return square(a+b , a+b);
}

int main()
{
   int sum = 1;
   for(int a = 0;a < 100; a++){
      sum = sum + fp(a);
   }
   return sum;
}
```

```
//编译器: armv7-a clang (trunk)
square(int， int):
      sub    sp， sp， #8    @sp减去8，意思为给square分配栈空间，只用2个栈空间完成计算
      str    r0，[sp， #4]  @第一个参数入栈
      str    r1，[sp]      @第二个参数入栈
      ldr    r1，[sp， #4]  @取出第一个参数给r1
      ldr    r2，[sp]      @取出第二个参数给r2
      mul    r0， r1， r2    @执行a*b给R0，返回值的工作一直是交给R0的
      add    sp， sp， #8    @函数执行完了，要释放申请的栈空间
      bx     lr           @子程序返回，等同于mov pc，lr，即跳到调用处
fp(int):
      push   {r11， lr}     @r11(fp)/lr入栈，保存调用者main的位置
      mov    r11， sp       @r11用于保存sp值，函数栈开始位置
      sub    sp， sp， #8    @sp减去8，意思为给fp分配栈空间，只用2个栈空间完成计算
      str    r0，[sp， #4]  @先保存参数值，放在SP+4，此时r0中存放的是参数
      mov    r0， #1        @r0=1
      str    r0，[sp]      @再把1也保存在SP的位置
      ldr    r0，[sp]      @把SP的值给R0
      ldr    r1，[sp， #4]  @把SP+4的值给R1
      add    r1， r0， r1    @执行r1=a+b
      mov    r0， r1        @r0=r1，用r0，r1传参
      bl     square(int， int)@先mov lr， pc 再mov pc square(int， int)
      mov    sp， r11       @函数执行完了，要释放申请的栈空间
      pop    {r11， lr}     @弹出r11和lr，lr是专用标签，弹出就自动复制给lr寄存器
      bx     lr           @子程序返回，等同于mov pc，lr，即跳到调用处
main:
      push   {r11， lr}     @r11(fp)/lr入栈，保存调用者的位置
      mov    r11， sp       @r11用于保存sp值，函数栈开始位置
      sub    sp， sp， #16   @sp减去8，意思为给main分配栈空间，只用2个栈空间完成计算
      mov    r0， #0        @初始化r0
      str    r0，[r11， #-4] @作用是保存SUM的初始值
      str    r0，[sp， #8]  @sum将始终占用SP+8的位置
      str    r0，[sp， #4]  @a将始终占用SP+4的位置
      b      .LBB1_1       @跳到循环开始位置
.LBB1_1:                   @循环开始位置入口
      ldr    r0，[sp， #4]  @取出a的值给r0
      cmp    r0， #99       @跟99比较
      bgt    .LBB1_4       @大于99，跳出循环 mov pc .LBB1_4
      b      .LBB1_2       @继续循环，直接 mov pc .LBB1_2
.LBB1_2:                   @符合循环条件入口
      ldr    r0，[sp， #8]  @取出sum的值给r0，sp+8用于写SUM的值
      str    r0，[sp]      @先保存SUM的值，SP的位置用于读SUM值
      ldr    r0，[sp， #4]  @r0用于传参，取出A的值给r0作为fp的参数
      bl     fp(int)       @先mov lr， pc再mov pc fp(int)
      mov    r1， r0        @fp的返回值为r0，保存到r1
      ldr    r0，[sp]      @取出SUM的值
      add    r0， r0， r1    @计算新sum的值，由R0保存
      str    r0，[sp， #8]  @将新sum保存到SP+8的位置
      b      .LBB1_3       @无条件跳转，直接 mov pc .LBB1_3
.LBB1_3:                   @完成a++操作入口
      ldr    r0，[sp， #4]  @SP+4中记录是a的值，赋给r0
      add    r0， r0， #1    @r0增加1
      str    r0，[sp， #4]  @把新的a值放回SP+4里去
      b      .LBB1_1       @跳转到比较 a < 100 处
.LBB1_4:                   @循环结束入口
      ldr    r0，[sp， #8]  @最后SUM的结果给R0，返回值的工作一直是交给R0的
      mov    sp， r11       @函数执行完了，要释放申请的栈空间
      pop    {r11， lr}     @弹出r11和lr，lr是专用标签，弹出就自动复制给lr寄存器
      bx     lr           @子程序返回，跳转到lr处等同于 MOV PC， LR
```

这个简单的汇编并不是鸿蒙的汇编，只是先打个底，由浅入深， 但看懂了它基本理解鸿蒙汇编代码没有问题， 后续将详细分析鸿蒙内核各个汇编文件的作用。 开始分析上面的汇编代码。

第一： 上面的代码和鸿蒙内核用栈方式一样，都采用了递减满栈的方式， 什么是递减满栈？ 递减指的是栈底地址高于栈顶地址，满栈指的是SP指针永远在栈顶。一定要理解递减满栈，否则读不懂内核汇编代码。举例说明:

```
square(int， int):
      sub    sp， sp， #8    @sp减去8，意思为给square分配栈空间，只用2个栈空间完成计算
      str    r0，[sp， #4]  @第一个参数入栈
      str    r1，[sp]      @第二个参数入栈
      ldr    r1，[sp， #4]  @取出第一个参数给r1
      ldr    r2，[sp]      @取出第二个参数给r2
      mul    r0， r1， r2    @执行a*b给R0，返回值的工作一直是交给R0的
      add    sp， sp， #8    @函数执行完了，要释放申请的栈空间
      bx     lr           @子程序返回，等同于mov pc，lr，即跳到调用处
```

首句汇编的含义就是申请栈空间， `sp = sp - 8` ，一个栈内单元(栈空间)占4个字节，申请2个栈空间搞定函数的计算，仔细看下代码除了在函数的末尾 `sp = sp + 8` 又恢复在之前的位置的中间过程，SP的值是没有任务变化，它的指向是不动的， 这跟很多人对栈的认知是不一样的，它只是被用于计算，例如 `ldr r1，[sp， #4]` 的意思是取出SP+4这个虚拟地址的值给r1寄存器，SP的值并没有改变的，为什么要+呢，因为SP是指向栈顶的，地址是最小的。 满栈就是用栈过程中对地址的操作不能超过SP，所以你很少在计算过程中看到 把sp-4地址中的值给某个寄存器， 除非是特别的指令，否则不可能有这样的指令。

第二： `sub sp， sp， #8` 和 `add sp， sp， #8` 是成对出现的，这就跟申请内存，释放内存的道理一样，这是内核对任务的运行栈管理方式，一样用多少申请多少，用完释放。空间大小就是栈帧，这是栈帧的本质含义。

第三： `push {r11， lr}` 和 `pop {r11， lr}` 也是成对出现的，主要是用于函数调用，例如 A -> B， B要保存A的栈帧范围和指令位置， lr保存是是A函数执行到哪个指令的位置， r11干了fp的工作，其实就是指向 A的栈顶位置，如此B执行完后return回A的时候，先mov pc，lr 内核就知道改执行A的哪条指令了，同时又知道了A的栈顶位置。

第四: 频繁出现的R0寄存器的作用用于传参和返回值， A调用B之前，假如有两个参数，就把参数给 `r0，r1` 记录，充当了A的变量， 到了B中后，先让 `r0，r1` 入栈，目的是保存参数值， 因为 B中要用 `r0，r1` ，他们变成B的变量用了。 返回值都是默认统一给r0保存。 B中将返回值给r0，回到A中取出R0值对A来说这就是B的返回值。

这是以上为汇编代码的分析，追问两个问题

第一：如果是可变参数怎么办？ 100个参数怎么整， 通过寄存器总共就12个，不够传参啊 第二：返回值可以有多个吗？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

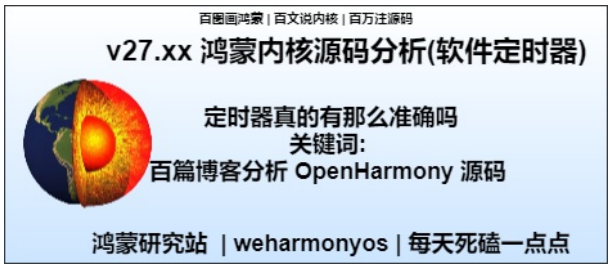
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

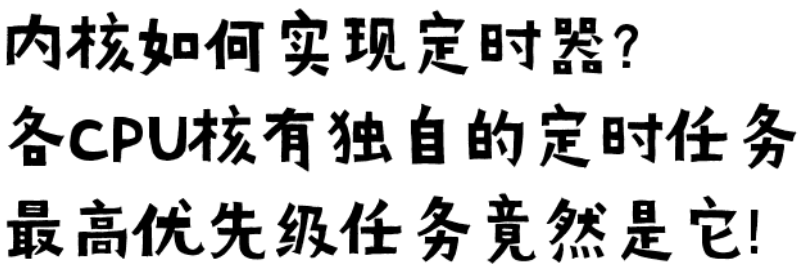
关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

27_软件定时器篇

本篇关键词：、、、





下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

本篇说清楚定时器的实现

读本篇之前建议先读v08.xx 鸿蒙内核源码分析(总目录) 其余篇。

运作机制

- 软件定时器，是基于系统Tick时钟中断且由软件来模拟的定时器。当经过设定的Tick数后，会触发用户自定义的回调函数。
- 软件定时器是系统资源，在模块初始化的时候已经分配了一块连续内存。
- 软件定时器使用了系统的一个队列和一个任务资源，软件定时器的触发遵循队列规则，先进先出。定时时间短的定时器总是比定时时间长的靠近队列头，满足优先触发的准则。
- 软件定时器以Tick为基本计时单位，当创建并启动一个软件定时器时，鸿蒙会根据当前系统Tick时间及设置的定时时长确定该定时器的到期Tick时间，并将该定时器控制结构挂入计时全局链表。
- 当Tick中断到来时，在Tick中断处理函数中扫描软件定时器的计时全局链表，检查是否有定时器超时，
- 若有则将超时的定时器记录下来。Tick中断处理函数结束后，软件定时器任务（优先级为最高）被唤醒，在该任务中调用已经记录下来的定时器的回调函数。

定时器长什么样？

```
typedef VOID (*SWTMR_PROC_FUNC)(UINTPTR arg);//函数指针， 赋值给 SWTMR_CTRL_S->pfnHandler，回调处理
typedef struct tagSwTmrCtrl {//软件定时器控制块
```

```
    SortLinkList stSortList;//通过它挂到对应CPU核定时器链表上
    UINT8 ucState;      /**< Software timer state *///软件定时器的状态
    UINT8 ucMode;       /**< Software timer mode *///软件定时器的模式
    UINT8 ucOverrun;    /**< Times that a software timer repeats timing *///软件定时器重复计时的次数
    UINT16 usTimerID;   /**< Software timer ID *///软件定时器ID，唯一标识，由软件计时器池分配
    UINT32 uwCount;     /**< Times that a software timer works *///软件定时器工作的时间
    UINT32 uwInterval;  /**< Timeout interval of a periodic software timer *///周期性软件定时器的超时间隔
    UINT32 uwExpiry;    /**< Timeout interval of an one-off software timer *///一次性软件定时器的超时间隔
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 uwCpuid;     /**< The cpu where the timer running on *///多核情况下，定时器运行的cpu
#endif
    UINTPTR uwArg;      /**< Parameter passed in when the callback function
                         that handles software timer timeout is called *///回调函数的参数
    SWTMR_PROC_FUNC pfnHandler; /**< Callback function that handles software timer timeout */ //处理软件计时器超时的回调函数
    UINT32          uwOwnerPid; /** Owner of this software timer *///软件定时器所属进程ID号
} SWTMR_CTRL_S;///变量前缀 uc:UINT8  us:UINT16 uw:UINT32
```

解读

- 在多CPU核情况下，定时器是跟着CPU走的，每个CPU核都维护着独立的定时任务链表，上面挂的都是CPU核要处理的定时器。
- `stSortList` 的背后是双向链表，这对钩子在定时器创建的那一刻会钩到CPU的`swtmrSortLink`上去。
- `pfnHandler`定时器时间到了的执行函数，由外界指定。`uwArg`为回调函数的参数
- `ucMode` 为定时器模式，软件定时器提供了三类模式
  > 单次触发定时器，这类定时器在启动后只会触发一次定时器事件，然后定时器自动删除。 周期触发定时器，这类定时器会周期性的触发定时器事件，直到用户手动停止定时器，否则将永远持续执行下去。 单次触发定时器，但这类定时器超时触发后不会自动删除，需要调用定时器删除接口删除定时器。
- `ucState` 定时器状态。
  > OS_SWTMR_STATUS_UNUSED（定时器未使用） 系统在定时器模块初始化时，会将系统中所有定时器资源初始化成该状态。
  > OS_SWTMR_STATUS_TICKING（定时器处于计数状态） 在定时器创建后调用LOS_SwtmrStart接口启动，定时器将变成该状态，是定时器运行时的状态。 OS_SWTMR_STATUS_CREATED（定时器创建后未启动，或已停止） 定时器创建后，不处于计数状态时，定时器将变成该状态。

定时器分类

定时器是指从指定的时刻开始，经过一定的指定时间后触发一个事件，例如定个时间提醒晚上9点准时秒杀。定时器有硬件定时器和软件定时器之分：

- 硬件定时器是芯片本身提供的定时功能。一般是由外部晶振提供给芯片输入时钟，芯片向软件模块提供一组配置寄存器，接受控制输入，到达设定时间值后芯片中断控制器产生时钟中断。硬件定时器的精度一般很高，可以达到纳秒级别，并且是中断触发方式。
- 软件定时器是由操作系统提供的一类系统接口，它构建在硬件定时器基础之上，使系统能够提供不受数目限制的定时器服务。

鸿蒙内核提供软件实现的定时器，以时钟节拍（OS Tick）的时间长度为单位，即定时数值必须是 OS Tick 的整数倍，例如鸿蒙内核默认是10ms触发一次，那么上层软件定时器只能是 10ms，20ms，100ms 等，而不能定时为 15ms。

定时器怎么管理？

```
LITE_OS_SEC_BSS SWTMR_CTRL_S    *g_swtmrCBArray = NULL;     /* First address in Timer memory space *///定时器池
LITE_OS_SEC_BSS UINT8           *g_swtmrHandlerPool = NULL; /* Pool of Swtmr Handler *///用于注册软时钟的回调函数
LITE_OS_SEC_BSS LOS_DL_LIST     g_swtmrFreeList;            /* Free list of Software Timer *///空闲定时器链表

typedef struct {//处理软件定时器超时的回调函数的结构体
    SWTMR_PROC_FUNC handler;    /**< Callback function that handles software timer timeout  */ //处理软件定时器超时的回调函数
    UINTPTR arg;                /**< Parameter passed in when the callback function
                                 that handles software timer timeout is called */ //调用处理软件计时器超时的回调函数时传入的参数
} SwtmrHandlerItem;
```

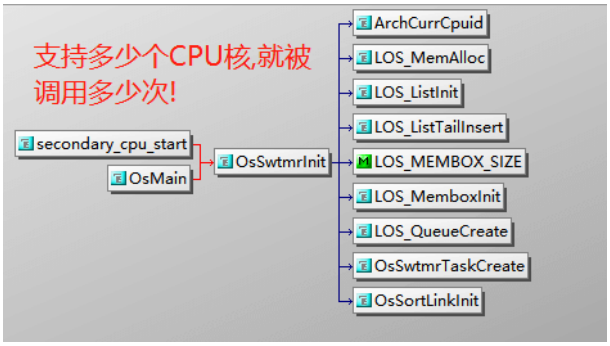
解读

三个全局变量可知，定时器是通过池来管理，在初始化阶段赋值。

- `g_swtmrCBArray` 定时器池，初始化中一次性创建1024个定时器控制块供使用
- `g_swtmrHandlerPool` 回调函数池，回调函数也是统一管理的，申请了静态内存保存。 池中放的是 `SwtmrHandlerItem` 回调函数描述符。
- `g_swtmrFreeList` 空闲可供分配的定时器链表，鸿蒙的进程池，任务池，事件池都是这么处理的，没有印象的自行去翻看。
  `g_swtmrFreeList`上挂的是一个个的 `SWTMR_CTRL_S`

- 要搞明白 `SWTMR_CTRL_S` 和 `SwtmrHandlerItem` 的关系，前者是一个定时器，后者是定时器时间到了去哪里干活。

初始化 -> OsSwtmrInit




```
#define LOSCFG_BASE_CORE_SWTMR_LIMIT 1024 // 最大支持的软件定时器数
LITE_OS_SEC_TEXT_INIT UINT32 OsSwtmrInit(VOID)
{
    UINT32 size;
    UINT16 index;
    UINT32 ret;
    SWTMR_CTRL_S *swtmr = NULL;
    UINT32 swtmrHandlePoolSize;
    UINT32 cpuid = ArchCurrCpuid();
    if (cpuid == 0) {//确保以下代码块由一个CPU执行，g_swtmrCBArray和g_swtmrHandlerPool 是所有CPU共用的
        size = sizeof(SWTMR_CTRL_S) * LOSCFG_BASE_CORE_SWTMR_LIMIT;//申请软时钟内存大小
        swtmr = (SWTMR_CTRL_S *)LOS_MemAlloc(m_aucSysMem0， size); /* system resident resource */ //常驻内存
        if (swtmr == NULL) {
            return LOS_ERRNO_SWTMR_NO_MEMORY;
        }

        (VOID)memset_s(swtmr， size， 0， size);//清0
        g_swtmrCBArray = swtmr;//软时钟
        LOS_ListInit(&g_swtmrFreeList);//初始化空闲链表
        for (index = 0; index < LOSCFG_BASE_CORE_SWTMR_LIMIT; index++， swtmr++) {
            swtmr->usTimerID = index;//按顺序赋值
            LOS_ListTailInsert(&g_swtmrFreeList， &swtmr->stSortList.sortLinkNode);//通过sortLinkNode将节点挂到空闲链表
        }
        //想要用静态内存池管理，就必须要使用LOS_MEMBOX_SIZE来计算申请的内存大小，因为需要点前缀内存承载头部信息。
        swtmrHandlePoolSize = LOS_MEMBOX_SIZE(sizeof(SwtmrHandlerItem)， OS_SWTMR_HANDLE_QUEUE_SIZE);//计算所有注册函数内存大小
        //规划一片内存区域作为软时钟处理函数的静态内存池。
        g_swtmrHandlerPool = (UINT8 *)LOS_MemAlloc(m_aucSysMem1， swtmrHandlePoolSize); /* system resident resource *///常驻内存
        if (g_swtmrHandlerPool == NULL) {
            return LOS_ERRNO_SWTMR_NO_MEMORY;
        }

        ret = LOS_MemboxInit(g_swtmrHandlerPool， swtmrHandlePoolSize， sizeof(SwtmrHandlerItem));//初始化软时钟注册池
        if (ret != LOS_OK) {
            return LOS_ERRNO_SWTMR_HANDLER_POOL_NO_MEM;
        }
    }
    //每个CPU都会创建一个属于自己的 OS_SWTMR_HANDLE_QUEUE_SIZE 的队列
    ret = LOS_QueueCreate(NULL， OS_SWTMR_HANDLE_QUEUE_SIZE， &g_percpu[cpuid].swtmrHandlerQueue， 0， sizeof(CHAR *));//为当前CPU c
    if (ret != LOS_OK) {
        return LOS_ERRNO_SWTMR_QUEUE_CREATE_FAILED;
    }

    ret = OsSwtmrTaskCreate();//每个CPU独自创建属于自己的软时钟任务，统一处理队列
    if (ret != LOS_OK) {
        return LOS_ERRNO_SWTMR_TASK_CREATE_FAILED;
    }

    ret = OsSortLinkInit(&g_percpu[cpuid].swtmrSortLink);//每个CPU独自对自己软时钟链表排序初始化，为啥要排序因为每个定时器的时间不一样，鸿蒙把用时
    if (ret != LOS_OK) {
        return LOS_ERRNO_SWTMR_SORTLINK_CREATE_FAILED;
    }
```

```
    return LOS_OK;
}
```

解读:

- 每个CPU核都是独立处理定时器任务的，所以需要独自管理。 `OsSwtmrInit` 是负责初始化各CPU核定时模块功能的，注意在多CPU核时， `OsSwtmrInit` 会被多次调用。
- `cpuid == 0` 代表主CPU核， 它最早执行这个函数，所以 `g_swtmrCBArray` 和 `g_swtmrHandlerPool` 是共用的，系统默认最多支持 1024 个定时器和回调函数。
- 每个CPU核都创建了自己独立的 `LOS_QueueCreate` 队列和任务 `OsSwtmrTaskCreate` ，并初始化了 `swtmrSortLink` 链表，关于链表排序可前往系列篇总目录 排序链表篇查看。

定时任务 -> 最高优先级

```
LITE_OS_SEC_TEXT_INIT UINT32 OsSwtmrTaskCreate(VOID)
{
    UINT32 ret， swtmrTaskID;
    TSK_INIT_PARAM_S swtmrTask;
    UINT32 cpuid = ArchCurrCpuid();//获取当前CPU id

    (VOID)memset_s(&swtmrTask， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));//清0
    swtmrTask.pfnTaskEntry = (TSK_ENTRY_FUNC)OsSwtmrTask;//入口函数
    swtmrTask.uwStackSize = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;//16K默认内核任务栈
    swtmrTask.pcName = "Swt_Task";//任务名称
    swtmrTask.usTaskPrio = 0;//哇塞! 逮到一个最高优先级的任务 @note_thinking 这里应该用 OS_TASK_PRIORITY_HIGHEST 表示
    swtmrTask.uwResved = LOS_TASK_STATUS_DETACHED;//分离模式
#if (LOSCFG_KERNEL_SMP == YES)
    swtmrTask.usCpuAffiMask   = CPUID_TO_AFFI_MASK(cpuid);//交给当前CPU执行这个任务
#endif
    ret = LOS_TaskCreate(&swtmrTaskID， &swtmrTask);//创建任务并申请调度
    if (ret == LOS_OK) {
        g_percpu[cpuid].swtmrTaskID = swtmrTaskID;//全局变量记录 软时钟任务ID
        OS_TCB_FROM_TID(swtmrTaskID)->taskStatus |= OS_TASK_FLAG_SYSTEM_TASK;//告知这是一个系统任务
    }

    return ret;
}
```

解读:

- 内核为每个CPU处理单独创建任务来处理定时器， 任务即线程， 外界可理解为内核开设了一个线程跑定时器。
- 注意看任务的优先级 `swtmrTask.usTaskPrio = 0;` 0是最高优先级! 这并不多见! 内核会在第一时间响应软时钟任务。
- 系列篇CPU篇中讲过每个CPU都有自己的任务链表和定时器任务， `g_percpu[cpuid].swtmrTaskID = swtmrTaskID;` 表示创建的任务和CPU具体核进行了捆绑。从此swtmrTaskID负责这个CPU的定时器处理。
- 定时任务是一个系统任务，除此之外还有哪些是系统任务？
- 任务入口函数 `OsSwtmrTask` ，是任务的执行体，类似于[Java 线程中的run()函数]
- `usCpuAffiMask` 代表这个任务只能由这个CPU核来跑

队列消费者 -> OsSwtmrTask

```
//软时钟的入口函数，拥有任务的最高优先级 0 级!
LITE_OS_SEC_TEXT VOID OsSwtmrTask(VOID)
{
    SwtmrHandlerItemPtr swtmrHandlePtr = NULL;
    SwtmrHandlerItem swtmrHandle;
    UINT32 ret， swtmrHandlerQueue;

    swtmrHandlerQueue = OsPercpuGet()->swtmrHandlerQueue;//获取定时器超时队列
    for (;;) {//死循环获取队列item，一直读干净为止
        ret = LOS_QueueRead(swtmrHandlerQueue， &swtmrHandlePtr， sizeof(CHAR *)， LOS_WAIT_FOREVER);//一个一个读队列
        if ((ret == LOS_OK) && (swtmrHandlePtr != NULL)) {
            swtmrHandle.handler = swtmrHandlePtr->handler;//超时中断处理函数，也称回调函数
            swtmrHandle.arg = swtmrHandlePtr->arg;//回调函数的参数
            (VOID)LOS_MemboxFree(g_swtmrHandlerPool， swtmrHandlePtr);//静态释放内存，注意在鸿蒙内核只有软时钟注册用到了静态内存
            if (swtmrHandle.handler != NULL) {
```

```
            swtmrHandle.handler(swtmrHandle.arg);//回调函数处理函数
        }
      }
    }
}
```

解读

- OsSwtmrTask是任务的执行体，只做一件事，消费定时器回调函数队列。
- 任务在跑一个死循环，不断在读队列。关于队列的具体操作不在此处细说，系列篇中已有专门的文章讲解，可前往查看。
- 每个CPU核都有属于自己的定时器回调函数队列，里面存放的是时间到了回调函数。
- 但队列的数据怎么来呢？ `OsSwtmrTask` 只是在不断的消费队列，那生产者在哪里呢？ 就是 `OsSwtmrScan`

队列生产者 -> OsSwtmrScan

```
LITE_OS_SEC_TEXT VOID OsSwtmrScan(VOID)//扫描定时器，如果碰到超时的，就放入超时队列
{
    SortLinkList *sortList = NULL;
    SWTMR_CTRL_S *swtmr = NULL;
    SwtmrHandlerItemPtr swtmrHandler = NULL;
    LOS_DL_LIST *listObject = NULL;
    SortLinkAttribute* swtmrSortLink = &OsPercpuGet()->swtmrSortLink;//拿到当前CPU的定时器链表

    swtmrSortLink->cursor = (swtmrSortLink->cursor + 1) & OS_TSK_SORTLINK_MASK;
    listObject = swtmrSortLink->sortLink + swtmrSortLink->cursor;
 //由于swtmr是在特定的sortlink中，所以需要很小心的处理它，但其他CPU Core仍然有机会处理它，比如停止计时器
    /*
     * it needs to be carefully coped with， since the swtmr is in specific sortlink
     * while other cores still has the chance to process it， like stop the timer.
     */
    LOS_SpinLock(&g_swtmrSpin);

    if (LOS_ListEmpty(listObject)) {
        LOS_SpinUnlock(&g_swtmrSpin);
        return;
    }
    sortList = LOS_DL_LIST_ENTRY(listObject->pstNext， SortLinkList， sortLinkNode);
    ROLLNUM_DEC(sortList->idxRollNum);

    while (ROLLNUM(sortList->idxRollNum) == 0) {
        sortList = LOS_DL_LIST_ENTRY(listObject->pstNext， SortLinkList， sortLinkNode);
        LOS_ListDelete(&sortList->sortLinkNode);
        swtmr = LOS_DL_LIST_ENTRY(sortList， SWTMR_CTRL_S， stSortList);

        swtmrHandler = (SwtmrHandlerItemPtr)LOS_MemboxAlloc(g_swtmrHandlerPool);//取出一个可用的软时钟处理项
        if (swtmrHandler != NULL) {
            swtmrHandler->handler = swtmr->pfnHandler;
            swtmrHandler->arg = swtmr->uwArg;

            if (LOS_QueueWrite(OsPercpuGet()->swtmrHandlerQueue， swtmrHandler， sizeof(CHAR *)， LOS_NO_WAIT)) {
                (VOID)LOS_MemboxFree(g_swtmrHandlerPool， swtmrHandler);
            }
        }

        if (swtmr->ucMode == LOS_SWTMR_MODE_ONCE) {
            OsSwtmrDelete(swtmr);

            if (swtmr->usTimerID < (OS_SWTMR_MAX_TIMERID - LOSCFG_BASE_CORE_SWTMR_LIMIT)) {
                swtmr->usTimerID += LOSCFG_BASE_CORE_SWTMR_LIMIT;
            } else {
                swtmr->usTimerID %= LOSCFG_BASE_CORE_SWTMR_LIMIT;
            }
        } else if (swtmr->ucMode == LOS_SWTMR_MODE_NO_SELFDELETE) {
            swtmr->ucState = OS_SWTMR_STATUS_CREATED;
        } else {
            swtmr->ucOverrun++;
            OsSwtmrStart(swtmr);
        }
```

```
        if (LOS_ListEmpty(listObject)) {
            break;
        }

        sortList = LOS_DL_LIST_ENTRY(listObject->pstNext， SortLinkList， sortLinkNode);
    }

    LOS_SpinUnlock(&g_swtmrSpin);
}
```

解读

- OsSwtmrScan 函数是在系统时钟处理函数 `OsTickHandler` 中调用的，它就干一件事，不停的比较定时器是否超时
- 一旦超时就把定时器的回调函数扔到队列中，让 `OsSwtmrTask` 去消费。

总结

- 定时器池 `g_swtmrCBArray` 存储内核所有的定时器，默认1024个，各CPU共享这个池
- 定时器响应函数池 `g_swtmrHandlerPool` 存储内核所有的定时器响应函数，默认1024个，各CPU也共享这个池
- 每个CPU核都有独立的任务(线程)来处理定时器， 这个任务叫定时任务
- 每个CPU核都有独立的响应函数队列 `swtmrHandlerQueue` ，队列中存放该核时间到了的响应函数 `SwtmrHandlerItem`
- 定时任务的优先级最高，循环读取队列 `swtmrHandlerQueue` ， `swtmrHandlerQueue` 中存放是定时器时间到了的响应函数。并一一回调这些响应函数。
- OsSwtmrScan负责扫描定时器的时间是否到了，到了就往队列 `swtmrHandlerQueue` 中扔。
- 定时器有多种模式，包括单次，循环。所以循环类定时器的响应函数会多次出现在 `swtmrHandlerQueue` 中。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

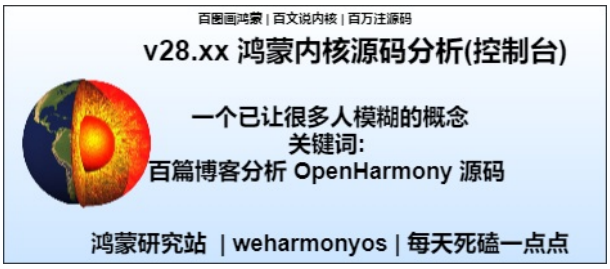
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

28_控制台篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

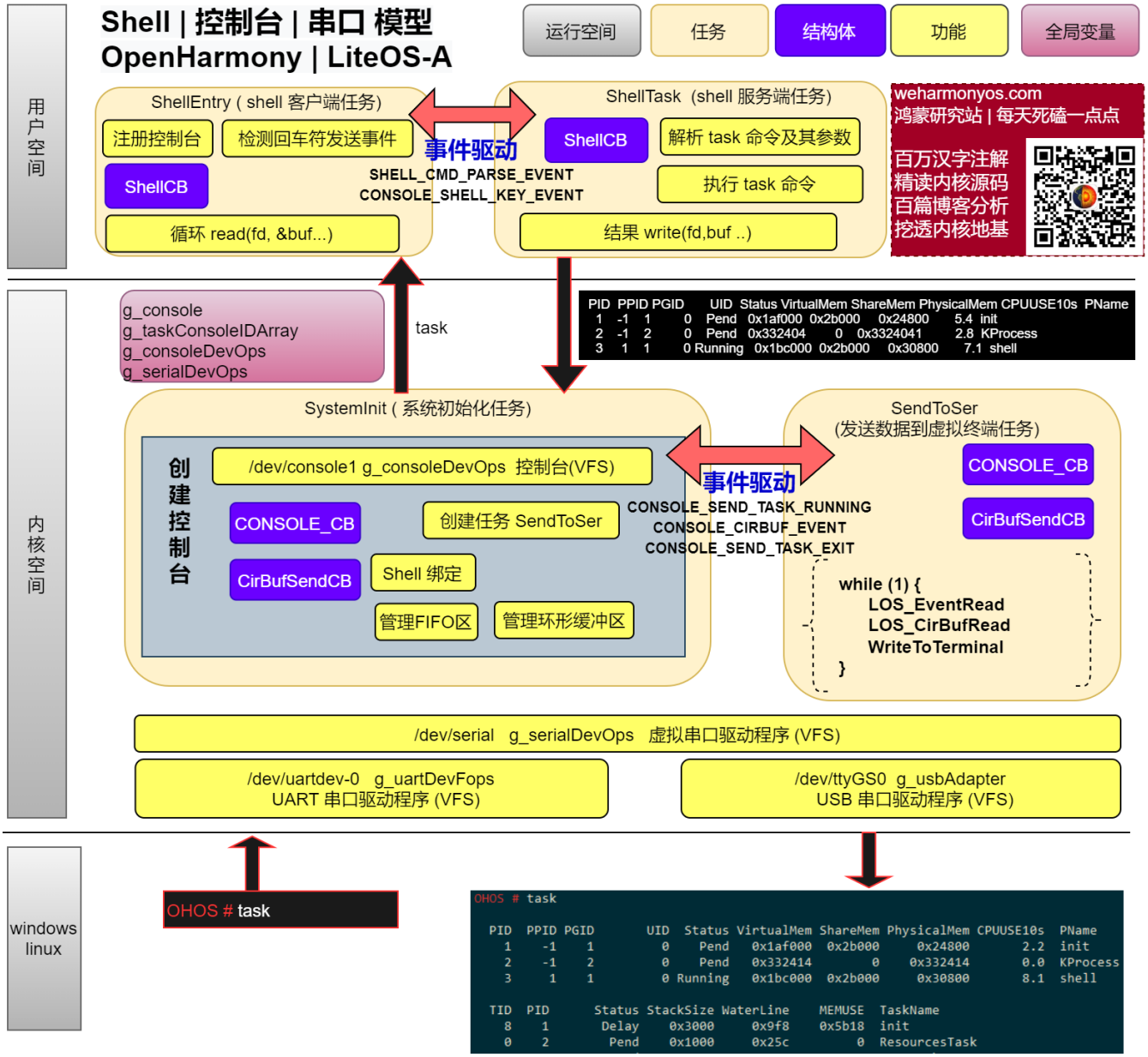
任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

本篇尝试讲明白控制台实现以及Shell如何依赖控制台工作。涉及源码部分只列出关键代码。 详细代码前往 >> 中文注解鸿蒙内核源码 查看

Shell | 控制台 | 串口模型

下图为看完鸿蒙内核Shell和控制台源码后整理的模型图



模型说明

- 模型涉及四个任务， 两个在用户空间，两个在内核空间。用户空间的在系列篇Shell部分中已有详细说明，请前往查看。
- `SystemInit` 任务是在内核 OsMain 中创建的系统初始化任务，其中初始化了根文件系统，串口，控制台等内核模块
- 在控制台模块中创建 `SendToSer` 任务，这是一个负责将控制台结果输出到终端的任务。
- 结构体 `CONSOLE_CB` ， `CirBufSendCB` 承载了控制台的实现过程。

代码实现

每个模块都有一个核心结构体，控制台则是

结构体 | CONSOLE_CB

```
/**
 * @brief 控制台控制块(描述符)
 */
typedef struct {
    UINT32 consoleID; ///< 控制台ID    例如：1 | 串口 ， 2 | 远程登录
    UINT32 consoleType; ///< 控制台类型
    UINT32 consoleSem; ///< 控制台信号量
```

```
    UINT32 consoleMask; ///< 控制台掩码
    struct Vnode *devVnode; ///< 索引节点
    CHAR *name; ///< 名称 例如: /dev/console1
    INT32 fd; ///< 系统文件句柄， 由内核分配
    UINT32 refCount; ///< 引用次数，用于判断控制台是否被占用
    UINT32 shellEntryId; ///<  负责接受来自终端信息的 "ShellEntry"任务，这个值在运行过程中可能会被换掉，它始终指向当前正在运行的shell客户端
    INT32 pgrpId; ///< 进程组ID
    BOOL isNonBlock; ///< 是否无锁方式
#ifdef LOSCFG_SHELL
    VOID *shellHandle; ///< shell句柄，本质是 shell控制块 ShellCB
#endif
    UINT32 sendTaskID; ///< 创建任务通过事件接收数据， 见于OsConsoleBufInit
    CirBufSendCB *cirBufSendCB; ///< 循环缓冲发送控制块
    UINT8 fifo[CONSOLE_FIFO_SIZE]; ///< termios 规范模式(ICANON mode )下使用 size:1K
    UINT32 fifoOut; ///< 对fifo的标记，输出位置
    UINT32 fifoIn; ///< 对fifo的标记，输入位置
    UINT32 currentLen; ///< 当前fifo位置
    struct termios consoleTermios; ///< 线路规程
} CONSOLE_CB;
```

解析

- 创建控制台的过程是给 `CONSOLE_CB` 赋值的过程，如下

```
STATIC CONSOLE_CB *OsConsoleCreate(UINT32 consoleID， const CHAR *deviceName)
{
  INT32 ret;
  CONSOLE_CB *consoleCB = OsConsoleCBInit(consoleID);//初始化控制台
  ret = (INT32)OsConsoleBufInit(consoleCB);//控制台buf初始化，创建 ConsoleSendTask 任务
  ret = (INT32)LOS_SemCreate(1, &consoleCB->consoleSem);//创建控制台信号量
  ret = OsConsoleDevInit(consoleCB, deviceName);//控制台设备初始化，注意这步要在 OsConsoleFileInit 的前面。
  ret = OsConsoleFileInit(consoleCB); //为 /dev/console(n|1:2)分配fd(3)
  OsConsoleTermiosInit(consoleCB, deviceName);//控制台线路规程初始化
  return consoleCB;
 }
```

- `Shell` 是用户空间进程， 负责解析和执行用户输入的命令。 但前提是得先拿到用户的输入数据。 不管数据是从串口进来，还是远程登录进来，必须得先经过内核， 而控制台的作用就是帮你拿到数据再交给 `shell` 处理， `shell` 再将要显示的处理结果通过控制台返回给终端用户， 那数据怎么传给 `shell` 呢？ 很显然用户进程只能通过系统调用 `read(fd，...)` 来读取内核数据， 因为应用程序的视角是只认 `fd` 。通用的办法是通过文件路径来打开文件来获取 `fd` 。

- 还有一种办法是内核先打开文件，获取 `fd` 后，用户任务通过捆绑的方式获取 `fd` ，而 `shell` 和 `console` 之间正是通过这种方式勾搭在一块的。具体在创建 `ShellEntry` 任务时将自己与控制台进行捆绑。看源码实现

```
///进入shell客户端任务初始化，这个任务负责编辑命令，处理命令产生的过程，例如如何处理方向键，退格键，回车键等
  LITE_OS_SEC_TEXT_MINOR UINT32 ShellEntryInit(ShellCB *shellCB)
  {
     UINT32 ret;
     CHAR *name = NULL;
     TSK_INIT_PARAM_S initParam = {0};
     if (shellCB->consoleID == CONSOLE_SERIAL) {
        name = SERIAL_ENTRY_TASK_NAME;
     } else if (shellCB->consoleID == CONSOLE_TELNET) {
        name = TELNET_ENTRY_TASK_NAME;
     } else {
        return LOS_NOK;
     }
     initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ShellEntry;//任务入口函数
     initParam.usTaskPrio   = 9; /* 9:shell task priority */
     initParam.auwArgs[0]   = (UINTPTR)shellCB;
     initParam.uwStackSize  = 0x1000;
     initParam.pcName       = name; //任务名称
     initParam.uwResved     = LOS_TASK_STATUS_DETACHED;
     ret = LOS_TaskCreate(&shellCB->shellEntryHandle, &initParam);//创建shell任务
  #ifdef LOSCFG_PLATFORM_CONSOLE
     (VOID)ConsoleTaskReg((INT32)shellCB->consoleID, shellCB->shellEntryHandle);//将shell捆绑到控制台
  #endif
     return ret;
```

```
    }
```

`ConsoleTaskReg` 将 `shellCB` 和 `consoleCB` 捆绑在一块，二者可以相互查找。 `ShellEntry` 任务个人更愿意称之为 `shell` 的客户端任务，用死循环不断一个字符一个字符的读取用户的输入，为何要单字符 读取可翻看系列篇的**Shell编辑篇**，简单的说是因为要处理控制字符(如:删除，回车==)

```
LITE_OS_SEC_TEXT_MINOR UINT32 ShellEntry(UINTPTR param)
{
    CHAR ch;
    INT32 n = 0;
    ShellCB *shellCB = (ShellCB *)param;
    CONSOLE_CB *consoleCB = OsGetConsoleByID((INT32)shellCB->consoleID);//获取绑定的控制台，目的是从控制台读数据
    (VOID)memset_s(shellCB->shellBuf, SHOW_MAX_LEN, 0, SHOW_MAX_LEN);//重置shell命令buf
    while (1) {
        n = read(consoleCB->fd, &ch, 1);//系统调用，从控制台读取一个字符内容，字符一个个处理
        if (n == 1) {//如果能读到一个字符
            ShellCmdLineParse(ch, (pf_OUTPUT)dprintf, shellCB);
        }
    }
}
```

- `read` 函数的 `consoleCB->fd` 是个虚拟字符设备文件 如: `/dev/console1` ，对文件的操作由 `g_consoleDevOps` 实现。 `read` 最终会调用 `ConsoleRead` ，再往下会调用到 `UART_Read`

```
/*! console device driver function structure | 控制台设备驱动程序，统一的vfs接口的实现 */
STATIC const struct file_operations_vfs g_consoleDevOps = {
    .open = ConsoleOpen,   /* open */
    .close = ConsoleClose, /* close */
    .read = ConsoleRead,   /* read */
    .write = ConsoleWrite, /* write */
    .seek = NULL,
    .ioctl = Consoleioctl,
    .mmap = NULL,
#ifndef CONFIG_DISABLE_POLL
    .poll = ConsolePoll,
#endif
};
```

- `fifo` 用于 `termios` (线路规程)的规范模式，输入数据基于行进行处理。在用户输入一个行结束符（回车符、EOF等）之前，系统调用read()读不到用户输入的任何字符。除了EOF之外的行结束符（回车符等），与普通字符一样会被read()读到缓冲区 `fifo` 中。在规范模式中，可以进行行编辑，而且一次调用read()最多只能读取一行数据。如果read()请求读取的数据字节少于当前行可读取的字节，则read()只读取被请求的字节数，剩下的字节下次再读。详细内容见系列篇之 **线路规程篇**

- `CirBufSendCB` 是专用于 `SendToSer` 任务的结构体，任务之间通过事件相互驱动，控制台通知 `SendToSer` 将数据发送给终端

```
/**
 * @brief 发送环形buf控制块，通过事件发送
 */
typedef struct {
    CirBuf cirBufCB;        /* Circular buffer CB | 循环缓冲控制块 */
    EVENT_CB_S sendEvent;   /* Inform telnet send task | 例如: 给SendToSer任务发送事件*/
} CirBufSendCB;
```

发送数据给终端的任务 | ConsoleSendTask

`ConsoleSendTask` 只干一件事，将数据发送给串口或远程登录，任务优先级与 `shell` 同级，为 `9` ，它由系统初始化任务 `SystemInit` 创建， 具体可翻看系列篇之**内核启动篇**

```
/// 控制台缓存初始化，创建一个 发送任务
STATIC UINT32 OsConsoleBufInit(CONSOLE_CB *consoleCB)
{
    UINT32 ret;
    TSK_INIT_PARAM_S initParam = {0};
    consoleCB->cirBufSendCB = ConsoleCirBufCreate();//创建控制台
    if (consoleCB->cirBufSendCB == NULL) {
        return LOS_NOK;
```

```
    }
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)ConsoleSendTask;//控制台发送任务入口函数
    initParam.usTaskPrio   = SHELL_TASK_PRIORITY; //优先级9
    initParam.auwArgs[0]   = (UINTPTR)consoleCB; //入口函数的参数
    initParam.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE; //16K
    if (consoleCB->consoleID == CONSOLE_SERIAL) {//控制台的两种方式
        initParam.pcName   = "SendToSer"; //任务名称(发送数据到串口)
    } else {
        initParam.pcName   = "SendToTelnet";//任务名称(发送数据到远程登录)
    }
    initParam.uwResved     = LOS_TASK_STATUS_DETACHED; //使用任务分离模式
    ret = LOS_TaskCreate(&consoleCB->sendTaskID, &initParam);//创建task 并加入就绪队列，申请立即调度
    if (ret != LOS_OK) { //创建失败处理
        ConsoleCirBufDelete(consoleCB->cirBufSendCB);//释放循环buf
        consoleCB->cirBufSendCB = NULL;//置NULL
        return LOS_NOK;
    }//永久等待读取 CONSOLE_SEND_TASK_RUNNING 事件，CONSOLE_SEND_TASK_RUNNING 由 ConsoleSendTask 发出。
    (VOID)LOS_EventRead(&consoleCB->cirBufSendCB->sendEvent, CONSOLE_SEND_TASK_RUNNING,
                        LOS_WAITMODE_OR | LOS_WAITMODE_CLR, LOS_WAIT_FOREVER);
  // ... 读取到 CONSOLE_SEND_TASK_RUNNING 事件才会往下执行
    return LOS_OK;
}
```

任务的入口函数`ConsoleSendTask`实现也很简单，此处全部贴出来，死循环等待事件的发送。说到死循环多说两句，不要被`while (1)`吓倒，认为内核会卡死在这里玩不下去，那是应用程序员看待死循环的视角，其实在内核当等待的事件没有到来的时，这个任务并不会往下执行，而是处于挂起状态，当事件到来时才会切换回来继续往下走，那如何知道事件到来了呢？ 可翻看系列篇之**事件控制篇**

```
STATIC UINT32 ConsoleSendTask(UINTPTR param)
{
    CONSOLE_CB *consoleCB = (CONSOLE_CB *)param;
    CirBufSendCB *cirBufSendCB = consoleCB->cirBufSendCB;
    CirBuf *cirBufCB = &cirBufSendCB->cirBufCB;
    UINT32 ret, size;
    UINT32 intSave;
    CHAR *buf = NULL;
    (VOID)LOS_EventWrite(&cirBufSendCB->sendEvent, CONSOLE_SEND_TASK_RUNNING);//发送一个控制台任务正在运行的事件
    while (1) {//读取 CONSOLE_CIRBUF_EVENT | CONSOLE_SEND_TASK_EXIT 这两个事件
        ret = LOS_EventRead(&cirBufSendCB->sendEvent, CONSOLE_CIRBUF_EVENT | CONSOLE_SEND_TASK_EXIT,
                            LOS_WAITMODE_OR | LOS_WAITMODE_CLR, LOS_WAIT_FOREVER);//读取循环buf或任务退出的事件
        if (ret == CONSOLE_CIRBUF_EVENT) {//控制台循环buf事件发生
            size =  LOS_CirBufUsedSize(cirBufCB);//循环buf使用大小
            if (size == 0) {
                continue;
            }
            buf = (CHAR *)LOS_MemAlloc(m_aucSysMem1, size + 1);//分配接收cirbuf的内存
            if (buf == NULL) {
                continue;
            }
            (VOID)memset_s(buf, size + 1, 0, size + 1);//清0
            LOS_CirBufLock(cirBufCB, &intSave);
            (VOID)LOS_CirBufRead(cirBufCB, buf, size);//读取循环cirBufCB至  buf
            LOS_CirBufUnlock(cirBufCB, intSave);

            (VOID)WriteToTerminal(consoleCB, buf, size);//将buf数据写到控制台终端设备
            (VOID)LOS_MemFree(m_aucSysMem1, buf);//清除buf
        } else if (ret == CONSOLE_SEND_TASK_EXIT) {//收到任务退出的事件， 由 OsConsoleBufDeinit 发出事件。
            break;//退出循环
        }
    }
    ConsoleCirBufDelete(cirBufSendCB);//删除循环buf，归还内存
    return LOS_OK;
}
```

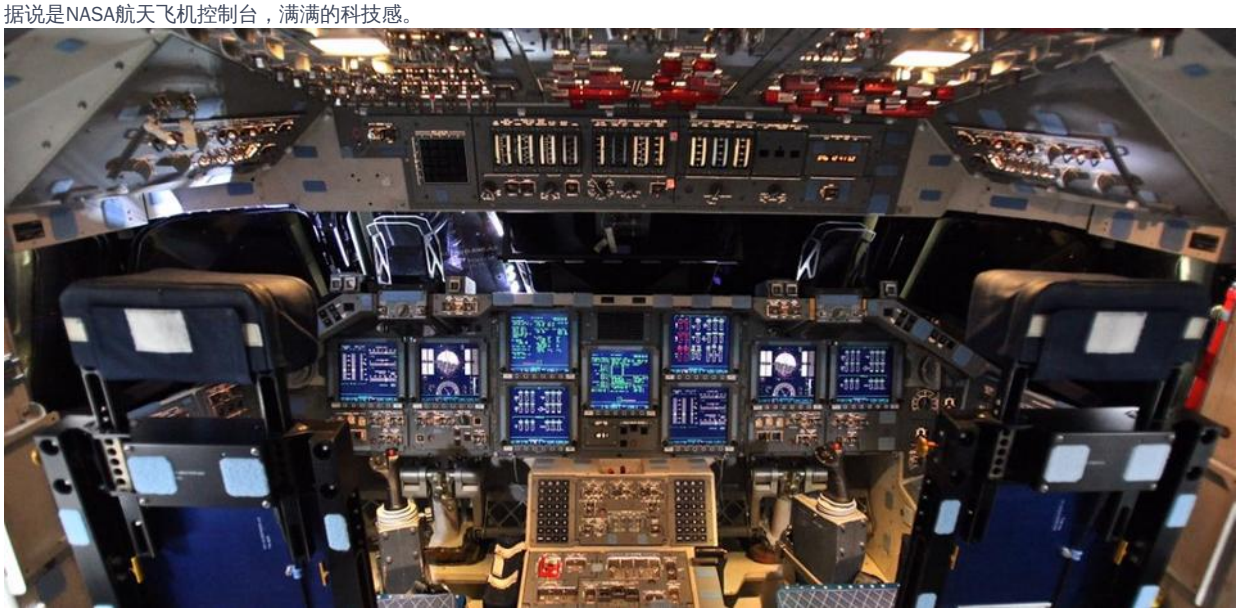
上面提到了控制台和终端，是经常容易搞混的又变得越来越模糊两个概念，简单说明下。

传统的控制台和终端

控制台(console)和终端(terminal)有什么区别？ 看张古老的图

这个不陌生吧，实现中虽很少看到，可电影里可没少出现。

据说是NASA航天飞机控制台，满满的科技感。



这就是控制台。早期控制台其实是给系统管理人员使用的。因为机器很大，价格很贵，不可能让每个人都拥有一个真正物理上属于自己的计算机，但是只让一个人用那其他人怎么办？ 效率太低，就出现了多用户多任务计算机，让一台计算机多个人同时登录使用的情况， 给每个人面前放个简单设备(只有键盘和屏幕)连接到主机上，如图所示

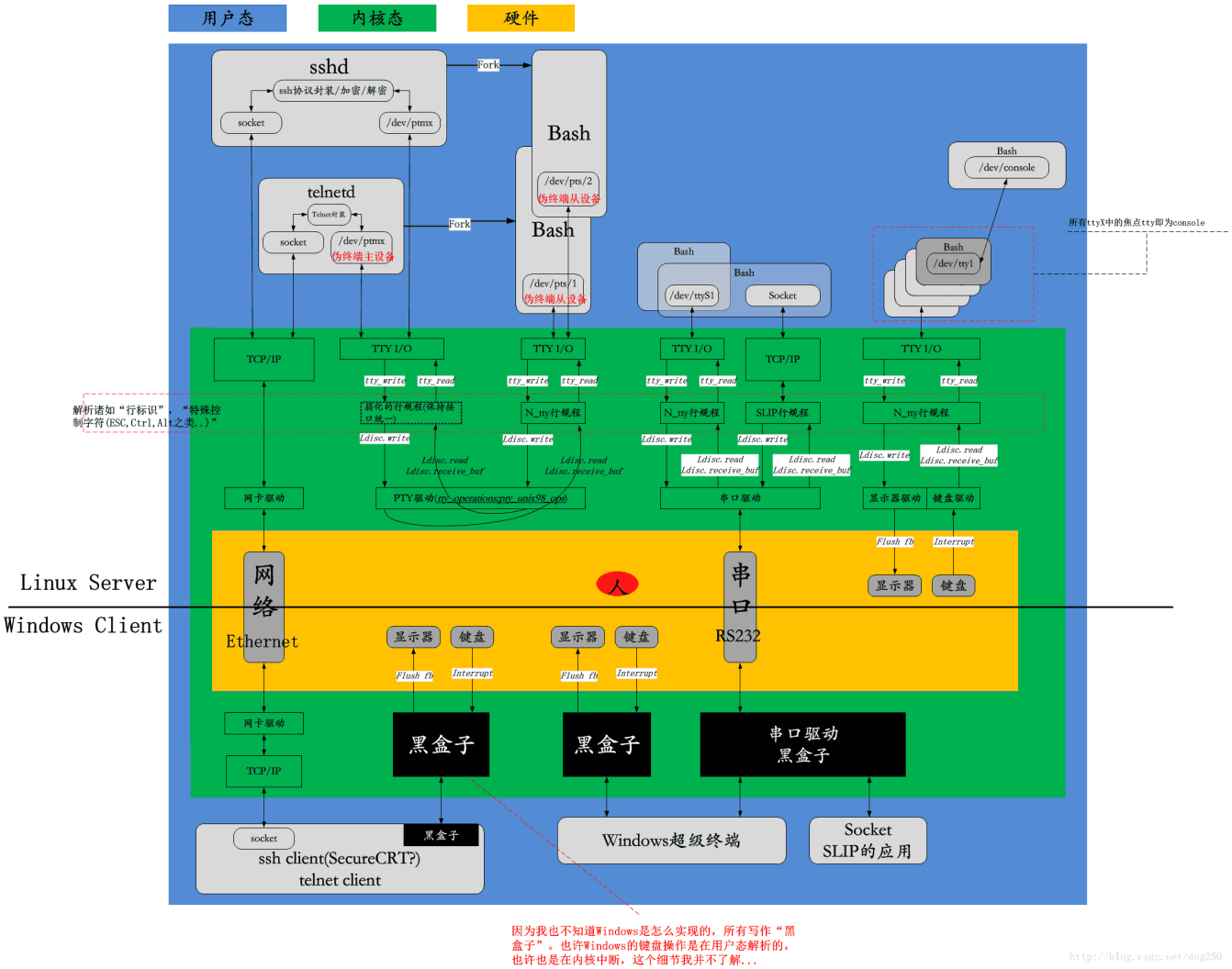
这个就叫**终端** ，注意别看那么大，长得很像一体机，但其实它只是一台显示器。这是给普通用户使用，权限也有限，核心功能权限还是在操作控制台的系统管理员手上。

综上所述，用图表列出二者早期差异

区别	终端（terminal）	控制台(console)
设备属性	外挂的附加设备	自带的基本设备
数量	多个	一个
主机信任度	低	高
输出内容	主机处理的信息	主机核心/自身信息
操作员	普通用户	管理员

现在的控制台和终端

由于时代的发展计算机的硬件越来越便宜，现在都是一个人独占一台计算机（个人电脑），已经不再需要传统意义上的硬件终端。现在终端和控制台都由硬件概念，逐渐演化成了软件的概念。终端和控制台的界限也慢慢模糊了，复杂了，甚至控制台也变成了终端， 现在要怎么理解它们，推荐一篇文章，请自行前往搜看。 << **彻底理解Linux的各种终端类型以及概念** >>



本篇内容与图中右上角的`/dev/console`那部分相关。 从鸿蒙内核视角来看，控制台和终端还是有很大差别的。

百文说内核 | 抓住主脉络

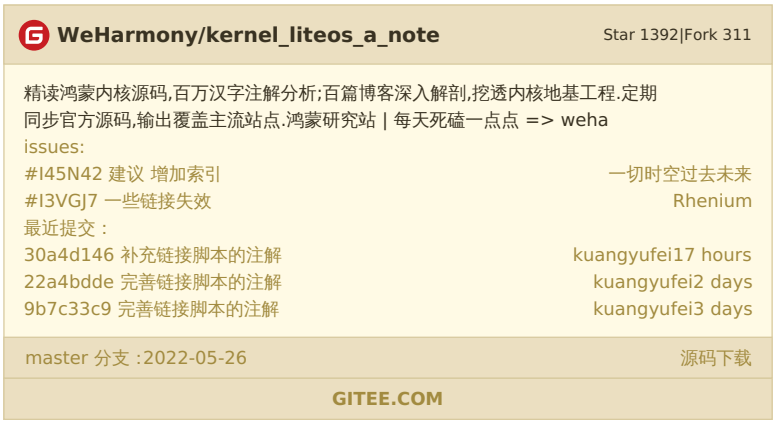
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

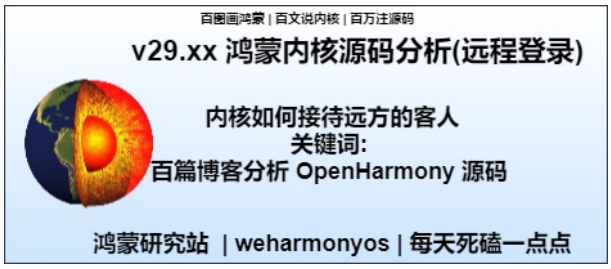


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

29_远程登录篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

什么是远程登录？

- 每个人都有上门做客的经历，抖音也一直在教我们做人，做客不要空手去，总得带点东西，而对中国人你就不能送**钟**，不能送**梨**，最好也别送**鞋**，因他们与 **终 离 邪** 谐音，犯忌讳。 这是人情世故，叫礼仪，是中华文明圈的共识，是相互交流信任的基础。
- 那互联网圈有没有这种共识呢？ 当然有，互联网世界的人情世故就是协议， 种种协议映射到人类社会来说就是种种礼仪，协议有`TCP`，`HTTP`，`SSH`，`Telnet`等等，就如同礼仪分商业礼仪，外交礼仪，校园礼仪，家庭礼仪等等。 孔圣人不也说 **不学礼，无以立** 应该就是这个道理， 登门拜访的礼仪可类比远程登录协议`Telnet`。 来了就跟自己家一样， 我家的东西就是你家的，随便用，甭客气。

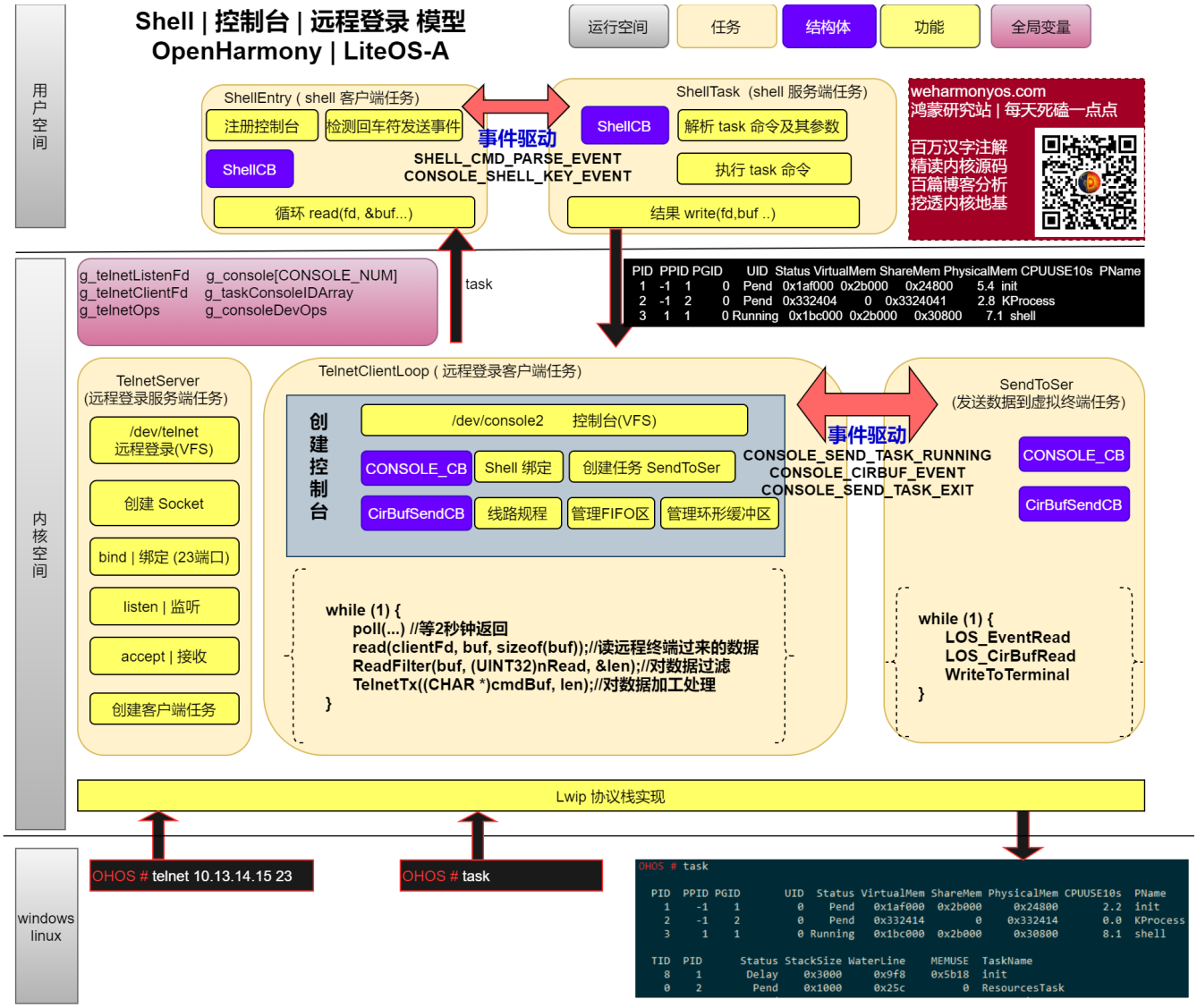
`Telnet`协议的具体内容可以查看以下文档。

协议	时间	英文版	中文版	标题
Telnet	1983	rfc854	rfc854	TELNET PROTOCOL SPECIFICATION(远程登录协议规范)
Telnet	1983	rfc855	rfc855	TELNET OPTION SPECIFICATIONS(远程登录选项规范)

`Telnet`协议细节不是本篇讨论的重点，后续会有专门的 **Lwip协议栈** 系列博客说清楚。本篇要说清楚的是内核如何接待远方的客人。

Shell | 控制台 | 远程登录模型

对远程登录来有客户端和服务端的说法，跟别人来你家你是主人和你去别人家你是客人一样，身份不同，职责不同，主人要做的事明显要更多，本篇只说鸿蒙对`telnet`服务端的实现，说清楚它是如何接待外面来的客人。至于图中提到的客户端任务是指主人为每个客人专门提供了一个对接人的意思。**下图**为看完三部分源码后整理的模型图



模型解释

- 通过本地的 shell 命令 telnet on 启动远程登录模块，由此创建Telnet的服务任务 TelnetServer
- TelnetServer 任务，创建 socket 监听 23 端口，接受来自远程终端的 telnet xx.xx.xx.xx 23 请求
- 收到请求后创建一个 TelnetClientLoop 用于接待客户的任务，对接详细的客户需求。
- 在接待客户期间创建一个远程登录类型的控制台，来处理和转发远程客户的请求给 shell 进程最终执行远程命令。
- shell 处理完成后通过专门的任务 SendToSer 回写远程终端，控制台部分详细看 系列篇的(控制台篇)

鸿蒙是如何实现的？

1. 启动 Telnet

```
//SHELLCMD_ENTRY(telnet_shellcmd， CMD_TYPE_EX， "telnet"， 1， (CmdCallBackFunc)TelnetCmd);/// 以静态方式注册shell 命令
/// 本命令用于启动或关闭telnet server服务
INT32 TelnetCmd(UINT32 argc, const CHAR **argv)
{
    if (strcmp(argv[0], "on") == 0) { // 输入 telnet on
        /* telnet on: try to start telnet server task */
        TelnetdTaskInit(); //启动远程登录 服务端任务
        return 0;
    }
    if (strcmp(argv[0], "off") == 0) {// 输入 telnet off
        /* telnet off: try to stop clients, then stop server task */
        TelnetdTaskDeinit();//关闭所有的客户端,并关闭服务端任务
```

```
        return 0;
    }
    return 0;
}
```

2. 创建Telnet服务端任务

```
STATIC VOID TelnetdTaskInit(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S initParam = {0};
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)TelnetdMain; // telnet任务入口函数
    initParam.uwStackSize = TELNET_TASK_STACK_SIZE; // 8K
    initParam.pcName = "TelnetServer"; //任务名称
    initParam.usTaskPrio = TELNET_TASK_PRIORITY; //优先级 9,和 shell 的优先级一样
    initParam.uwResved = LOS_TASK_STATUS_DETACHED; //独立模式
    if (atomic_read(&g_telnetTaskId) != 0) {//只支持一个 telnet 服务任务
        PRINT_ERR("telnet server is already running!\n");
        return;
    }
    ret = LOS_TaskCreate((UINT32 *)&g_telnetTaskId, &initParam);//创建远程登录服务端任务并发起调度
}
```

3. Telnet服务端任务入口函数

```
//远程登录操作命令
STATIC const struct file_operations_vfs g_telnetOps = {
    TelnetOpen,
    TelnetClose,
    TelnetRead,
    TelnetWrite,
    NULL,
    TelnetIoctl,
    NULL,
#ifndef CONFIG_DISABLE_POLL
    TelnetPoll,
#endif
    NULL,
};
STATIC INT32 TelnetdMain(VOID)
{
    sock = TelnetdInit(TELNETD_PORT);//1.初始化创建 socket ,socket的本质就是打开了一个虚拟文件
    TelnetLock();
    ret = TelnetedRegister();//2.注册驱动程序 /dev/telnet ,g_telnetOps g_telnetDev
    TelnetUnlock();
    TelnetdAcceptLoop(sock);//3.等待连接,处理远程终端过来的命令 例如#task 命令
    return 0;
}
```

4. 循环等待远程终端的连接请求

```
STATIC VOID TelnetdAcceptLoop(INT32 listenFd)
{
    while (g_telnetListenFd >= 0) {//必须启动监听
        TelnetUnlock();
        (VOID)memset_s(&inTelnetAddr, sizeof(inTelnetAddr), 0, sizeof(inTelnetAddr));
        clientFd = accept(listenFd, (struct sockaddr *)&inTelnetAddr, (socklen_t *)&len);//接收数据
        if (TelnetdAcceptClient(clientFd, &inTelnetAddr) == 0) {//
            /*
             * Sleep sometime before next loop: mostly we already have one connection here,
             * and the next connection will be declined. So don't waste our cpu.
             | 在下一个循环来临之前休息片刻,因为鸿蒙只支持一个远程登录,此时已经有一个链接,
             在TelnetdAcceptClient中创建线程不会立即调度, 休息下任务会挂起,重新调度
             */
            LOS_Msleep(TELNET_ACCEPT_INTERVAL);//以休息的方式发起调度. 直接申请调度也未尝不可吧 @note_thinking
        } else {
```

```
            return;
        }
        TelnetLock();
    }
    TelnetUnlock();
}
```

5. 远方的客人到来,安排专人接待

鸿蒙目前只支持接待一位远方的客人，`g_telnetClientFd`是个全局变量，创建专门任务接待客人。

```
STATIC INT32 TelnetdAcceptClient(INT32 clientFd, const struct sockaddr_in *inTelnetAddr)
{
    g_telnetClientFd = clientFd;
 //创建一个线程处理客户端的请求
    if (pthread_create(&tmp, &useAttr, TelnetClientLoop, (VOID *)(UINTPTR)clientFd) != 0) {
        PRINT_ERR("Failed to create client handle task\n");
        g_telnetClientFd = -1;
        goto ERROUT_UNLOCK;
    }
}
```

6. 接待员做好接待工作

因接待工作很重要，这边把所有代码贴出来，并加上了大量的注释，目的只有一个，让咱客人爽。

```
STATIC VOID *TelnetClientLoop(VOID *arg)
{
    struct pollfd pollFd;
    INT32 ret;
    INT32 nRead;
    UINT32 len;
    UINT8 buf[TELNET_CLIENT_READ_BUF_SIZE];
    UINT8 *cmdBuf = NULL;
    INT32 clientFd = (INT32)(UINTPTR)arg;
    (VOID)prctl(PR_SET_NAME, "TelnetClientLoop", 0, 0, 0);
    TelnetLock();
    if (TelnetClientPrepare(clientFd) != 0) {//做好准备工作
        TelnetUnlock();
        (VOID)close(clientFd);
        return NULL;
    }
    TelnetUnlock();
    while (1) {//死循环接受远程输入的数据
        pollFd.fd = clientFd;
        pollFd.events = POLLIN | POLLRDHUP;//监听读数据和挂起事件
        pollFd.revents = 0;
 /*
 POLLIN 普通或优先级带数据可读
 POLLRDNORM 普通数据可读
 POLLRDBAND 优先级带数据可读
 POLLPRI 高优先级数据可读
 POLLOUT 普通数据可写
 POLLWRNORM 普通数据可写
 POLLWRBAND 优先级带数据可写
 POLLERR 发生错误
 POLLHUP 发生挂起
 POLLNVAL 描述字不是一个打开的文件
 poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，
 如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，
 直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。
      这个过程经历了多次无谓的遍历。
      poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。
      poll与select的不同，通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，
  pollfd中的events字段和revents分别用于标示关注的事件和发生的事件，故pollfd数组只需要被初始化一次
      poll的实现机制与select类似，其对应内核中的sys_poll，只不过poll向内核传递pollfd数组，
 然后对pollfd中的每个描述符进行poll，相比处理fdset来说，poll效率更高。poll返回后，
 需要对pollfd中的每个元素检查其revents值，来得指事件是否发生。
```

```
    优点
    1）poll() 不要求开发者计算最大文件描述符加一的大小。
    2）poll() 在应付大数目的文件描述符的时候速度更快，相比于select。
    3）它没有最大连接数的限制，原因是它是基于链表来存储的。
    缺点
    1）大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。
    2）与select一样，poll返回后，需要轮询pollfd来获取就绪的描述符
   */
        ret = poll(&pollFd, 1, TELNET_CLIENT_POLL_TIMEOUT);//等2秒钟返回
        if (ret < 0) {//失败时，poll()返回-1
            break;
    /* ret < 0 各值
          EBADF       一个或多个结构体中指定的文件描述符无效。
          EFAULTfds     指针指向的地址超出进程的地址空间。
          EINTR         请求的事件之前产生一个信号，调用可以重新发起。
          EINVALnfds    参数超出PLIMIT_NOFILE值。
          ENOMEM        可用内存不足，无法完成请求

    */
        }
        if (ret == 0) {//如果在超时前没有任何事件发生，poll()返回0
            continue;
        }
        /* connection reset， maybe keepalive failed or reset by peer | 连接重置，可能keepalive失败或被peer重置*/
        if ((UINT16)pollFd.revents & (POLLERR | POLLHUP | POLLRDHUP)) {
            break;
        }
        if ((UINT16)pollFd.revents & POLLIN) {//数据事件
            nRead = read(clientFd, buf, sizeof(buf));//读远程终端过来的数据
            if (nRead <= 0) {
                /* telnet client shutdown */
                break;
            }
            cmdBuf = ReadFilter(buf, (UINT32)nRead, &len);//对数据过滤
            if (len > 0) {
                (VOID)TelnetTx((CHAR *)cmdBuf, len);//对数据加工处理
            }
        }
    }
    TelnetLock();
    TelnetClientClose();
    (VOID)close(clientFd);
    clientFd = -1;
    g_telnetClientFd = -1;
    TelnetUnlock();
    return NULL;
}
```

最后结语

理解远程登录的实现建议结合 `shell编辑篇` ， `shell执行篇` ， `控制台篇` 三篇来理解，实际上它们是上中下三层。

百文说内核 | 抓住主脉络

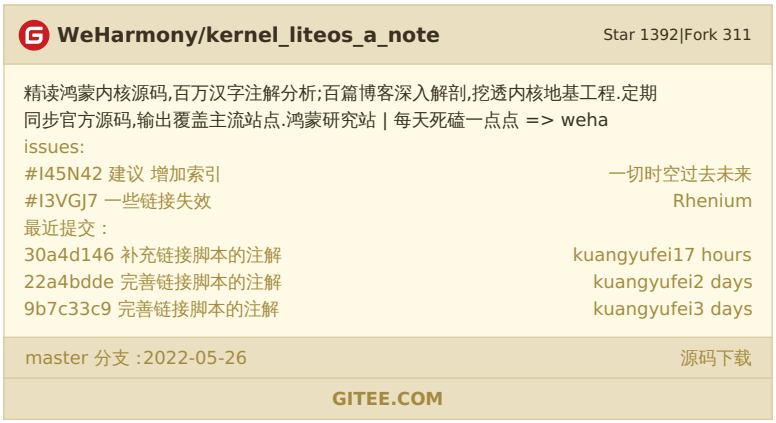
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

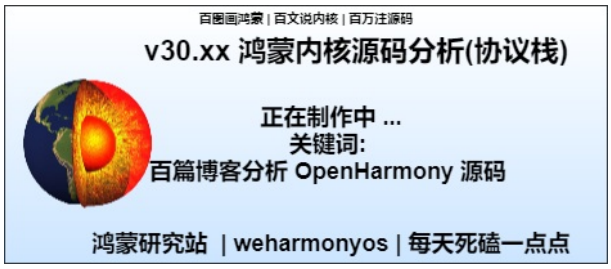


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

30_协议栈篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

任务管理相关篇为:

- v21.07 鸿蒙内核源码分析(任务控制块) | 内核最重要的概念
- v22.05 鸿蒙内核源码分析(并发并行) | 如何搞清楚它俩区分
- v23.03 鸿蒙内核源码分析(就绪队列) | 美好的事物永远值得等待
- v24.08 鸿蒙内核源码分析(调度机制) | 公平是相对的
- v25.05 鸿蒙内核源码分析(任务管理) | 如何管理任务池
- v26.03 鸿蒙内核源码分析(用栈方式) | 谁来提供程序运行场地
- v27.02 鸿蒙内核源码分析(软件定时器) | 内核最高级任务竟是它
- v28.01 鸿蒙内核源码分析(控制台) | 一个让很多人模糊的概念
- v29.01 鸿蒙内核源码分析(远程登录) | 内核如何接待远方的客人
- v30.01 鸿蒙内核源码分析(协议栈) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

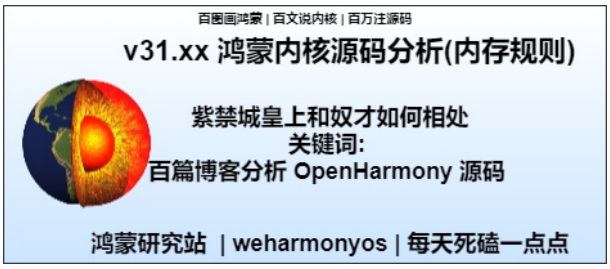
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

31_内存规则篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

主子和奴才

看本篇之前建议先看 鸿蒙内核源码分析(调度故事篇)。 请想一个问题，内核本身也是程序要在内存运行， 用户程序一样也要在内存运行，大家都在一个窝里吃饭， 你凭什么就管我了。好像内核程序是主子，用户程序是奴才似的。

哎! 其实用户进程就是内核的一个个奴才， 被捏的死死的。 按不住奴才那这主子就不合格，就不是一个稳定系统。 请想想实际内存就这么点大， 如何满足众多用户进程的需求？ 内核空间和用户空间如何隔离？ 如何防止访问乱串？ 如何分配/释放，防止碎片化？ 空间不够了又如何置换到硬盘？ 想想头都大了。内核这当家的主子真是不容易，这些都是他要解决的问题， 但欲戴其冠，必承其重。

先说如果没有内存管理会怎样？

那就是个奴才们能把主子给活活踩死， 想想主奴不分，吃喝拉撒睡都在一起，称兄道弟的想干啥？ 没规矩不成方圆嘛，这事业肯定搞不大，单片机时代就是这种情况。 裸机编程，指针可以随便乱飞，数据可以随意覆盖，没有划定边界，没有明确职责，没有特权指令，没有地址保护，你还想像java开发一样，只管new内存，不去释放，应用可以随便崩但系统跑的妥妥的？想的美! 直接系统死机，甚至开机都开不了，主板直接报废了。所以不能运行很复杂的程序，尽量可控，而且更是不可能支持应用的动态加载运行。 队伍大了就不好带了，方法得换， 游击队的做法不适合规模作战，内存就需要管理了，而且是 5A级的严格管理。

内存管理在管什么？

简单说就是给主子赋能，拥有超级权利，为什么就他有？ 因为他先来，掌握了先机。它定好了游戏规则，你们来玩。有哪些游戏规则？

- 第一： 主奴有别，主子即是裁判又是运动员，主子有主子地方，奴才们有奴才们待的地方，主子可以在你的空间走来走去，但你只能在主人划定的区域活动。奴才把自己玩崩了也只是奴才狗屁了， 但主人和其他人还会是好好的。 主子有所有特权，比如某个奴才太嚣张了，就直接拖到午门问斩。

- 第二： 奴奴有分，奴才们基本都是平等的，虽有高级和低级奴才区分，但本质都是奴才。奴才之间是不能随意勾连，登门问客的，防止一块搞政变。 他们都有属于自己的活动空间，而且活动空间还巨大巨大，大到奴才们觉得整个紫荆城都是他们家的，给你这么大空间你干活才有动力，奴才们是铆足了劲一个个尽情的表演各种剧本，有玩电子商务的，有玩游戏的，有搞直播的等等...不愧是紫荆城的主人很有一套，明明只有一个紫禁城，硬被他整出了N个紫荆城的感觉。而且这套驾奴本领还取了个很好听的名字叫：`虚拟内存`。

看图：

这是整个紫荆城的全貌图，里面的内核虚拟空间是主人专用的，里面放的是主人的资料，数据，奴才永远进不去，kernel heap 也是给主人专用的动态内存空间，管理奴才和日常运作开销很多时候需要动态申请内存，这个是专门用来提供给主人使用的。而所有奴才的空间都在叫用户空间的那一块。你没看错，是所有奴才的都在那。当然实际情况是用户空间比图中的大的多，因为主人其实用不了多少空间，大部分是留给奴才们干活用了，因

为篇幅的限制笔者把用户空间压缩了下。 再来看看奴才空间是啥样的。看图

这张图是第一张图的局部用户空间放大图。里面放的是奴才的私人用品，数据，task运行栈区，动态分配内存的堆区，堆区自下而上，栈区自上而下中间由映射区(L1，L2表)隔开。这么多奴才在里面不挤吗？答案是：真不挤 。主人手眼通天，因为用了一个好帮手解决了这个问题，这个帮手名叫 MMU（李大总管）

MMU是干什么事的？

看下某度对MMU定义：它是一种负责处理中央处理器（CPU）的内存访问请求的计算机硬件.它的功能包括虚拟地址到物理地址的转换（即虚拟内存管理）、内存保护、中央处理器高速缓存的控制.通过它的一番操作，把物理空间成倍成倍的放大，他们之间的映射关系存放在页面中。

好像看懂又好像没看懂是吧，到底是干啥的？其实就是个地址映射登记中心。记住这两个字：映射 看下图

物理内存可以理解为真实世界的紫禁城，虚拟内存就是被MMU虚拟出来的比物理页面大的多的空间。举例说明大概说明下过程：

有A(厨师)，B(文艺青年) 两个奴才来到紫禁城，每个人都很有抱负，主子规定要先跑去登记处登记活动范围，领回来一张表 叫 L1页表，上面说了大半个紫禁城你可以跑动，都是你的，L1页表记录你每个房间的编号。其实奴才们的表都一样，能跑的范围也都一样。 李大总管也有一张私人表叫TLB表，具体玩的呢，看个例子就明白了。

举例说明

TLB表(李总管的私人表)

```
|真实房间|当前谁在用|
--------------------
|7     |  A     |
|8     |  C     |
|9     |  B     |
```

李大总管的私人表叫 TLB（translation lookaside buffer）可翻译为“地址转换后援缓冲器”，也可简称为“快表”。从TLB表可以看出，有三个真实的房间， 7，8，9，目前是分配给了A，B，C使用。

奴才们的L1页表(当然可以有无数的奴才表，每个奴才人手一张)

```
|奴才 |虚拟房间|真实房间|作用|
---------------------------
|A奴才| 1   |7      |厨房拿菜|
|A奴才| 2   |8      |洗手间|
|A奴才| 3   |9      |卧室|

|   |虚拟房间|真实房间|作用|
----------------------------
|B奴才| 3 |8       |音乐室|
|B奴才| 1 |9       |美术室|
|B奴才| 2 |7       |武术室|
```

再模拟一个他们的活动场景:

```
|奴才|动作1|动作2|动作3|动作4|
----------------------------------
|A  |厨房拿菜|卧室睡觉|上洗手间|无|
|B  |武术室|美术室|无|音乐室|
```

第一： A要去1号间厨房拿菜，提交表给李总管，李总管拿表和自己的表对照，发现1号虚拟房间对应的是7号真实房间，7号刚好分配给了A用，盖章同意。A拿到了自己菜。

```
|真实房间|当前谁在用|
-----------------------
|7     |A         |
|8     |C         |
|9     |B         |
```

此时李总管的表没变化。 第二： B要去2号间练武术，提交表给李总管，李总管拿表和自己的表对照，发现1号虚拟房间对应的是7号真实房间，7号是A在用，不属于B，里面放的都还是菜呢，咋办？简单，把菜挪出去，把B奴才的武术设备装进来，更改自己的表变成了

```
|真实房间|当前谁在用|
------------------------
|7     |B     |
|8     |C     |
|9     |B     |
```

此时李总管的表变了，三个真实房间B用了两个了。 第三： A要去3号间睡觉了，又提交表给李总管，李总管拿表和自己的表对照，发现3号虚拟房间对应的是9号真实房间，9号刚好分配给了B用了，此时里面放的还是美术用品呢。咋办？简单，挪出去，把A奴才的睡觉设备装进来，再更改自己的表变成了

```
|真实房间|当前谁在用|
------------------------
|7     |B     |
|8     |C     |
|9     |A     |
```

此时李总管的表变了，9号给了A了，而8号一直在C手里，因为过程中没人用到了8号房。但继续跑下去肯定会易主。

明白了吗？ 这就是 **映射的核心思想!** 对A，B来说，它们只认 1，2，3房间，记得自己的房间是干什么用的就行，完全不必知道背后的7，8，9是谁在用， 用房间之前提交表单就行了，后面的不用管。 而且各自1，2，3可以重新映射到不一样的房间， A，B映射是完全独立的，看清没有它们的123对应的可不都是789的顺序。

上面的1，2，3就叫虚拟地址，也叫线性地址。 而789就是物理地址。 如此只有三个房间都可以给很多很多的奴才使用，让他们觉得这三个房间都是自己的。 完美!!! 当然AB也可以有自己虚拟地址789，例如:

```
|奴才|虚拟房间|真实房间|作用|
------------------------
|A奴才| 1 |7|厨房拿菜|
|A奴才| 2 |8|洗手间|
|A奴才| 3 |9|卧室|
|A奴才| 7 |19|洗澡|
|A奴才| 8 |88|去皇上寝宫偷看|
|A奴才| 9 |45|御膳房|
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

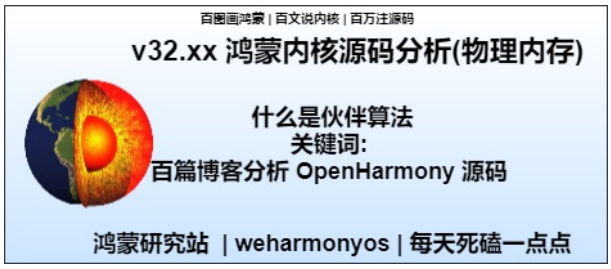
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

32_物理内存篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

如何初始化物理内存？

鸿蒙内核物理内存采用了段页式管理，先看两个主要结构体。结构体的每个成员变量的含义都已经注解出来，请结合源码理解。

```
#define VM_LIST_ORDER_MAX    9    //伙伴算法分组数量，从 2^0，2^1，...，2^8 (256*4K)=1M
#define VM_PHYS_SEG_MAX    32    //最大支持32个段

typedef struct VmPhysSeg {//物理段描述符
    PADDR_T start;            /* The start of physical memory area */ //物理内存段的开始地址
    size_t size;              /* The size of physical memory area */ //物理内存段的大小
    LosVmPage *pageBase;      /* The first page address of this area */ //本段首个物理页框地址
    SPIN_LOCK_S freeListLock; /* The buddy list spinlock */    //伙伴算法自旋锁，用于操作freeList上锁
    struct VmFreeList freeList[VM_LIST_ORDER_MAX];  /* The free pages in the buddy list */ //伙伴算法的分组，默认分成10组 2^0，2^1，...，2^VM_LI
    SPIN_LOCK_S lruLock;  //用于置换的自旋锁，用于操作lruList
    size_t lruSize[VM_NR_LRU_LISTS];  //5个双循环链表大小，如此方便得到size
    LOS_DL_LIST lruList[VM_NR_LRU_LISTS]; //页面置换算法，5个双循环链表头，它们分别描述五中不同类型的链表
} LosVmPhysSeg;


//注意: vmPage 中并没有虚拟地址，只有物理地址
typedef struct VmPage { //物理页框描述符
    LOS_DL_LIST         node;       /**< vm object dl list */ //虚拟内存节点，通过它挂/摘到全局g_vmPhysSeg[segID]->freeList[order]物理页框链表上
    UINT32              index;      /**< vm page index to vm object */ //索引位置
    PADDR_T             physAddr;    /**< vm page physical addr */  //物理页框起始物理地址，只能用于计算，不会用于操作(读/写数据==)
    Atomic              refCounts;   /**< vm page ref count */   //被引用次数，共享内存会被多次引用
    UINT32              flags;       /**< vm page flags */    //页标签，同时可以有多个标签（共享/引用/活动/被锁==）
    UINT8               order;       /**< vm page in which order list */ //被安置在伙伴算法的几号序列(            2^0，2^1，2^2，...，2^order)
    UINT8               segID;       /**< the segment id of vm page */ //所属段ID
    UINT16              nPages;      /**< the vm page is used for kernel heap */ //分配页数，标识从本页开始连续的几页将一块被分配
} LosVmPage;//注意:关于nPages和order的关系说明，当请求分配为5页时，order是等于3的，因为只有2^3才能满足5页的请求
```

理解它们是理解物理内存管理的关键，尤其是 **LosVmPage ，**鸿蒙内存模块代码通篇都能看到它的影子。内核默认最大允许管理32个段。

段页式管理简单说就是先将物理内存切成一段段，每段再切成单位为 4K 的物理页框， 页是在内核层的操作单元， 物理内存的分配，置换，缺页，内存共享，文件高速缓存的读写，都是以页为单位的，所以**LosVmPage 很重要，很重要!**

结构体的每个变量代表了一个个的功能点， 结构体中频繁了出现LOS_DL_LIST的身影，双向链表是鸿蒙内核最重要的结构体，在系列篇开篇就专门讲过它的重要性。

再比如 LosVmPage.refCounts 页被引用的次数，可理解被进程拥有的次数，当refCounts大于1时，被多个进程所拥有，说明这页就是共享页。当等于0时，说明没有进程在使用了，这时就可以被释放了。

看到这里熟悉JAVA的同学是不是似曾相识，这像是Java的内存回收机制。在内核层面，引用的概念不仅仅适用于内存模块，也适用于其他模块，比如文件/设备模块，同样都存在共享的场景。这些模块不在这里展开说，后续有专门的章节细讲。

段一开始是怎么划分的 ？ 需要方案提供商手动配置，存在静态的全局变量中，鸿蒙默认只配置了一段。

```
struct VmPhysSeg g_vmPhysSeg[VM_PHYS_SEG_MAX];//物理段数组，最大32段
INT32 g_vmPhysSegNum = 0; //总段数
LosVmPage *g_vmPageArray = NULL;//物理页框数组
size_t g_vmPageArraySize;//总物理页框数


/* Physical memory area array */
STATIC struct VmPhysArea g_physArea[] = {//这里只有一个区域，即只生成一个段
    {
        .start = SYS_MEM_BASE， //整个物理内存基地址，#define SYS_MEM_BASE          DDR_MEM_ADDR ， 0x80000000
        .size = SYS_MEM_SIZE_DEFAULT，//整个物理内存总大小 0x07f00000
    }，
};
```

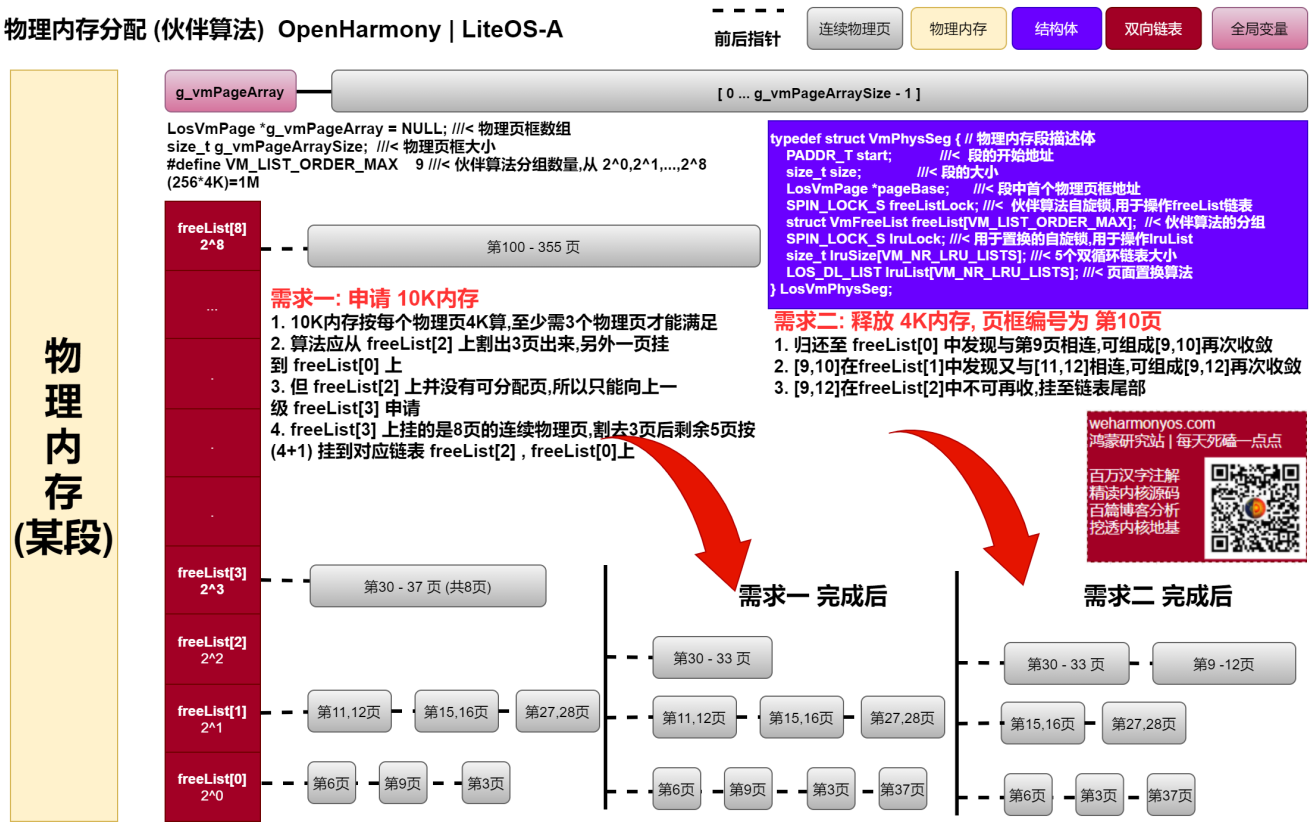
有了段和这些全局变量，就可以对内存初始化了。 OsVmPageStartup 是对物理内存的初始化， 它被整个系统内存初始化 OsSysMemInit所调用。 直接上代码。

```
/*****************************************************************************
 完成对物理内存整体初始化，本函数一定运行在实模式下
 1.申请大块内存g_vmPageArray存放LosVmPage，按4K一页划分物理内存存放在数组中。
*****************************************************************************/
VOID OsVmPageStartup(VOID)
{
    struct VmPhysSeg *seg = NULL;
    LosVmPage *page = NULL;
    paddr_t pa;
    UINT32 nPage;
    INT32 segID;

    OsVmPhysAreaSizeAdjust(ROUNDUP((g_vmBootMemBase - KERNEL_ASPACE_BASE)， PAGE_SIZE));//校正 g_physArea size

    nPage = OsVmPhysPageNumGet();//得到 g_physArea 总页数
    g_vmPageArraySize = nPage * sizeof(LosVmPage);//页表总大小
    g_vmPageArray = (LosVmPage *)OsVmBootMemAlloc(g_vmPageArraySize);//实模式下申请内存，此时还没有初始化MMU

    OsVmPhysAreaSizeAdjust(ROUNDUP(g_vmPageArraySize， PAGE_SIZE));//

    OsVmPhysSegAdd();// 完成对段的初始化
    OsVmPhysInit();// 加入空闲链表和设置置换算法，LRU(最近最久未使用)算法

    for (segID = 0; segID < g_vmPhysSegNum; segID++) {//遍历物理段，将段切成一页一页
        seg = &g_vmPhysSeg[segID];
        nPage = seg->size >> PAGE_SHIFT;//本段总页数
        for (page = seg->pageBase， pa = seg->start; page <= seg->pageBase + nPage;//遍历，算出每个页框的物理地址
             page++， pa += PAGE_SIZE) {
            OsVmPageInit(page， pa， segID);//对物理页框进行初始化，注意每页的物理地址都不一样
        }
        OsVmPageOrderListInit(seg->pageBase， nPage);//伙伴算法初始化，将所有页加入空闲链表供分配
    }
}
```

结合中文注释，代码很好理解， 此番操作之后全局变量里的值就都各就各位了，可以开始工作了。

如何分配/回收物理内存？ 答案是伙伴算法

伙伴算法系列篇中有说过好几篇，这里再看图理解下什么伙伴算法，伙伴算法注重**物理内存的连续性，注意是连续性!**



结合图比如，要分配4(2^2)页（16k）的内存空间，算法会先从free_area2中查看free链表是否为空，如果有空闲块，则从中分配，如果没有空闲块，就从它的上一级free_area3（每块32K）中分配出16K，并将多余的内存（16K）加入到free_area2中去。如果free_area3也没有空闲，则从更上一级申请空间，依次递推，直到free_area max_order，如果顶级都没有空间，那么就报告分配失败。

释放是申请的逆过程，当释放一个内存块时，先在其对于的free_area链表中查找是否有伙伴存在，如果没有伙伴块，直接将释放的块插入链表头。如果有或板块的存在，则将其从链表摘下，合并成一个大块，然后继续查找合并后的块在更大一级链表中是否有伙伴的存在，直至不能合并或者已经合并至最大块2^max_order为止。

看过系列篇文章的可能都发现了，笔者喜欢用讲故事和打比方来说明内核运作机制， 为了更好的理解，同样打个比方， 笔者认为伙伴算法很像是卖标准猪肉块的算法。

物理内存是一整头猪，已经切成了1斤1斤的了，但是还都连在一起，每一斤上都贴了个标号， 而且老板只按 1斤(2^0)， 2斤(2^1)， 4斤(2^2)，...256斤(2^8)的方式来卖。售货柜上分成了9组

张三来了要7斤猪肉，怎么办？ **给8斤，注意是给8斤啊 ，因为它要严格按它的标准来卖。**张三如果归还了，查看现有8斤组里有没有序号能连在一块的，有的话2个8斤合成16斤，放到16斤组里去。 如果没有这8斤猪肉将挂到上图中第2组(2^3)再卖。

大家脑海中有画面了吗？ 那么问题来了，它为什么要这么卖猪肉，好处是什么？ 简单啊:至少两个好处:

第一：卖肉速度快，效率高，标准化的东西最好卖了。

第二：可防止碎肉太多，后面的人想买大块的猪肉买不到了。 请仔细想想是不是这样的？如果每次客户来了要多少就割多少出去，运行一段时候后你还能买到10斤连在一块的猪肉吗？ 很可能给是一包碎肉，里面甚至还有一两一两的边角肉，碎肉的结果必然是管理麻烦，效率低啊。如果按伙伴算法的结果是运行一段时候后，图中0，1，2各组中都有可卖的猪肉啊，张三哥归还了那8斤(其实他指向要7斤)猪肉，王五兄弟来了要6斤，直接把张三哥归还的给王五就行了。效率极高。

那么问题又来了，凡事总有两面性，它的坏处是什么？ 也简单啊 :至少两个坏处:

第一：浪费了!，白给的三斤对王五没用啊，浪费的问题有其他办法解决，但不是在这个层面去解决，而是由 slab分配器解决，这里不重点说后续会专门讲slab分配器是如何解决这个问题的。

第二：合并要求太严格了，一定得是伙伴(连续)才能合并成更大的块。这样也会导致时间久了很难有大块的连续性的猪肉块。

比方打完了，鸿蒙内核是如何实现卖肉算法的呢？ 请看代码

```
LosVmPage *OsVmPhysPagesAlloc(struct VmPhysSeg *seg， size_t nPages)
{
    struct VmFreeList *list = NULL;
    LosVmPage *page = NULL;
    UINT32 order;
    UINT32 newOrder;

    if ((seg == NULL) || (nPages == 0)) {
        return NULL;
    }
 //因为伙伴算法分配单元是 1，2，4，8 页，比如nPages = 3时，就需要从 4号空闲链表中分，剩余的1页需要劈开放到1号空闲链表中
    order = OsVmPagesToOrder(nPages);//根据页数计算出用哪个块组
    if (order < VM_LIST_ORDER_MAX) {//order不能大于9 即:256*4K = 1M 可理解为向内核堆申请内存一次不能超过1M
        for (newOrder = order; newOrder < VM_LIST_ORDER_MAX; newOrder++) {//没有就找更大块
            list = &seg->freeList[newOrder];//从最合适的块处开始找
            if (LOS_ListEmpty(&list->node)) {//理想情况链表为空，说明没找到
                continue;//继续找更大块的
            }
            page = LOS_DL_LIST_ENTRY(LOS_DL_LIST_FIRST(&list->node)， LosVmPage， node);//找第一个节点就行，因为链表上挂的都是同样大小物理页框
            goto DONE;
        }
    }
    return NULL;
DONE:
    OsVmPhysFreeListDelUnsafe(page);//将物理页框从链表上摘出来
    OsVmPhysPagesSpiltUnsafe(page， order， newOrder);//将物理页框劈开，把用不了的页再挂到对应的空闲链表上
    return page;
}

/*****************************************************************************
 本函数很像卖猪肉的，拿一大块肉剁，先把多余的放回到小块肉堆里去。
 oldOrder:原本要买 2^2肉
 newOrder:却找到个 2^8肉块
*****************************************************************************/
STATIC VOID OsVmPhysPagesSpiltUnsafe(LosVmPage *page， UINT8 oldOrder， UINT8 newOrder)
{
    UINT32 order;
    LosVmPage *buddyPage = NULL;

    for (order = newOrder; order > oldOrder;) {//把肉剁碎的过程，把多余的肉块切成2^7，2^6...标准块，
        order--;//越切越小，逐一挂到对应的空闲链表上
        buddyPage = &page[VM_ORDER_TO_PAGES(order)];//@note_good 先把多余的肉割出来，这句代码很赞!因为LosVmPage本身是在一个大数组上，pag
        LOS_ASSERT(buddyPage->order == VM_LIST_ORDER_MAX);//没挂到伙伴算法对应组块空闲链表上的物理页框的order必须是VM_LIST_ORDER_MAX
```

```
            OsVmPhysFreeListAddUnsafe(buddyPage， order);//将劈开的节点挂到对应序号的链表上，buddyPage->order = order
        }
    }
```

为了方便理解代码细节， 这里说一种情况: 比如三哥要买3斤的，发现4斤，8斤的都没有了，只有16斤的怎么办？ 注意不会给16斤，只会给4斤。这时需要把肉劈开，劈成 8，4，4，其中4斤给张三哥，将剩下的8斤，4斤挂到对应链表上。 OsVmPhysPagesSpiltUnsafe 干的就是劈猪肉的活。

伙伴算法的链表是怎么初始化的，再看段代码

```
//初始化空闲链表，分配物理页框使用伙伴算法
STATIC INLINE VOID OsVmPhysFreeListInit(struct VmPhysSeg *seg)
{
    int i;
    UINT32 intSave;
    struct VmFreeList *list = NULL;

    LOS_SpinInit(&seg->freeListLock);//初始化用于分配的自旋锁

    LOS_SpinLockSave(&seg->freeListLock， &intSave);
    for (i = 0; i < VM_LIST_ORDER_MAX; i++) {//遍历伙伴算法空闲块组链表
        list = &seg->freeList[i]; //一个个来
        LOS_ListInit(&list->node); //LosVmPage。node将挂到list->node上
        list->listCnt = 0;   //链表上的数量默认0
    }
    LOS_SpinUnlockRestore(&seg->freeListLock， intSave);
}
```

鸿蒙是面向未来设计的系统，高瞻远瞩，格局远大，设计精良， 海量知识点， 对内核源码加上中文注解已有三个多月，越深入精读内核源码，越能感受到设计者的精巧用心，创新突破， 向开发者致敬。 可以毫不夸张的说鸿蒙内核源码可作为大学C语言，数据结构，操作系统，汇编语言 四门课程的教学项目。如此宝库，不深入研究实在是暴殄天物，于心不忍。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

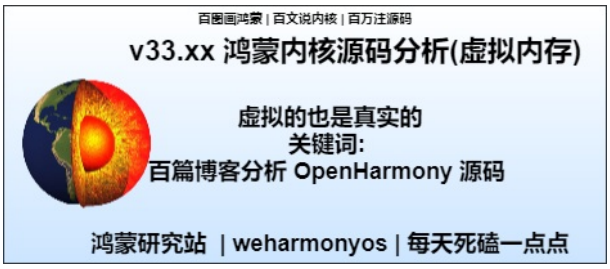
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

33_内存概念篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

RAM

随机存取存储器（Random Access Memory，RAM)

SRAM与DRMA

静态随机存取存储器（Static Random-Access Memory，SRAM） 动态随机存取存储器（Dynamic Random Access Memory，DRAM）

ROM

只读存储器(Read Only Memory，ROM)

FLASH闪存

```
/// 内存管理单元（英语：memory management unit，缩写为MMU），有时称作分页内存管理单元（英语：paged memory management unit，缩写为PMMU
typedef struct ArchMmu {//内存管理单元
#ifndef LOSCFG_PAGE_TABLE_FINE_LOCK
    SPIN_LOCK_S         lock;           /**< arch mmu page table entry modification spin lock */
#endif
    VADDR_T             *virtTtb;       /**< translation table base virtual addr | 注意:这里是个指针,内核操作都用这个地址*/
    PADDR_T             physTtb;        /**< translation table base phys addr | 注意:这里是个值,这个值是记录给MMU使用的,MMU只认它,内核是无法使用的*/
    UINT32              asid;           /**< TLB asid | 标识进程用的，由mmu初始化阶段申请分配，有了它在mmu层面才知道是哪个进程的虚拟地址*/
    LOS_DL_LIST         ptList;         /**< page table vm page list | L1 为表头，后面挂的是n多L2*/
} LosArchMmu;
```

百文说内核 | 抓住主脉络

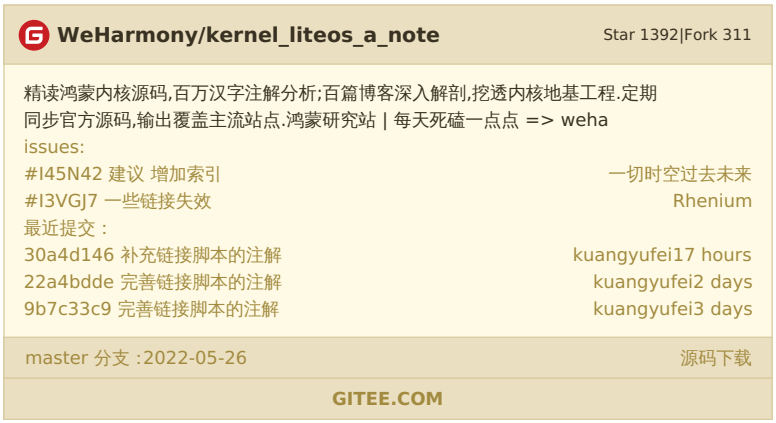
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

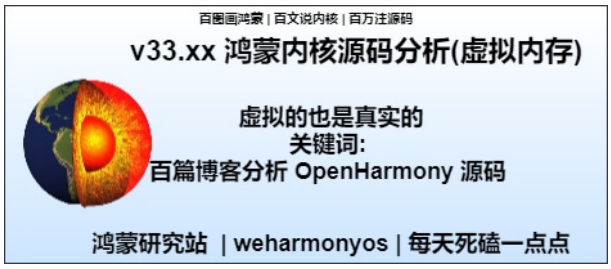


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

33_虚拟内存篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(虚拟内存) | 虚拟的也是真实的
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

```
/// 内存管理单元（英语：memory management unit，缩写为MMU），有时称作分页内存管理单元（英语：paged memory management unit，缩写为PMMU
typedef struct ArchMmu {//内存管理单元
#ifndef LOSCFG_PAGE_TABLE_FINE_LOCK
    SPIN_LOCK_S         lock;           /**< arch mmu page table entry modification spin lock */
#endif
    VADDR_T             *virtTtb;       /**< translation table base virtual addr | 注意:这里是个指针,内核操作都用这个地址*/
    PADDR_T             physTtb;        /**< translation table base phys addr | 注意:这里是个值,这个值是记录给MMU使用的,MMU只认它,内核是无法使用的*/
    UINT32              asid;           /**< TLB asid | 标识进程用的，由mmu初始化阶段申请分配，有了它在mmu层面才知道是哪个进程的虚拟地址*/
    LOS_DL_LIST         ptList;         /**< page table vm page list | L1 为表头，后面挂的是n多L2*/
} LosArchMmu;
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布， **鸿蒙研究站 | weharmonyos** 中回复 **百文** 可方便阅读。

按功能模块:

- 基础知识 >> 双向链表 | 内核概念 | 源码结构 | 地址空间 | 计时单位 | 优雅的宏 | 钩子框架 | 位图管理 | POSIX | main函数 |
- 进程管理 >> 调度故事 | 进程控制块 | 进程空间 | 线性区 | 红黑树 | 进程管理 | Fork进程 | 进程回收 | Shell编辑 | Shell解析 |
- 任务管理 >> 任务控制块 | 并发并行 | 就绪队列 | 调度机制 | 任务管理 | 用栈方式 | 软件定时器 | 控制台 | 远程登录 | 协议栈 |
- 内存管理 >> 内存规则 | 物理内存 | 虚拟内存 | 虚实映射 | 页表管理 | 静态分配 | TLFS算法 | 内存池管理 | 原子操作 | 圆整对齐 |
- 通讯机制 >> 通讯总览 | 自旋锁 | 互斥锁 | 快锁使用 | 快锁实现 | 读写锁 | 信号量 | 事件机制 | 信号生产 | 信号消费 | 消息队列 | 消息封装 | 消息映射 | 共享内存 |
- 文件系统 >> 文件概念 | 文件故事 | 索引节点 | VFS | 文件句柄 | 根文件系统 | 挂载机制 | 管道文件 | 文件映射 | 写时拷贝 |
- 硬件架构 >> 芯片模式 | ARM架构 | 指令集 | 协处理器 | 工作模式 | 寄存器 | 多核管理 | 中断概念 | 中断管理 |
- 内核汇编 >> 编码方式 | 汇编基础 | 汇编传参 | 链接脚本 | 开机启动 | 进程切换 | 任务切换 | 中断切换 | 异常接管 | 缺页中断 |
- 编译运行 >> 编译过程 | 编译构建 | GN语法 | 忍者无敌 | ELF格式 | ELF解析 | 静态链接 | 重定位 | 动态链接 | 进程映像 | 应用启动 | 系统调用 | VDSO |
- 调测工具 >> 模块监控 | 日志跟踪 | 系统安全 | 测试用例 |
- 前因后果 >> 总目录 | 源码注释 | 静态站点 | 参考手册 |

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。

- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码，**鸿蒙研究站 | weharmonyos** 中回复 **百万** 可方便阅读。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交 :
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

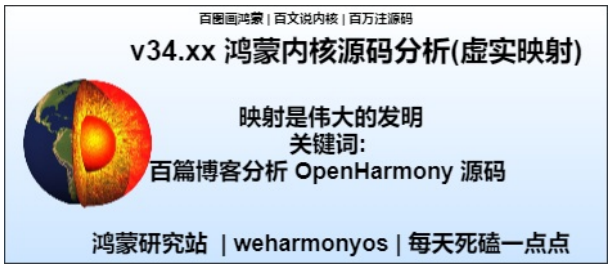
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢点赞分享的,后来都成了大神。:)

34_虚实映射篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

MMU的本质

虚拟地址(VA): 就是线性地址， 鸿蒙内存部分全是VA的身影， 是由编译器和链接器在定位程序时分配的，每个应用程序都使用相同的虚拟内存地址空间，而这些虚拟内存地址空间实际上分别映射到不同的实际物理内存空间上。CPU只知道虚拟地址，向虚拟地址要数据，但在其保护模式下很悲催地址信号在路上被MMU拦截了，MMU把虚拟地址换成了物理地址，从而拿到了真正的数据。

物理地址(PA)：程序的指令和常量数据，全局变量数据以及运行时动态申请内存所分配的实际物理内存存放位置。

MMU采用页表(page table)来实现虚实地址转换，页表项除了描述虚拟页到物理页直接的转换外，还提供了页的访问权限(读，写，可执行)和存储属性。 MMU的本质是拿虚拟地址的高位(20位)做文章，低12位是页内偏移地址不会变。也就是说虚拟地址和物理地址的低12位是一样的，本篇详细讲述MMU是如何变戏法的。

MMU是通过两级页表结构：L1和L2来实现映射功能的，鸿蒙内核当然也实现了这两级页表转换的实现。本篇是系列篇关于内存部分最满意的一篇，也是最不好理解的一篇， 强烈建议结合源码看。

一级页表L1

L1 页表将全部的 4G 地址空间划分为 4096 个 1 M的节，页表中每一项(页表项) 32 位，其内容是 L2页 表基地址或某个 1M 物理内存的基地址。虚拟地址的高 12 位用于对页表项定位，也就是 4096 个页面项的索引， L1 页表的基地址，也叫转换表基地址，存放在 CP15 的 C2（TTB） 寄存器中，鸿蒙内核源码分析(内存汇编篇)中有详细的描述，自行翻看。

L1页表项有三种描述格式，鸿蒙源码如下。

```
/* L1 descriptor type */
#define MMU_DESCRIPTOR_L1_TYPE_INVALID                  (0x0 << 0)
#define MMU_DESCRIPTOR_L1_TYPE_PAGE_TABLE               (0x1 << 0)
#define MMU_DESCRIPTOR_L1_TYPE_SECTION                  (0x2 << 0)
#define MMU_DESCRIPTOR_L1_TYPE_MASK                     (0x3 << 0)
```

第一种：Fault（INVALID)页表项，表示对应虚拟地址未被映射，访问将产生一个数据中止异常。

第二种：PAGE_TABLE页表项，指向L2页表的页表项，意思就是把1M分成更多的页（256*4K）

第三种：SECTION页表项 ，指向1M节的页表项的最低二位[1:0]，用于定义页表项的类型，section页表项对应1M的节，直接使用页表项的最高12位替代虚拟地址的高12位即可得到物理地址。还是直接看鸿蒙源码来的清晰，每一行都加了详细的注释。

LOS_ArchMmuQuery

```
//通过虚拟地址查询物理地址
STATUS_T LOS_ArchMmuQuery(const LosArchMmu *archMmu， VADDR_T vaddr， PADDR_T *paddr， UINT32 *flags)
{//archMmu->virtTtb:转换表基地址
    PTE_T l1Entry = OsGetPte1(archMmu->virtTtb， vaddr);//获取PTE vaddr右移20位 得到L1描述子地址
    PTE_T l2Entry;
    PTE_T* l2Base = NULL;

    if (OsIsPte1Invalid(l1Entry)) {//判断L1描述子地址是否有效
        return LOS_ERRNO_VM_NOT_FOUND;//无效返回虚拟地址未查询到
    } else if (OsIsPte1Section(l1Entry)) {// section页表项: l1Entry低二位是否为 10
        if (paddr != NULL) {//物理地址 = 节基地址(section页表项的高12位) + 虚拟地址低20位
            *paddr = MMU_DESCRIPTOR_L1_SECTION_ADDR(l1Entry) + (vaddr & (MMU_DESCRIPTOR_L1_SMALL_SIZE - 1));
        }

        if (flags != NULL) {
            OsCvtSecAttsToFlags(l1Entry， flags);//获取虚拟内存的flag信息
        }
    } else if (OsIsPte1PageTable(l1Entry)) {//PAGE_TABLE页表项: l1Entry低二位是否为 01
        l2Base = OsGetPte2BasePtr(l1Entry);//获取L2转换表基地址
        if (l2Base == NULL) {
            return LOS_ERRNO_VM_NOT_FOUND;
        }
        l2Entry = OsGetPte2(l2Base， vaddr);//获取L2描述子地址
        if (OsIsPte2SmallPage(l2Entry) || OsIsPte2SmallPageXN(l2Entry)) {
            if (paddr != NULL) {//物理地址 = 小页基地址(L2页表项的高20位) + 虚拟地址低12位
                *paddr = MMU_DESCRIPTOR_L2_SMALL_PAGE_ADDR(l2Entry) + (vaddr & (MMU_DESCRIPTOR_L2_SMALL_SIZE - 1));
            }

            if (flags != NULL) {
                OsCvtPte2AttsToFlags(l1Entry， l2Entry， flags);//获取虚拟内存的flag信息
            }
        } else if (OsIsPte2LargePage(l2Entry)) {//鸿蒙目前暂不支持64K大页，未来手机版应该会支持。
            LOS_Panic("%s %d， large page unimplemented\n"， __FUNCTION__， __LINE__);
        } else {
            return LOS_ERRNO_VM_NOT_FOUND;
        }
    }

    return LOS_OK;
}
```

这是鸿蒙内核对地址使用最频繁的功能，通过虚拟地址得到物理地址和flag信息，看下哪些地方会调用到它。

二级页表L2

L1页表项表示1M的地址范围，L2把1M分成更多的小页，鸿蒙内核 一页按4K算，所以被分成 256个小页。

L2页表中包含256个页表项，每个32位(4个字节)，L2页表需要 256*4 = 1K的空间，必须按1K对齐，每个L2页表项将4K的虚拟内存地址转换为物理地址，每个L2页面项都给出了一个4K的页基地址。

L2页表项有三种格式：

```
/* L2 descriptor type */
#define MMU_DESCRIPTOR_L2_TYPE_INVALID                  (0x0 << 0)
#define MMU_DESCRIPTOR_L2_TYPE_LARGE_PAGE               (0x1 << 0)
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE               (0x2 << 0)
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE_XN            (0x3 << 0)
#define MMU_DESCRIPTOR_L2_TYPE_MASK                     (0x3 << 0)
```

第一种：Fault（INVALID)页表项，表示对应虚拟地址未被映射，访问将产生一个数据中止异常。

第二种：大页表项，包含一个指向64K页的指针，但鸿蒙内核并没有实现大页表的支持，给出了未实现的提示

```
if (OsIsPte2LargePage(l2Entry)) {
        LOS_Panic("%s %d， large page unimplemented\n"， __FUNCTION__， __LINE__);
```

```
    }
```

第三种：小页表项，包含一个指向4K页的指针。

映射初始化的过程

先看调用和被调用的关系

```
//启动映射初始化
VOID OsInitMappingStartUp(VOID)
{
   OsArmInvalidateTlbBarrier();//使TLB失效

   OsSwitchTmpTTB();//切换到临时TTB

   OsSetKSectionAttr();//设置内核段(text，rodata，bss)映射

   OsArchMmuInitPerCPU();//初始化CPU与mmu相关信息
}
```

干脆利落，调用了四个函数，其中三个在鸿蒙内核源码分析(内存汇编篇)有涉及，不展开讲，这里说OsSetKSectionAttr

它实现了内核空间各个区的映射，内核本身也是程序，鸿蒙把内核空间在物理内存上就独立开来了，也就是说在物理内存上有一段区域是只给内核空间享用的，从根上就把内核和APP 空间隔离了，里面放的是内核的重要数据(包括代码，常量和全局变量)，具体看代码，代码很长，整个函数全贴出来了，都加上了注释。

OsSetKSectionAttr 内核空间的设置和映射

```
typedef struct ArchMmuInitMapping {
   PADDR_T phys;//物理地址
   VADDR_T virt;//虚拟地址
   size_t  size;//大小
   unsigned int flags;//标识 读/写/.. VM_MAP_REGION_FLAG_PERM_*
   const char *name;//名称
} LosArchMmuInitMapping;

VADDR_T *OsGFirstTableGet()
{
   return (VADDR_T *)g_firstPageTable;//UINT8 g_firstPageTable[MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS]
}
//设置内核空间段属性，可看出内核空间是固定映射到物理地址
STATIC VOID OsSetKSectionAttr(VOID)
{
   /* every section should be page aligned */
   UINTPTR textStart = (UINTPTR)&__text_start;//代码段开始位置
   UINTPTR textEnd = (UINTPTR)&__text_end;//代码段结束位置
   UINTPTR rodataStart = (UINTPTR)&__rodata_start;//常量只读段开始位置
   UINTPTR rodataEnd = (UINTPTR)&__rodata_end;//常量只读段结束位置
   UINTPTR ramDataStart = (UINTPTR)&__ram_data_start;//全局变量段开始位置
   UINTPTR bssEnd = (UINTPTR)&__bss_end;//bss结束位置
   UINT32 bssEndBoundary = ROUNDUP(bssEnd， MB);
   LosArchMmuInitMapping mmuKernelMappings[] = {
      {
         .phys = SYS_MEM_BASE + textStart - KERNEL_VMM_BASE，//映射物理内存位置
         .virt = textStart，//内核代码区
         .size = ROUNDUP(textEnd - textStart， MMU_DESCRIPTOR_L2_SMALL_SIZE)，//代码区大小
         .flags = VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_EXECUTE，//代码段可读，可执行
         .name = "kernel_text"
      }，
      {
         .phys = SYS_MEM_BASE + rodataStart - KERNEL_VMM_BASE，//映射物理内存位置
         .virt = rodataStart，//内核常量区
         .size = ROUNDUP(rodataEnd - rodataStart， MMU_DESCRIPTOR_L2_SMALL_SIZE)，//4K对齐
         .flags = VM_MAP_REGION_FLAG_PERM_READ，//常量段只读
         .name = "kernel_rodata"
      }，
      {
```

```
            .phys = SYS_MEM_BASE + ramDataStart - KERNEL_VMM_BASE，//映射物理内存位置
            .virt = ramDataStart，
            .size = ROUNDUP(bssEndBoundary - ramDataStart， MMU_DESCRIPTOR_L2_SMALL_SIZE)，
            .flags = VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_WRITE，//全局变量区可读可写
            .name = "kernel_data_bss"
        }
    };
    LosVmSpace *kSpace = LOS_GetKVmSpace();//获取内核空间
    status_t status;
    UINT32 length;
    paddr_t oldTtPhyBase;
    int i;
    LosArchMmuInitMapping *kernelMap = NULL;//内核映射
    UINT32 kmallocLength;

    /* use second-level mapping of default READ and WRITE */
    kSpace->archMmu.virtTtb = (PTE_T *)g_firstPageTable;//__attribute__((section(".bss.prebss.translation_table"))) UINT8 g_firstPageTable[MMU_D
    kSpace->archMmu.physTtb = LOS_PaddrQuery(kSpace->archMmu.virtTtb);//通过TTB虚拟地址查询TTB物理地址
    status = LOS_ArchMmuUnmap(&kSpace->archMmu， KERNEL_VMM_BASE，
                              (bssEndBoundary - KERNEL_VMM_BASE) >> MMU_DESCRIPTOR_L2_SMALL_SHIFT);//解绑 bssEndBoundary - KERNEL_VMM_BASE
    if (status != ((bssEndBoundary - KERNEL_VMM_BASE) >> MMU_DESCRIPTOR_L2_SMALL_SHIFT)) {//解绑失败
        VM_ERR("unmap failed， status: %d"， status);
        return;
    }
//映射 textStart - KERNEL_VMM_BASE 区
    status = LOS_ArchMmuMap(&kSpace->archMmu， KERNEL_VMM_BASE， SYS_MEM_BASE，
                            (textStart - KERNEL_VMM_BASE) >> MMU_DESCRIPTOR_L2_SMALL_SHIFT，
                            VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_WRITE |
                            VM_MAP_REGION_FLAG_PERM_EXECUTE);
    if (status != ((textStart - KERNEL_VMM_BASE) >> MMU_DESCRIPTOR_L2_SMALL_SHIFT)) {
        VM_ERR("mmap failed， status: %d"， status);
        return;
    }

    length = sizeof(mmuKernelMappings) / sizeof(LosArchMmuInitMapping);
    for (i = 0; i < length; i++) {//对mmuKernelMappings一一映射好
        kernelMap = &mmuKernelMappings[i];
        status = LOS_ArchMmuMap(&kSpace->archMmu， kernelMap->virt， kernelMap->phys，
                                kernelMap->size >> MMU_DESCRIPTOR_L2_SMALL_SHIFT， kernelMap->flags);
        if (status != (kernelMap->size >> MMU_DESCRIPTOR_L2_SMALL_SHIFT)) {
            VM_ERR("mmap failed， status: %d"， status);
            return;
        }
        LOS_VmSpaceReserve(kSpace， kernelMap->size， kernelMap->virt);//保留区
    }
//将剩余空间映射好
    kmallocLength = KERNEL_VMM_BASE + SYS_MEM_SIZE_DEFAULT - bssEndBoundary;
    status = LOS_ArchMmuMap(&kSpace->archMmu， bssEndBoundary，
                            SYS_MEM_BASE + bssEndBoundary - KERNEL_VMM_BASE，
                            kmallocLength >> MMU_DESCRIPTOR_L2_SMALL_SHIFT，
                            VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_WRITE);
    if (status != (kmallocLength >> MMU_DESCRIPTOR_L2_SMALL_SHIFT)) {
        VM_ERR("unmap failed， status: %d"， status);
        return;
    }
    LOS_VmSpaceReserve(kSpace， kmallocLength， bssEndBoundary);

    /* we need free tmp ttbase */
    oldTtPhyBase = OsArmReadTtbr0();//读取TTB值
    oldTtPhyBase = oldTtPhyBase & MMU_DESCRIPTOR_L2_SMALL_FRAME;
    OsArmWriteTtbr0(kSpace->archMmu.physTtb | MMU_TTBRx_FLAGS);//内核页表基地址写入CP15 c2(TTB寄存器)
    ISB;

    /* we changed page table entry， so we need to clean TLB here */
    OsCleanTLB();//清空TLB缓冲区

    (VOID)LOS_MemFree(m_aucSysMem0， (VOID *)(UINTPTR)(oldTtPhyBase - SYS_MEM_BASE + KERNEL_VMM_BASE));//释放内存池
}
```

LOS_ArchMmuMap

mmu的map 就是生成L1，L2页表项的过程，以供虚实地址的转换使用，还是直接看代码吧，代码说明一切！

```
//所谓的 map 就是 生成L1，L2页表项的过程
status_t LOS_ArchMmuMap(LosArchMmu *archMmu， VADDR_T vaddr， PADDR_T paddr， size_t count， UINT32 flags)
{
    PTE_T l1Entry;
    UINT32 saveCounts = 0;
    INT32 mapped = 0;
    INT32 checkRst;

    checkRst = OsMapParamCheck(flags， vaddr， paddr);//检查参数
    if (checkRst < 0) {
        return checkRst;
    }

    /* see what kind of mapping we can use */
    while (count > 0) {
        if (MMU_DESCRIPTOR_IS_L1_SIZE_ALIGNED(vaddr) && //虚拟地址和物理地址对齐 0x100000（1M）时采用
            MMU_DESCRIPTOR_IS_L1_SIZE_ALIGNED(paddr) && //section页表项格式
            count >= MMU_DESCRIPTOR_L2_NUMBERS_PER_L1) { //MMU_DESCRIPTOR_L2_NUMBERS_PER_L1 = 0x100
            /* compute the arch flags for L1 sections cache， r ，w ，x， domain and type */
            saveCounts = OsMapSection(archMmu， flags， &vaddr， &paddr， &count);//生成L1 section类型页表项并保存
        } else {
            /* have to use a L2 mapping， we only allocate 4KB for L1， support 0 ~ 1GB */
            l1Entry = OsGetPte1(archMmu->virtTtb， vaddr);//获取L1页面项
            if (OsIsPte1Invalid(l1Entry)) {//L1 fault页面项类型
                OsMapL1PTE(archMmu， &l1Entry， vaddr， flags);//生成L1 page table类型页表项并保存
                saveCounts = OsMapL2PageContinous(l1Entry， flags， &vaddr， &paddr， &count);//生成L2 页表项目并保存
            } else if (OsIsPte1PageTable(l1Entry)) {//L1 page table页面项类型
                saveCounts = OsMapL2PageContinous(l1Entry， flags， &vaddr， &paddr， &count);//生成L2 页表项目并保存
            } else {
                LOS_Panic("%s %d， unimplemented tt_entry %x\n"， __FUNCTION__， __LINE__， l1Entry);
            }
        }
        mapped += saveCounts;
    }

    return mapped;
}

STATIC UINT32 OsMapL2PageContinous(PTE_T pte1， UINT32 flags， VADDR_T *vaddr， PADDR_T *paddr， UINT32 *count)
{
    PTE_T *pte2BasePtr = NULL;
    UINT32 archFlags;
    UINT32 saveCounts;

    pte2BasePtr = OsGetPte2BasePtr(pte1);
    if (pte2BasePtr == NULL) {
        LOS_Panic("%s %d， pte1 %#x error\n"， __FUNCTION__， __LINE__， pte1);
    }

    /* compute the arch flags for L2 4K pages */
    archFlags = OsCvtPte2FlagsToAttrs(flags);
    saveCounts = OsSavePte2Continuous(pte2BasePtr， OsGetPte2Index(*vaddr)， *paddr | archFlags， *count);
    *paddr += (saveCounts << MMU_DESCRIPTOR_L2_SMALL_SHIFT);
    *vaddr += (saveCounts << MMU_DESCRIPTOR_L2_SMALL_SHIFT);
    *count -= saveCounts;
    return saveCounts;
}
```

OsMapL2PageContinous 没有加注释，希望你别太懒，赶紧动起来，到这里应该都能看懂了！最好能结合 鸿蒙内核源码分析(内存汇编篇)一起看理解会更深透。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很

重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。

- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

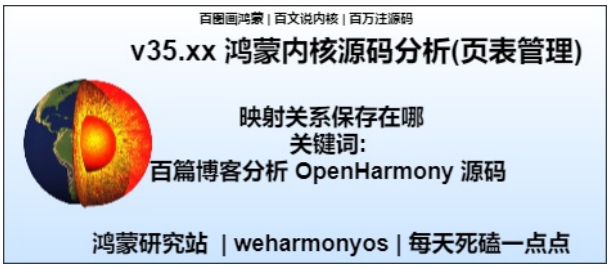
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

35_页表管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

什么是页表

在前几篇中会反复提到 **页表**，其作用是记录虚拟地址和物理地址的映射关系的。它也需要内存空间存放内容，便于查询，本篇说页表的使用流程和实现过程。

- 鸿蒙有三种虚拟空间(LosVmSpace)：
  - 内核空间(g_kVmSpace)：数量一个，内核也是程序，需要有容身之所，需要通过外部工具先烧录至 flash 指定位置，再启动加载至内存固定位置开始运行，存放内核数据代码的这部分空间称之为内核空间，其包括中断向量表(vectors)，代码区(.text)，只读数据区(.rodata)，可读写数据区(.data)，未初始化的全局变量(.bbs)和页表区，这部分内容可查看系列篇之 **(开机启动篇)**
  - 内核分配空间(g_vMallocSpace)： 数量一个，内核启动后，运行期间需要不断的申请和释放内存，这些内存从哪里来呢？由内核分配空间提供，注意此处是动态以内存池的方式分配没错，但不能简单的理解为堆空间，因为内核态栈空间(stack)，堆空间(heap)，映射区(map)都是由它提供，它们没有明显的地址边界，你无法只从地址判断运行逻辑，这部分详细内容可查看系列篇之 **(内存池管理)**
  - 用户空间 ：数量多个，是应用程序运行的空间，在这个空间中 **栈区**，**堆区**，**映射区** ，**代码区**，**数据区** 会有明显的边界，**栈区**在高地址位，向下生长，**堆区**在低地址位，向上生长，**数据区**在更低的地址位，**代码区**在更更低的地址位。详细看下图

页表是记录这三种虚拟空间地址映射关系的 ，而内核空间与内核分配空间的虚拟地址不会重叠所以可设计成共用一张页表，存储在内核空间中，而所有用户空间的虚拟地址范围是一致的，必须独立记录映射关系，统一存储在内核空间的页表区，当用户进程切换时便提供这份页表给 MMU ， MMU 将对这份页表增删改查。所谓的 **缺页中断** 就是在这份页表中没有查到虚拟地址映射的物理地址，处理缺页中断是将内容调入物理内存并更新页表的过程。再比如**打开某个文件** 背后需要做 **文件映射** ，为这个文件单独开辟一个线性区，将文件内容以页为单位加载到物理内存页帧中，由页表保存线性区地址和物理地址的映射关系。

内核页表 | g_firstPageTable

此处会涉及到编译器的知识， __attribute__ 这个关键词是 GNU 编译器中的编译属性， __attribute__((section("section_name"))) ，含义是将作用的函数或数据放入指定名为 section_name 对应的段中。再说的简单点是编译器帮我们在数据区的指定的位置开了一个全局变量名叫 section_name 代码见于 los_arch_mmu.c

```
#define MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS                0x4000U //16K
__attribute__((aligned(MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS))) \
    __attribute__((section(".bss.prebss.translation_table"))) UINT8 \
    g_firstPageTable[MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS];
#ifdef LOSCFG_KERNEL_SMP
__attribute__((aligned(MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS))) \
    __attribute__((section(".bss.prebss.translation_table"))) UINT8 \
    g_tempPageTable[MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS];
```

```
UINT8 *g_mmuJumpPageTable = g_tempPageTable;
#else
extern CHAR __mmu_ttlb_begin; /* defined in .ld script | 内核临时页表在系统使能mmu到使用虚拟地址运行这段期间使用,其虚拟地址保存在g_mmuJumpPag
UINT8 *g_mmuJumpPageTable = (UINT8 *)&__mmu_ttlb_begin; /* temp page table, this is only used when system power up | 临时页表,用于系统启动
#endif
```

- `UINT8` 分配了 `16K` 等于分配了 `4` 个物理页，一个物理页的单元大小是按 `8` 位算的 `4K` 。能存储 `2^10 = 1024` 个 `UINT32` 数据，虚拟地址的长度为 `UINT32` 。
- 为什么要这么做呢？切换进程就需要提供页表的位置，用户进程的页表是在内核创建用户进程的时候就提供好了，就已经知道了具体位置，那内核的页表呢？它同样也需要在内核运行之前就提供好具体位置，注意，此处说的是页面的位置，而非页表内容。我们写的普通代码使用的全局变量并不能设定其在数据区的具体地址，你是做不到让程序指定一个变量的地址必须是地址 `(0x2345)` 对不对？所以只能由编译器来指定内核页表的具体地址。
- 抛个问题，从代码中知道内核还有一个临时页表 `g_tempPageTable` ，为何要有临时内核页表呢？

用户页表

```
/// 创建用户进程空间
LosVmSpace *OsCreateUserVmSpace(VOID)
{
    BOOL retVal = FALSE;
    LosVmSpace *space = LOS_MemAlloc(m_aucSysMem0, sizeof(LosVmSpace));//在内核空间申请用户进程空间
    if (space == NULL) {
        return NULL;
    }
 //此处为何直接申请物理页帧存放用户进程的页表,大概是因为所有页表都被存放在内核空间(g_kVmSpace)而非内核分配空间(g_vMallocSpace)
    VADDR_T *ttb = LOS_PhysPagesAllocContiguous(1);//分配一个物理页用于存放虚实映射关系表, 即:L1表
    if (ttb == NULL) {//若连页表都没有,剩下的也别玩了.
        (VOID)LOS_MemFree(m_aucSysMem0, space);
        return NULL;
    }
    (VOID)memset_s(ttb, PAGE_SIZE, 0, PAGE_SIZE);//4K空间置0
    retVal = OsUserVmSpaceInit(space, ttb);//初始化用户空间,mmu
    LosVmPage *vmPage = OsVmVaddrToPage(ttb);//找到所在物理页框
    if ((retVal == FALSE) || (vmPage == NULL)) {
        (VOID)LOS_MemFree(m_aucSysMem0, space);
        LOS_PhysPagesFreeContiguous(ttb, 1);
        return NULL;
    }
    LOS_ListAdd(&space->archMmu.ptList, &(vmPage->node));//页表链表,先挂上L1,后续还会挂上 N个L2表
    return space;
}
```

解读

- 用户空间的页表由内核空间提供，因为页表大小和物理页框对应，默认都是 `4K` ，所以直接申请物理页，页表的作用是存储虚拟地址和物理地址映射关系的，但它自身也是需要映射的，又该如何记录这种关系呢？鸿蒙使用了一个很巧妙的办法 **偏移法** 。 `KERNEL_ASPACE_BASE` 为内核空间的起始地址， `SYS_MEM_BASE` 为物理内存的起始地址

```
    #define KERNEL_VADDR_BASE       0x40000000
    #define KERNEL_VMM_BASE         U32_C(KERNEL_VADDR_BASE) ///< 速度快,使用cache
    #define KERNEL_ASPACE_BASE      KERNEL_VMM_BASE ///< 内核空间基地址
    #define SYS_MEM_BASE            DDR_MEM_ADDR ///< 物理内存基地址
    ///分配连续的物理页
    VOID *LOS_PhysPagesAllocContiguous(size_t nPages)
    {
        LosVmPage *page = NULL;
        if (nPages == 0) {
            return NULL;
        }
        //鸿蒙 nPages 不能大于 2^8 次方,即256个页,1M内存,仅限于内核态,用户态不限制分配大小.
        page = OsVmPhysPagesGet(nPages);//通过伙伴算法获取物理上连续的页
        if (page == NULL) {
            return NULL;
        }
        return OsVmPageToVaddr(page);//通过物理页找虚拟地址
    }
```

```
VOID *OsVmPageToVaddr(LosVmPage *page)//
{
    VADDR_T vaddr;
    vaddr = KERNEL_ASPACE_BASE + page->physAddr - SYS_MEM_BASE;//page->physAddr - SYS_MEM_BASE 得到物理地址的偏移量
    //因在整个虚拟内存中内核空间和用户空间是通过地址隔离的，如此很巧妙的就把该物理页映射到了内核空间
    //内核空间的vmPage是不会被置换的，因为是常驻内存，内核空间初始化mmu时就映射好了L1表
    return (VOID *)(UINTPTR)vaddr;
}
```

vaddr = KERNEL_ASPACE_BASE + page->physAddr - SYS_MEM_BASE; 表示申请的物理地址在物理空间的偏移量等于映射的虚拟地址在内核空间的偏移量，不需要存储映射关系，这简直就是神来之笔，拍案叫绝。但也由此可知 每个进程的页表(L1，L2)在逻辑地址层面不在一起，因为物理地址是不可能在一起的。

MMU页表

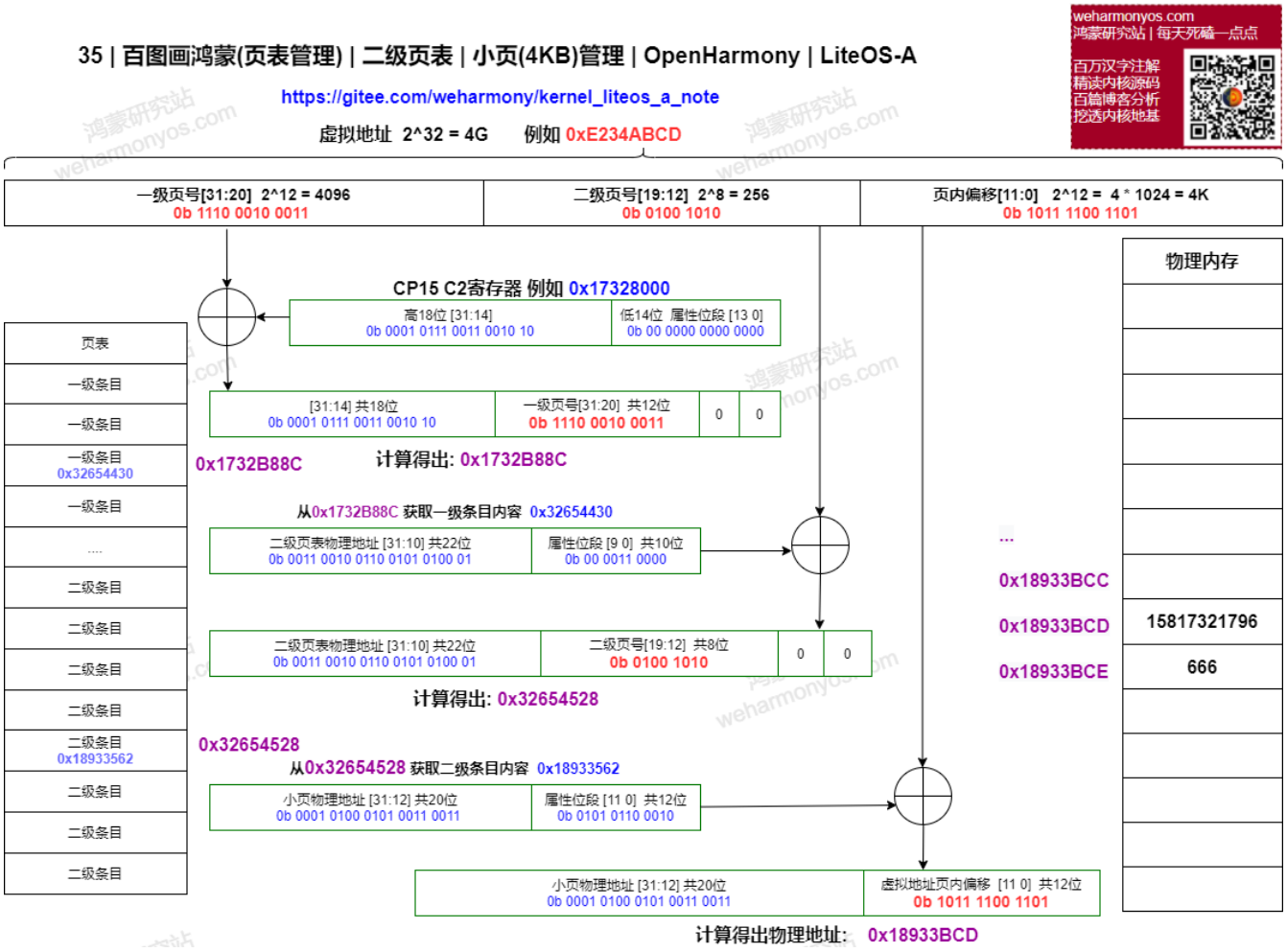
MMU 地址映射是连续的物理地址映射到连续的虚拟地址，切成一定的块大小映射，鸿蒙内核 MMU 一级条目分成 **段(1MB)** ， **页** 两种：

```
#define MMU_DESCRIPTOR_L1_TYPE_PAGE_TABLE               (0x1 << 0) ///< 一级条目类型按页分
#define MMU_DESCRIPTOR_L1_TYPE_SECTION                  (0x2 << 0) ///< 一级条目类型按段分
```

二级条目分成 **大页(64KB)** ， **小页(4KB)** ， **极小页(1KB)** 三种：

```
#define MMU_DESCRIPTOR_L2_TYPE_LARGE_PAGE               (0x1 << 0) ///< 二级条目类型按大页分
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE               (0x2 << 0) ///< 二级条目类型按小页分
#define MMU_DESCRIPTOR_L2_TYPE_SMALL_PAGE_XN              (0x3 << 0) ///< 二级条目类型按极小页分
```

下图绘制了 **小页(4KB)** 获取物理地址内容的全过程 ，将步骤和数据放在一块理解



过程解读

- 第一步：`CPU` 提供虚拟地址 **0xE234ABCD**
- 第二步：计算L1地址，从 `CP15` 的 `C2` 寄存器取出高 `18` 位用于高位，将虚拟地址高 `12` 用于中位，低二位补 `0` ， 得出L1地址：**0x1732B88C**
- 第三步：从L1地址：**0x1732B88C** 中取出内容L1内容：**0x34564430** 用于计算L2地址
- 第四步：计算L2地址，从 **0x34564430** 取出高 `22` 位用于高位，将虚拟地址中 `8` 用于中位，低二位补 `0` ， 得出L2地址：**0x32654528**
- 第五步：从L2地址：**0x32654528** 中取出内容 L2内容：**0x18933562** 用于计算物理地址
- 第六步：计算物理地址，从 **0x32654528** 取出高 `20` 位用于高位，将虚拟地址低 `12` 用于低位， 得出最后的物理地址：**0x18933BCD**
- 第七步：从物理地址 **0x18933BCD** 获取数据内容 **15817321796** 即虚拟地址**0xE234ABCD**获取的最终数据 关于 `CP15` 协处理部分请翻看系列篇的 **(协处理器篇)**

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

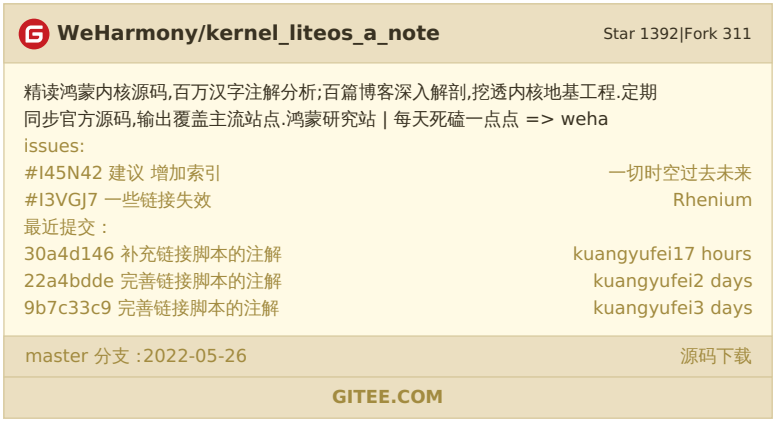
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

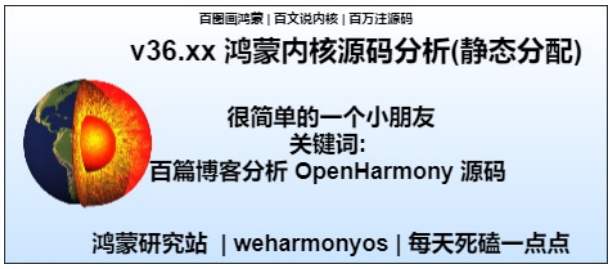


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

36_静态分配篇

本篇关键词：**池头、池体、节头、节块**



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

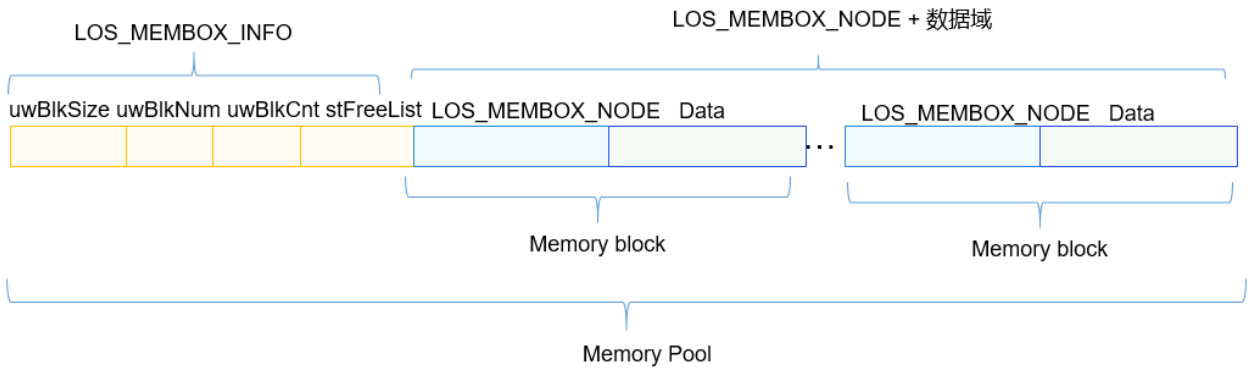
- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

静态分配

相比动态分配，静态内存池的分配就是个小弟弟，非常的简单，**两个结构体** + **一张图** 就能说明白。

```
typedef struct {//静态内存池信息结构体
    UINT32 uwBlkSize;           /**< Block size | 块大小*/
    UINT32 uwBlkNum;            /**< Block number | 块数量*/
    UINT32 uwBlkCnt;            /**< The number of allocated blocks | 已经被分配的块数量*/
    LOS_MEMBOX_NODE stFreeList; /**< Free list | 空闲链表*/
} LOS_MEMBOX_INFO;

typedef struct tagMEMBOX_NODE { //内存池中空闲节点的结构,是个单向的链表
    struct tagMEMBOX_NODE *pstNext; /**< Free node's pointer to the next node in a memory pool | 指向内存池中下一个空闲节点的指针*/
} LOS_MEMBOX_NODE;
```

下图来源于官网



解读

- 静态内存池在概念上由 **池头** 和 **池体** 两部分组成，池体由众多**节块**组成，节块由 **节头**和 **节体** 两部分组成
- 在数据结构上表现为 `LOS_MEMBOX_INFO` (池头) + [`LOS_MEMBOX_NODE` (节头) + `data` (节体)] + ... + [`LOS_MEMBOX_NODE` (节头) + `data` (节体)] ，在虚拟地址上它们是连在一起的。
- **池头** 记录总信息，包括 节块大小，总节块数量，已分配节块数量，空闲节块链表表头， `stFreeList` 将所有空闲节块链接到一起，分配内存根本不需要遍历， `stFreeList` 指向的下一个不为 `null` 代表还有空闲节块。
- **节头**只有一个指向下一个空闲链表的`pstNext`指针，简单但足以。
- 静态分配的优缺点是很明显的，总结下：
  - 负责管理的结构体简单，会占用很少的空间，这点优于动态分配。
  - 分配速度最快，一步到位。
  - 缺点是浪费严重，僵硬不灵活，很计划经济，给每一个家庭每月口粮就这么多，高矮胖瘦都不会管。

因代码量不大，但很精彩，看这种代码是种享受，本篇详细列出静态内存代码层面的实现，关键处已添加注释。

初始化

```
///初始化一个静态内存池，根据入参设定其起始地址、总大小及每个内存块大小
LITE_OS_SEC_TEXT_INIT UINT32 LOS_MemboxInit(VOID *pool, UINT32 poolSize, UINT32 blkSize)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;//在内存起始处放置控制头
    LOS_MEMBOX_NODE *node = NULL;
    //...
    UINT32 index;
    UINT32 intSave;
    MEMBOX_LOCK(intSave);
    boxInfo->uwBlkSize = LOS_MEMBOX_ALIGNED(blkSize + OS_MEMBOX_NODE_HEAD_SIZE); //节块总大小(节头+节体)
    boxInfo->uwBlkNum = (poolSize - sizeof(LOS_MEMBOX_INFO)) / boxInfo->uwBlkSize;//总节块数量
    boxInfo->uwBlkCnt = 0; //已分配的数量
    if (boxInfo->uwBlkNum == 0) {///只有0块的情况
        MEMBOX_UNLOCK(intSave);
        return LOS_NOK;
    }
    node = (LOS_MEMBOX_NODE *)(boxInfo + 1);//去除池头,找到第一个节块位置
    boxInfo->stFreeList.pstNext = node;//池头空闲链表指向第一个节块
    for (index = 0; index < boxInfo->uwBlkNum - 1; ++index) {//切割节块,挂入空闲链表
        node->pstNext = OS_MEMBOX_NEXT(node, boxInfo->uwBlkSize);//按块大小切割好,统一由pstNext指向
        node = node->pstNext;//node存储了下一个节点的地址信息
    }
    node->pstNext = NULL;//最后一个为null
    MEMBOX_UNLOCK(intSave);
    return LOS_OK;
}
```

申请

```
///从指定的静态内存池中申请一块静态内存块,整个内核源码只有 OsSwtmrScan中用到了静态内存.
LITE_OS_SEC_TEXT VOID *LOS_MemboxAlloc(VOID *pool)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;
    LOS_MEMBOX_NODE *node = NULL;
    LOS_MEMBOX_NODE *nodeTmp = NULL;
    UINT32 intSave;
    if (pool == NULL) {
        return NULL;
    }
    MEMBOX_LOCK(intSave);
    node = &(boxInfo->stFreeList);//拿到空闲单链表
    if (node->pstNext != NULL) {//不需要遍历链表,因为这是空闲链表
        nodeTmp = node->pstNext;//先记录要使用的节点
        node->pstNext = nodeTmp->pstNext;//不再空闲了,把节点摘出去了.
        OS_MEMBOX_SET_MAGIC(nodeTmp);//为已使用的节块设置魔法数字
        boxInfo->uwBlkCnt++;//已使用块数增加
    }
    MEMBOX_UNLOCK(intSave);
    return (nodeTmp == NULL) ? NULL : OS_MEMBOX_USER_ADDR(nodeTmp);//返回可用的虚拟地址
}
```

释放

```
/// 释放指定的一块静态内存块
LITE_OS_SEC_TEXT UINT32 LOS_MemboxFree(VOID *pool, VOID *box)
{
    LOS_MEMBOX_INFO *boxInfo = (LOS_MEMBOX_INFO *)pool;
    UINT32 ret = LOS_NOK;
    UINT32 intSave;
    if ((pool == NULL) || (box == NULL)) {
        return LOS_NOK;
    }
    MEMBOX_LOCK(intSave);
    do {
        LOS_MEMBOX_NODE *node = OS_MEMBOX_NODE_ADDR(box);//通过节体获取节块首地址
        if (OsCheckBoxMem(boxInfo, node) != LOS_OK) {
            break;
        }
        node->pstNext = boxInfo->stFreeList.pstNext;//节块指向空闲链表表头
        boxInfo->stFreeList.pstNext = node;//空闲链表表头反指向它,意味节块排到第一,下次申请将首个分配它
        boxInfo->uwBlkCnt--;//已经使用的内存块减一
        ret = LOS_OK;
    } while (0);//将被编译时优化
    MEMBOX_UNLOCK(intSave);
    return ret;
}
```

使用

鸿蒙内核目前只有软时钟处理使用了静态内存池，直接上代码

```
///软时钟初始化 ,注意函数在多CPU情况下会执行多次
STATIC UINT32 SwtmrBaseInit(VOID)
{
    UINT32 ret;
    UINT32 size = sizeof(SWTMR_CTRL_S) * LOSCFG_BASE_CORE_SWTMR_LIMIT;
    SWTMR_CTRL_S *swtmr = (SWTMR_CTRL_S *)LOS_MemAlloc(m_aucSysMem0, size); /* system resident resource */
    if (swtmr == NULL) {
        return LOS_ERRNO_SWTMR_NO_MEMORY;
    }
    (VOID)memset_s(swtmr, size, 0, size);//清0
    g_swtmrCBArray = swtmr;//软时钟
    LOS_ListInit(&g_swtmrFreeList);//初始化空闲链表
    for (UINT16 index = 0; index < LOSCFG_BASE_CORE_SWTMR_LIMIT; index++, swtmr++) {
            swtmr->usTimerID = index;//按顺序赋值
            LOS_ListTailInsert(&g_swtmrFreeList, &swtmr->stSortList.sortLinkNode);//通过sortLinkNode将节点挂到空闲链表
    }
 //想要用静态内存池管理,就必须要使用LOS_MEMBOX_SIZE来计算申请的内存大小,因为需要点前缀内存承载头部信息.
    size = LOS_MEMBOX_SIZE(sizeof(SwtmrHandlerItem), OS_SWTMR_HANDLE_QUEUE_SIZE);//规划一片内存区域作为软时钟处理函数的静态内存池。
    g_swtmrHandlerPool = (UINT8 *)LOS_MemAlloc(m_aucSysMem1, size); /* system resident resource */
    if (g_swtmrHandlerPool == NULL) {
        return LOS_ERRNO_SWTMR_NO_MEMORY;
    }
    ret = LOS_MemboxInit(g_swtmrHandlerPool, size, sizeof(SwtmrHandlerItem));
    if (ret != LOS_OK) {
        return LOS_ERRNO_SWTMR_HANDLER_POOL_NO_MEM;
    }
    for (UINT16 index = 0; index < LOSCFG_KERNEL_CORE_NUM; index++) {
        SwtmrRunQue *srq = &g_swtmrRunQue[index];
        /* The linked list of all cores must be initialized at core 0 startup for load balancing */
        OsSortLinkInit(&srq->swtmrSortLink);
        LOS_ListInit(&srq->swtmrHandlerQueue);
        srq->swtmrTask = NULL;
    }
    SwtmrDebugDataInit();
    return LOS_OK;
}
```

```
typedef VOID (*SWTMR_PROC_FUNC)(UINTPTR arg); //函数指针, 赋值给 SWTMR_CTRL_S->pfnHandler,回调处理
typedef struct {//处理软件定时器超时的回调函数的结构体
    SWTMR_PROC_FUNC handler;    /**< Callback function that handles software timer timeout  */ //处理软件定时器超时的回调函数
    UINTPTR arg;                /**< Parameter passed in when the callback function
                                 that handles software timer timeout is called */ //调用处理软件计时器超时的回调函数时传入的参数
    LOS_DL_LIST node;
#ifdef LOSCFG_SWTMR_DEBUG
    UINT32 swtmrID;
#endif
} SwtmrHandlerItem;
```

关于软定时器可以查看系列相关篇，请想想为何软件定时器会使用静态内存。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

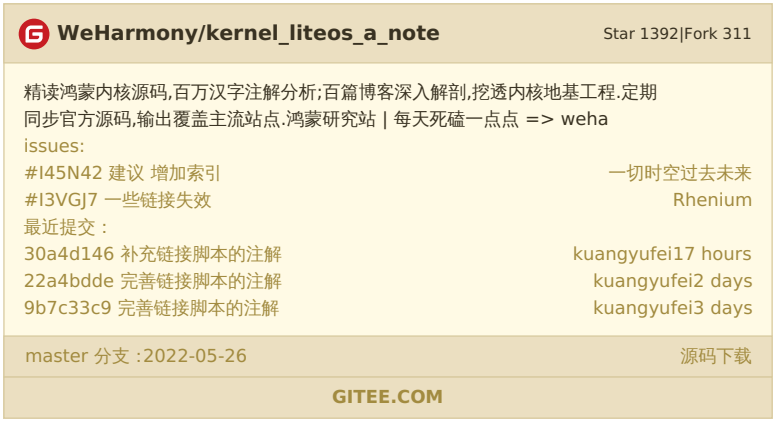
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

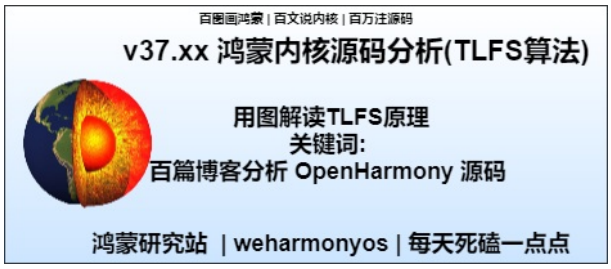


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

37_TLFS算法篇

本篇关键词：TLFS 、内存池 、malloc、free



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

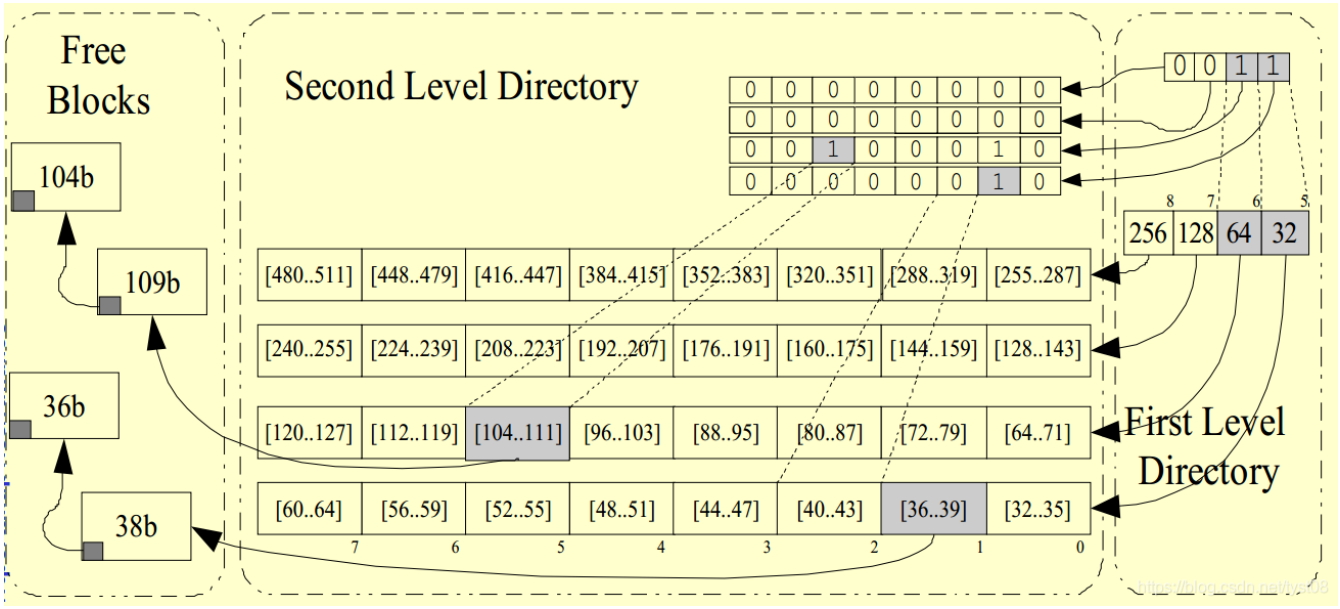
动态分配

本篇开始说一个耳朵听起老茧的概念 **动态分配**，将分成上下两篇，本篇为上篇，看完能快速理解下篇鸿蒙内核源码对动态内存的具体实现。

- **鸿蒙内核源码分析(TLFS算法)** 结合图表从理论视角说清楚 TLFS 算法
- **鸿蒙内核源码分析(内存池管理)** 结合源码说清楚鸿蒙内核动态内存池实现过程，个人认为这部分代码很精彩，简洁高效，尤其对空闲节点和已使用节点的实现令人称奇。

TLFS 原理

TLSF(Two-Level Segregate Fit) 是一种用于实时操作系统的内存分配算法，用两级结构对空闲块按大小进行划分，采用两级链表/索引的方式来加快查找。详细可查看 >> TLSF 论文 。



把上图看懂基本能明白 TLFS 的原理，请尝试自己先理解一遍再看本篇。

解读

- 为方便理解 ，将上图做成(**表一**)，中间过程请查看图表变化

步骤	操作	一级位图 (FL_bitmap)	二级位图 (SL_bitmaps[])	空闲链表(大小-虚拟地址)
第一步	**初始阶段**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 38b(0x3457) --> 36b(0xed31)

- 右边为第一级链表 `First List` (简称 `fl`)。将空闲内存块的大小根据 `2` 的幂进行分类，如（32、64、128、...），这跟伙伴算法很类似，伙伴算法是物理内存的分配算法，这样做的好处是防止外部碎片化，是否空闲用位图标识 **FL_bitmap | 一维数组**， `0011` 代表 `[32-64]`、`[64-128]` 这个区间有内存可以申请，例如： **malloc(37)** 时，查到在区间 `[32-64]` 中，为 `1` 代表本次可能可以申请到内存，但具体行不行得进入第二级查看。

- 中间为第二级链表 `Second List` (简称 `sl`)。第二层链表在第一层的基础上，按照一定的间隔，线性分段，图中将其分成 `8` 等份，对于 `[32-64]` 来说 `1/8` 为 `4` ，对于 `[64 - 128]` 来说 `1/8` 为 `8` ，可以确定的是等份也是 `2` 的倍数，同样是否空闲用位图标识**SL_bitmaps[] | 二维数组** ，每个bit代表是否空闲，图中代表 `[36 - 39]` 有内存可供分配，再查看其空闲链表发现真正可供分配的空间有两块， `38` 和 `36` ，自然将 `38` 给 **malloc(37)** 返回，此时空闲链表中还剩 `36` 节点 ，所以 一二级位图数据不会有任何的变化。

- 左边为空闲链表块，上面挂 `fl` ， `sl` 都为 `1` 时的空闲内存块，块大小为区间范围值，图中有两个空闲块 `38b --> 36b` ， `109b --> 104b` ，在实际运行过程往往出现同样大小的内存块例如 `38b --> 36b--> 36b`

申请过程

用二次申请说明详细过程

- **malloc(37)** ，发现值在区间 `[32-64]` 并对应 `fl` 的位图为 `1` ，说明 `sl` 中肯定会有一个 `1` ，但并不能保证能申请到。得细看第二步 `sl` 发现区间 `[36 - 39]` 的位图为 `1` ，说明空闲链表中肯定会有一块内存， ，但也不能保证大小适合 `37` 。再看最后一步，发现有两块内存 `38b --> 36b` ，只有 `38b` 符合，于是 **malloc(37)** 的结果是获得了一块大小 `38b` 的内存块，相差的一个 `1b` 称为内部碎片，这种碎片无法避免。将(**表一**)更新为(**表二**)

步骤	操作	一级位图 (FL_bitmap)	二级位图 (SL_bitmaps[])	空闲链表(大小-虚拟地址)
第一步	**初始阶段**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 38b(0x3457) --> 36b(0xed31)
第二步	**malloc(37)** **返回地址0x3457**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 36b(0xed31)

- **malloc(50)**，同样落在 `[32-64]` 查看 `fl` 的位图为 `1` ，查看第二步 `sl` 只有 `[36 - 39]` 的位图为 `1` ，而 **50 > 39**，不能满足所以没必要看第三步，得返回第一步往上走发现 `[64-128]` 的 `fl` 的位图为 `1` ，**50 < 64** 说明 **malloc(50)** 这次肯定能申请到内存了，查看 `[64-128]` 对应的 `sl` 发现 `[104-111]` 的位图为 `1` ，到第三步发现有 `109b --> 104b` 两块，选择其中更小 `104b` 的块切割成 `50` ， `54` 两小块，此时要对 `54` 处理挂入空闲链表， `54` 处于 `fl = [32-64]` ， `sl = [52-55]` 区间，地址为: **0x6838+50=0x686A** 。所以将 `[52-55]` 区间位图置为 `1` ，并挂入空闲链表。将(**表二**)更新为(**表三**)

步骤	操作	一级位图 (FL_bitmap)	二级位图 (SL_bitmaps[])	空闲链表(大小-虚拟地址)
第一步	**初始阶段**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 38b(0x3457) --> 36b(0xed31)
第二步	**malloc(37)** **返回地址0x3457**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 36b(0xed31)
第三步	**malloc(50)** **返回地址0x6838**	0011	00000000 00000000 00100000 00100010	109b(0x5625) 54b(0x686A) 36b(0xed31)

注意此时 `[32-64]` 的二级位图变成了 00100010 有两个 `1`

释放过程

同样用二次释放说明详细过程

- **free(0x3457)** 从地址可知正是上面的 **malloc(37)的内存，与分配切割相对应的是释放有合并的步骤，但malloc(37)虽然申请是 `37` ，但其实内核记录的内存块大小是 `38` ，所以会找寻地址为 0x3457 + 38 = 0x348F 的地址是否也处于空闲，如果是则合并，由表三**可知 并没有 **0x348F**的空闲块将，而 `38` 位于 `fl = [32-64]` ， `sl = [36-39]` 区间，所以挂回该空闲链表等待后续再分配使用，由此(**表三**)更新为(**表四**)

步骤	操作	一级位图 (FL_bitmap)	二级位图 (SL_bitmaps[])	空闲链表(大小-虚拟地址)
第一步	**初始阶段**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 38b(0x3457) --> 36b(0xed31)
第二步	**malloc(37) 返回地址0x3457**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 36b(0xed31)
第三步	**malloc(50) 返回地址0x6838**	0011	00000000 00000000 00100000 00100010	109b(0x5625) 54b(0x686A) 36b(0xed31)
第四步	**free(0x3457)**	0011	00000000 00000000 00100000 00100010	109b(0x5625) 54b(0x686A) 38b(0x3457) --> 36b(0xed31)

- **free(0x5610)** 这里假设内核记录该内存块大小为 **0x15**，归还的同时会找寻**0x5610 + 0x15 = 0x5625** 是否有空闲块，发现 `sl = [104-111]` 有一块 `109b` 空闲块，两块合成一块大小为 **109 + 0x15 = 109 + 21 = 130** ，位于 `fl = [128-255]` ， `sl = [128-143]` 区间，由此(**表四**)再更新为(**表五**)

步骤	操作	一级位图 (FL_bitmap)	二级位图 (SL_bitmaps[])	空闲链表(大小-虚拟地址)
第一步	**初始阶段**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 38b(0x3457) --> 36b(0xed31)
第二步	**malloc(37) 返回地址0x3457**	0011	00000000 00000000 00100000 00000010	109b(0x5625) --> 104b(0x6838) 36b(0xed31)
第三步	**malloc(50) 返回地址0x6838**	0011	00000000 00000000 00100000 00100010	109b(0x5625) 54b(0x686A) 36b(0xed31)
第四步	**free(0x3457)**	0011	00000000 00000000 00100000 00100010	109b(0x5625) 54b(0x686A) 38b(0x3457) --> 36b(0xed31)
第五步	**free(0x5610)**	0101	00000000 00000001 00000000 00100010	130b(0x5610) 54b(0x686A) 38b(0x3457) --> 36b(0xed31)

总结

TLSF(Two-Level Segregate Fit) 有两大优点：

- 实时性，执行速度快，只需查询位图就能知道结果，最多查询两次一级位图，时间复杂度为O(1)。
- 碎片少，浪费少，利用率高，因采用2次幂的方式，切割和合并非常的方便，很少出现外部碎片。

真正的鸿蒙内存动态分配实现过程比这些要复杂些，但有了本文算法基础做铺垫看源码实现会容易很多。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

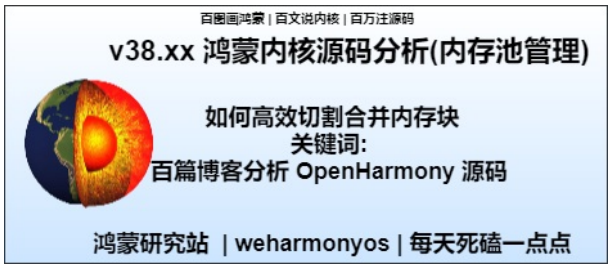
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

38_内存池管理

本篇关键词：内存池、哨兵节点、动态扩展、吃水线



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

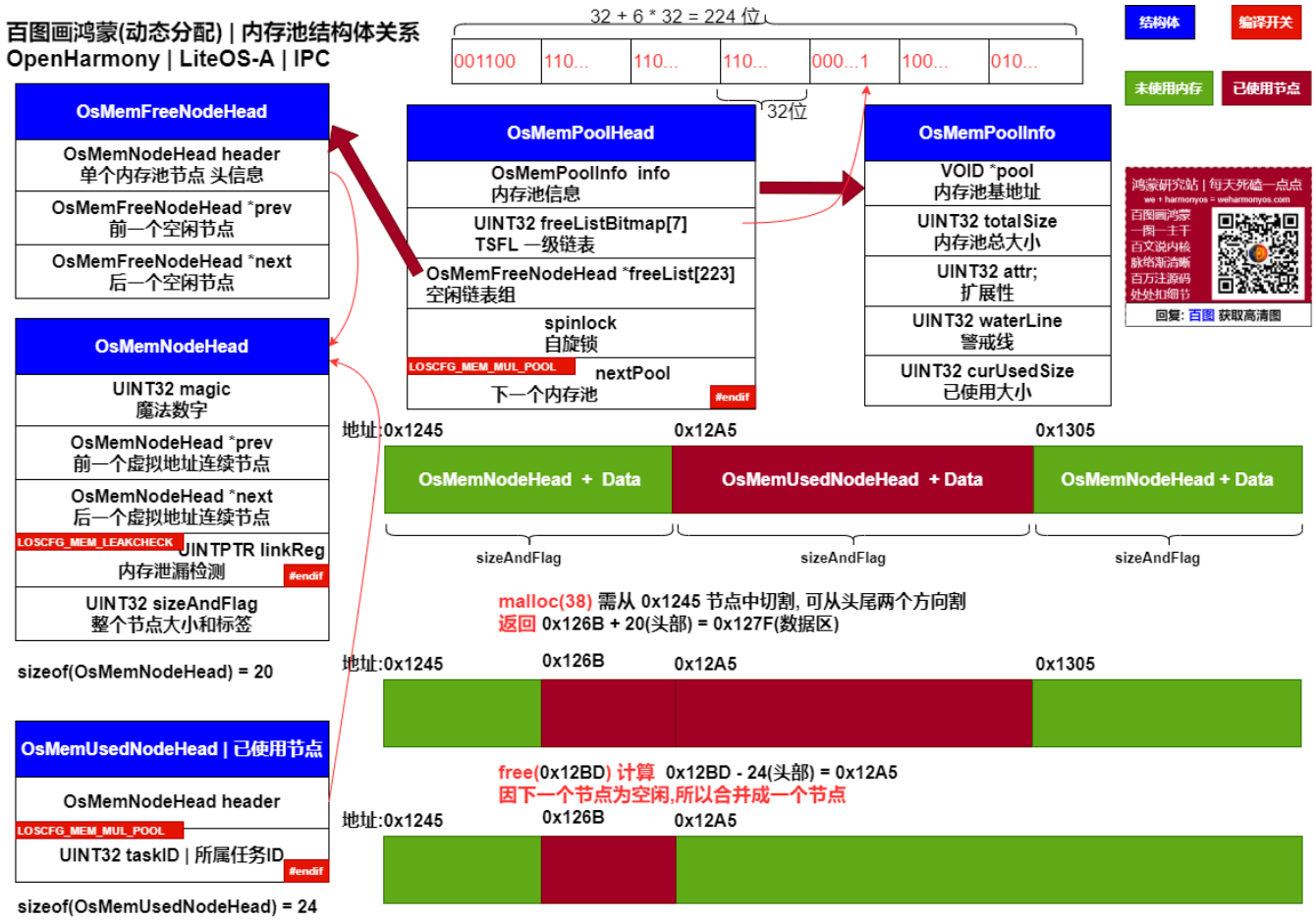
- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

动态分配

系列篇将动态分配分成上下两篇，本篇为下篇，阅读之前建议翻看上篇。

- **鸿蒙内核源码分析(TLFS算法)** 结合图表从理论视角说清楚 TLFS 算法
- **鸿蒙内核源码分析(内存池管理)** 结合源码说清楚鸿蒙内核动态内存池实现过程，个人认为这部分代码很精彩，简洁高效，尤其对空闲节点和已使用节点的实现令人称奇。

为了便于理解源码，站长画了以下图，图中列出主要结构体，位图，分配和释放信息，逐一说明。



- 请将内存池想成一条画好了网格虚线的大白纸，会有两种角色往白纸上画东西，一个是内核画管理数据，一个外部程序画业务数据，内核先画，外部程序想画需申请大小，申请成功内核会提供个地址给外部使用，例如申请 20 个格子，成功后内核返回一个 (5，8) 坐标，表示从第五行第八列开始往后的连续 20 个格子你可以使用。用完了释放只需要告诉内核一个坐标 (5,8) 而不需要大小，内核就知道回收多少格子。但内核凭什么知道要释放多少个格子呢？一定有个格子给记录下来了对不对，实际中存大小的格子坐标就是 (5，7) 。其值是在申请的时候或更早的时候填进去的。而且不一定是 20 ，但一定不小于 20 。如果您能完全理解以上这段话，那可能已经理解了内存池的管理的方式，不用往下看了。

内存池 | OsMemPoolHead

```
/// 内存池头信息
struct OsMemPoolHead {
    struct OsMemPoolInfo info; ///< 记录内存池的信息
    UINT32 freeListBitmap[OS_MEM_BITMAP_WORDS]; ///< 空闲位图 int[7] = 32 * 7 = 224
    struct OsMemFreeNodeHead *freeList[OS_MEM_FREE_LIST_COUNT];///< 空闲节点链表 32 + 24 * 8 = 224
    SPIN_LOCK_S spinlock; ///< 操作本池的自旋锁,涉及CPU多核竞争,所以必须得是自旋锁
#ifdef LOSCFG_MEM_MUL_POOL
    VOID *nextPool; ///< 指向下一个内存池 OsMemPoolHead 类型
#endif
};
/// 内存池信息
struct OsMemPoolInfo {
    VOID *pool;    ///< 指向内存块基地址,仅做记录而已,真正的分配内存跟它没啥关系
    UINT32 totalSize; ///< 总大小,确定了内存池的边界
    UINT32 attr;  ///< 属性 default attr: lock, not expand.
#ifdef LOSCFG_MEM_WATERLINE
    UINT32 waterLine;   /* Maximum usage size in a memory pool | 内存吃水线*/
    UINT32 curUsedSize; /* Current usage size in a memory pool | 当前已使用大小*/
#endif
};
```

解读

- `OsMemPoolInfo.pool` 是整个内存池的第一个格子，里面放的是一个内存池起始虚拟地址。
- `OsMemPoolInfo.totalSize` 表示这张纸有多少个格子。
- `OsMemPoolInfo.attr` 表示池子还能不能再变大。
- `OsMemPoolInfo.waterLine` 池子水位警戒线，跟咱三峡大坝发洪水时的警戒线 **175米** 类似，告知上限，水一旦漫过此线就有重大风险， `waterLine` 一词很形象，内核很多思想真来源于生活。
- `OsMemPoolInfo.curUsedSize` 所有已分配内存大小的叠加。
- `freeListBitmap` 空闲位图，这是 `tlfs` 算法的一二级表示，是个长度为 `7` 的整型数组

```
#define OS_MEM_BITMAP_WORDS    ((OS_MEM_FREE_LIST_COUNT >> 5) + 1)
#define OS_MEM_FREE_LIST_COUNT  (OS_MEM_SMALL_BUCKET_COUNT + (OS_MEM_LARGE_BUCKET_COUNT << OS_MEM_SLI))
#define OS_MEM_LARGE_START_BUCKET       7 /// 大桶的开始下标
#define OS_MEM_SMALL_BUCKET_COUNT       31 ///< 小桶的偏移单位 从 4 ~ 124 ,共32级
#define OS_MEM_SLI                      3 ///< 二级小区间级数,
```

  这一坨坨的宏看着有点绕，简单说就是鸿蒙对申请大小分成两种情况
  - 第一种：小桶申请** 当小于128个字节大小的需求平均分成了 `([0-4] , [4-8] , ... , [124-128])` 共 `32` 个等级，而 `freeListBitmap[0]` 为一个 `UINT32` ，共 32 位刚好表示这 `32` 个等级是否有空闲块。例如: 当 `freeListBitmap[0] = 0b...101` 时，如果此时 `malloc(3)` 到来，因 `101` 对应的是 `12` , `8` , `4` 等级，而且 `12` ， `4` 位图位为 `1` ，说明在 `4` 的等级上有空闲内存块可以满足 `malloc(3)` ，需要注意的是虽然 `malloc(3)` 但因为 `4` 等级上只有一种单位 `4` 所以 `malloc(3)` 最后实际得到的是 `4` ，而如果 `malloc(7)` 到来时，正常需要 `8` 等级来满足，但 `8` 等级位图位为 `0` 表示没有空闲内存块，就需要向上找位图为 `1` 的 `12` 等级来申请，于是 `12` 将被分成 `8` ， `4` 两块， `8` 提供给 `malloc(7)` ，剩下的 `4` 挂入等级为 `4` 的空闲链表上。
  - 第二种：大桶申请** 将占用 `freeListBitmap` 的剩余 `6` 个 `UINT32` 整型变量，共可以表示 `32 * 6 = 192` 位 ，同时 `192 = 24 * 8` ，鸿蒙将大于 `128` 个字节的申请按 `2` 次幂分成 `24` 大等级，每个等级又分成 `8` 个小等级 即 **TLFS** 算法 `24` 级对应的范围为 `([2^7-2^8-1] , [2^8-2^9-1] , ... , [2^30-2^31-1])` 而每大级被平均分成 `8` 小级， 例如最小的 `[2^7-2^8-1]` 将被分成每份递增 `2^4 = 16` 大小的八份 `([2^7-2^7+2^4] , [2^7+2^4-2^7+2^4*2] , ... , [2^7+2^4*7-2^8-1])` 而最大的 `[2^30-2^31-1]` 将被分成每份递增 `2^27 = 134 217 728` 大小的八份，请记住 `2^27` 这个数，后面还会说它。 `([2^30-2^30+2^27] , [2^30+2^4-2^30+2^27*2] , ... , [2^30+2^4*7-2^31-1])`
- `OsMemFreeNodeHead freeList[..]` 是空闲链表数组，大小 `224` 个，即每个 `freeListBitmap` 等级都对应了一个链表

```
/// 内存池空闲节点
struct OsMemFreeNodeHead {
    struct OsMemNodeHead header; ///< 内存池节点
    struct OsMemFreeNodeHead *prev; ///< 前一个空闲前驱节点
    struct OsMemFreeNodeHead *next; ///< 后一个空闲后继节点
};
```

  `prev` ， `next` ，指向同级前后节点， 节点的内容在 `OsMemNodeHead` 中，这是一个关键结构体，需单独讲。

内存池节点 | OsMemNodeHead

```
/// 内存池节点
struct OsMemNodeHead {
    UINT32 magic; ///< 魔法数字 0xABCDDCBA
    union {//注意这里的前后指向的是连续的地址节点，用于分割和合并
        struct OsMemNodeHead *prev; /* The prev is used for current node points to the previous node | prev 用于当前节点指向前一个节点*/
        struct OsMemNodeHead *next; /* The next is used for last node points to the expand node | next 用于最后一个节点指向展开节点*/
    } ptr;
#ifdef LOSCFG_MEM_LEAKCHECK //内存泄漏检测
    UINTPTR linkReg[LOS_RECORD_LR_CNT];///< 存放左右节点地址,用于检测
#endif
    UINT32 sizeAndFlag; ///< 数据域大小
};
/// 已使用内存池节点
struct OsMemUsedNodeHead {
    struct OsMemNodeHead header;///< 已被使用节点
#if OS_MEM_FREE_BY_TASKID
    UINT32 taskID; ///< 使用节点的任务ID
#endif
};
```

解读

- `magic` 魔法数字多次提高，内核很多模块都用到了它，比如 **栈顶** ，存在的意义是防止越界，栈溢出栈顶元素就一定会被修改。同理使用了大于申请的内存会导致紧挨着的内存块魔法数字被修改，从而判定为内存溢出。
- 出现一个联合体，其中的 `prev` ，是指向前节点的 **虚拟地址** 或者叫 **线性地址** 也可以叫 **逻辑地址** ， 这些地址是 **连续** 的，注意 **连续性** 很重要，

它是内存块合并和分割的前提，回到图中的 `0x1245` ， `0x12A5` ， `0x1305` 来看，三个内存块节点的地址是**逻辑地址**相连的，内存块节点由头体两部分组成，头部放的是该节点的信息，体是 **malloc(..)** 的返回地址，所以当释放 **free(0xXXX)** 某块内存时很容易知道本节点的起始地址是多少，但向前合并就得知道前节点 `prev` 的地址，而后节点 `next` 的地址可通过 `0xXXX + sizeAndFlag - 头部 = next` 计算得到。既然不需要 `next` 那联合体出现在的 `next` 有什么意思呢? 这个 `next` 是指该块内存的尾节点的意思，当内存池允许扩展大小时，新旧两块内存之间就会产生一个连接处，它们的**线性地址**是不可能连续的，所以不存在合并的问题， `prev` 于它而言没有意义，需要记录下一个内存块的地址，这个工作就交给了联合体中的 `next` 。

- 一个内存池可以由多个内存块组成，每个内存块都有独立的尾节点，指向下一块内存的开始地址，最后一个内存块的尾节点也称为**哨兵节点**，它像个哨兵一样为整个内存池站岗，风餐露宿，固守边疆。当扩大版图之后它又跑到下一站，一个内存池只有一个哨兵，它是最可爱的人，此处应有掌声。
- `linkReg` 用于检测内存泄漏，这部分内容在 **鸿蒙内核源码分析(模块监控)** 已有详细说明，此处不再赘述。
- `UINT32 sizeAndFlag` ，表示**总大小** 包括(头部和体部)和 **标签** ，上面已经让大家记住 `2^27` 这个数，这是动态内存能分配的最大的尺寸。 `UINT32` 中留 `28` 位给它足以，剩下的高 `4` 位就留给 `Flag` 。每位又分别表示以下含义

```
#define OS_MEM_NODE_USED_FLAG      0x80000000U ///< 已使用标签
#define OS_MEM_NODE_ALIGNED_FLAG   0x40000000U ///< 对齐标签
#define OS_MEM_NODE_LAST_FLAG      0x20000000U  /* Sentinel Node | 哨兵节点标签，最后一个节点*/
#define OS_MEM_NODE_ALIGNED_AND_USED_FLAG (OS_MEM_NODE_USED_FLAG | OS_MEM_NODE_ALIGNED_FLAG | OS_MEM_NODE_LAST_FL
```

- 从联合体和 `sizeAndFlag` 可以看出鸿蒙的设计思想，充分利用空间，准确区分概念，一张卫生纸擦完嘴还要接着擦地，节俭之家必有余粮啊，这是非常有必要的，因为内存资源太稀缺了。在实际运行过程中，分配节点常数以万计，每个能省一个 `UINT32` ，就是一万个 `UINT32` ，约等于 `39KB` ，非常可观。 这也是为什么站长始终觉得鸿蒙是个大宝藏的原因。
- `OsMemUsedNodeHead.taskID` 已使用节点比空闲节点头部多了一个使用该节点任务的标记，由开关宏 `OS_MEM_FREE_BY_TASKID` 控制，默认是关闭的。

代码实现

有了这么长的铺垫，再来看鸿蒙内核动态内存管理的代码简直就是易如反掌，此处拆解 **节点切割** ，**节点合并** ，**内存池扩展** 三段代码。都已添加详细的注解 ，所有注解代码请前往 百万汉字注解鸿蒙内核 | kernel_liteos_a_note 仓库查看

节点切割 | OsMemSplitNode

```
/// 切割节点
STATIC INLINE VOID OsMemSplitNode(VOID *pool, struct OsMemNodeHead *allocNode, UINT32 allocSize)
{
    struct OsMemFreeNodeHead *newFreeNode = NULL;
    struct OsMemNodeHead *nextNode = NULL;
    newFreeNode = (struct OsMemFreeNodeHead *)(VOID *)((UINT8 *)allocNode + allocSize);//切割后出现的新空闲节点,在分配节点的右侧
    newFreeNode->header.ptr.prev = allocNode;//新节点指向前节点,说明是从左到右切割
    newFreeNode->header.sizeAndFlag = allocNode->sizeAndFlag - allocSize;//新空闲节点大小
    allocNode->sizeAndFlag = allocSize;//分配节点大小
    nextNode = OS_MEM_NEXT_NODE(&newFreeNode->header);//获取新节点的下一个节点
    if (!OS_MEM_NODE_GET_LAST_FLAG(nextNode->sizeAndFlag)) {//如果下一个节点不是哨兵节点(末尾节点)
        nextNode->ptr.prev = &newFreeNode->header;//下一个节点的前节点为新空闲节点
        if (!OS_MEM_NODE_GET_USED_FLAG(nextNode->sizeAndFlag)) {//如果下一个节点也是空闲的
            OsMemFreeNodeDelete(pool, (struct OsMemFreeNodeHead *)nextNode);//删除下一个节点信息
            OsMemMergeNode(nextNode);//下一个节点和新空闲节点 合并成一个新节点
        }
    }
    OsMemFreeNodeAdd(pool, newFreeNode);//挂入空闲链表
}
```

节点合并 | OsMemMergeNode

```
/// 合并节点,和前面的节点合并 node 消失
STATIC INLINE VOID OsMemMergeNode(struct OsMemNodeHead *node)
{
    struct OsMemNodeHead *nextNode = NULL;
    node->ptr.prev->sizeAndFlag += node->sizeAndFlag; //前节点长度变长
    nextNode = (struct OsMemNodeHead *)((UINTPTR)node + node->sizeAndFlag); // 下一个节点位置
    if (!OS_MEM_NODE_GET_LAST_FLAG(nextNode->sizeAndFlag)) {//不是哨兵节点
        nextNode->ptr.prev = node->ptr.prev;//后一个节点的前节点变成前前节点
    }
```

```
}
```

内存池扩展

```
/// 内存池扩展实现
STATIC INLINE INT32 OsMemPoolExpandSub(VOID *pool, UINT32 size, UINT32 intSave)
{
    UINT32 tryCount = MAX_SHRINK_PAGECACHE_TRY;
    struct OsMemPoolHead *poolInfo = (struct OsMemPoolHead *)pool;
    struct OsMemNodeHead *newNode = NULL;
    struct OsMemNodeHead *endNode = NULL;
    size = ROUNDUP(size + OS_MEM_NODE_HEAD_SIZE, PAGE_SIZE);//圆整
    endNode = OS_MEM_END_NODE(pool, poolInfo->info.totalSize);//获取哨兵节点
RETRY:
    newNode = (struct OsMemNodeHead *)LOS_PhysPagesAllocContiguous(size >> PAGE_SHIFT);//申请新的内存池 | 物理内存
    if (newNode == NULL)
        return -1;
    newNode->sizeAndFlag = (size - OS_MEM_NODE_HEAD_SIZE);//设置新节点大小
    newNode->ptr.prev = OS_MEM_END_NODE(newNode, size);//新节点的前节点指向新节点的哨兵节点
    OsMemSentinelNodeSet(endNode, newNode, size);//设置老内存池的哨兵节点信息,其实就是指向新内存块
    OsMemFreeNodeAdd(pool, (struct OsMemFreeNodeHead *)newNode);//将新节点加入空闲链表
    endNode = OS_MEM_END_NODE(newNode, size);//获取新节点的哨兵节点
    (VOID)memset(endNode, 0, sizeof(*endNode));//清空内存
    endNode->ptr.next = NULL;//新哨兵节点没有后续指向,因为它已成为最后
    endNode->magic = OS_MEM_NODE_MAGIC;//设置新哨兵节的魔法数字
    OsMemSentinelNodeSet(endNode, NULL, 0); //设置新哨兵节点内容
    OsMemWaterUsedRecord(poolInfo, OS_MEM_NODE_HEAD_SIZE);//更新内存池警戒线
    return 0;
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

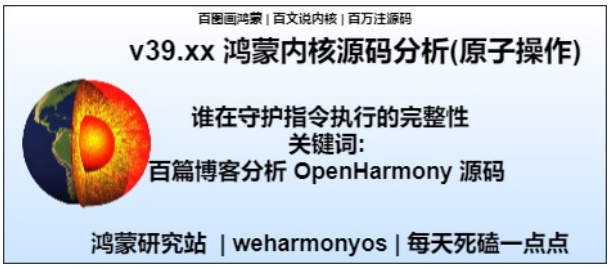
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

39_原子操作篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

本篇说清楚原子操作

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 系列篇。

基本概念

在支持多任务的操作系统中，修改一块内存区域的数据需要“读取-修改-写入”三个步骤。然而同一内存区域的数据可能同时被多个任务访问，如果在修改数据的过程中被其他任务打断，就会造成该操作的执行结果无法预知。

使用开关中断的方法固然可以保证多任务执行结果符合预期，但这种方法显然会影响系统性能。

`ARMv6` 架构引入了 `LDREX` 和 `STREX` 指令，以支持对共享存储器更缜密的非阻塞同步。由此实现的原子操作能确保对同一数据的“读取-修改-写入”操作在它的执行期间不会被打断，即操作的原子性。

有多个任务对同一个内存数据进行加减或交换操作时，使用原子操作保证结果的可预知性。

看过 v08.xx 鸿蒙内核源码分析(总目录) 自旋锁篇的应该对LDREX和STREX指令不陌生的，自旋锁的本质就是对某个变量的原子操作，而且一定要通过汇编代码实现，也就是说 `LDREX` 和 `STREX` 指令保证了原子操作的底层实现。 回顾下自旋锁申请和释放锁的汇编代码。

ArchSpinLock 申请锁代码

```
FUNCTION(ArchSpinLock)  @死守，非要拿到锁
    mov     r1， #1     @r1=1
1:                  @循环的作用，因SEV是广播事件。不一定lock->rawLock的值已经改变了
    ldrex   r2， [r0]   @r0 = &lock->rawLock， 即 r2 = lock->rawLock
    cmp     r2， #0     @r2和0比较
    wfene               @不相等时，说明资源被占用，CPU核进入睡眠状态
    strexeq r2， r1， [r0]@此时CPU被重新唤醒，尝试令lock->rawLock=1，成功写入则r2=0
    cmpeq   r2， #0     @再来比较r2是否等于0，如果相等则获取到了锁
    bne     1b          @如果不相等，继续进入循环
    dmb                 @用DMB指令来隔离，以保证缓冲中的数据已经落实到RAM中
    bx      lr          @此时是一定拿到锁了，跳回调用ArchSpinLock函数
```

ArchSpinUnlock 释放锁代码

```
FUNCTION(ArchSpinUnlock)   @释放锁
    mov    r1， #0        @r1=0
    dmb                  @数据存储隔离，以保证缓冲中的数据已经落实到RAM中
    str    r1， [r0]      @令lock->rawLock = 0
    dsb                  @数据同步隔离
    sev                  @给各CPU广播事件，唤醒沉睡的CPU们
    bx     lr            @跳回调用ArchSpinLock函数
```

运作机制

鸿蒙通过对 `ARMv6` 架构中的 `LDREX` 和 `STREX` 进行封装，向用户提供了一套原子操作接口。

- LDREX Rx， [Ry] 读取内存中的值，并标记对该段内存为独占访问：
  - 读取寄存器Ry指向的4字节内存数据，保存到Rx寄存器中。
  - 对Ry指向的内存区域添加独占访问标记。
- STREX Rf， Rx， [Ry] 检查内存是否有独占访问标记，如果有则更新内存值并清空标记，否则不更新内存：
  - 有独占访问标记
    - 将寄存器Rx中的值更新到寄存器Ry指向的内存。
    - 标志寄存器Rf置为0。
  - 没有独占访问标记
    - 不更新内存。
    - 标志寄存器Rf置为1。
- 判断标志寄存器 标志寄存器为0时，退出循环，原子操作结束。 标志寄存器为1时，继续循环，重新进行原子操作。

功能列表

原子数据包含两种类型Atomic（有符号32位数）与 Atomic64（有符号64位数）。原子操作模块为用户提供下面几种功能，接口详细信息可以查看源码。

功能分类	接口名	描述
读	LOS_AtomicRead	读取内存数据
写	LOS_AtomicSet	写入内存数据
加	LOS_AtomicAdd	对内存数据做加法
	LOS_AtomicSub	对内存数据做减法
	LOS_AtomicInc	对内存数据加1
	LOS_AtomicIncRet	对内存数据加1并返回运算结果
减	LOS_AtomicDec	对内存数据减1
	LOS_AtomicDecRet	对内存数据减1并返回运算结果
交换	LOS_AtomicXchg32bits	交换内存数据，原内存中的值以返回值的方式返回
	LOS_AtomicCmpXchg32bits	比较并交换内存数据，返回比较结果

此处讲述 `LOS_AtomicAdd` ， `LOS_AtomicSub` ，`LOS_AtomicRead` ，`LOS_AtomicSet` 理解了函数的汇编代码是理解的原子操作的关键。

LOS_AtomicAdd

```
//对内存数据做加法
```

```
STATIC INLINE INT32 LOS_AtomicAdd(Atomic *v， INT32 addVal)
{
   INT32 val;
   UINT32 status;
   do {
      __asm__ __volatile__("ldrex   %1， [%2]\n"
                           "add   %1， %1， %3\n"
                           "strex   %0， %1， [%2]"
                           : "=&r"(status)， "=&r"(val)
                           : "r"(v)， "r"(addVal)
                           : "cc");
   } while (__builtin_expect(status != 0， 0));
   return val;
}
```

这是一段C语言内嵌汇编，逐一解读

- 先将 `status` `val` `v` `addVal` 的值交由通用寄存器(R0~R3)接管。
- %2代表了入参v，[%2]代表的是参数v指向地址的值，也就是 *v ，函数要独占的就是它
- %0 ~ %3 对应 `status` `val` `v` `addVal`
- ldrex %1， [%2] 表示 val = *v ;
- add %1， %1， %3 表示 val = val + addVal;
- strex %0， %1， [%2] 表示 *v = val;
- status 表示是否更新成功，成功了置0，不成功则为 1
- __builtin_expect是结束循环的判断语句，将最有可能执行的分支告诉编译器。 这个指令的写法为：__builtin_expect(EXP， N)。

  > 意思是：EXP==N 的概率很大。
  >
  > 综合理解__builtin_expect(status != 0， 0)
  >
  > 说的是status = 0 的可能性很大，不成功就会重新来一遍，直到strex更新成(status == 0)为止。

- "=&r"(val) 被修饰的操作符作为输出，即将寄存器的值回给val，val为函数的返回值
- "cc"向编译器声明以上信息。

LOS_AtomicSub

```
//对内存数据做减法
STATIC INLINE INT32 LOS_AtomicSub(Atomic *v， INT32 subVal)
{
   INT32 val;
   UINT32 status;
   do {
      __asm__ __volatile__("ldrex   %1， [%2]\n"
                           "sub   %1， %1， %3\n"
                           "strex   %0， %1， [%2]"
                           : "=&r"(status)， "=&r"(val)
                           : "r"(v)， "r"(subVal)
                           : "cc");
   } while (__builtin_expect(status != 0， 0));
   return val;
}
```

解读

- 同 `LOS_AtomicAdd` 解读

volatile

这里要重点说下`volatile`，`volatile` 提醒编译器它后面所定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，**都**

要直接从变量地址中读取数据。如果没有 volatile 关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

```
//读取内存数据
STATIC INLINE INT32 LOS_AtomicRead(const Atomic *v)
{
    return *(volatile INT32 *)v;
}
//写入内存数据
STATIC INLINE VOID LOS_AtomicSet(Atomic *v， INT32 setVal)
{
    *(volatile INT32 *)v = setVal;
}
```

编程实例

调用原子操作相关接口，观察结果：

1。创建两个任务

- 任务一用LOS_AtomicAdd对全局变量加100次。
- 任务二用LOS_AtomicSub对全局变量减100次。

2。子任务结束后在主任务中打印全局变量的值。

```
#include "los_hwi.h"
#include "los_atomic.h"
#include "los_task.h"

UINT32 g_testTaskId01;
UINT32 g_testTaskId02;
Atomic g_sum;
Atomic g_count;

UINT32 Example_Atomic01(VOID)
{
    int i = 0;
    for(i = 0; i < 100; ++i) {
        LOS_AtomicAdd(&g_sum , 1);
    }
    LOS_AtomicAdd(&g_count , 1);
    return LOS_OK;
}

UINT32 Example_Atomic02(VOID)
{
    int i = 0;
    for(i = 0; i < 100; ++i) {
        LOS_AtomicSub(&g_sum , 1);
    }
    LOS_AtomicAdd(&g_count , 1);
    return LOS_OK;
}

UINT32 Example_TaskEntry(VOID)
{
    TSK_INIT_PARAM_S stTask1={0};
    stTask1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_Atomic01;
    stTask1.pcName       = "TestAtomicTsk1";
    stTask1.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    stTask1.usTaskPrio   = 4;
    stTask1.uwResved     = LOS_TASK_STATUS_DETACHED;
    TSK_INIT_PARAM_S stTask2={0};
    stTask2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_Atomic02;
    stTask2.pcName       = "TestAtomicTsk2";
    stTask2.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    stTask2.usTaskPrio   = 4;
    stTask2.uwResved     = LOS_TASK_STATUS_DETACHED;
```

```
    LOS_TaskLock();
    LOS_TaskCreate(&g_testTaskId01， &stTask1);
    LOS_TaskCreate(&g_testTaskId02， &stTask2);
    LOS_TaskUnlock();
    while(LOS_AtomicRead(&g_count) != 2);
    dprintf("g_sum = %d\n"， g_sum);
    return LOS_OK;
}
```

结果验证

```
g_sum = 0
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

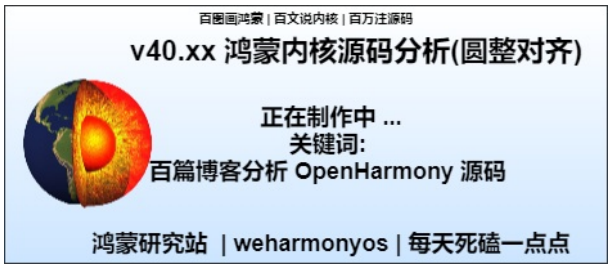
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

40_圆整对齐篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内存管理相关篇为:

- v31.02 鸿蒙内核源码分析(内存规则) | 内存管理到底在管什么
- v32.04 鸿蒙内核源码分析(物理内存) | 真实的可不一定精彩
- v33.04 鸿蒙内核源码分析(内存概念) | RAM & ROM & Flash
- v34.03 鸿蒙内核源码分析(虚实映射) | 映射是伟大的发明
- v35.02 鸿蒙内核源码分析(页表管理) | 映射关系保存在哪
- v36.03 鸿蒙内核源码分析(静态分配) | 很简单的一位小朋友
- v37.01 鸿蒙内核源码分析(TLFS算法) | 图表解读TLFS原理
- v38.01 鸿蒙内核源码分析(内存池管理) | 如何高效切割合并内存块
- v39.04 鸿蒙内核源码分析(原子操作) | 谁在守护指令执行的完整性
- v40.01 鸿蒙内核源码分析(圆整对齐) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看 页表按 16Kb 对齐

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

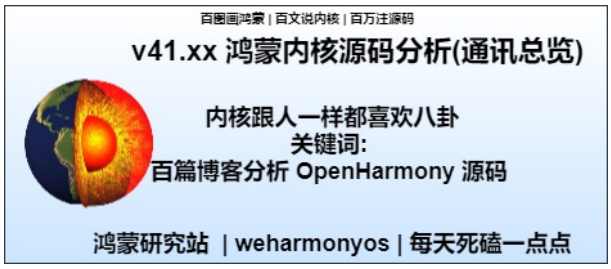
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

41_通讯总览篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

通讯需求

鸿蒙内核默认支持 `64` 个进程和 `128` 个任务，由进程池和任务池统一管理。内核设计尽量不去打扰它们，让各自过好各自的小日子，但大家毕竟在一口锅里吃饭，再宅的宅男也要出门办事，出了门磕磕碰碰就在所难免了，有了纠纷就需要解决，社区问题可以找居委会刘大妈，家庭问题可以找德高望重的七舅姥爷，社会问题可以报警，法律问题可以打官司仲裁。总之天塌不下来，需求决定了解决方案，不存在只有需求没有解决方案的情况，至少不会长期存在，想想是不是这个道理。

通信方式:

(1).数据传输： 一个进程需要将它的数据发送给另一个进程，发送的数据量在一个字节到KB字节之间。(liteipc消息队列默认1K)

(2).共享数据: 多个进程想要操作共享数据，一个进程对共享数据的修改，别的进程应该立刻看到。

(3).通知事件： 一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。

进程间九种通讯方式

1.管道pipe(fs_syscall.c)

管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。 调用pipe系统函数即可创建一个管道。有如下特质：

1. 其本质是一个伪文件(实为内核缓冲区)
2. 由两个文件描述符引用，一个表示读端，一个表示写端。
3. 规定数据从管道的写端流入管道，从读端流出。

管道的原理: 管道实为内核使用环形队列机制，借助内核缓冲区(4k)实现。 管道的局限性： ① 数据自己读不能自己写。 ② 数据一旦被读走，便不在管道中存在，不可反复读取。 ③ 由于管道采用半双工通信方式。因此，数据只能在一个方向上流动。 ④ 只能在有公共祖先的进程间使用管道。 常见的通信方式有，单工通信、半双工通信、全双工通信。

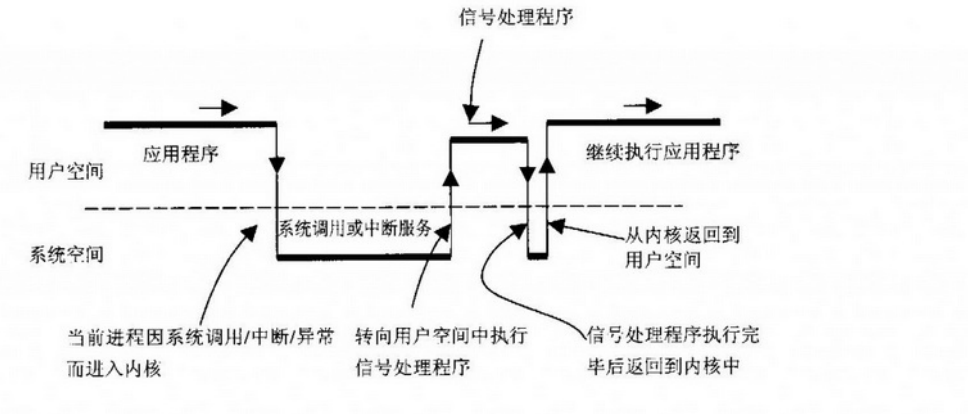
这部分后系列篇文件相关篇中会重点讲，敬请关注。 详细看 SysPipe 函数。

2.信号(los_signal.c)

信号思想来自Unix，在发展了50年之后，许多方面都没有发生太大的变化。信号可以由内核产生，也可以由用户进程产生，并由内核传送给特定的进程或线程(组)，若这个进程注册/安装了自己的信号处理程序，则内核会调用这个函数去处理信号，否则则执行默认的函数或者忽略。信号分为两大类：可靠信号与不可靠信号，前32种信号为不可靠信号，后32种为可靠信号。长这样:

```
#define SIGHUP    1 //终端挂起或者控制进程终止
#define SIGINT    2 //键盘中断（如break键被按下）
#define SIGQUIT   3 //键盘的退出键被按下
#define SIGILL    4 //非法指令
#define SIGTRAP   5 //跟踪陷阱（trace trap），启动进程，跟踪代码的执行
#define SIGABRT   6 //由abort(3)发出的退出指令
#define SIGIOT    SIGABRT
#define SIGBUS    7 //总线错误
#define SIGFPE    8 //浮点异常
#define SIGKILL   9  //常用的命令 kill 9 13
#define SIGUSR1   10 //用户自定义信号1
```

信号为系统提供了一种进程间异步通讯的方式，一个进程不必通过任何操作来等待信号的到达。事实上，进程也不可能知道信号到底什么时候到达。一般来说，只需用户进程提供信号处理函数，内核会想方设法调用信号处理函数，处理过程如图所示:



个人把这种异步通讯过程理解为生产者(安装和发送信号)和消费者(捕捉和处理信号)两个部分，分姊妹两篇已完成

- v48.xx (信号生产篇) | 年过半百，依然活力十足
- v49.xx (信号消费篇) | 谁让CPU连续四次换栈运行

3.消息队列(los_queue.c)

基本概念

队列又称消息队列，是一种常用于任务间通信的数据结构。队列接收来自任务或中断的 不固定长度消息，并根据不同的接口确定传递的消息是否存放在队列空间中。

任务能够从队列里面读取消息，当队列中的消息为空时，挂起读取任务；当队列中有新消息时， 挂起的读取任务被唤醒并处理新消息。任务也能够往队列里写入消息，当队列已经写满消息时， 挂起写入任务；当队列中有空闲消息节点时，挂起的写入任务被唤醒并写入消息。如果将 读队列和写队列的超时时间设置为0，则不会挂起任务，接口会直接返回，这就是非阻塞模式。

消息队列提供了异步处理机制，允许将一个消息放入队列，但不立即处理。同时队列还有缓冲消息的作用。

队列特性

消息以先进先出的方式排队，支持异步读写。 读队列和写队列都支持超时机制。 每读取一条消息，就会将该消息节点设置为空闲。 发送消息类型由通信双方约定，可以允许不同长度（不超过队列的消息节点大小）的消息。 一个任务能够从任意一个消息队列接收和发送消息。 多个任务能够从同一个消息队列接收和发送消息。 创建队列时所需的队列空间，默认支持接口内系统自行动态申请内存的方式，同时也支持将用户分配的队列空间作为接口入参传入的方式。

详细可前往查看:

- v33.xx (消息队列篇) | 进程间如何异步解耦传递大数据 ？

4.共享内存(shm.c)

共享内存是进程间通信中最简单的方式之一。共享内存允许两个或更多进程访问同一块物理内存，每个进程都要单独对这块物理内存进行映射。当一个进程改变了这块地址中的内容的时候，该物理页框将被标记为脏页，如此其它进程都会知道内容发生了更改。

这部分后系列篇内存相关篇中会重点讲，内存部分虽已写过几篇，但是没讲透，要重新再梳理。

5.信号量(los_sem.c)

基本概念

信号量（Semaphore）是一种实现任务间通信的机制，可以实现任务间同步或共享资源的互斥访问。 一个信号量的数据结构中，通常有一个计数值，用于对有效资源数的计数，表示剩下的可被使用的共享资源数。

对信号量有个形象的比喻 停车场的停车位， 进停车场前看下屏幕上实时显示剩余车位，0表示不能进，只有大于0才能进入，进入后自动减1，出口处也加了监测，出去后剩余车位增加1个。

使用场景

在多任务系统中，信号量是一种非常灵活的同步方式，可以运用在多种场合中，实现锁、同步、资源计数等功能， 也能方便的用于任务与任务，中断与任务的同步中。常用于协助一组相互竞争的任务访问共享资源。

详细可前往查看:

- v29.xx (信号量篇) | 信号量解决任务同步问题

6.互斥锁 (los_mux.c) :

基本概念

互斥锁又称互斥型信号量，是一种特殊的二值性信号量，用于实现对临界资源的独占式处理。 另外，互斥锁可以解决信号量存在的优先级翻转问题。 任意时刻互斥锁只有两种状态，开锁或闭锁。当任务持有时，这个任务获得该互斥锁的所有权， 互斥锁处于闭锁状态。当该任务释放锁后，任务失去该互斥锁的所有权，互斥锁处于开锁状态。 当一个任务持有互斥锁时，其他任务不能再对该互斥锁进行开锁或持有。

详细可前往查看:

- v27.xx (互斥锁篇) | 互斥锁比自旋锁可丰满许多
- v26.xx (自旋锁篇) | 真的好想为自旋锁立贞节牌坊!

7.快锁 (los_futex.c)

futex 是Fast Userspace muTexes的缩写(快速用户空间互斥体)，是一种用户态和内核态混合的同步机制。首先，同步的进程间通过mmap共享一段内存，futex变量就位于这段共享的内存中且操作是原子的，当进程尝试进入互斥区或者退出互斥区的时候，先去查看共享内存中的futex变量，如果没有竞争发生，则只修改futex，而不用再执行系统调用了。当通过访问futex变量告诉进程有竞争发生，则还是得执行系统调用去完成相应的处理(wait 或者 wake up)。

注解版同步到官方最新源码后，发现快锁的部分改动很大，这部分要重新注解，敬请留意。

8.事件 (los_event.c)

基本概念 事件（Event）是一种任务间通信的机制，可用于任务间的同步。

多任务环境下，任务之间往往需要同步操作，一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。 一对多同步模型：一个任务等待多个事件的触发。可以是任意一个事件发生时唤醒任务处理事件，也可以是几个事件都发生后才唤醒任务处理事件。 多对多同步模型：多个任务等待多个事件的触发。

事件特点

任务通过创建事件控制块来触发事件或等待事件。 事件间相互独立，内部实现为一个32位无符号整型，每一位标识一种事件类型。第25位不可用，因此最多可支持31种事件类型。 事件仅用于任务间的同步，不提供数据传输功能。 多次向事件控制块写入同一事件类型，在被清零前等效于只写入一次。 多个任务可以对同一事件进行读写操作。 支持事件读写超时机制。

事件可应用于多种任务同步场景，在某些同步场景下可替代信号量。

使用场景

队列用于任务间通信，可以实现消息的异步处理。同时消息的发送方和接收方不需要彼此联系，两者间是解耦的。

详细可前往查看:

- v30.xx (事件控制篇) | 任务间多对多的同步方案

9.文件消息队列 (hm_liteipc.c)

基于文件实现的消息队列，特点是队列中消息数量多(256个)，传递消息内容大(可到1K)

```
#define IPC_MSG_DATA_SZ_MAX 1024 //最大的消息内容 1K ，posix最大消息内容 64个字节
#define IPC_MSG_OBJECT_NUM_MAX 256 //最大的消息数量256 ，posix最大消息数量 16个
```

文件消息队列隐约感觉鸿蒙的分布式通讯，跨屏之类的功能是靠它实现的，分布式的代码还没研究，尚不清楚，如果有了解的请告知。后续要重点研究下跨应用通讯的技术实现。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

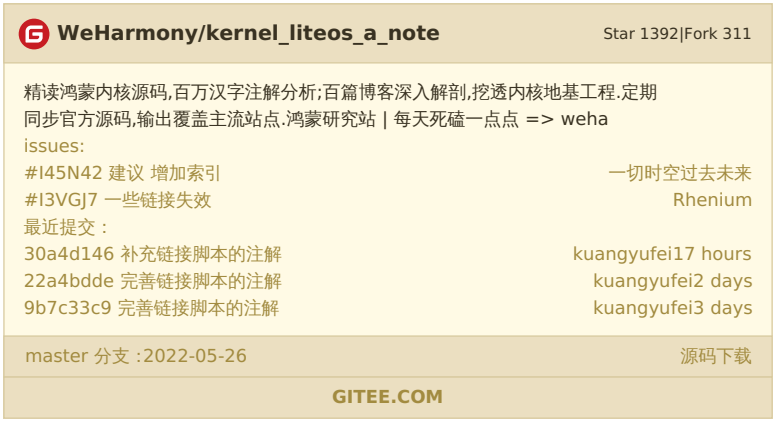
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

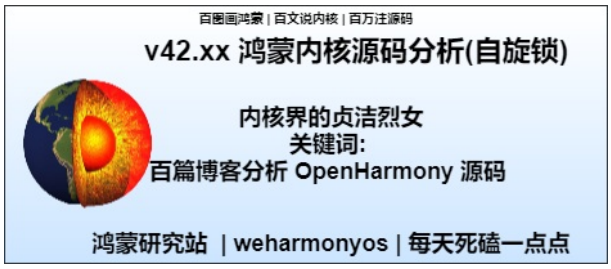


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

42_自旋锁篇

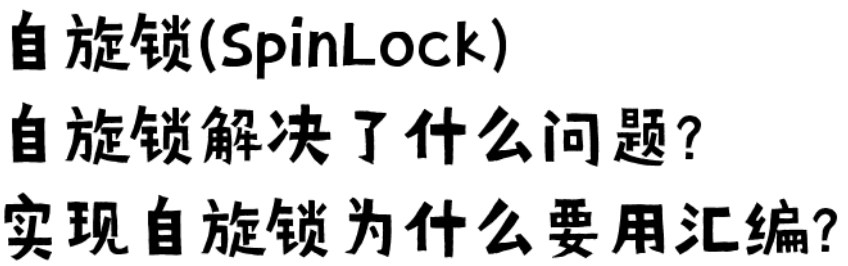
本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

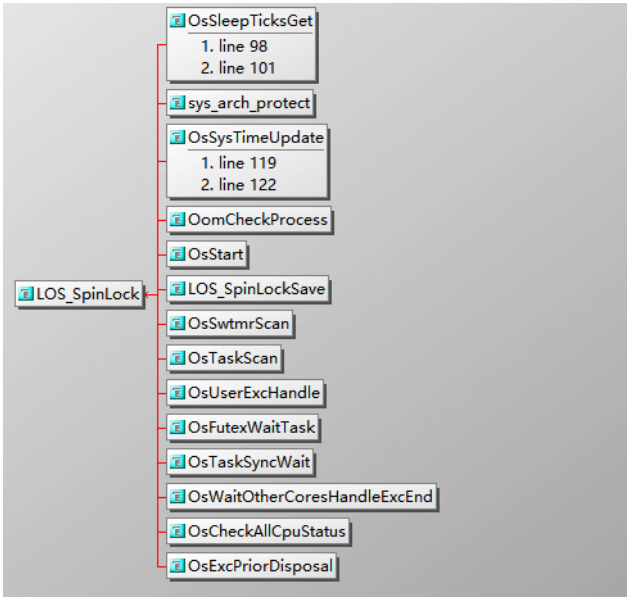
- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Litelpc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Litelpc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式



本篇说清楚自旋锁

读本篇之前建议先读系列篇 进程/线程篇。

内核中哪些地方会用到自旋锁？看图:



概述

自旋锁 顾名思义，是一把自动旋转的锁，这很像厕所里的锁，进入前标记是绿色可用的，进入格子间后，手一带，里面的锁转个圈，外面标记变成了红色表示在使用，外面的只能等待。这是形象的比喻，但实际也是如此。

在多 CPU 核环境中，由于使用相同的内存空间，存在对同一资源进行访问的情况，所以需要互斥访问机制来保证同一时刻只有一个核进行操作，自旋锁就是这样的一种机制。

- 自旋锁是指当一个线程在获取锁时，如果锁已经被其它 CPU 中的线程获取，那么该线程将循环等待，并不断判断是否能够成功获取锁，直到其它 CPU 释放锁后，等锁CPU才会退出循环。
- 自旋锁的设计理念是它仅会被持有非常短的时间，锁只能被一个任务持有，而且持有自旋锁的CPU是不可以进入睡眠模式的，因为其他的CPU在等待锁，为了防止死锁上下文交换也是不允许的，是禁止发生调度的。
- 自旋锁与互斥锁比较类似，它们都是为了解决对共享资源的互斥使用问题。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个持有者。但是两者在调度机制上略有不同，对于互斥锁，如果锁已经被占用，锁申请者会被阻塞；但是自旋锁不会引起调用者阻塞，会一直循环检测自旋锁是否已经被释放。

虽然都是共享资源竞争，但自旋锁强调的是 CPU 核间的竞争，而互斥量强调的是任务(包括同一 CPU 核)之间的竞争。

自旋锁长什么样？

```
typedef struct Spinlock {//自旋锁结构体
    size_t      rawLock;//原始锁
#if (LOSCFG_KERNEL_SMP_LOCKDEP == YES) // 死锁检测模块开关
    UINT32      cpuid; //持有锁的CPU
    VOID        *owner; //持有锁任务
    const CHAR  *name; //锁名称
#endif
} SPIN_LOCK_S;
```

结构体很简单，里面有个宏，用于死锁检测，默认情况下是关闭的。所以真正的被使用的变量只有 rawLock 一个。但C语言代码中找不到变量的变化过程，而是通过一段汇编代码来实现。看完本篇会明白也只能通过汇编代码来实现自旋锁。

自旋锁使用流程

自旋锁用于多 CPU 核的情况，解决的是 CPU 之间竞争资源的问题。使用流程很简单，三步走。

- 创建自旋锁：使用 LOS_SpinInit 初始化自旋锁，或者使用 SPIN_LOCK_INIT 初始化静态内存的自旋锁。
- 申请自旋锁：使用接口 LOS_SpinLock LOS_SpinTrylock LOS_SpinLockSave 申请指定的自旋锁，申请成功就继续往后执行锁保护的代码；申请失败在自旋锁申请中忙等，直到申请到自旋锁为止。
- 释放自旋锁：使用 LOS_SpinUnlock LOS_SpinUnlockRestore 接口释放自旋锁。锁保护代码执行完毕后，释放对应的自旋锁，以便其他核申请自

旋锁。

几个关键函数

自旋锁模块由内联函数实现，见于 los_spinlock.h 代码不多，主要是三个函数。

```
ArchSpinLock(&lock->rawLock);
ArchSpinTrylock(&lock->rawLock)
ArchSpinUnlock(&lock->rawLock);
```

可以说掌握了它们就掌握了自旋锁，但这三个函数全由汇编实现。见于 los_dispatch.S 文件 因为系列篇已有两篇讲过汇编代码，所以很容易理解这三段代码。函数的参数由 r0 记录，即 r0 保存了 lock->rawLock 的地址，拿锁/释放锁是让 lock->rawLock 在0，1切换 下面逐一说明自旋锁的汇编代码。

ArchSpinLock 汇编代码

```
FUNCTION(ArchSpinLock)  @死守，非要拿到锁
    mov     r1， #1     @r1=1
1:                  @循环的作用，因SEV是广播事件。不一定lock->rawLock的值已经改变了
    ldrex   r2， [r0]   @r0 = &lock->rawLock， 即 r2 = lock->rawLock
    cmp     r2， #0     @r2和0比较
    wfene               @不相等时，说明资源被占用，CPU核进入睡眠状态
    strexeq r2， r1， [r0]@此时CPU被重新唤醒，尝试令lock->rawLock=1，成功写入则r2=0
    cmpeq   r2， #0     @再来比较r2是否等于0，如果相等则获取到了锁
    bne     1b          @如果不相等，继续进入循环
    dmb                 @用DMB指令来隔离，以保证缓冲中的数据已经落实到RAM中
    bx      lr          @此时是一定拿到锁了，跳回调用ArchSpinLock函数
```

看懂了这段汇编代码就理解了自旋锁实现的真正机制，为什么一定要用汇编来实现。 因为 CPU 宁愿睡眠也非要拿到锁不可的， 注意这里可不是让线程睡眠，而是让 CPU 进入睡眠状态，能让 CPU 进入睡眠的只能通过汇编实现。C语言根本就写不出让 CPU 真正睡眠的代码。

ArchSpinTrylock 汇编代码

如果不看下面这段汇编代码，你根本不可能知道 ArchSpinTrylock 和 ArchSpinLock 的真正区别是什么。

```
FUNCTION(ArchSpinTrylock)   @尝试拿锁，拿不到就撤
    mov     r1， #1         @r1=1
    mov     r2， r0         @r2 = r0
    ldrex   r0， [r2]       @r2 = &lock->rawLock， 即 r0 = lock->rawLock
    cmp     r0， #0         @r0和0比较
    strexeq r0， r1， [r2]   @尝试令lock->rawLock=1，成功写入则r0=0，否则 r0 =1
    dmb                     @数据存储隔离，以保证缓冲中的数据已经落实到RAM中
    bx      lr              @跳回调用ArchSpinLock函数
```

比较两段汇编代码可知， ArchSpinTrylock 即没有循环也不会让 CPU 进入睡眠，直接返回了，而 ArchSpinLock 会睡了醒， 醒了睡，一直守到丈夫(lock->rawLock = 0 的广播事件发生)回来才肯罢休。 笔者代码注释到此处那真是心潮澎湃，心碎了老一地， 真想给 ArchSpinLock 立一个贞节牌坊!

ArchSpinUnlock 汇编代码

```
FUNCTION(ArchSpinUnlock)   @释放锁
    mov     r1， #0         @r1=0
    dmb                   @数据存储隔离，以保证缓冲中的数据已经落实到RAM中
    str     r1， [r0]       @令lock->rawLock = 0
    dsb                   @数据同步隔离
    sev                   @给各CPU广播事件，唤醒沉睡的CPU们
    bx      lr            @跳回调用ArchSpinLock函数
```

代码中涉及到几个不常用的汇编指令，一一说明:

汇编指令之 WFI / WFE / SEV

WFI (Wait for interrupt):等待中断到来指令。 WFI 一般用于 cpuidle ， WFI 指令是在处理器发生中断或类似异常之前不需要做任何事情。

在鸿蒙源码分析系列篇(总目录)线程篇中已说过，每个 CPU 都有自己的 idle 任务， CPU 没事干的时候就待在里面，就一个死循环守着WFI指令，有中断来了就触发 CPU 起床干活。 中断分硬中断和软中断，系统调用就是通过软中断实现的，而设备类的就属于硬中断，都能触发 CPU 干活。 具体看下 CPU 空闲的时候在干嘛，代码超级简单:

```
LITE_OS_SEC_TEXT WEAK VOID OsIdleTask(VOID) //CPU没事干的时候待在这里
{
   while (1) {//只有一个死循环
      Wfi();//WFI指令:arm core 立即进入low-power standby state，等待中断，进入休眠模式。
   }
}
```

WFE (Wait for event):等待事件的到来指令 WFE 指令是在 SEV 指令生成事件之前不需要执行任何操作，所以用WFE的地方，后续一定会对应一个SEV的指令去唤醒它。 WFE的一个典型使用场景，是用在自旋锁中， spinlock 的功能，是在不同CPU core之间，保护共享资源。使用 WFE 的流程是：

- 开始之初资源空闲
- CPU核1 访问资源，持有锁，获得资源
- CPU核2 访问资源，此时资源不空闲，执行WFE指令，让core进入low-power state(睡眠)
- CPU核1 释放资源，释放锁，释放资源，同时执行 SEV 指令，唤醒CPU核2
- CPU核2 获得资源

另外说一下 以往的自旋锁，在获得不到资源时，让 CPU 核进入死循环，而通过插入 WFE 指令，则大大节省功耗。

SEV (send event):发送事件指令，SEV是一条广播指令，它会将事件发送到多处理器系统中的所有处理器，以唤醒沉睡的 CPU 。

SEV 和 WFE 的实现很像设计模式的观察者模式。

汇编指令之 LDREX / STREX

LDREX 用来读取内存中的值，并标记对该段内存的独占访问：

LDREX Rx，[Ry] 上面的指令意味着，读取寄存器 Ry 指向的4字节内存值，将其保存到Rx寄存器中，同时标记对 Ry 指向内存区域的独占访问。

如果执行 LDREX 指令的时候发现已经被标记为独占访问了，并不会对指令的执行产生影响。

而STREX在更新内存数值时，会检查该段内存是否已经被标记为独占访问，并以此来决定是否更新内存中的值：

STREX Rx， Ry，[Rz] 如果执行这条指令的时候发现已经被标记为独占访问了，则将寄存器Ry中的值更新到寄存器 Rz 指向的内存，并将寄存器 Rx 设置成0。指令执行成功后，会将独占访问标记位清除。

而如果执行这条指令的时候发现没有设置独占标记，则不会更新内存，且将寄存器 Rx 的值设置成1。

一旦某条 STREX 指令执行成功后，以后再对同一段内存尝试使用 STREX 指令更新的时候，会发现独占标记已经被清空了，就不能再更新了，从而实现独占访问的机制。

编程实例

本实例实现如下流程。

- 任务Example_TaskEntry初始化自旋锁，创建两个任务Example_SpinTask1、Example_SpinTask2，分别运行于两个核。
- Example_SpinTask1、Example_SpinTask2中均执行申请自旋锁的操作，同时为了模拟实际操作，在持有自旋锁后进行延迟操作，最后释放自旋锁。
- 300Tick后任务Example_TaskEntry被调度运行，删除任务Example_SpinTask1和Example_SpinTask2。

```
#include "los_spinlock.h"
#include "los_task.h"

/* 自旋锁句柄id */
SPIN_LOCK_S g_testSpinlock;
/* 任务ID */
UINT32 g_testTaskId01;
UINT32 g_testTaskId02;
```

```
VOID Example_SpinTask1(VOID)
{
    UINT32 i;
    UINTPTR intSave;

    /* 申请自旋锁 */
    dprintf("task1 try to get spinlock\n");
    LOS_SpinLockSave(&g_testSpinlock， &intSave);
    dprintf("task1 got spinlock\n");
    for(i = 0; i < 5000; i++) {
        asm volatile("nop");
    }

    /* 释放自旋锁 */
    dprintf("task1 release spinlock\n");
    LOS_SpinUnlockRestore(&g_testSpinlock， intSave);

    return;
}

VOID Example_SpinTask2(VOID)
{
    UINT32 i;
    UINTPTR intSave;

    /* 申请自旋锁 */
    dprintf("task2 try to get spinlock\n");
    LOS_SpinLockSave(&g_testSpinlock， &intSave);
    dprintf("task2 got spinlock\n");
    for(i = 0; i < 5000; i++) {
        asm volatile("nop");
    }

    /* 释放自旋锁 */
    dprintf("task2 release spinlock\n");
    LOS_SpinUnlockRestore(&g_testSpinlock， intSave);

    return;
}

UINT32 Example_TaskEntry(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S stTask1;
    TSK_INIT_PARAM_S stTask2;

    /* 初始化自旋锁 */
    LOS_SpinInit(&g_testSpinlock);

    /* 创建任务1 */
    memset(&stTask1， 0， sizeof(TSK_INIT_PARAM_S));
    stTask1.pfnTaskEntry  = (TSK_ENTRY_FUNC)Example_SpinTask1;
    stTask1.pcName        = "SpinTsk1";
    stTask1.uwStackSize   = LOSCFG_TASK_MIN_STACK_SIZE;
    stTask1.usTaskPrio    = 5;
#ifdef LOSCFG_KERNEL_SMP
    /* 绑定任务到CPU0运行 */
    stTask1.usCpuAffiMask = CPUID_TO_AFFI_MASK(0);
#endif
    ret = LOS_TaskCreate(&g_testTaskId01， &stTask1);
    if(ret != LOS_OK) {
        dprintf("task1 create failed .\n");
        return LOS_NOK;
    }

    /* 创建任务2 */
    memset(&stTask2， 0， sizeof(TSK_INIT_PARAM_S));
    stTask2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_SpinTask2;
    stTask2.pcName       = "SpinTsk2";
    stTask2.uwStackSize  = LOSCFG_TASK_MIN_STACK_SIZE;
    stTask2.usTaskPrio   = 5;
```

```
#ifdef LOSCFG_KERNEL_SMP
    /* 绑定任务到CPU1运行 */
    stTask1.usCpuAffiMask = CPUID_TO_AFFI_MASK(1);
#endif
    ret = LOS_TaskCreate(&g_testTaskId02， &stTask2);
    if(ret != LOS_OK) {
        dprintf("task2 create failed .\n");
        return LOS_NOK;
    }

    /* 任务休眠300Ticks */
    LOS_TaskDelay(300);

    /* 删除任务1 */
    ret = LOS_TaskDelete(g_testTaskId01);
    if(ret != LOS_OK) {
        dprintf("task1 delete failed .\n");
        return LOS_NOK;
    }
    /* 删除任务2 */
    ret = LOS_TaskDelete(g_testTaskId02);
    if(ret != LOS_OK) {
        dprintf("task2 delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

运行结果

```
task2 try to get spinlock
task2 got spinlock
task1 try to get spinlock
task2 release spinlock
task1 got spinlock
task1 release spinlock
```

总结

- 自旋锁用于解决CPU核间竞争资源的问题
- 因为自旋锁会让CPU陷入睡眠状态，所以锁的代码不能太长，否则容易导致意外出现，也影响性能。
- 必须由汇编代码实现，因为C语言写不出让CPU进入真正睡眠，核间竞争的代码。

百文说内核 | 抓住主脉络

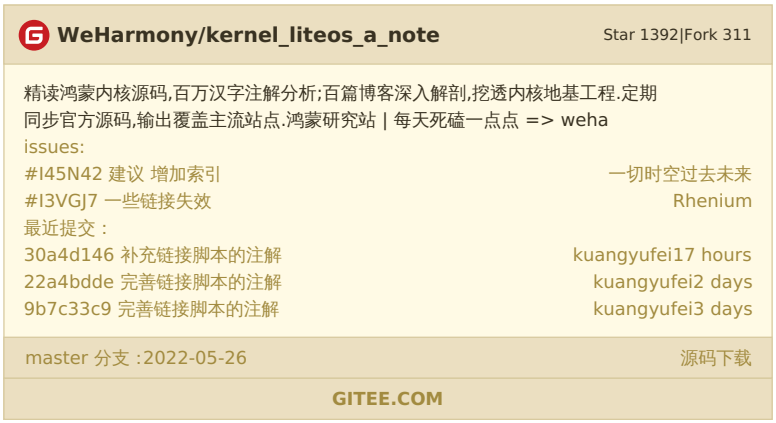
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

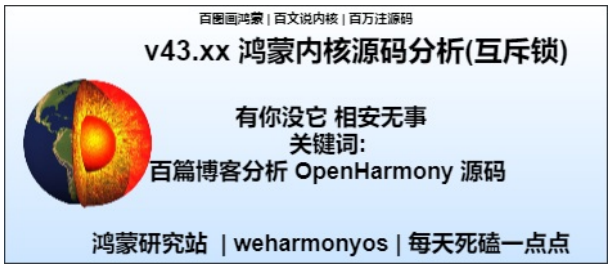


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

43_互斥锁篇

本篇关键词：、、、



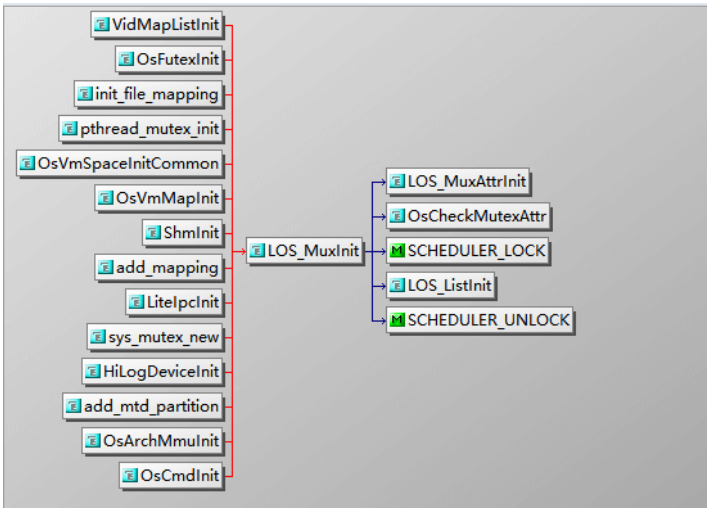
下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

本篇说清楚互斥锁

内核中哪些地方会用到互斥锁？看图:



图中是内核有关模块对互斥锁初始化，有文件，有内存，用消息队列等等，使用面非常的广。其实在给内核源码加注的过程中，会看到大量的自旋锁和互斥锁，它们的存在有序的保证了内核和应用程序的正常运行。是非常基础和重要的功能。

概述

自旋锁 和 **互斥锁** 虽都是锁，但解决的问题不同， 自旋锁解决用于CPU核间共享内存的竞争，而互斥锁解决线程(任务)间共享内存的竞争。

自旋锁的特点是死守共享资源，拿不到锁，CPU选择忙等(busy waiting)，等待其他CPU释放资源。所以共享代码段不能太复杂，否则容易死锁，休克。

互斥锁的特点是拿不到锁往往原任务阻塞，切换到新任务运行。CPU是会一直跑的。这样很容易会想到几个问题:

第一：会出现很多任务在等同一把锁的情况出现，因为切换新任务也可能因要同一把锁而被阻塞，CPU又被调去跑新新任务了。这样就会出现一个等锁的链表。

第二：持有锁的一方再申请同一把锁时还能成功吗？ 答案是可以的，这种锁叫递归锁，是鸿蒙内核默认方式。

第三：当优先级很高的A任务要锁失败，主动让出CPU进入睡眠，而如果持有锁的B任务优先级很低， 迟迟等不到调度不到B任务运行，无法释放锁怎么办？ 答案是会临时调整B任务的优先级，调到A一样高，这样B能很快的被调度到，等B释放锁后其优先级又会被打回原形。所以一个任务的优先级会看情况时高时低。

第四:B任务释放锁之后要主动唤醒等锁的任务链表，使他们能加入就绪队列，等待被调度。调度算法是一视同仁的，它只看优先级。

带着这些问题，进入鸿蒙内核互斥锁的实现代码，本篇代码量较大， 每行代码都一一注解说明。

互斥锁长什么样？

```
enum {
   LOS_MUX_PRIO_NONE = 0， //线程的优先级和调度不会受到互斥锁影响，先来后到，普通排队。
   LOS_MUX_PRIO_INHERIT = 1， //当高优先级的等待低优先级的线程释放锁时，低优先级的线程以高优先级线程的优先级运行。
        //当线程解锁互斥量时，线程的优先级自动被将到它原来的优先级
   LOS_MUX_PRIO_PROTECT = 2 //详见:OsMuxPendOp中的注解，详细说明了LOS_MUX_PRIO_PROTECT的含义
};
enum {
   LOS_MUX_NORMAL = 0， //非递归锁 只有[0。1]两个状态，不做任何特殊的错误检，不进行deadlock detection(死锁检测)
   LOS_MUX_RECURSIVE = 1， //递归锁 允许同一线程在互斥量解锁前对该互斥量进行多次加锁。递归互斥量维护锁的计数，在解锁次数和加锁次数不相同的情况下
   LOS_MUX_ERRORCHECK = 2， //进行错误检查，如果一个线程企图对一个已经锁住的mutex进行relock或对未加锁的unlock，将返回一个错误。
   LOS_MUX_DEFAULT = LOS_MUX_RECURSIVE //鸿蒙系统默认使用递归锁
};
typedef struct { //互斥锁的属性
   UINT8 protocol; //协议
   UINT8 prioceiling; //优先级上限
   UINT8 type;  //类型属性
   UINT8 reserved; //保留字段
} LosMuxAttr;

typedef struct OsMux { //互斥锁结构体
   UINT32 magic;       /**< magic number */ //魔法数字
   LosMuxAttr attr;    /**< Mutex attribute */ //互斥锁属性
   LOS_DL_LIST holdList; /**< The task holding the lock change */ //当有任务拿到本锁时，通过holdList节点把锁挂到该任务的锁链表上
   LOS_DL_LIST muxList; /**< Mutex linked list */ //等这个锁的任务链表，上面挂的都是任务，注意和holdList的区别。
   VOID *owner;        /**< The current thread that is locking a mutex */ //当前拥有这把锁的任务
   UINT16 muxCount;    /**< Times of locking a mutex */ //锁定互斥体的次数，递归锁允许多次
} LosMux;
```

这互斥锁长的明显的比自旋锁丰满多啦，还记得自旋锁的样子吗，就一个变量，单薄到令人心疼。

初始化

```
LITE_OS_SEC_TEXT UINT32 LOS_MuxInit(LosMux *mutex， const LosMuxAttr *attr)
{  //...
   SCHEDULER_LOCK(intSave); //拿到调度自旋锁
   mutex->muxCount = 0;  //锁定互斥量的次数
   mutex->owner = NULL;  //持有该锁的任务
   LOS_ListInit(&mutex->muxList); //初始化等待该锁的任务链表
   mutex->magic = OS_MUX_MAGIC; //固定标识，互斥锁的魔法数字
   SCHEDULER_UNLOCK(intSave); //释放调度自旋锁
   return LOS_OK;
}
```

留意mutex->muxList，这又是一个双向链表， 双向链表是内核最重要的结构体，不仅仅是鸿蒙内核，在linux内核中(list_head)又何尝不是，牢牢的寄生在宿主结构体上。muxList上挂的是未来所有等待这把锁的任务。

三种申请模式

申请互斥锁有三种模式：无阻塞模式、永久阻塞模式、定时阻塞模式。

无阻塞模式：即任务申请互斥锁时，入参timeout等于0。若当前没有任务持有该互斥锁，或者持有该互斥锁的任务和申请该互斥锁的任务为同一个任务，则申请成功，否则立即返回申请失败。

永久阻塞模式：即任务申请互斥锁时，入参timeout等于0xFFFFFFFF。若当前没有任务持有该互斥锁，则申请成功。否则，任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后，直到有其他任务释放该互斥锁，阻塞任务才会重新得以执行。

定时阻塞模式：即任务申请互斥锁时，0<timeout<0xFFFFFFFF。若当前没有任务持有该互斥锁，则申请成功。否则该任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后，超时前如果有其他任务释放该互斥锁，则该任务可成功获取互斥锁继续执行，若超时前未获取到该互斥锁，接口将返回超时错误码。

如果有任务阻塞于该互斥锁，则唤醒被阻塞任务中优先级最高的，该任务进入就绪态，并进行任务调度。 如果没有任务阻塞于该互斥锁，则互斥锁释放成功。

申请互斥锁主函数 OsMuxPendOp

```
//互斥锁的主体函数，由OsMuxlockUnsafe调用，互斥锁模块最重要的几个函数之一
//最坏情况就是拿锁失败，让出CPU，变成阻塞任务，等别的任务释放锁后排到自己了接着执行。
STATIC UINT32 OsMuxPendOp(LosTaskCB *runTask， LosMux *mutex， UINT32 timeout)
{
    UINT32 ret;
    LOS_DL_LIST *node = NULL;
    LosTaskCB *owner = NULL;

    if ((mutex->muxList.pstPrev == NULL) || (mutex->muxList.pstNext == NULL)) {//列表为空时的处理
        /* This is for mutex macro initialization. */
        mutex->muxCount = 0;//锁计数器清0
        mutex->owner = NULL;//锁没有归属任务
        LOS_ListInit(&mutex->muxList);//初始化锁的任务链表，后续申请这把锁任务都会挂上去
    }

    if (mutex->muxCount == 0) {//无task用锁时，肯定能拿到锁了.在里面返回
        mutex->muxCount++;    //互斥锁计数器加1
        mutex->owner = (VOID *)runTask; //当前任务拿到锁
        LOS_ListTailInsert(&runTask->lockList， &mutex->holdList);//持有锁的任务改变了，节点挂到当前task的锁链表
        if ((runTask->priority > mutex->attr.prioceiling) && (mutex->attr.protocol == LOS_MUX_PRIO_PROTECT)) {//看保护协议的做法是怎样的？
            LOS_BitmapSet(&runTask->priBitMap， runTask->priority);//1.priBitMap是记录任务优先级变化的位图，这里把任务当前的优先级记录在priBitMap
            OsTaskPriModify(runTask， mutex->attr.prioceiling);//2.把高优先级的mutex->attr.prioceiling设为当前任务的优先级.
        }//注意任务优先级有32个， 是0最高，31最低!!!这里等于提高了任务的优先级，目的是让其在下次调度中继续提高被选中的概率，从而快速的释放锁.
        return LOS_OK;
    }
//递归锁muxCount>0 如果是递归锁就要处理两种情况 1.runtask持有锁 2.锁被别的任务拿走了
    if (((LosTaskCB *)mutex->owner == runTask) && (mutex->attr.type == LOS_MUX_RECURSIVE)) {//第一种情况 runtask是锁持有方
        mutex->muxCount++; //递归锁计数器加1，递归锁的目的是防止死锁，鸿蒙默认用的就是递归锁(LOS_MUX_DEFAULT = LOS_MUX_RECURSIVE)
        return LOS_OK;  //成功退出
    }
//到了这里说明锁在别的任务那里，当前任务只能被阻塞了。
    if (!timeout) {//参数timeout表示等待多久再来拿锁
        return LOS_EINVAL;//timeout = 0表示不等了，没拿到锁就返回不纠结，返回错误。见于LOS_MuxTrylock
    }
//自己要被阻塞，只能申请调度，让出CPU core 让别的任务上
    if (!OsPreemptableInSched()) {//不能申请调度 (不能调度的原因是因为没有持有调度任务自旋锁)
        return LOS_EDEADLK;//返回错误，自旋锁被别的CPU core 持有
    }

    OsMuxBitmapSet(mutex， runTask， (LosTaskCB *)mutex->owner);//设置锁位图，尽可能的提高锁持有任务的优先级

    owner = (LosTaskCB *)mutex->owner; //记录持有锁的任务
    runTask->taskMux = (VOID *)mutex; //记下当前任务在等待这把锁
    node = OsMuxPendFindPos(runTask， mutex);//在等锁链表中找到一个优先级比当前任务更低的任务
    ret = OsTaskWait(node， timeout， TRUE);//task陷入等待状态 TRUE代表需要调度
    if (ret == LOS_ERRNO_TSK_TIMEOUT) {//这行代码虽和OsTaskWait挨在一起，但要过很久才会执行到，因为在OsTaskWait中CPU切换了任务上下文
        runTask->taskMux = NULL;// 所以重新回到这里时可能已经超时了
        ret = LOS_ETIMEDOUT;//返回超时
    }

    if (timeout != LOS_WAIT_FOREVER) {//不是永远等待的情况
        OsMuxBitmapRestore(mutex， runTask， owner);//恢复锁的位图
    }
```

```
    return ret;
}
```

释放锁的主体函数 OsMuxPostOp

```
//是否有其他任务持有互斥锁而处于阻塞状，如果是就要唤醒它，注意唤醒一个任务的操作是由别的任务完成的
//OsMuxPostOp只由OsMuxUnlockUnsafe，参数任务归还锁了，自然就会遇到锁要给谁用的问题， 因为很多任务在申请锁，由OsMuxPostOp来回答这个问题
STATIC UINT32 OsMuxPostOp(LosTaskCB *taskCB， LosMux *mutex， BOOL *needSched)
{
    LosTaskCB *resumedTask = NULL;

    if (LOS_ListEmpty(&mutex->muxList)) {//如果互斥锁列表为空
        LOS_ListDelete(&mutex->holdList);//把持有互斥锁的节点摘掉
        mutex->owner = NULL;
        return LOS_OK;
    }

    resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(mutex->muxList)));//拿到等待互斥锁链表的第一个任务实体，接下来要唤醒任务
    if (mutex->attr.protocol == LOS_MUX_PRIO_INHERIT) {//互斥锁属性协议是继承会怎么操作？
        if (resumedTask->priority > taskCB->priority) {//拿到锁的任务优先级低于参数任务优先级
            if (LOS_HighBitGet(taskCB->priBitMap) != resumedTask->priority) {//参数任务bitmap中最低的优先级不等于等待锁的任务优先级
                LOS_BitmapClr(&taskCB->priBitMap， resumedTask->priority);//把等待任务锁的任务的优先级记录在参数任务的bitmap中
            }
        } else if (taskCB->priBitMap != 0) {//如果bitmap不等于0说明参数任务至少有任务调度的优先级
            OsMuxPostOpSub(taskCB， mutex);//
        }
    }
    mutex->muxCount = 1;//互斥锁数量为1
    mutex->owner = (VOID *)resumedTask;//互斥锁的持有人换了
    resumedTask->taskMux = NULL;//resumedTask不再等锁了
    LOS_ListDelete(&mutex->holdList);//自然要从等锁链表中把自己摘出去
    LOS_ListTailInsert(&resumedTask->lockList， &mutex->holdList);//把锁挂到恢复任务的锁链表上，lockList是任务持有的所有锁记录
    OsTaskWake(resumedTask);//resumedTask有了锁就唤醒它，因为当初在没有拿到锁时处于了pend状态
    if (needSched != NULL) {//如果不为空
        *needSched = TRUE;//就走起再次调度流程
    }

    return LOS_OK;
}
```

编程实例

本实例实现如下流程。

- 任务Example_TaskEntry创建一个互斥锁，锁任务调度，创建两个任务Example_MutexTask1、Example_MutexTask2.Example_MutexTask2优先级高于Example_MutexTask1，解锁任务调度，然后Example_TaskEntry任务休眠300Tick。
- Example_MutexTask2被调度，以永久阻塞模式申请互斥锁，并成功获取到该互斥锁，然后任务休眠100Tick，Example_MutexTask2挂起，Example_MutexTask1被唤醒。
- Example_MutexTask1以定时阻塞模式申请互斥锁，等待时间为10Tick，因互斥锁仍被Example_MutexTask2持有，Example_MutexTask1挂起。10Tick超时时间到达后，Example_MutexTask1被唤醒，以永久阻塞模式申请互斥锁，因互斥锁仍被Example_MutexTask2持有，Example_MutexTask1挂起。
- 100Tick休眠时间到达后，Example_MutexTask2被唤醒， 释放互斥锁，唤醒Example_MutexTask1。Example_MutexTask1成功获取到互斥锁后，释放锁。
- 300Tick休眠时间到达后，任务Example_TaskEntry被调度运行，删除互斥锁，删除两个任务。

```
/* 互斥锁句柄id */
UINT32 g_testMux;
/* 任务ID */
UINT32 g_testTaskId01;
UINT32 g_testTaskId02;

VOID Example_MutexTask1(VOID)
{
    UINT32 ret;

    printf("task1 try to get  mutex， wait 10 ticks.\n");
```

```
    /* 申请互斥锁 */
    ret = LOS_MuxPend(g_testMux， 10);

    if (ret == LOS_OK) {
        printf("task1 get mutex g_testMux.\n");
        /* 释放互斥锁 */
        LOS_MuxPost(g_testMux);
        return;
    } else if (ret == LOS_ERRNO_MUX_TIMEOUT ) {
            printf("task1 timeout and try to get mutex， wait forever.\n");
            /* 申请互斥锁 */
            ret = LOS_MuxPend(g_testMux， LOS_WAIT_FOREVER);
            if (ret == LOS_OK) {
                printf("task1 wait forever， get mutex g_testMux.\n");
                /* 释放互斥锁 */
                LOS_MuxPost(g_testMux);
                return;
            }
    }
    return;
}

VOID Example_MutexTask2(VOID)
{
    printf("task2 try to get  mutex， wait forever.\n");
    /* 申请互斥锁 */
    (VOID)LOS_MuxPend(g_testMux， LOS_WAIT_FOREVER);

    printf("task2 get mutex g_testMux and suspend 100 ticks.\n");

    /* 任务休眠100Ticks */
    LOS_TaskDelay(100);

    printf("task2 resumed and post the g_testMux\n");
    /* 释放互斥锁 */
    LOS_MuxPost(g_testMux);
    return;
}

UINT32 Example_TaskEntry(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S task1;
    TSK_INIT_PARAM_S task2;

    /* 创建互斥锁 */
    LOS_MuxCreate(&g_testMux);

    /* 锁任务调度 */
    LOS_TaskLock();

    /* 创建任务1 */
    memset(&task1， 0， sizeof(TSK_INIT_PARAM_S));
    task1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_MutexTask1;
    task1.pcName       = "MutexTsk1";
    task1.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    task1.usTaskPrio   = 5;
    ret = LOS_TaskCreate(&g_testTaskId01， &task1);
    if (ret != LOS_OK) {
        printf("task1 create failed.\n");
        return LOS_NOK;
    }

    /* 创建任务2 */
    memset(&task2， 0， sizeof(TSK_INIT_PARAM_S));
    task2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_MutexTask2;
    task2.pcName       = "MutexTsk2";
    task2.uwStackSize  = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    task2.usTaskPrio   = 4;
    ret = LOS_TaskCreate(&g_testTaskId02， &task2);
    if (ret != LOS_OK) {
```

```
        printf("task2 create failed.\n");
        return LOS_NOK;
    }

    /* 解锁任务调度 */
    LOS_TaskUnlock();
    /* 休眠300Ticks */
    LOS_TaskDelay(300);

    /* 删除互斥锁 */
    LOS_MuxDelete(g_testMux);

    /* 删除任务1 */
    ret = LOS_TaskDelete(g_testTaskId01);
    if (ret != LOS_OK) {
        printf("task1 delete failed .\n");
        return LOS_NOK;
    }
    /* 删除任务2 */
    ret = LOS_TaskDelete(g_testTaskId02);
    if (ret != LOS_OK) {
        printf("task2 delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

结果验证

```
task2 try to get  mutex ,  wait forever.
task2 get mutex g_testMux and suspend 100 ticks.
task1 try to get  mutex ,  wait 10 ticks.
task1 timeout and try to get mutex ,  wait forever.
task2 resumed and post the g_testMux
task1 wait forever , get mutex g_testMux.
```

总结

1.互斥锁解决的是任务间竞争共享内存的问题.

2.申请锁失败的任务会进入睡眠OsTaskWait，内核会比较持有锁的任务和申请锁任务的优先级，把持有锁的任务优先级调到尽可能的高，以便更快的被调度执行，早日释放锁.

3.释放锁的任务会在等锁链表中找一个高优先级任务，通过OsTaskWake唤醒它，并向调度算法申请调度.但要注意，调度算法只是按优先级来调度，并不保证调度后的任务一定是要唤醒的任务.

4.互斥锁篇关键是看懂 OsMuxPendOp 和 OsMuxPostOp 两个函数。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

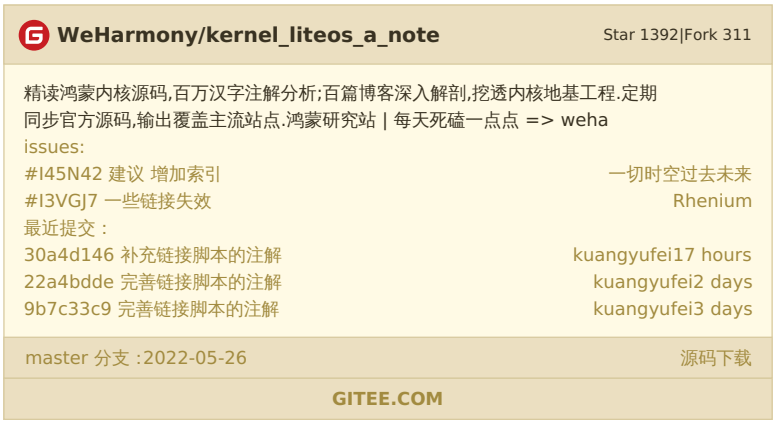
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

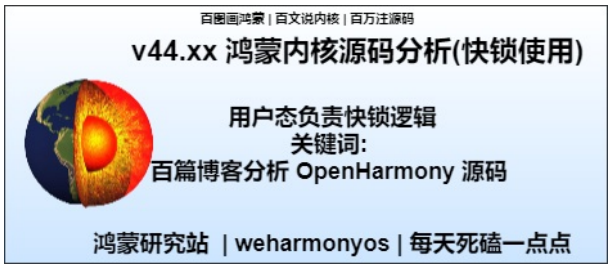


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

44_快锁使用篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

快锁上下篇

鸿蒙内核实现了`Futex`，系列篇将用两篇来介绍快锁，主要两个原因:

- 网上介绍`Futex`的文章很少，全面深入内核介绍的就更少，所以来一次详细整理和挖透。
- 涉及用户态和内核态打配合，共同作用，既要说用户态的使用又要说清楚内核态的实现。 本篇为上篇，用户态下如何使用`Futex`，并借助一个`demo`来说清楚整个过程。

基本概念

`Futex`(`Fast userspace mutex`，用户态快速互斥锁)，系列篇简称 **快锁** ，是一个在`Linux`上实现锁定和构建高级抽象锁如信号量和`POSIX`互斥的基本工具，它第一次出现在`linux`内核开发的`2.5.7`版；其语义在`2.5.40`固定下来，然后在`2.6.x`系列稳定版内核中出现，是内核提供的一种系统调用能力。通常作为基础组件与用户态的相关锁逻辑结合组成用户态锁，是一种用户态与内核态共同作用的锁，其用户态部分负责锁逻辑，内核态部分负责锁调度。

当用户态线程请求锁时，先在用户态进行锁状态的判断维护，若此时不产生锁的竞争，则直接在用户态进行上锁返回；反之，则需要进行线程的挂起操作，通过`Futex`系统调用请求内核介入来挂起线程，并维护阻塞队列。

当用户态线程释放锁时，先在用户态进行锁状态的判断维护，若此时没有其他线程被该锁阻塞，则直接在用户态进行解锁返回；反之，则需要进行阻塞线程的唤醒操作，通过`Futex`系统调用请求内核介入来唤醒阻塞队列中的线程。

存在意义

- **互斥锁**(`mutex`)是必须进入内核态才知道锁可不可用，没人跟你争就拿走锁回到用户态，有人争就得干等 (包括 有限时间等和无限等待两种，都需让出`CPU`执行权) 或者放弃本次申请回到用户态继续执行。那为何**互斥锁**一定要陷入内核态检查呢? 互斥锁(`mutex`)本质是竞争内核空间的某个全局变量(`LosMux`结构体)。应用程序也有全局变量，但其作用域只在自己的用户空间中有效，属于内部资源，有竞争也是应用程序自己内部解决。而应用之间的资源竞争(即内核资源)就需要内核程序来解决，内核空间只有一个，内核的全局变量当然要由内核来管理。应用程序想用内核资源就必须经过系统调用陷入内核态，由内核程序接管`CPU`，所谓接管本质是要改变程序状态寄存器，`CPU`将从用户态栈切换至内核态栈运行，执行完成后又要切回用户态栈中继续执行，如此一来栈间上下文的切换就存在系统性能的损耗。没看明白的请前往系列篇 **(互斥锁篇)** 翻看。

- **快锁** 解决思路是能否在用户态下就知道锁可不可用，因为竞争并不是时刻出现，跑到内核态一看其实往往没人给你争，白跑一趟来回太浪费性能。那问题来了，用户态下如何知道锁可不可用呢？ 因为不陷入内核态就访问不到内核的全局变量。而自己私有空间的变量对别的进程又失效不能用。越深入研究内核越有一种这样的感觉，内核的实现可以像数学一样推导出来，非常有意思。数学其实是基于几个常识公理推导出了整个数学体系，因为不如此逻辑就无法自洽。如果对内核有一定程度的了解，这里自然能推导出可以借助 **共享内存** 来实现！

使用过程

看个linux futex官方`demo`详细说明下用户态下使用`Futex`的整个过程，代码不多，但涉及内核的知识点很多，通过它可以检验出内核基本功扎实程度。

```
//futex_demo.c
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <stdatomic.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <sys/time.h>
#define errExit(msg)    do { perror(msg); exit(EXIT_FAILURE); \
                        } while (0)
static uint32_t *futex1, *futex2, *iaddr;
/// 快速系统调用
static int futex(uint32_t *uaddr, int futex_op, uint32_t val,
       const struct timespec *timeout, uint32_t *uaddr2, uint32_t val3)
{
   return syscall(SYS_futex, uaddr, futex_op, val,
                  timeout, uaddr2, val3);
}
/// 申请快锁
static void fwait(uint32_t *futexp)
{
   long s;
   while (1) {
      const uint32_t one = 1;
      if (atomic_compare_exchange_strong(futexp, &one, 0))
         break; //申请快锁成功
      //申请快锁失败,需等待
      s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
      if (s == -1 && errno != EAGAIN)
         errExit("futex-FUTEX_WAIT");
   }
}
/// 释放快锁
static void fpost(uint32_t *futexp)
{
   long s;
   const uint32_t zero = 0;
   if (atomic_compare_exchange_strong(futexp, &zero, 1)) {//释放快锁成功
      s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);//唤醒等锁 进程/线程
      if (s  == -1)
         errExit("futex-FUTEX_WAKE");
   }
}
/// 父子进程竞争快锁
int main(int argc, char *argv[])
{
   pid_t childPid;
   int nloops;
   setbuf(stdout, NULL);
   nloops = (argc > 1) ? atoi(argv[1]) : 3;
   iaddr = mmap(NULL, sizeof(*iaddr) * 2, PROT_READ | PROT_WRITE,
            MAP_ANONYMOUS | MAP_SHARED, -1, 0);//创建可读可写匿名共享内存
   if (iaddr == MAP_FAILED)
      errExit("mmap");
```

```
    futex1 = &iaddr[0]; //绑定锁一地址
    futex2 = &iaddr[1]; //绑定锁二地址
    *futex1 = 0; // 锁一不可申请
    *futex2 = 1; // 锁二可申请
    childPid = fork();
    if (childPid == -1)
        errExit("fork");
    if (childPid == 0) {//子进程返回
        for (int j = 0; j < nloops; j++) {
            fwait(futex1);//申请锁一
            printf("子进程  (%jd) %d\n", (intmax_t) getpid(), j);
            fpost(futex2);//释放锁二
        }
        exit(EXIT_SUCCESS);
    }
    // 父进程返回执行
    for (int j = 0; j < nloops; j++) {
        fwait(futex2);//申请锁二
        printf("父进程 (%jd) %d\n", (intmax_t) getpid(), j);
        fpost(futex1);//释放锁一
    }
    wait(NULL);
    exit(EXIT_SUCCESS);
}
```

代码在 wsl2 上编译运行结果如下:

```
root@DESKTOP-5PBPDNG:/home/turing# gcc ./futex_demo.c -o futex_demo
root@DESKTOP-5PBPDNG:/home/turing# ./futex_demo
父进程 (283) 0
子进程 (284) 0
父进程 (283) 1
子进程 (284) 1
父进程 (283) 2
子进程 (284) 2
```

解读

- 通过系统调用 mmap 创建一个可读可写的共享内存 iaddr[2] 整型数组，完成两个 futex 锁的初始化。内核会在内存分配一个共享线性区(MAP_ANONYMOUS | MAP_SHARED)，该线性区可读可写(PROT_READ | PROT_WRITE)

  ```
  futex1 = &iaddr[0]; //绑定锁一地址
  futex2 = &iaddr[1]; //绑定锁二地址
  *futex1 = 0; // 锁一不可申请
  *futex2 = 1; // 锁二可申请
  ```

  如此 futex1 和 futex2 有初始值并都是共享变量，想详细了解 mmap 内核实现的可查看系列篇 **（线性区篇）** 和 **（共享内存篇）** 有详细介绍。
- childPid = fork(); 创建了一个子进程，fork会拷贝父进程线性区的映射给子进程，导致的结果就是父进程的共享线性区到子进程这也是共享线性区，映射的都是相同的物理地址。对 fork 不熟悉的请前往翻看，系列篇 **（fork篇）| 一次调用，两次返回** 专门说它。
- fwait (申请锁)与 fpost (释放锁)成对出现，单独看下申请锁过程

  ```
  /// 申请快锁
  static void fwait(uint32_t *futexp)
  {
      long s;
      while (1) {
          const uint32_t one = 1;
          if (atomic_compare_exchange_strong(futexp, &one, 0))
              break; //申请快锁成功
          //申请快锁失败,需等待
          s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
          if (s == -1 && errno != EAGAIN)
              errExit("futex-FUTEX_WAIT");
      }
  }
  ```

  死循环的break条件是 atomic_compare_exchange_strong 为真，这是个原子比较操作，此处必须这么用，至于为什么请前往翻看系列篇 **（原子**

操作篇）| 谁在为完整性保驾护航 ，注意它是理解 Futex 的关键所在，它的含义是

```
在头文件<stdatomic.h>中定义
_Bool atomic_compare_exchange_strong（volatile A * obj，C * expected，C desired）；
```

将所指向的值obj与所指向的值进行原子比较 expected ，如果相等，则用前者替换前者 desired （执行读取 - 修改 - 写入操作）。否则，加载实际值所指向的 obj 进入 *expected （进行负载操作）。 什么意思？来个直白的解释：

- 如果 futexp == 1 则 atomic_compare_exchange_strong 返回真，同时将 futexp 的值变成 0 ，1代表可以持有锁，一旦持有立即变0，别人就拿不到了。所以此处甚秒。而且这发生在用户态。
- 如果 futexp == 0 atomic_compare_exchange_strong 返回假，没有拿到锁，就需要陷入内核态去挂起任务等待锁的释放

```
futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0) //执行一个等锁的系统调用
```

参数四为 NULL 代表不在内核态停留直接返回用户态，后续将在内核态部分详细说明。

- childPid == 0 是子进程的返回。不断地申请 futex1 释放 futex2

```
if (childPid == 0) {//子进程返回
    for (int j = 0; j < nloops; j++) {
        fwait(futex1);
        printf("子进程  (%jd) %d\n", (intmax_t) getpid(), j);
        fpost(futex2);
    }
    exit(EXIT_SUCCESS);
}
```

- 最后的父进程的返回，不断地申请 futex2 释放 futex1

```
// 父进程返回执行
for (int j = 0; j < nloops; j++) {
    fwait(futex2);
    printf("父进程 (%jd) %d\n", (intmax_t) getpid(), j);
    fpost(futex1);
}
wait(NULL);
exit(EXIT_SUCCESS);
```

- 两把锁的初值为 *futex1 = 0; *futex2 = 1; ，父进程在 fwait(futex2) 所以父进程的 printf 将先执行， *futex2 = 0; 锁二变成不可申请，打印完成后释放 fpost(futex1) 使其结果为 *futex1 = 1; 表示锁一可以申请了，而子进程在等 fwait(futex1) ，交替下来执行的结果为

```
父进程 (283) 0
子进程 (284) 0
父进程 (283) 1
子进程 (284) 1
父进程 (283) 2
子进程 (284) 2
```

几个问题

以上是个简单的例子，只发生在两个进程抢一把锁的情况下，如果再多几个进程抢一把锁时情况就变复杂多了。 例如会遇到以下情况:

- 鸿蒙内核进程池默认上限是 64 个，除去两个内核进程外，剩下的都归属用户进程，理论上用户进程可以创建很多快锁,这些快锁可以用于进程间(共享快锁)也可以用于线程间(私有快锁)，在快锁的生命周期中该如何保存？
- 无锁时，前面已经有进程在申请锁时，如何处理好新等锁进程和旧等锁进程的关系？
- 释放锁时，需要唤醒已经在等锁的进程，唤醒的顺序由什么条件决定？

这些工作在用户态下肯定没办法完成，需要内核处理，请查看 **(快锁实现篇) | 内核态下的快锁Futex(下)**，详细解构其实现过程。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

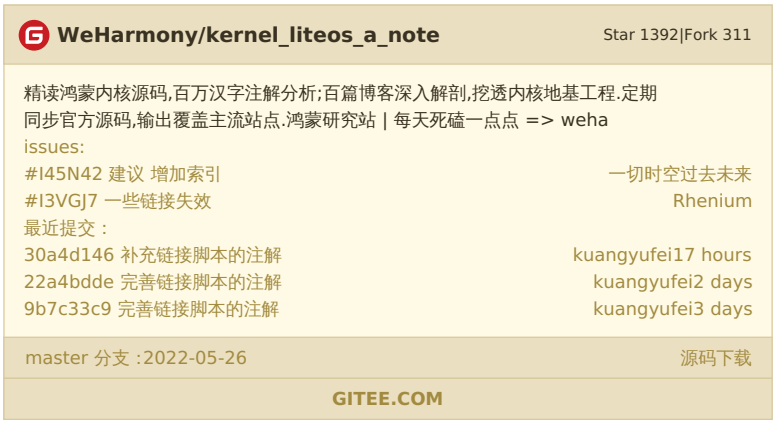
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

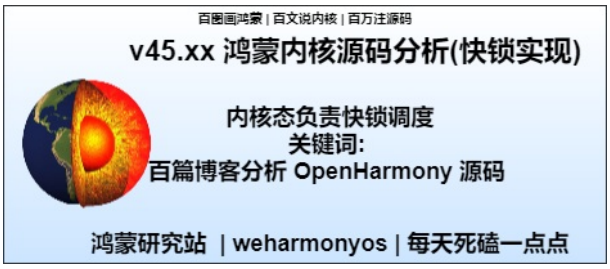


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

45_快锁实现篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

本篇为快锁下篇，说清楚快锁在内核态的实现，解答以下问题，它们在上篇的末尾被提出来。

- 鸿蒙内核进程池默认上限是`64`个，除去两个内核进程外，剩下的都归属用户进程，理论上用户进程可以创建很多快锁，这些快锁可以用于进程间(共享快锁)也可以用于线程间(私有快锁)，在快锁的生命周期中该如何保存？
- 无锁时，前面已经有进程在申请锁时，如何处理好新等锁进程和旧等锁进程的关系？
- 释放锁时，需要唤醒已经在等锁的进程，唤醒的顺序由什么条件决定？

系列篇多次提过，线程在内核层面叫任务，在内核任务比进程重要得多，调度也好，竞争也罢，都是围绕任务展开的。竞争快锁是任务间的竞争，自然会和任务(`task`)有紧密的联系，其在内核的表达也出现在了任务表达之中。

```
typedef struct { // 任务控制块
    ...
    LOS_DL_LIST     pendList;          /**< Task pend node | 如果任务阻塞时就通过它挂到各种阻塞情况的链表上,比如OsTaskWait时 */
    ...
    FutexNode       futex;  ///< 指明任务在等待哪把快锁，一次只等一锁，锁和任务的关系是(1:N)关系
} LosTaskCB;
```

对 **任务** 不清楚的请翻看系列相关篇，一定要搞懂，它是内核最重要的概念，甚至没有之一，搞不懂任务就一定搞不懂内核整体的运行机制。

快锁节点 | 内核表达

`FutexNode`(快锁节点) 是快锁模块核心结构体，熟悉这块源码的钥匙。

```
typedef struct {
    UINTPTR      key;          /* private:uvaddr | 私有锁，用虚拟地址         shared:paddr | 共享锁，用物理地址 */
    UINT32       index;        /* hash bucket index | 哈希桶索引 OsFutexKeyToIndex */
    UINT32       pid;          /* private:process id   shared:OS_INVALID(-1) | 私有锁:进程ID      ，共享锁为 -1 */
    LOS_DL_LIST  pendList;     /* point to pendList in TCB struct | 指向 TCB 结构中的 pendList, 通过它找到任务*/
    LOS_DL_LIST  queueList;    /* thread list blocked by this lock | 挂等待这把锁的任务，其实这里挂到是FutexNode.queueList，通过 queueList 可以找到 p
    LOS_DL_LIST  futexList;    /* point to the next FutexNode | 下一把快锁节点*/
```

```
} FutexNode;
```

解读

- 首先要明白 **快锁** 和 **快锁节点** 的区别，否则看内核代码一定会懵圈，内核并没有**快锁**这个结构体， key 就是快锁，它们的关系是 1:N 的关系，快锁分成了 **私有锁** 和 **共享锁** 两种类型。用 key 表示唯一性。共享锁用物理地址，私有锁用虚拟地址。为什么要这么做呢？
  - 私有锁的意思是进程私有，作用于同一个进程的不同任务间， 因为任务是共享进程空间的， 所以可以用虚拟地址来表示进程内的唯一性 。但两个不同的进程会出现两个虚拟地址一样的快锁。
  - 共享锁的意思是进程共享，作用于不同进程的不同任务间，因为不同的进程都会有相同的虚拟地址范围， 所以不能用虚拟地址来表示唯一性 ，只能用物理地址。虚拟地址：物理地址 = N: 1 ，不清楚的请查看系列篇之内存映射相关篇。
- index 内核使用哈希桶来检索快锁 ， index 和 key 的关系通过哈希算法(FNV-1a)来映射。注意会有同一个哈希桶中两个 key 一样的锁，虽然它会以极低概率出现。快锁的内核实现代码部分，个人觉得可以优化的空间很大，应好好测试下这块 ，说不定会有意想不到的 bug 。
- pid 指快锁节点进程归属，作用于私有锁。
- pendList 指向 LosTaskCB.pendList ， 通过它去唤醒和挂起任务，但并没有在源码中看到指向动作，如有看到的请告知。
- queueList 具有相同 key 值的节点被 queue_list 串联起来表示被同一把锁阻塞的任务队列，意思就是 queueList 上面挂的都是等值为相同 key 的快锁，并按快锁背后的任务优先级排好序。任务优先级高的可以先获取快锁使用权。
- futexList 指向下一把快锁, 虽然挂的也是 FutexNode ，但是意义不一样！是指 queueList 链表上的首个快锁节点，即不同 key 的快锁。能理解吗？好吧 ，我承认这里面有点绕 。

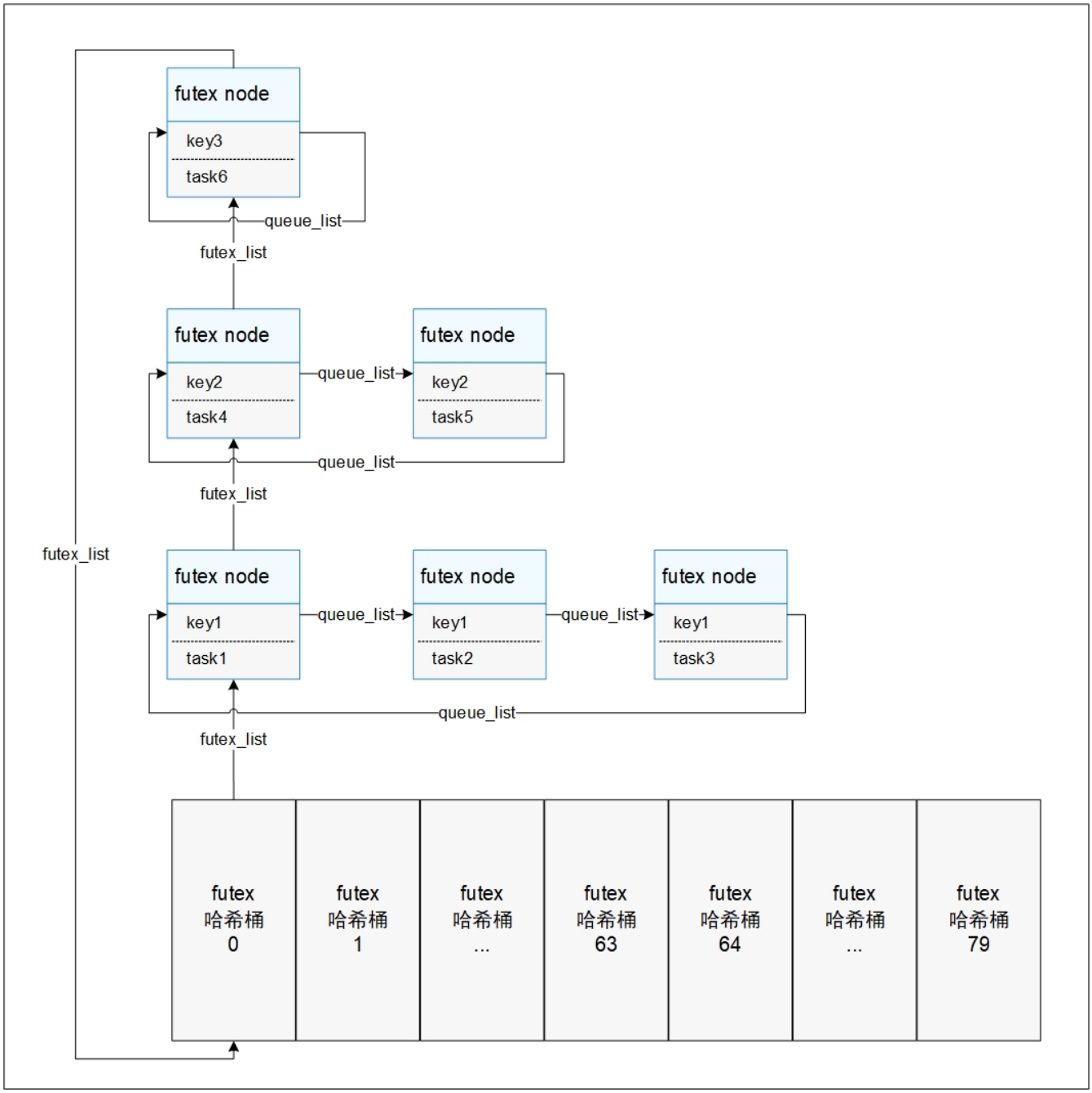
哈希桶 | 管理快锁

当用户态产生锁的竞争或释放需要进行相关线程的调度操作时，会触发 Futex 系统调用进入内核，此时会将用户态锁的地址传入内核，并在内核的 Futex 中以锁地址来区分用户态的每一把锁，因为用户态可用虚拟地址空间为 1GiB ，为了便于查找、管理，内核 Futex 采用哈希桶来存放用户态传入的锁。

哈希桶共有 80 个， 0~63 号桶用于存放私有锁（以虚拟地址进行哈希）， 64~79 号桶用于存放共享锁（以物理地址进行哈希），所有相同的 key 都掉进了同一个桶里。私有/共享属性通过用户态锁的初始化以及 Futex 系统调用入参确定。

```
#define FUTEX_INDEX_PRIVATE_MAX     64 ///< 0~63号桶用于存放私有锁（以虚拟地址进行哈希）,同一进程不同线程共享futex变量，表明变量在进程地址空间
///< 它告诉内核，这个futex是进程专有的，不可以与其他进程共享。它仅仅用作同一进程的线程间同步。
#define FUTEX_INDEX_SHARED_MAX      16 ///< 64~79号桶用于存放共享锁（以物理地址进行哈希）,不同进程间通过文件共享futex变量，表明该变量在文件中
#define FUTEX_INDEX_MAX             (FUTEX_INDEX_PRIVATE_MAX + FUTEX_INDEX_SHARED_MAX) ///< 80个哈希桶
#define FUTEX_INDEX_SHARED_POS      FUTEX_INDEX_PRIVATE_MAX ///< 共享锁开始位置
FutexHash g_futexHash[FUTEX_INDEX_MAX];///< 默认80个哈希桶

typedef struct {
    LosMux      listLock;///< 内核操作lockList的互斥锁
    LOS_DL_LIST lockList;///< 用于挂载 FutexNode (Fast userspace mutex，用户态快速互斥锁)
} FutexHash;
```

结构体很简单，没什么可说的，一把互斥锁确保一个链表的操作。 下图来源于官方文档，基本能准确的描述管理方式，暂且使用此图(后续可能重画)，有了这张图理解上面 FutexNode 会更轻松



任务调度

- 无锁时就需要将当前任务挂起，可详细跟踪函数 `OsFutexWaitTask` ，无非就是根据任务的优先级调整 `queueList` `futexList` `queueList` 这些链表上的位置

```
/// 将当前任务挂入等待链表中
STATIC INT32 OsFutexWaitTask(const UINT32 *userVaddr, const UINT32 flags, const UINT32 val, const UINT32 timeOut)
{
    INT32 futexRet;
    UINT32 intSave, lockVal;
    LosTaskCB *taskCB = NULL;
    FutexNode *node = NULL;
    UINTPTR futexKey = OsFutexFlagsToKey(userVaddr, flags);//通过地址和flags 找到 key
    UINT32 index = OsFutexKeyToIndex(futexKey, flags);//通过key找到哈希桶
    FutexHash *hashNode = &g_futexHash[index];

    if (OsFutexLock(&hashNode->listLock)) {//操作快锁节点链表前先上互斥锁
```

```
        return LOS_EINVAL;
    }
    //userVaddr必须是用户空间虚拟地址
    if (LOS_ArchCopyFromUser(&lockVal, userVaddr, sizeof(UINT32))) {//将值拷贝到内核空间
        PRINT_ERR("Futex wait param check failed! copy from user failed!\n");
        futexRet = LOS_EINVAL;
        goto EXIT_ERR;
    }

    if (lockVal != val) {//对参数内部逻辑检查
        futexRet = LOS_EBADF;
        goto EXIT_ERR;
    }
    //注意第二个参数 FutexNode *node = NULL
    if (OsFutexInsertTaskToHash(&taskCB, &node, futexKey, flags)) {// node = taskCB->futex
        futexRet = LOS_NOK;
        goto EXIT_ERR;
    }

    SCHEDULER_LOCK(intSave);
    OsTaskWaitSetPendMask(OS_TASK_WAIT_FUTEX, futexKey, timeOut);
    OsSchedTaskWait(&(node->pendList), timeOut, FALSE);
    OsSchedLock();
    LOS_SpinUnlock(&g_taskSpin);

    futexRet = OsFutexUnlock(&hashNode->listLock);
    if (futexRet) {
        OsSchedUnlock();
        LOS_IntRestore(intSave);
        goto EXIT_UNLOCK_ERR;
    }

    LOS_SpinLock(&g_taskSpin);
    OsSchedUnlock();

    /*
    * it will immediately do the scheduling, so there's no need to release the
    * task spinlock. when this task's been rescheduled, it will be holding the spinlock.
    */
    OsSchedResched();

    if (taskCB->taskStatus & OS_TASK_STATUS_TIMEOUT) {
        taskCB->taskStatus &= ~OS_TASK_STATUS_TIMEOUT;
        SCHEDULER_UNLOCK(intSave);
        return OsFutexDeleteTimeoutTaskNode(hashNode, node);
    }

    SCHEDULER_UNLOCK(intSave);
    return LOS_OK;

EXIT_ERR:
    (VOID)OsFutexUnlock(&hashNode->listLock);
EXIT_UNLOCK_ERR:
    return futexRet;
}
```

针对本篇开始的问题二，等锁新任务来临后由任务优先级决定在 queueList 中的位置, OsFutexInsertTasktoPendList

```
///< 将快锁挂到任务的阻塞链表上
STATIC INT32 OsFutexInsertTasktoPendList(FutexNode **firstNode, FutexNode *node, const LosTaskCB *run)
{
    LosTaskCB *taskHead = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&((*firstNode)->pendList)));//获取阻塞链表首个任务
    LOS_DL_LIST *queueList = &((*firstNode)->queueList);
    FutexNode *tailNode = NULL;
    LosTaskCB *taskTail = NULL;

    if (run->priority < taskHead->priority) {//任务的优先级比较
        /* The one with the highest priority is inserted at the top of the queue */
        LOS_ListTailInsert(queueList, &(node->queueList));//查到queueList的尾部
        OsFutexReplaceQueueListHeadNode(*firstNode, node);//同时交换futexList链表上的位置
```

```
        *firstNode = node;
        return LOS_OK;
    }
    //如果等锁链表上没有任务或者当前任务大于链表首个任务
    if (LOS_ListEmpty(queueList) && (run->priority >= taskHead->priority)) {
        /* Insert the next position in the queue with equal priority */
        LOS_ListHeadInsert(queueList, &(node->queueList));//从头部插入当前任务,当前任务是要被挂起的
        return LOS_OK;
    }

    tailNode = OS_FUTEX_FROM_QUEUELIST(LOS_DL_LIST_LAST(queueList));//获取尾部节点
    taskTail = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(tailNode->pendList)));//获取阻塞任务的最后一个
    if ((run->priority >= taskTail->priority) ||//当前任务优先级比最后一个更高,或者 ... 没看懂, 为啥要这样 ? @notethinking
        ((run->priority - taskHead->priority) > (taskTail->priority - run->priority))) {//跟最后一个比较优先级
        return OsFutexInsertFindFormBackToFront(queueList, run, node);//从后往前插入
    }

    return OsFutexInsertFindFromFrontToBack(queueList, run, node);//否则从前往后插入
}
```

- 释放锁时就需要将 queueList 上挂起任务唤醒，可详细跟踪函数 OsFutexWaitTask ，如果没有任务再等锁了就 DeleteKey

```
STATIC INT32 OsFutexWakeTask(UINTPTR futexKey, UINT32 flags, INT32 wakeNumber, FutexNode **newHeadNode, BOOL *wakeAny)
{
    UINT32 intSave;
    FutexNode *node = NULL;
    FutexNode *headNode = NULL;
    UINT32 index = OsFutexKeyToIndex(futexKey, flags);
    FutexHash *hashNode = &g_futexHash[index];
    FutexNode tempNode = { //先组成一个临时快锁节点,目的是为了找到哈希桶中是否有这个节点
        .key = futexKey,
        .index = index,
        .pid = (flags & FUTEX_PRIVATE) ? LOS_GetCurrProcessID() : OS_INVALID,
    };

    node = OsFindFutexNode(&tempNode);//找快锁节点
    if (node == NULL) {
        return LOS_EBADF;
    }

    headNode = node;

    SCHEDULER_LOCK(intSave);
    OsFutexCheckAndWakePendTask(headNode, wakeNumber, hashNode, newHeadNode, wakeAny);//再找到等这把锁的唤醒指向数量的任务
    if ((*newHeadNode) != NULL) {
        OsFutexReplaceQueueListHeadNode(headNode, *newHeadNode);
        OsFutexDeinitFutexNode(headNode);
    } else if (headNode->index < FUTEX_INDEX_MAX) {
        OsFutexDeleteKeyFromFutexList(headNode);
        OsFutexDeinitFutexNode(headNode);
    }
    SCHEDULER_UNLOCK(intSave);

    return LOS_OK;
}
```

百文说内核 | 抓住主脉络

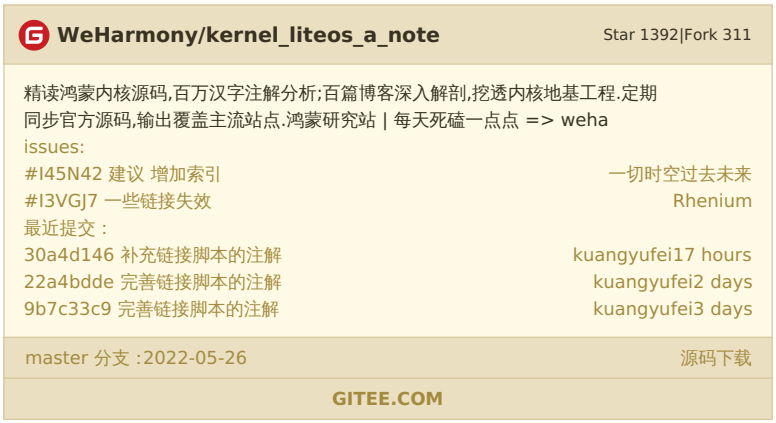
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

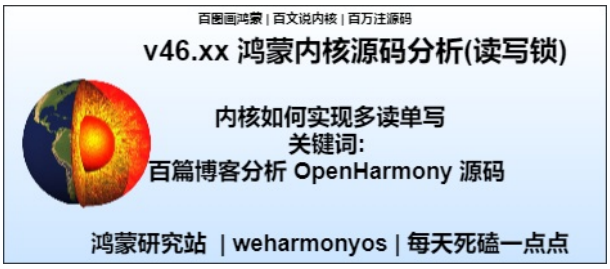


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

46_读写锁篇

本篇关键词：多读锁、单写锁、多核唤醒、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

特点&场景

读写锁 ：是计算机程序的并发控制的一种同步机制，也称“共享-互斥锁”、多读者-单写者锁。读操作可并发重入，写操作是互斥的。 鸿蒙实现的读写锁有几个特点：

- 一把锁分成 **读/写** 两种操作方式，读操作和写操作本身是互斥的，待操作任务按优先级存在两个独立的链表中，是读模式还是写模式，全凭任务的优先级而定，谁高就切到哪种模式。
- 一旦切到读模式，待读链表中优先级高于待写链表中**最高**优先级的任务们可以同时进行读操作，这些任务**并行**完成后便切到写模式。
- 一旦切到写模式，待写链表中优先级高于待读链表中**最高**优先级的任务们不可以同时进行写操作，只能一个个执行，这些任务**串行**完成后便切到读模式。
- 模式可以被高优先级任务打断，例如写模式时有高优先级的待读任务到来时，将切到读模式，反之亦然。

在实际很多业务场景中读写操作的频率是不同的，读往往高几个数量级，因读操作并不改变业务数据结构，所以读锁也称为**共享锁**。写操作会改变数据结构，数据之间须同步，修改注定是排他的，所以也称为**排它锁/互斥锁**，**读写锁**很好的解决了这种读写不对称的业务场景。

本篇详细解剖鸿蒙是如何实现读写锁的，代码部分均有注释，尤其释放锁部分的实现很精彩，让人有种酣畅淋漓的感觉。

鸿蒙实现

OsRwlock

系列篇多次说过，内核相对独立的功能 ，都有一个核心结构体 ，是一把吃透该功能的钥匙， 读写锁便是 `LosRwlock`。

```
typedef struct OsRwlock {//读写锁
    INT32 magic:24;        /**< Magic number | 魔法数字*/
    INT32 rwCount:8;       /**< Times of locking the rwlock, rwCount > 0 when rwkick is read mode, rwCount < 0
                            when the rwlock is write mode, rwCount = 0 when the lock is free.
                            大于0时为读模式，小于0时为写模式 等于0为自由模式
                            rwCount为读锁和写锁的数量,注意它们并不是此消彼长的行为模式*/
    VOID *writeOwner;      /**< The current write thread that is locking the rwlock | 拥有写权限的任务*/
```

```
    LOS_DL_LIST readList;  /**< Read waiting list | 等待读操作的任务链表*/
    LOS_DL_LIST writeList; /**< Write waiting list | 等待写操作的任务链表*/
} LosRwlock;
```

解读

- 魔法数字不会陌生，很多内核模块中都会使用到 ，比如栈顶就会有 ，以此判断栈是否溢出 (值被修改表示溢出)
- `rwCount` 记录读或者写时的任务数量，读写锁分五种操作模式：

```
    enum RwlockMode {
        RWLOCK_NONE_MODE, ///< 自由模式: 读写链表都没有内容
        RWLOCK_READ_MODE, ///< 读模式: 读链表有数据,写链表没有数据
        RWLOCK_WRITE_MODE,  ///< 写模式: 写链表有数据,读链表没有数据
        RWLOCK_READFIRST_MODE, ///< 读优先模式: 读链表中的任务最高优先级高于写链表中任务最高优先级
        RWLOCK_WRITEFIRST_MODE ///< 写优先模式: 写链表中的任务最高优先级高于读链表中任务最高优先级
    };
```

- `writeOwner` 持有写权限的任务 ，写操作是互斥操作 ，这里记录下最后执行写操作的任务
- `readList` 所有等待读操作的任务都会挂到这里 ，并按优先级排好序
- `writeList` 所有等待写操作的任务都会挂到这里 ，并按优先级排好序

初始化读写锁

```
UINT32 LOS_RwlockInit(LosRwlock *rwlock)
{
    UINT32 intSave;
    if (rwlock == NULL) {
        return LOS_EINVAL;
    }
    SCHEDULER_LOCK(intSave);
    if ((rwlock->magic & RWLOCK_COUNT_MASK) == OS_RWLOCK_MAGIC) {
        SCHEDULER_UNLOCK(intSave);
        return LOS_EPERM;
    }
    rwlock->rwCount = 0;
    rwlock->writeOwner = NULL;
    LOS_ListInit(&(rwlock->readList));
    LOS_ListInit(&(rwlock->writeList));
    rwlock->magic = OS_RWLOCK_MAGIC;
    SCHEDULER_UNLOCK(intSave);
    return LOS_OK;
}
```

优先级 | 找位置

不论读/写，都得按优先级排序，所以当一个申请任务到来时，通过 `OsSchedLockPendFindPos` 找到合适的位置将挂入链表的操作就很重要了

```
/// 找到合适的位置 将当前任务插入到 读/写锁链表中
LOS_DL_LIST *OsSchedLockPendFindPos(const LosTaskCB *runTask, LOS_DL_LIST *lockList)
{
    LOS_DL_LIST *node = NULL;
    if (LOS_ListEmpty(lockList)) {
        node = lockList;
    } else {
        LosTaskCB *pendedTask1 = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(lockList));//找到首任务
        LosTaskCB *pendedTask2 = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_LAST(lockList)); //找到尾任务
        if (pendedTask1->priority > runTask->priority) { //首任务比当前任务优先级低，所以runTask可以排第一
            node = lockList->pstNext;//下一个，注意priority越大 优先级越低
        } else if (pendedTask2->priority <= runTask->priority) {//尾任务比当前任务优先级高，所以runTask排最后
            node = lockList;
        } else {//两头比较没结果，陷入中间比较
            node = OsSchedLockPendFindPosSub(runTask, lockList);//进入子级循环找
        }
    }
    return node;
}
```

```
/// 找到合适的位置 将当前任务插入到 读/写锁链表中
STATIC INLINE LOS_DL_LIST *OsSchedLockPendFindPosSub(const LosTaskCB *runTask, const LOS_DL_LIST *lockList)
{
    LosTaskCB *pendedTask = NULL;
    LOS_DL_LIST *node = NULL;
    LOS_DL_LIST_FOR_EACH_ENTRY(pendedTask, lockList, LosTaskCB, pendList) {//遍历阻塞链表
        if (pendedTask->priority < runTask->priority) {//当前优先级更小,位置不宜,继续
            continue;
        } else if (pendedTask->priority > runTask->priority) {//找到合适位置
            node = &pendedTask->pendList;
            break;
        } else {//找到合适位置
            node = pendedTask->pendList.pstNext;
            break;
        }
    }
    return node;
}
```

申请读锁 | OsRwlockRdPendOp

申请读模式下的锁,分三种情况：

-
  - 若无人持有锁，读任务可获得锁。
-
  - 若有人持有锁，读任务可获得锁，读取顺序按照任务优先级。
-
  - 若有人（非自己）持有写模式下的锁，则当前任务无法获得锁，直到写模式下的锁释放。

```
LOS_RwlockRdLock
    OsRwlockRdUnsafe
        OsRwlockRdPendOp
```

```
STATIC UINT32 OsRwlockRdPendOp(LosTaskCB *runTask, LosRwlock *rwlock, UINT32 timeout)
{
    UINT32 ret;
    if (rwlock->rwCount >= 0) {//第一和第二种情况
        if (OsRwlockPriCompare(runTask, &(rwlock->writeList))) {//读优先级低于写优先级,意思就是必须先写再读
            if (rwlock->rwCount == INT8_MAX) {//读锁任务达到上限
                return LOS_EINVAL;
            }
            rwlock->rwCount++;//拿读锁成功
            return LOS_OK;
        }
    }
    if (!timeout) {
        return LOS_EINVAL;
    }
    if (!OsPreemptableInSched()) {//不可抢占时
        return LOS_EDEADLK;
    }
    /* The current task is not allowed to obtain the write lock when it obtains the read lock.
    | 当前任务在获得读锁时不允许获得写锁 */
    if ((LosTaskCB *)(rwlock->writeOwner) == runTask) { //拥有写锁任务是否为当前任务
        return LOS_EINVAL;
    }
    /*
     * When the rwlock mode is write mode or the priority of the current read task
     * is lower than the first pended write task, current read task will be pended.
     | 当 rwlock 模式为写模式或当前读任务的优先级低于第一个挂起的写任务时，当前读任务将被挂起。
     反正就是写锁任务优先
     */
    LOS_DL_LIST *node = OsSchedLockPendFindPos(runTask, &(rwlock->readList));//找到要挂入的位置
    //例如现有链表内任务优先级为 0 3 8 9 23 当前为 10 时, 返回的是 9 这个节点
    ret = OsSchedTaskWait(node, timeout, TRUE);//从尾部插入读锁链表 由此变成了 0 3 8 9 10 23
    if (ret == LOS_ERRNO_TSK_TIMEOUT) {
```

```
        return LOS_ETIMEDOUT;
    }
    return ret;
}
```

申请写锁 | OsRwlockWrPendOp

申请写模式下的锁，分两种情况：

- 若该锁当前没有任务持有，或者持有该读模式下的锁的任务和申请该锁的任务为同一个任务，则申请成功，可立即获得写模式下的锁。
- 若该锁当前已经存在读模式下的锁，且读取任务优先级较高，则当前任务挂起，直到读模式下的锁释放。

```
STATIC UINT32 OsRwlockWrPendOp(LosTaskCB *runTask, LosRwlock *rwlock, UINT32 timeout)
{
    UINT32 ret;
    /* When the rwlock is free mode, current write task can obtain this rwlock.
| 若该锁当前没有任务持有，或者持有该读模式下的锁的任务和申请该锁的任务为同一个任务，则申请成功，可立即获得写模式下的锁。*/
    if (rwlock->rwCount == 0) {
        rwlock->rwCount = -1;
        rwlock->writeOwner = (VOID *)runTask;//直接给当前进程锁
        return LOS_OK;
    }
    /* Current write task can use one rwlock once again if the rwlock owner is it.
| 如果 rwlock 拥有者是当前写入任务，则它可以再次使用该锁。*/
    if ((rwlock->rwCount < 0) && ((LosTaskCB *)(rwlock->writeOwner) == runTask)) {
        if (rwlock->rwCount == INT8_MIN) {
            return LOS_EINVAL;
        }
        rwlock->rwCount--;//注意再次拥有算是两把写锁了.
        return LOS_OK;
    }
    if (!timeout) {
        return LOS_EINVAL;
    }
    if (!OsPreemptableInSched()) {
        return LOS_EDEADLK;
    }
    /*
     * When the rwlock is read mode or other write task obtains this rwlock, current
     * write task will be pended. | 当 rwlock 为读模式或其他写任务获得该 rwlock 时，当前的写任务将被挂起。直到读模式下的锁释放
     */
    LOS_DL_LIST *node =  OsSchedLockPendFindPos(runTask, &(rwlock->writeList));//找到要挂入的位置
    ret = OsSchedTaskWait(node, timeout, TRUE);//从尾部插入写锁链表
    if (ret == LOS_ERRNO_TSK_TIMEOUT) {
        ret = LOS_ETIMEDOUT;
    }
    return ret;
}
```

释放读写锁 | OsRwlockPostOp

通常释放操作很简单 ，不会特书大书 ，但**读写锁**不一样，释放操作反而是精彩部分，因为释放就意味着等这把锁的任务有机会运行了，这些任务需要被唤醒 ，而读模式下能同时进行读操作，意味着内核要连续的唤醒等待读操作的任务 。所以出现了以下代码，站长认为这部分代码写的很精彩，必须要点赞 `@note_good`

```
/// 释放锁
STATIC UINT32 OsRwlockPostOp(LosRwlock *rwlock, BOOL *needSched)
{
    UINT32 rwlockMode;
    LosTaskCB *resumedTask = NULL;
    UINT16 pendedWriteTaskPri;

    rwlock->rwCount = 0;
    rwlock->writeOwner = NULL;
    rwlockMode = OsRwlockGetMode(&(rwlock->readList), &(rwlock->writeList));//先获取模式
    if (rwlockMode == RWLOCK_NONE_MODE) {//自由模式则正常返回
        return LOS_OK;
```

```
    }
    /* In this case, rwlock will wake the first pended write task. | 在这种情况下，rwlock 将唤醒第一个挂起的写任务。 */
    if ((rwlockMode == RWLOCK_WRITE_MODE) || (rwlockMode == RWLOCK_WRITEFIRST_MODE)) {//如果当前是写模式 (有任务在等写锁涅)
        resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->writeList)));//获取任务实体
        rwlock->rwCount = -1;//直接干成-1,注意这里并不是 --
        rwlock->writeOwner = (VOID *)resumedTask;//有锁了则唤醒等锁的任务(写模式)
        OsSchedTaskWake(resumedTask);
        if (needSched != NULL) {
            *needSched = TRUE;
        }
        return LOS_OK;
    }
    /* In this case, rwlock will wake the valid pended read task. 在这种情况下，rwlock 将唤醒有效的挂起读取任务。 */
    if (rwlockMode == RWLOCK_READFIRST_MODE) { //如果是读优先模式
        pendedWriteTaskPri = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->writeList)))->priority;//取出写锁任务的最高优先级
    }
    resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->readList)));//获取最高优先级读锁任务
    rwlock->rwCount = 1; //直接干成1,因为是释放操作
    OsSchedTaskWake(resumedTask);//有锁了则唤醒等锁的任务(读模式)
    while (!LOS_ListEmpty(&(rwlock->readList))) {//遍历读链表,目的是要唤醒其他读模式的任务(优先级得要高于pendedWriteTaskPri才行)
        resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->readList)));
        if ((rwlockMode == RWLOCK_READFIRST_MODE) && (resumedTask->priority >= pendedWriteTaskPri)) {//低于写模式的优先级
            break;//跳出循环
        }
        if (rwlock->rwCount == INT8_MAX) {
            return EINVAL;
        }
        rwlock->rwCount++;//读锁任务数量增加
        OsSchedTaskWake(resumedTask);//不断唤醒读锁任务,由此实现了允许多个读操作并发,因为在多核情况下resumedTask很大可能
        //与当前任务并不在同一个核上运行, 此处非常的有意思,点赞! @note_good
    }
    if (needSched != NULL) {
        *needSched = TRUE;
    }
    return LOS_OK;
}
```

解读

- 因写模式为互斥操作，背后的含义就是只能唤醒一个任务运行，所以取出首个(最高优先级)任务后，将任务唤醒加入就绪队列 `OsSchedTaskWake` ，申请调度 。

```
if ((rwlockMode == RWLOCK_WRITE_MODE) || (rwlockMode == RWLOCK_WRITEFIRST_MODE)) {//如果当前是写模式 (有任务在等写锁涅)
    resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->writeList)));//获取任务实体
    rwlock->rwCount = -1;//直接干成-1,注意这里并不是 --
    rwlock->writeOwner = (VOID *)resumedTask;//有锁了则唤醒等锁的任务(写模式)
    OsSchedTaskWake(resumedTask);
    if (needSched != NULL) {
        *needSched = TRUE;
    }
    return LOS_OK;
 }
```

- 而读模式为共享模式，读任务可以同时执行，这里的同时指的是并行，在多个 `CPU` 上同时执行读操作。能同时执行多少个取决于写链表上的优先级，所以会先拿到最高写任务优先级 `pendedWriteTaskPri` 用于连续唤醒读任务的优先级比较。

```
if (rwlockMode == RWLOCK_READFIRST_MODE) { //如果是读优先模式
    pendedWriteTaskPri = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->writeList)))->priority;//取出写锁任务的最高优先级
}
```

  通过循环连续的将读任务加入就绪队列，这才是真正做到能同时进行读操作(共享锁)的代码 !!! 这段代码一定要点赞，建议反复理解，有种醍醐灌顶 ，豁然开朗的奇妙感觉。

```
while (!LOS_ListEmpty(&(rwlock->readList))) {//遍历读链表,目的是要唤醒其他读模式的任务(优先级得要高于pendedWriteTaskPri才行)
    resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(rwlock->readList)));
    if ((rwlockMode == RWLOCK_READFIRST_MODE) && (resumedTask->priority >= pendedWriteTaskPri)) {//低于写模式的优先级
        break;//跳出循环
    }
    if (rwlock->rwCount == INT8_MAX) {
```

```
        return EINVAL;
    }
    rwlock->rwCount++;//读锁任务数量增加
    OsSchedTaskWake(resumedTask);//不断唤醒读锁任务,由此实现了允许多个读操作并发,因为在多核情况下resumedTask很大可能
    //与当前任务并不在同一个核上运行, 此处非常的有意思,点赞! @note_good
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

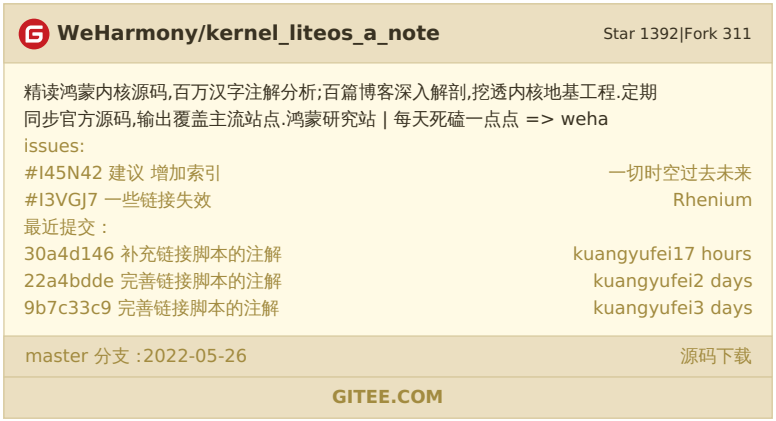
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

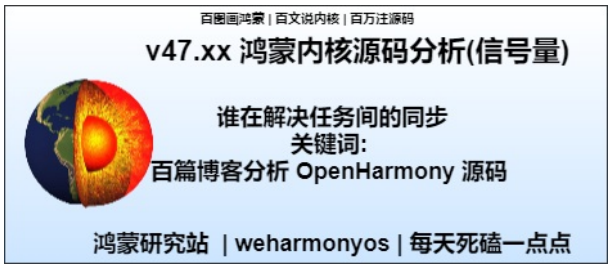


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

47_信号量篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

本篇说清楚信号量

读本篇之前建议先读v08.xx 鸿蒙内核源码分析(总目录) 其他篇幅。

基本概念

信号量（Semaphore） 是一种实现任务间通信的机制，可以实现任务间同步或共享资源的互斥访问。 一个信号量的数据结构中，通常有一个计数值，用于对有效资源数的计数，表示剩下的可被使用的共享资源数，其值的含义分两种情况：

> 0，表示该信号量当前不可获取，因此可能存在正在等待该信号量的任务。 正值，表示该信号量当前可被获取。

以同步为目的的信号量和以互斥为目的的信号量在使用上有如下不同：

> 用作互斥时，初始信号量计数值不为0，表示可用的共享资源个数。在需要使用共享资源前，先获取信号量，然后使用一个共享资源，使用完毕后释放信号量。这样在共享资源被取完，即信号量计数减至0时，其他需要获取信号量的任务将被阻塞，从而保证了共享资源的互斥访问。另外，当共享资源数为1时，建议使用二值信号量，一种类似于互斥锁的机制。

> 用作同步时，初始信号量计数值为0。任务1获取信号量而阻塞，直到任务2或者某中断释放信号量，任务1才得以进入Ready或Running态，从而达到了任务间的同步。

信号量运作原理

信号量初始化，为配置的N个信号量申请内存（N值可以由用户自行配置，通过 LOSCFG_BASE_IPC_SEM_LIMIT 宏实现），并把所有信号量初始化成未使用，加入到未使用链表中供系统使用。

- 信号量创建，从未使用的信号量链表中获取一个信号量，并设定初值。
- 信号量申请，若其计数器值大于0，则直接减1返回成功。否则任务阻塞，等待其它任务释放该信号量， 等待的超时时间可设定。当任务被一个信号量阻塞时，将该任务挂到信号量等待任务队列的队尾。
- 信号量释放，若没有任务等待该信号量，则直接将计数器加1返回。否则唤醒该信号量等待任务队列上的第一个任务。
- 信号量删除，将正在使用的信号量置为未使用信号量，并挂回到未使用链表。

> 信号量允许多个任务在同一时刻访问共享资源，但会限制同一时刻访问此资源的最大任务数目。 当访问资源的任务数达到该资源允许的最大数量时，会阻塞其他试图获取该资源的任务，直到有任务释放该信号量。

信号量长什么样？

```
typedef struct {
    UINT8 semStat; /**< Semaphore state *///信号量的状态
    UINT16 semCount; /**< Number of available semaphores *///有效信号量的数量
    UINT16 maxSemCount;  /**< Max number of available semaphores *///有效信号量的最大数量
    UINT32 semID; /**< Semaphore control structure ID *///信号量索引号
    LOS_DL_LIST semList; /**< Queue of tasks that are waiting on a semaphore *///等待信号量的任务队列，任务通过阻塞节点挂上去
} LosSemCB;
```

`semList` ，这又是一个双向链表， 双向链表是内核最重要的结构体， 可前往 v08.xx 鸿蒙内核源码分析(总目录) 查看双向链表篇， `LOS_DL_LIST` 像狗皮膏药一样牢牢的寄生在宿主结构体上`semList`上挂的是未来所有等待这个信号量的任务。

初始化信号量模块

```
#ifndef LOSCFG_BASE_IPC_SEM_LIMIT
#define LOSCFG_BASE_IPC_SEM_LIMIT 1024 //信号量的最大个数
#endif

LITE_OS_SEC_TEXT_INIT UINT32 OsSemInit(VOID)//信号量初始化
{
    LosSemCB *semNode = NULL;
    UINT32 index;

    LOS_ListInit(&g_unusedSemList);//初始
    /* system resident memory， don't free */
    g_allSem = (LosSemCB *)LOS_MemAlloc(m_aucSysMem0， (LOSCFG_BASE_IPC_SEM_LIMIT * sizeof(LosSemCB)));//分配信号池
    if (g_allSem == NULL) {
        return LOS_ERRNO_SEM_NO_MEMORY;
    }

    for (index = 0; index < LOSCFG_BASE_IPC_SEM_LIMIT; index++) {
        semNode = ((LosSemCB *)g_allSem) + index;//拿信号控制块， 可以直接g_allSem[index]来嘛
        semNode->semID = SET_SEM_ID(0， index);//保存ID
        semNode->semStat = OS_SEM_UNUSED;//标记未使用
        LOS_ListTailInsert(&g_unusedSemList， &semNode->semList);//通过semList把 信号块挂到空闲链表上
    }

    if (OsSemDbgInitHook() != LOS_OK) {
        return LOS_ERRNO_SEM_NO_MEMORY;
    }
    return LOS_OK;
}
```

分析如下:

- 初始化创建了信号量池来统一管理信号量， 默认 1024 个信号量
- 信号ID范围从 [0，1023]
- 未分配使用的信号量都挂到了全局变量 `g_unusedSemList` 上。

小建议:鸿蒙内核其他池(如进程池，任务池)都采用`free`来命名空闲链表，而此处使用`unused` ，命名风格不太严谨，有待改善。

创建信号量

```
LITE_OS_SEC_TEXT_INIT UINT32 OsSemCreate(UINT16 count， UINT16 maxCount， UINT32 *semHandle)
{
    unusedSem = LOS_DL_LIST_FIRST(&g_unusedSemList);//从未使用信号量池中取首个
    LOS_ListDelete(unusedSem);//从空闲链表上摘除
    semCreated = GET_SEM_LIST(unusedSem);//通过semList挂到链表上的，这里也要通过它把LosSemCB头查到。 进程，线程等结构体也都是这么干的。
    semCreated->semCount = count;//设置数量
```

```
    semCreated->semStat = OS_SEM_USED;//设置可用状态
    semCreated->maxSemCount = maxCount;//设置最大信号数量
    LOS_ListInit(&semCreated->semList);//初始化链表，后续阻塞任务通过task->pendList挂到semList链表上，就知道哪些任务在等它了。
    *semHandle = semCreated->semID;//参数带走 semID
    OsSemDbgUpdateHook(semCreated->semID， OsCurrTaskGet()->taskEntry， count);
    return LOS_OK;

ERR_HANDLER:
    OS_RETURN_ERROR_P2(errLine， errNo);
}
```

分析如下:

- 从未使用的空闲链表中拿首个信号量供分配使用。
- 信号量的最大数量和信号量个数都由参数指定。
- 信号量状态由 `OS_SEM_UNUSED` 变成了 `OS_SEM_USED`
- `semHandle` 带走信号量ID，外部由此知道成功创建了一个编号为 `*semHandle` 的信号量

申请信号量

```
LITE_OS_SEC_TEXT UINT32 LOS_SemPend(UINT32 semHandle， UINT32 timeout)
{
    UINT32 intSave;
    LosSemCB *semPended = GET_SEM(semHandle);//通过ID拿到信号体
    UINT32 retErr = LOS_OK;
    LosTaskCB *runTask = NULL;

    if (GET_SEM_INDEX(semHandle) >= (UINT32)LOSCFG_BASE_IPC_SEM_LIMIT) {
        OS_RETURN_ERROR(LOS_ERRNO_SEM_INVALID);
    }

    if (OS_INT_ACTIVE) {
        PRINT_ERR("!!!LOS_ERRNO_SEM_PEND_INTERR!!!\n");
        OsBackTrace();
        return LOS_ERRNO_SEM_PEND_INTERR;
    }

    runTask = OsCurrTaskGet();//获取当前任务
    if (runTask->taskStatus & OS_TASK_FLAG_SYSTEM_TASK) {
        OsBackTrace();
        return LOS_ERRNO_SEM_PEND_IN_SYSTEM_TASK;
    }

    SCHEDULER_LOCK(intSave);

    if ((semPended->semStat == OS_SEM_UNUSED) || (semPended->semID != semHandle)) {
        retErr = LOS_ERRNO_SEM_INVALID;
        goto OUT;
    }

    /* Update the operate time， no matter the actual Pend success or not */
    OsSemDbgTimeUpdateHook(semHandle);

    if (semPended->semCount > 0) {//还有资源可用，返回肯定得成功，semCount=0时代表没资源了，task会必须去睡眠了
        semPended->semCount--;//资源少了一个
        goto OUT;//注意这里 retErr = LOS_OK ，所以返回是OK的
    } else if (!timeout) {
        retErr = LOS_ERRNO_SEM_UNAVAILABLE;
        goto OUT;
    }

    if (!OsPreemptableInSched()) {//不能申请调度 (不能调度的原因是因为没有持有调度任务自旋锁)
        PRINT_ERR("!!!LOS_ERRNO_SEM_PEND_IN_LOCK!!!\n");
        OsBackTrace();
        retErr = LOS_ERRNO_SEM_PEND_IN_LOCK;
        goto OUT;
    }
```

```
    runTask->taskSem = (VOID *)semPended;//标记当前任务在等这个信号量
    retErr = OsTaskWait(&semPended->semList， timeout， TRUE);//任务进入等待状态，当前任务会挂到semList上，并在其中切换任务上下文
    if (retErr == LOS_ERRNO_TSK_TIMEOUT) {//注意:这里是涉及到task切换的，把自己挂起，唤醒其他task
        runTask->taskSem = NULL;
        retErr = LOS_ERRNO_SEM_TIMEOUT;
    }

OUT:
    SCHEDULER_UNLOCK(intSave);
    return retErr;
}
```

分析如下: 这个函数有点复杂，大量的 `goto` ，但别被它绕晕了，盯着返回值看。 先说结果只有一种情况下申请信号量能成功(即 `retErr == LOS_OK`)

```
    if (semPended->semCount > 0) {//还有资源可用，返回肯定得成功，semCount=0时代表没资源了，task会必须去睡眠了
        semPended->semCount--;//资源少了一个
        goto OUT;//注意这里 retErr = LOS_OK ，所以返回是OK的
    }
```

其余申请失败的原因有:

- 信号量ID超出范围(默认1024)
- 中断发生期间
- 系统任务
- 信号量状态不对，信号量ID不匹配

以上都是异常的判断，再说正常情况下 `semPended->semCount = 0` 时的情况，没有资源了怎么办？ 任务进入 `OsTaskWait` 睡眠状态，怎么睡，睡多久，由参数 `timeout` 定 `timeout` 值分以下三种模式:

> 无阻塞模式：即任务申请信号量时，入参 `timeout` 等于0。若当前信号量计数值不为0，则申请成功，否则立即返回申请失败。

> 永久阻塞模式：即任务申请信号量时，入参 `timeout` 等于0xFFFFFFFF。若当前信号量计数值不为0，则申请成功。 否则该任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后，直到有其他任务释放该信号量，阻塞任务才会重新得以执行。

> 定时阻塞模式：即任务申请信号量时， `0<timeout<0xFFFFFFFF` 。若当前信号量计数值不为0，则申请成功。 否则，该任务进入阻塞态，系统切换到就绪任务中优先级最高者继续执行。任务进入阻塞态后， 超时前如果有其他任务释放该信号量，则该任务可成功获取信号量继续执行，若超时前未获取到信号量，接口将返回超时错误码。

在 `OsTaskWait` 中，任务将被挂入 `semList` 链表， `semList` 上挂的都是等待这个信号量的任务。

释放信号量

```
LITE_OS_SEC_TEXT UINT32 OsSemPostUnsafe(UINT32 semHandle， BOOL *needSched)
{
    LosSemCB *semPosted = NULL;
    LosTaskCB *resumedTask = NULL;

    if (GET_SEM_INDEX(semHandle) >= LOSCFG_BASE_IPC_SEM_LIMIT) {
        return LOS_ERRNO_SEM_INVALID;
    }

    semPosted = GET_SEM(semHandle);
    if ((semPosted->semID != semHandle) || (semPosted->semStat == OS_SEM_UNUSED)) {
        return LOS_ERRNO_SEM_INVALID;
    }

    /* Update the operate time， no matter the actual Post success or not */
    OsSemDbgTimeUpdateHook(semHandle);

    if (semPosted->semCount == OS_SEM_COUNT_MAX) {//当前信号资源不能大于最大资源量
        return LOS_ERRNO_SEM_OVERFLOW;
    }
    if (!LOS_ListEmpty(&semPosted->semList)) {//当前有任务挂在semList上，要去唤醒任务
        resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&(semPosted->semList)));//semList上面挂的都是task->pendlist节点，取第一个tas
        resumedTask->taskSem = NULL;//任务不用等信号了，重新变成NULL值
        OsTaskWake(resumedTask);//唤醒任务，注意resumedTask一定不是当前任务，OsTaskWake里面并不会自己切换任务上下文，只是设置状态
        if (needSched != NULL) {//参数不为空，就返回需要调度的标签
```

```
            *needSched = TRUE;//TRUE代表需要调度
        }
    } else {//当前没有任务挂在semList上，
        semPosted->semCount++;//信号资源多一个
    }

    return LOS_OK;
}

LITE_OS_SEC_TEXT UINT32 LOS_SemPost(UINT32 semHandle)
{
    UINT32 intSave;
    UINT32 ret;
    BOOL needSched = FALSE;

    SCHEDULER_LOCK(intSave);
    ret = OsSemPostUnsafe(semHandle， &needSched);
        SCHEDULER_UNLOCK(intSave);
    if (needSched) {//需要调度的情况
        LOS_MpSchedule(OS_MP_CPU_ALL);//向所有CPU发送调度指令
        LOS_Schedule();////发起调度
    }

    return ret;
}
```

分析如下:

- 注意看在什么情况下 `semPosted->semCount` 才会 ++ ，是在 `LOS_ListEmpty` 为真的时候， `semList` 是等待这个信号量的任务。 `semList` 上的任务是在 `OsTaskWait` 中挂入的。都在等这个信号。
- 每次 `OsSemPost` 都会唤醒 `semList` 链表上一个任务，直到 `semList` 为空。
- 掌握信号量的核心是理解 `LOS_SemPend` 和 `LOS_SemPost`

编程示例

本实例实现如下功能：

- 测试任务Example_TaskEntry创建一个信号量，锁任务调度，创建两个任务Example_SemTask1、Example_SemTask2，Example_SemTask2优先级高于Example_SemTask1，两个任务中申请同一信号量，解锁任务调度后两任务阻塞，测试任务Example_TaskEntry释放信号量。
- Example_SemTask2得到信号量，被调度，然后任务休眠20Tick，Example_SemTask2延迟，Example_SemTask1被唤醒。
- Example_SemTask1定时阻塞模式申请信号量，等待时间为10Tick，因信号量仍被Example_SemTask2持有，Example_SemTask1挂起，10Tick后仍未得到信号量， Example_SemTask1被唤醒，试图以永久阻塞模式申请信号量，Example_SemTask1挂起。
- 20Tick后Example_SemTask2唤醒， 释放信号量后，Example_SemTask1得到信号量被调度运行，最后释放信号量。
- Example_SemTask1执行完，40Tick后任务Example_TaskEntry被唤醒，执行删除信号量，删除两个任务。

```
/* 任务ID */
static UINT32 g_testTaskId01;
static UINT32 g_testTaskId02;
/* 测试任务优先级 */
#define TASK_PRIO_TEST  5
/* 信号量结构体id */
static UINT32 g_semId;

VOID Example_SemTask1(VOID)
{
    UINT32 ret;

    printf("Example_SemTask1 try get sem g_semId ，timeout 10 ticks.\n");
    /* 定时阻塞模式申请信号量，定时时间为10ticks */
    ret = LOS_SemPend(g_semId， 10);

    /*申请到信号量*/
    if (ret == LOS_OK) {
        LOS_SemPost(g_semId);
        return;
```

```
    }
    /* 定时时间到，未申请到信号量 */
    if (ret == LOS_ERRNO_SEM_TIMEOUT) {
        printf("Example_SemTask1 timeout and try get sem g_semId wait forever.\n");
        /*永久阻塞模式申请信号量*/
        ret = LOS_SemPend(g_semId， LOS_WAIT_FOREVER);
        printf("Example_SemTask1 wait_forever and get sem g_semId .\n");
        if (ret == LOS_OK) {
            LOS_SemPost(g_semId);
            return;
        }
    }
}

VOID Example_SemTask2(VOID)
{
    UINT32 ret;
    printf("Example_SemTask2 try get sem g_semId wait forever.\n");
    /* 永久阻塞模式申请信号量 */
    ret = LOS_SemPend(g_semId， LOS_WAIT_FOREVER);

    if (ret == LOS_OK) {
        printf("Example_SemTask2 get sem g_semId and then delay 20ticks .\n");
    }

    /* 任务休眠20 ticks */
    LOS_TaskDelay(20);

    printf("Example_SemTask2 post sem g_semId .\n");
    /* 释放信号量 */
    LOS_SemPost(g_semId);
    return;
}

UINT32 ExampleTaskEntry(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S task1;
    TSK_INIT_PARAM_S task2;

   /* 创建信号量 */
    LOS_SemCreate(0，&g_semId);

    /* 锁任务调度 */
    LOS_TaskLock();

    /*创建任务1*/
    (VOID)memset_s(&task1， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));
    task1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_SemTask1;
    task1.pcName       = "TestTsk1";
    task1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    task1.usTaskPrio   = TASK_PRIO_TEST;
    ret = LOS_TaskCreate(&g_testTaskId01， &task1);
    if (ret != LOS_OK) {
        printf("task1 create failed .\n");
        return LOS_NOK;
    }

    /* 创建任务2 */
    (VOID)memset_s(&task2， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));
    task2.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_SemTask2;
    task2.pcName       = "TestTsk2";
    task2.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    task2.usTaskPrio   = (TASK_PRIO_TEST - 1);
    ret = LOS_TaskCreate(&g_testTaskId02， &task2);
    if (ret != LOS_OK) {
        printf("task2 create failed .\n");
        return LOS_NOK;
    }

    /* 解锁任务调度 */
```

```
    LOS_TaskUnlock();

    ret = LOS_SemPost(g_semId);

    /* 任务休眠40 ticks */
    LOS_TaskDelay(40);

    /* 删除信号量 */
    LOS_SemDelete(g_semId);

    /* 删除任务1 */
    ret = LOS_TaskDelete(g_testTaskId01);
    if (ret != LOS_OK) {
        printf("task1 delete failed .\n");
        return LOS_NOK;
    }
    /* 删除任务2 */
    ret = LOS_TaskDelete(g_testTaskId02);
    if (ret != LOS_OK) {
        printf("task2 delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

实例运行结果:

```
Example_SemTask2 try get sem g_semId wait forever.
Example_SemTask1 try get sem g_semId  , timeout 10 ticks.
Example_SemTask2 get sem g_semId and then delay 20ticks .
Example_SemTask1 timeout and try get sem g_semId wait forever.
Example_SemTask2 post sem g_semId .
Example_SemTask1 wait_forever and get sem g_semId .
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

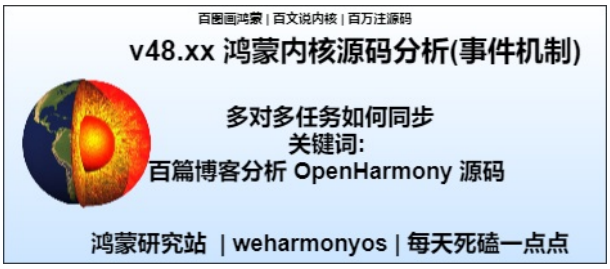
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

48_事件控制篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

本篇说清楚事件（Event）

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 其他篇。

官方概述

先看官方对事件的描述。

事件（Event）是一种任务间通信的机制，可用于任务间的同步。

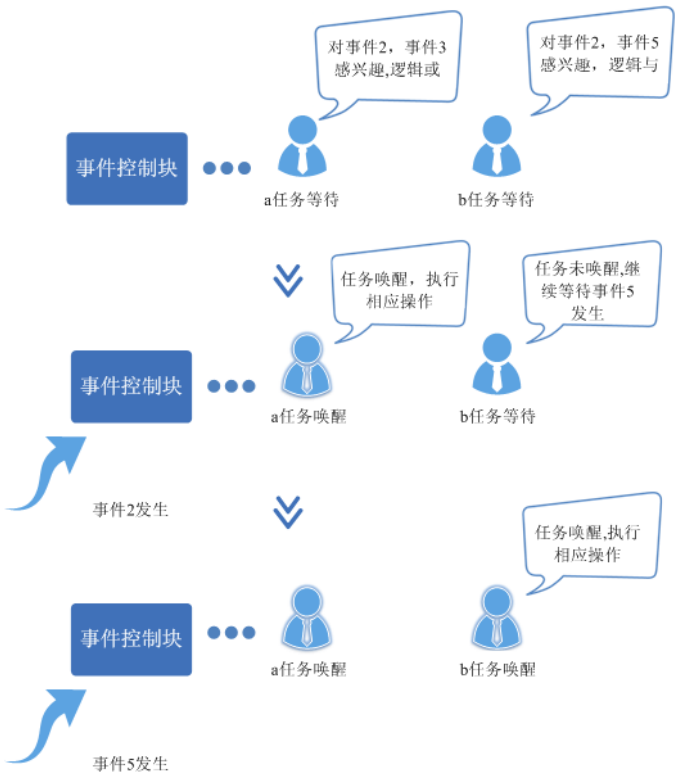
多任务环境下，任务之间往往需要同步操作，一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。

- 一对多同步模型：一个任务等待多个事件的触发。可以是任意一个事件发生时唤醒任务处理事件，也可以是几个事件都发生后才唤醒任务处理事件。
- 多对多同步模型：多个任务等待多个事件的触发。

鸿蒙提供的事件具有如下特点：

- 任务通过创建事件控制块来触发事件或等待事件。
- 事件间相互独立，内部实现为一个32位无符号整型，每一位标识一种事件类型。第25位不可用，因此最多可支持31种事件类型。
- 事件仅用于任务间的同步，不提供数据传输功能。
- 多次向事件控制块写入同一事件类型，在被清零前等效于只写入一次。
- 多个任务可以对同一事件进行读写操作。
- 支持事件读写超时机制。

再看事件图



注意图中提到了三个概念 `事件控制块` `事件` `任务` 接下来结合代码来理解事件模块的实现。

事件控制块长什么样？

```
typedef struct tagEvent {
    UINT32 uwEventID;       /**< Event mask in the event control block，//标识发生的事件类型位，事件ID，每一位标识一种事件类型
                            indicating the event that has been logically processed. */
    LOS_DL_LIST stEventList; /**< Event control block linked list *///读取事件任务链表
} EVENT_CB_S，*PEVENT_CB_S;
```

简单是简单，就两个变量，如下: `uwEventID` ：用于标识该任务发生的事件类型，其中每一位表示一种事件类型（0表示该事件类型未发生、1表示该事件类型已经发生），一共31种事件类型，第25位系统保留。

`stEventList` ，这又是一个双向链表， 双向链表是内核最重要的结构体， 可前往 v08.xx 鸿蒙内核源码分析(总目录) 查看双向链表篇。
`LOS_DL_LIST` 像狗皮膏药一样牢牢的寄生在宿主结构体上 `stEventList` 上挂的是所有等待这个事件的任务。

事件控制块<>事件<>任务 三者关系

一定要搞明白这三者的关系，否则搞不懂事件模块是如何运作的。

- 任务是事件的生产者，通过 `LOS_EventWrite` ，向外部广播发生了XX事件，并唤醒此前已在事件控制块中登记过的要等待XX事件发生的XX任务。
- 事件控制块 `EVENT_CB_S` 是记录者，只干两件事件:

  1. `uwEventID` 按位记录哪些事件发生了，它只是记录，怎么消费它不管的。

  2. `stEventList` 记录哪些任务在等待事件，但任务究竟在等待哪些事件它也是不记录的

- 任务也是消费者，通过 `LOS_EventRead` 消费，只有任务自己清楚要以什么样的方式，消费什么样的事件。 先回顾下任务结构体 `LosTaskCB` 对事件部分的描述如下:

```
typedef struct {
    //...去掉不相关的部分
    VOID          *taskEvent;  //和任务发生关系的事件控制块
    UINT32        eventMask;   //对哪些事件进行屏蔽
    UINT32        eventMode;   //事件三种模式(LOS_WAITMODE_AND，LOS_WAITMODE_OR，LOS_WAITMODE_CLR)
```

```
} LosTaskCB;
```

`taskEvent` 指向的就是 `EVENT_CB_S`

`eventMask` 屏蔽掉 事件控制块 中的哪些事件

`eventMode` 已什么样的方式去消费事件，三种读取模式

```
#define LOS_WAITMODE_AND                4U
#define LOS_WAITMODE_OR                 2U
#define LOS_WAITMODE_CLR                1U
```

- 所有事件（`LOS_WAITMODE_AND`）：逻辑与，基于接口传入的事件类型掩码`eventMask`，只有这些事件都已经发生才能读取成功，否则该任务将阻塞等待或者返回错误码。
- 任一事件（`LOS_WAITMODE_OR`）：逻辑或，基于接口传入的事件类型掩码`eventMask`，只要这些事件中有任一种事件发生就可以读取成功，否则该任务将阻塞等待或者返回错误码。
- 清除事件（`LOS_WAITMODE_CLR`）：这是一种附加读取模式，需要与所有事件模式或任一事件模式结合使用（`LOS_WAITMODE_AND | LOS_WAITMODE_CLR` 或 `LOS_WAITMODE_OR | LOS_WAITMODE_CLR`）。在这种模式下，当设置的所有事件模式或任一事件模式读取成功后，会自动清除事件控制块中对应的事件类型位。

- 一个事件控制块`EVENT_CB_S`中的事件可以来自多个任务，多个任务也可以同时消费事件控制块中的事件，并且这些任务之间可以没有任何关系!

函数列表

事件可应用于多种任务同步场景，在某些同步场景下可替代信号量。

功能分类	接口名	描述
初始化事件	LOS_EventInit	初始化一个事件控制块
读/写事件	LOS_EventRead	读取指定事件类型，超时时间为相对时间：单位为Tick
	LOS_EventWrite	写指定的事件类型
清除事件	LOS_EventClear	清除指定的事件类型
校验事件掩码	LOS_EventPoll	根据用户传入的事件ID、事件掩码及读取模式，返回用户传入的事件是否符合预期
销毁事件	LOS_EventDestroy	销毁指定的事件控制块

其中读懂 `OsEventWrite` 和 `OsEventRead` 就明白了事件模块。

事件初始化 -> LOS_EventInit

```
//初始化一个事件控制块
LITE_OS_SEC_TEXT_INIT UINT32 LOS_EventInit(PEVENT_CB_S eventCB)
{
    UINT32 intSave;
    intSave = LOS_IntLock();//锁中断
    eventCB->uwEventID = 0; //其中每一位表示一种事件类型（0表示该事件类型未发生、1表示该事件类型已经发生）
    LOS_ListInit(&eventCB->stEventList);//事件链表初始化
```

```
    LOS_IntRestore(intSave);//恢复中断
    return LOS_OK;
}
```

代码解读:

- 事件是共享资源，所以操作期间不能产生中断。
- 初始化两个记录者 `uwEventID` `stEventList`

事件生产过程 -> OsEventWrite


```
LITE_OS_SEC_TEXT VOID OsEventWriteUnsafe(PEVENT_CB_S eventCB， UINT32 events， BOOL once， UINT8 *exitFlag)
{
    LosTaskCB *resumedTask = NULL;
    LosTaskCB *nextTask = NULL;
    BOOL schedFlag = FALSE;

    eventCB->uwEventID |= events;//对应位贴上标签
    if (!LOS_ListEmpty(&eventCB->stEventList)) {//等待事件链表判断，处理等待事件的任务
        for (resumedTask = LOS_DL_LIST_ENTRY((&eventCB->stEventList)->pstNext， LosTaskCB， pendList);
             &resumedTask->pendList != &eventCB->stEventList;) {//循环获取任务链表
            nextTask = LOS_DL_LIST_ENTRY(resumedTask->pendList.pstNext， LosTaskCB， pendList);//获取任务实体
            if (OsEventResume(resumedTask， eventCB， events)) {//是否恢复任务
                schedFlag = TRUE;//任务已加至就绪队列，申请发生一次调度
            }
            if (once == TRUE) {//是否只处理一次任务
                break;//退出循环
            }
            resumedTask = nextTask;//检查链表中下一个任务
        }
    }

    if ((exitFlag != NULL) && (schedFlag == TRUE)) {//是否让外面调度
        *exitFlag = 1;
    }
}
//写入事件
LITE_OS_SEC_TEXT STATIC UINT32 OsEventWrite(PEVENT_CB_S eventCB， UINT32 events， BOOL once)
{
    UINT32 intSave;
    UINT8 exitFlag = 0;

    SCHEDULER_LOCK(intSave); //禁止调度
    OsEventWriteUnsafe(eventCB， events， once， &exitFlag);//写入事件
    SCHEDULER_UNLOCK(intSave); //允许调度
```

```
    if (exitFlag == 1) { //需要发生调度
        LOS_MpSchedule(OS_MP_CPU_ALL);//通知所有CPU调度
        LOS_Schedule();//执行调度
    }
    return LOS_OK;
}
```

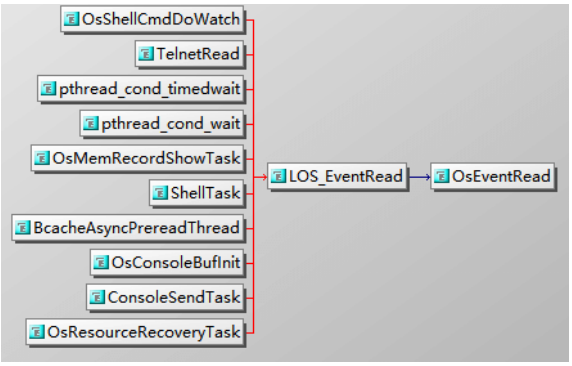
代码解读:

1. 给对应位贴上事件标签， `eventCB->uwEventID |= events;` 注意uwEventID是按位管理的。每个位代表一个事件是否写入，例如 `uwEventID = 00010010` 代表产生了 1，4 事件
2. 循环从 `stEventList` 链表中取出等待这个事件的任务判断是否唤醒任务。 `OsEventResume`

```
//事件恢复，判断是否唤醒任务
LITE_OS_SEC_TEXT STATIC UINT8 OsEventResume(LosTaskCB *resumedTask， const PEVENT_CB_S eventCB， UINT32 events)
{
    UINT8 exitFlag = 0;//是否唤醒

    if (((resumedTask->eventMode & LOS_WAITMODE_OR) && ((resumedTask->eventMask & events) != 0)) ||
        ((resumedTask->eventMode & LOS_WAITMODE_AND) &&
        ((resumedTask->eventMask & eventCB->uwEventID) == resumedTask->eventMask))) {//逻辑与 和 逻辑或 的处理
        exitFlag = 1;

        resumedTask->taskEvent = NULL;
        OsTaskWake(resumedTask);//唤醒任务，加入就绪队列
    }

    return exitFlag;
}
```

3.唤醒任务 `OsTaskWake` 只是将任务重新加入就绪队列，需要立即申请一次调度 `LOS_Schedule` 。

事件消费过程 -> OsEventRead




```
LITE_OS_SEC_TEXT STATIC UINT32 OsEventRead(PEVENT_CB_S eventCB， UINT32 eventMask， UINT32 mode， UINT32 timeout，
                                           BOOL once)
{
    UINT32 ret;
    UINT32 intSave;
    SCHEDULER_LOCK(intSave);
    ret = OsEventReadImp(eventCB， eventMask， mode， timeout， once);//读事件实现函数
    SCHEDULER_UNLOCK(intSave);
    return ret;
}

//读取指定事件类型的实现函数，超时时间为相对时间：单位为Tick
LITE_OS_SEC_TEXT STATIC UINT32 OsEventReadImp(PEVENT_CB_S eventCB， UINT32 eventMask， UINT32 mode，
                                              UINT32 timeout， BOOL once)
{
    UINT32 ret = 0;
    LosTaskCB *runTask = OsCurrTaskGet();
    runTask->eventMask = eventMask;
```

```
    runTask->eventMode = mode;
    runTask->taskEvent = eventCB;//事件控制块
    ret = OsTaskWait(&eventCB->stEventList， timeout， TRUE);//任务进入等待状态，挂入阻塞链表
    if (ret == LOS_ERRNO_TSK_TIMEOUT) {//如果返回超时
        runTask->taskEvent = NULL;
        return LOS_ERRNO_EVENT_READ_TIMEOUT;
    }
    ret = OsEventPoll(&eventCB->uwEventID， eventMask， mode);//检测事件是否符合预期
    return ret;
}
```

代码解读:

- 事件控制块是给任务使用的， 任务给出读取一个事件的条件
  - `eventMask` 告诉系统屏蔽掉这些事件，对屏蔽的事件不感冒.
  - `eventMode` 已什么样的方式去消费事件，是必须都满足给的条件，还是只满足一个就响应.
  - 条件给完后，自己进入等待状态 `OsTaskWait` ，等待多久 `timeout` 决定，任务自己说了算.
  - `OsEventPoll` 检测事件是否符合预期，啥意思？看下它的代码就知道了

```
//根据用户传入的事件值、事件掩码及校验模式，返回用户传入的事件是否符合预期
LITE_OS_SEC_TEXT UINT32 OsEventPoll(UINT32 *eventID， UINT32 eventMask， UINT32 mode)
{
    UINT32 ret = 0;//事件是否发生了

    LOS_ASSERT(OsIntLocked());//断言不允许中断了
    LOS_ASSERT(LOS_SpinHeld(&g_taskSpin));//任务自旋锁

    if (mode & LOS_WAITMODE_OR) {//如果模式是读取掩码中任意事件
        if ((*eventID & eventMask) != 0) {
            ret = *eventID & eventMask; //发生了
        }
    } else {//等待全部事件发生
        if ((eventMask != 0) && (eventMask == (*eventID & eventMask))) {//必须满足全部事件发生
            ret = *eventID & eventMask; //发生了
        }
    }

    if (ret && (mode & LOS_WAITMODE_CLR)) {//是否清除事件
        *eventID = *eventID & ~ret;
    }

    return ret;
}
```

编程实例

本实例实现如下流程。

示例中，任务Example_TaskEntry创建一个任务Example_Event，Example_Event读事件阻塞，Example_TaskEntry向该任务写事件。可以通过示例日志中打印的先后顺序理解事件操作时伴随的任务切换。

- 在任务Example_TaskEntry创建任务Example_Event，其中任务Example_Event优先级高于Example_TaskEntry。
- 在任务Example_Event中读事件0x00000001，阻塞，发生任务切换，执行任务Example_TaskEntry。
- 在任务Example_TaskEntry向任务Example_Event写事件0x00000001，发生任务切换，执行任务Example_Event。
- Example_Event得以执行，直到任务结束。
- Example_TaskEntry得以执行，直到任务结束。

```
#include "los_event.h"
#include "los_task.h"
#include "securec.h"

/* 任务ID */
UINT32 g_testTaskId;

/* 事件控制结构体 */
EVENT_CB_S g_exampleEvent;
```

```
/* 等待的事件类型 */
#define EVENT_WAIT 0x00000001

/* 用例任务入口函数 */
VOID Example_Event(VOID)
{
    UINT32 ret;
    UINT32 event;

    /* 超时等待方式读事件，超时时间为100 ticks， 若100 ticks后未读取到指定事件，读事件超时，任务直接唤醒 */
    printf("Example_Event wait event 0x%x \n"， EVENT_WAIT);

    event = LOS_EventRead(&g_exampleEvent， EVENT_WAIT， LOS_WAITMODE_AND， 100);
    if (event == EVENT_WAIT) {
        printf("Example_Event，read event :0x%x\n"， event);
    } else {
        printf("Example_Event，read event timeout\n");
    }
}

UINT32 Example_TaskEntry(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S task1;

    /* 事件初始化 */
    ret = LOS_EventInit(&g_exampleEvent);
    if (ret != LOS_OK) {
        printf("init event failed .\n");
        return -1;
    }

    /* 创建任务 */
    (VOID)memset_s(&task1， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));
    task1.pfnTaskEntry = (TSK_ENTRY_FUNC)Example_Event;
    task1.pcName       = "EventTsk1";
    task1.uwStackSize  = OS_TSK_DEFAULT_STACK_SIZE;
    task1.usTaskPrio   = 5;
    ret = LOS_TaskCreate(&g_testTaskId， &task1);
    if (ret != LOS_OK) {
        printf("task create failed .\n");
        return LOS_NOK;
    }

    /* 写g_testTaskId 等待事件 */
    printf("Example_TaskEntry write event .\n");

    ret = LOS_EventWrite(&g_exampleEvent， EVENT_WAIT);
    if (ret != LOS_OK) {
        printf("event write failed .\n");
        return LOS_NOK;
    }

    /* 清标志位 */
    printf("EventMask:%d\n"， g_exampleEvent.uwEventID);
    LOS_EventClear(&g_exampleEvent， ~g_exampleEvent.uwEventID);
    printf("EventMask:%d\n"， g_exampleEvent.uwEventID);

    /* 删除任务 */
    ret = LOS_TaskDelete(g_testTaskId);
    if (ret != LOS_OK) {
        printf("task delete failed .\n");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

运行结果

```
Example_Event wait event 0x1
Example_TaskEntry write event .
Example_Event , read event :0x1
EventMask:1
EventMask:0
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

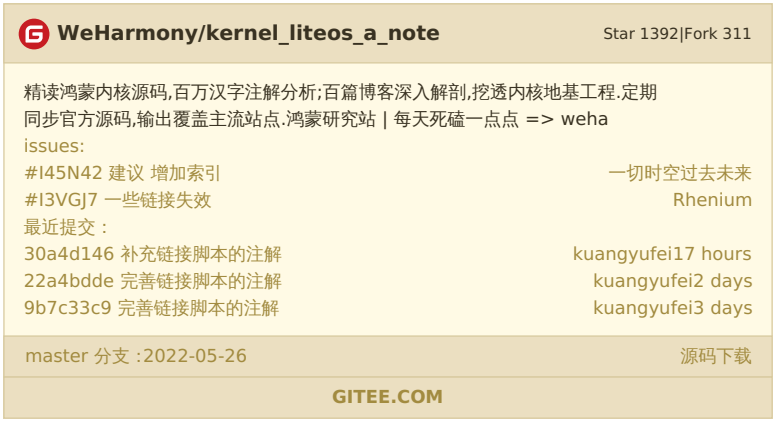
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

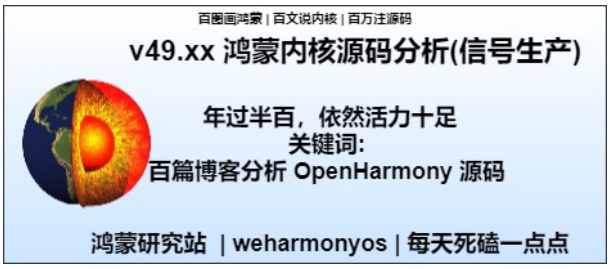


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

49_信号生产篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

信号生产

关于信号篇，本只想写一篇，但发现把它想简单了，内容不多，难度极大。整理了好长时间，理解了为何<<深入理解linux内核>>要单独为它开一章，原因有二

- 信号相关的结构体多，而且还容易搞混。所以看本篇要注意结构体的名字和作用。
- 系统调用太多了，涉及面广，信号的来源分硬件和软件。相当于软中断和硬中断，这就会涉及到汇编代码，但信号的处理函数又在用户空间，CPU是禁止内核态执行用户态代码的，所以运行过程需在用户空间和内核空间来回的折腾，频繁的切换上下文。

信号思想来自Unix，它老人家已经五十多岁了，但很有活力，许多方面几乎没发生大的变化。信号可以由内核产生，也可以由用户进程产生，并由内核传送给特定的进程或线程(组)，若这个进程定义了自己的信号处理程序，则调用这个程序去处理信号，否则则执行默认的程序或者忽略。

信号为系统提供了一种进程间异步通讯的方式，一个进程不必通过任何操作来等待信号的到达。事实上，进程也不可能知道信号到底什么时候到达。一般来说，只需用户进程提供信号处理函数，内核会想方设法调用信号处理函数，网上查阅了很多的关于信号的资料。个人想换个视角去看信号。把异步过程理解为生产者(安装和发送信号)和消费者(捕捉和处理信号)两个过程。鉴于此，系列篇将分成两篇说明，本篇为信号生产篇:

- **v49.xx (信号生产篇) | 年过半百，依然活力十足**
- **v50.xx (信号消费篇) | 谁让CPU连续四次换栈运行**

信号分类

每个信号都有一个名字和编号，这些名字都以 `SIG` 开头，例如 `SIGQUIT` 、 `SIGCHLD` 等等。 信号定义在signal。h头文件中，信号名都定义为正整数。 具体的信号名称可以使用 `kill -l` 来查看信号的名字以及序号，信号是从1开始编号的，不存在0号信号。不过 `kill` 对于信号0有特殊的应用。啥用呢？ 可用来查询进程是否还在。 敲下 `kill 0 pid` 就知道了。

信号分为两大类：可靠信号与不可靠信号，前32种信号为不可靠信号，后32种为可靠信号。

- 不可靠信号： 也称为非实时信号，不支持排队，信号可能会丢失， 比如发送多次相同的信号， 进程只能收到一次。 信号值取值区间为1~31；
- 可靠信号： 也称为实时信号，支持排队， 信号不会丢失， 发多少次， 就可以收到多少次。 信号值取值区间为32~64

```
#define SIGHUP    1 //终端挂起或者控制进程终止
#define SIGINT    2 //键盘中断（ctrl + c）
#define SIGQUIT   3 //键盘的退出键被按下
#define SIGILL    4 //非法指令
#define SIGTRAP   5 //跟踪陷阱（trace trap），启动进程，跟踪代码的执行
#define SIGABRT   6 //由abort(3)发出的退出指令
#define SIGIOT    SIGABRT //abort发出的信号
#define SIGBUS    7 //总线错误
#define SIGFPE    8 //浮点异常
#define SIGKILL   9 //常用的命令 kill 9 123 | 不能被忽略、处理和阻塞
#define SIGUSR1   10 //用户自定义信号1
#define SIGSEGV   11 //无效的内存引用， 段违例（segmentation    violation），进程试图去访问其虚地址空间以外的位置
#define SIGUSR2   12 //用户自定义信号2
#define SIGPIPE   13 //向某个非读管道中写入数据
#define SIGALRM   14 //由alarm(2)发出的信号，默认行为为进程终止
#define SIGTERM   15 //终止信号
#define SIGSTKFLT 16 //栈溢出
#define SIGCHLD   17 //子进程结束信号
#define SIGCONT   18 //进程继续（曾被停止的进程）
#define SIGSTOP   19 //终止进程    | 不能被忽略、处理和阻塞
#define SIGTSTP   20 //控制终端（tty）上 按下停止键
#define SIGTTIN   21 //进程停止，后台进程企图从控制终端读
#define SIGTTOU   22 //进程停止，后台进程企图从控制终端写
#define SIGURG    23 //I/O有紧急数据到达当前进程
#define SIGXCPU   24 //进程的CPU时间片到期
#define SIGXFSZ   25 //文件大小的超出上限
#define SIGVTALRM 26 //虚拟时钟超时
#define SIGPROF   27 //profile时钟超时
#define SIGWINCH  28 //窗口大小改变
#define SIGIO     29 //I/O相关
#define SIGPOLL   29 //
#define SIGPWR    30 //电源故障，关机
#define SIGSYS    31 //系统调用中参数错，如系统调用号非法
#define SIGUNUSED SIGSYS//不使用

#define _NSIG 65
```

信号来源

信号来源分为硬件类和软件类：

- 硬件类
  - 用户输入：比如在终端上按下组合键`ctrl+C`，产生`SIGINT`信号；
  - 硬件异常：CPU检测到内存非法访问等异常，通知内核生成相应信号，并发送给发生事件的进程；
- 软件类
  - 通过系统调用，发送signal信号：`kill()`，`raise()`，`sigqueue()`，`alarm()`，`setitimer()`，`abort()`
    - `kill`命令就是一个发送信号的工具，用于向进程或进程组发送信号。例如: `kill 9 PID`（`SIGKILL`)来杀死`PID`进程。
    - sigqueue()：只能向一个进程发送信号，不能向进程组发送信号；主要针对实时信号提出，与sigaction()组合使用，当然也支持非实时信号的发送；
    - alarm()：用于调用进程指定时间后发出SIGALARM信号；
    - setitimer()：设置定时器，计时达到后给进程发送SIGALRM信号，功能比alarm更强大；
    - abort()：向进程发送SIGABORT信号，默认进程会异常退出。
    - raise()：用于向进程自身发送信号；

信号与进程的关系

主要是通过系统调用`sigaction`将用户态信号处理函数注册到PCB保存。所有进程的任务都共用这个信号注册函数`sigHandler`，在信号的消费阶段内核用一种特殊的方式'回调'它。

```
typedef struct ProcessCB {//PCB中关于信号的信息
    UINTPTR             sigHandler;   /**< signal handler */   //捕捉信号后的处理函数
    sigset_t            sigShare;     /**< signal share bit */ //信号共享位，64个信号各站一位
}LosProcessCB;
typedef unsigned _Int64 sigset_t; //一个64位的变量，每个信号代表一位。
struct sigaction {//信号处理机制结构体
 union {
```

```
 void (*sa_handler)(int); //信号处理函数——普通版
 void (*sa_sigaction)(int， siginfo_t *， void *);//信号处理函数——高级版
} __sa_handler;
sigset_t sa_mask;//指定信号处理程序执行过程中需要阻塞的信号；
int sa_flags;  //标示位
    // SA_RESTART：使被信号打断的syscall重新发起。
    // SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号。
    // SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到SIGCHLD信号，这时子进程如果退出也不会成为僵 尸进程。
    // SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。
    // SA_RESETHAND：信号处理之后重新设置为默认的处理方式。
    // SA_SIGINFO：使用sa_sigaction成员而不是sa_handler作为信号处理函数。
void (*sa_restorer)(void);
};
typedef struct sigaction sigaction_t;
```

解读

- 每个信号都对应一个位。 信号从1开始编号 [1 ~ 64] 对应 `sigShare` 的[0 ~ 63]位，所以中间会差一个。记住这点，后续代码会提到。
- `sigHandler` 信号处理函数的注册过程，由系统调用 `sigaction` (用户空间) -> `OsSigAction` (内核空间)完成绑定动作。

```
#include <signal.h>
int sigaction(int signo， const struct sigaction *act， struct sigaction *oact);
int OsSigAction(int sig， const sigaction_t *act， sigaction_t *oact)
{
    UINTPTR addr;
    sigaction_t action;
    if (!GOOD_SIGNO(sig) || sig < 1 || act == NULL) {
        return -EINVAL;
    }
    //将数据从用户空间拷贝到内核空间
    if (LOS_ArchCopyFromUser(&action， act， sizeof(sigaction_t)) != LOS_OK) {
        return -EFAULT;
    }
    if (sig == SIGSYS) {//鸿蒙此处通过错误的系统调用 来安装信号处理函数，有点巧妙。
        addr = OsGetSigHandler();//获取进程信号处理函数
        if (addr == 0) {//进程没有设置信号处理函数时
            OsSetSigHandler((unsigned long)(UINTPTR)action.sa_handler);//设置进程信号处理函数——普通版
            return LOS_OK;
        }
        return -EINVAL;
    }
    return LOS_OK;
}
```

  - `sigaction(...)` 第一个参数是要安装的信号; 第二个参数与sigaction函数同名的结构体，这里会让人很懵，函数名和结构体一直，没明白为毛要这么搞？ 结构体内定义了信号处理方法;第三个为输出参数，将信号的当前的sigaction结构带回。但鸿蒙显然没有认真对待第三个参数。把 musl 实现给阉割了。
  - 对结构体的 `sigaction` 鸿蒙目前只支持信号处理函数——普通版， `sa_handler` 表示自定义信号处理函数，该函数返回值为void，可以带一个int参数，通过参数可以得知当前信号的编号，这样就可以用同一个函数处理多种信号。
  - `sa_mask` 指定信号处理程序执行过程中需要阻塞的信号。
  - `sa_flags` 字段包含一些选项，具体看注释
  - `sa_sigaction` 是实时信号的处理函数， `union` 二选一。鸿蒙暂时不支持这种方式。

信号与任务的关系

```
typedef struct {//TCB中关于信号的信息
    sig_cb          sig; //信号控制块，用于异步通信，类似于 linux singal模块
} LosTaskCB;
typedef struct {//信号控制块(描述符)
    sigset_t sigFlag;  //不屏蔽的信号标签集
    sigset_t sigPendFlag; //信号阻塞标签集，记录因哪些信号被阻塞
    sigset_t sigprocmask; /* Signals that are blocked            */ //进程屏蔽了哪些信号
    sq_queue_t sigactionq; //信号捕捉队列
    LOS_DL_LIST waitList; //等待链表，上面挂的是等待信号到来的任务， 可查找 OsTaskWait(&sigcb->waitList， timeout， TRUE) 理解
    sigset_t sigwaitmask; /* Waiting for pending signals         */ //任务在等待阻塞信号
```

```
    siginfo_t sigunbinfo; /* Signal info when task unblocked     */ //任务解锁时的信号信息
    sig_switch_context context; //信号切换上下文， 用于保存切换现场， 比如发生系统调用时的返回，涉及同一个任务的两个栈进行切换
} sig_cb;
```

解读

- 系列篇已多次说过，进程只是管理资源的容器，真正让cpu干活的是任务`task`，所以发给进程的信号最终还是需要分发给具体任务来处理。所以能想到的是关于任务部分会更复杂。
- `context`信号处理很复杂的原因在于信号的发起在用户空间，发送需要系统调用，而处理信号的函数又是用户空间提供的， 所以需要反复的切换任务上下文。而且还有硬中断的问题，比如 ctrl + c ，需要从硬中断中回调用户空间的信号处理函数，处理完了再回到内核空间，最后回到用户空间。没听懂吧，我自己都说晕了，所以需要专门的一篇来说清楚信号的处理问题。本篇不展开说。
- `sig_cb`结构体是任务处理信号的结构体，要响应，屏蔽哪些信号等等都由它完成，这个结构体虽不复杂，但是很绕，很难搞清楚它们之间的区别。笔者是经过一番痛苦的阅读理解后才明白各自的含义。并想通过用打比方的例子试图让大家明白。
- 以下用追女孩打比方理解。任务相当于某个男，没错说的就是屏幕前的你，除了苦逼的码农谁会有耐心能坚持看到这里。64个信号对应64个女孩。允许一男同时追多个女孩，女孩也可同时被多个男追。女孩也可以主动追男的。理解如下:
- `waitList`等待信号的任务链表，上面挂的是因等待信号而被阻塞的任务。众男在排队追各自心爱的女孩们，处于无所事事的挂起的状态，等待女孩们的出现。
- `sigwaitmask`任务在等待的信号集合，只有这些信号能唤醒任务。相当于列出喜欢的各位女孩，只要出现一位就能让你满血复活。
- `sigprocmask`指任务对哪些信号不感冒。来了也不处理。相当于列出不喜欢的各位女孩，请她们别来骚扰你，嘚瑟。
- `sigPendFlag`信号到达但并未唤醒任务。相当于喜欢你的女孩来追你，但她不在你喜欢的列表内，结果是不搭理人家继续等喜欢的出现。
- `sigFlag`记录不屏蔽的信号集合，相当于你并不反感的女孩们。记录来过的那些女孩(除掉你不喜欢的)。

信号发送过程

用户进程调用`kill()`的过程如下:

```
kill(pid_t pid， int sig) - 系统调用
|                  用户空间
------------------------------------------------------------------------------------
|                  内核空间
SysKill(...)
|---> OsKillLock(...)
   |---> OsKill(..， OS_USER_KILL_PERMISSION)
      |---> OsDispatch()  //鉴权，向进程发送信号
         |---> OsSigProcessSend()    //选择任务发送信号
            |---> OsSigProcessForeachChild(..，ForEachTaskCB handler，..)
               |---> SigProcessKillSigHandler() //处理 SIGKILL
                  |---> OsTaskWake() //唤醒所有等待任务
                  |---> OsSigEmptySet() //清空信号等待集
               |---> SigProcessSignalHandler()
                  |---> OsTcbDispatch() //向目标任务发送信号
                     |---> OsTaskWake() //唤醒任务
                     |---> OsSigEmptySet() //清空信号等待集
```

流程

- 通过 系统调用 `kill` 陷入内核空间
- 因为是用户态进程，使用`OS_USER_KILL_PERMISSION`权限发送信号

  ```
  #define OS_KERNEL_KILL_PERMISSION 0U //内核态 kill 权限
  #define OS_USER_KILL_PERMISSION   3U //用户态 kill 权限
  ```

- 鉴权之后进程轮询任务组，向目标任务发送信号。这里分三种情况:
  - `SIGKILL`信号，将所有等待任务唤醒，拉入就绪队列等待被调度执行，并情况信号等待集
  - 非`SIGKILL`信号时，将通过`sigwaitmask`和`sigprocmask`过滤，找到一个任务向它发送信号`OsTcbDispatch`。

代码细节

```
int OsKill(pid_t pid， int sig， int permission)
{
    siginfo_t info;
```

```
    int ret;
    /* Make sure that the para is valid */
    if (!GOOD_SIGNO(sig) || pid < 0) {//有效信号 [0，64]
        return -EINVAL;
    }
    if (OsProcessIDUserCheckInvalid(pid)) {//检查参数进程
        return -ESRCH;
    }
    /* Create the siginfo structure */ //创建信号结构体
    info.si_signo = sig; //信号编号
    info.si_code = SI_USER; //来自用户进程信号
    info.si_value.sival_ptr = NULL;
    /* Send the signal */
    ret = OsDispatch(pid， &info， permission);//发送信号
    return ret;
}
//信号分发
int OsDispatch(pid_t pid， siginfo_t *info， int permission)
{
    LosProcessCB *spcb = OS_PCB_FROM_PID(pid);//找到这个进程
    if (OsProcessIsUnused(spcb)) {//进程是否还在使用，不一定是当前进程但必须是个有效进程
        return -ESRCH;
    }
#ifdef LOSCFG_SECURITY_CAPABILITY //启用能力安全模式
    LosProcessCB *current = OsCurrProcessGet();//获取当前进程
    /* If the process you want to kill had been inactive， but still exist. should return LOS_OK */
    if (OsProcessIsInactive(spcb)) {//如果要终止的进程处于非活动状态，但仍然存在，应该返回OK
        return LOS_OK;
    }
    /* Kernel process always has kill permission and user process should check permission *///内核进程总是有kill权限，用户进程需要检查权限
    if (OsProcessIsUserMode(current) && !(current->processStatus & OS_PROCESS_FLAG_EXIT)) {//用户进程检查能力范围
        if ((current != spcb) && (!IsCapPermit(CAP_KILL)) && (current->user->userID != spcb->user->userID)) {
            return -EPERM;
        }
    }
#endif
    if ((permission == OS_USER_KILL_PERMISSION) && (OsSignalPermissionToCheck(spcb) < 0)) {
        return -EPERM;
    }
    return OsSigProcessSend(spcb， info);//给参数进程发送信号
}
//给参数进程发送参数信号
int OsSigProcessSend(LosProcessCB *spcb， siginfo_t *sigInfo)
{
    int ret;
    struct ProcessSignalInfo info = {
        .sigInfo = sigInfo， //信号内容
        .defaultTcb = NULL， //以下四个值将在OsSigProcessForeachChild中根据条件完善
        .unblockedTcb = NULL，
        .awakenedTcb = NULL，
        .receivedTcb = NULL
    };
    //总之是要从进程中找个至少一个任务来接受这个信号，优先级
    //awakenedTcb > receivedTcb > unblockedTcb > defaultTcb
    /* visit all taskcb and dispatch signal */ //访问所有任务和分发信号
    if ((info.sigInfo != NULL) && (info.sigInfo->si_signo == SIGKILL)) {//需要干掉进程时 SIGKILL = 9， #linux kill 9 14
        (void)OsSigProcessForeachChild(spcb， SigProcessKillSigHandler， &info);//进程要被干掉了，通知所有task做善后处理
        OsSigAddSet(&spcb->sigShare， info.sigInfo->si_signo);
        OsWaitSignalToWakeProcess(spcb);//等待信号唤醒进程
        return 0;
    } else {
        ret = OsSigProcessForeachChild(spcb， SigProcessSignalHandler， &info);//进程通知所有task处理信号
    }
    if (ret < 0) {
        return ret;
    }
    SigProcessLoadTcb(&info， sigInfo);
    return 0;
}
//让进程的每一个task执行参数函数
int OsSigProcessForeachChild(LosProcessCB *spcb， ForEachTaskCB handler， void *arg)
```

```
{
    int ret;
    /* Visit the main thread last (if present) */
    LosTaskCB *taskCB = NULL;//遍历进程的 threadList 链表，里面存放的都是task节点
    LOS_DL_LIST_FOR_EACH_ENTRY(taskCB， &(spcb->threadSiblingList)， LosTaskCB， threadList) {//遍历进程的任务列表
        ret = handler(taskCB， arg);//回调参数函数
        OS_RETURN_IF(ret != 0， ret);//这个宏的意思就是只有ret = 0时，啥也不处理。其余就返回 ret
    }
    return LOS_OK;
}
```

- 如果是 `SIGKILL` 信号，让 `spcb` 的所有任务执行 `SigProcessKillSigHandler` 函数，查看旗下的所有任务是否又在等待这个信号的，如果有就将任务唤醒，放在就绪队列等待被调度执行。

```
//进程收到 SIGKILL 信号后，通知任务tcb处理.
static int SigProcessKillSigHandler(LosTaskCB *tcb， void *arg)
{
    struct ProcessSignalInfo *info = (struct ProcessSignalInfo *)arg;//转参
    if ((tcb != NULL) && (info != NULL) && (info->sigInfo != NULL)) {//进程有信号
        sig_cb *sigcb = &tcb->sig;
        if (!LOS_ListEmpty(&sigcb->waitList) && OsSigIsMember(&sigcb->sigwaitmask， info->sigInfo->si_signo)) {//如果任务在等待这个信号
            OsTaskWake(tcb);//唤醒这个任务，加入进程的就绪队列，并不申请调度
            OsSigEmptySet(&sigcb->sigwaitmask);//清空信号等待位，不等任何信号了。因为这是SIGKILL信号
        }
    }
    return 0;
}
```

- 非 `SIGKILL` 信号，让 `spcb` 的所有任务执行 `SigProcessSignalHandler` 函数

```
static int SigProcessSignalHandler(LosTaskCB *tcb， void *arg)
{
    struct ProcessSignalInfo *info = (struct ProcessSignalInfo *)arg;//先把参数解出来
    int ret;
    int isMember;
    if (tcb == NULL) {
        return 0;
    }
    /* If the default tcb is not setted， then set this one as default. */
    if (!info->defaultTcb) {//如果没有默认发送方的任务，即默认参数任务。
        info->defaultTcb = tcb;
    }
    isMember = OsSigIsMember(&tcb->sig.sigwaitmask， info->sigInfo->si_signo);//任务是否在等待这个信号
    if (isMember && (!info->awakenedTcb)) {//是在等待，并尚未向该任务时发送信号时
        /* This means the task is waiting for this signal. Stop looking for it and use this tcb.
        * The requirement is: if more than one task in this task group is waiting for the signal，
        * then only one indeterminate task in the group will receive the signal.
        */
        ret = OsTcbDispatch(tcb， info->sigInfo);//发送信号，注意这是给其他任务发送信号，tcb不是当前任务
        OS_RETURN_IF(ret < 0， ret);//这种写法很有意思
        /* set this tcb as awakenedTcb */
        info->awakenedTcb = tcb;
        OS_RETURN_IF(info->receivedTcb != NULL， SIG_STOP_VISIT); /* Stop search */
    }
    /* Is this signal unblocked on this thread？ */
    isMember = OsSigIsMember(&tcb->sig.sigprocmask， info->sigInfo->si_signo);//任务是否屏蔽了这个信号
    if ((!isMember) && (!info->receivedTcb) && (tcb != info->awakenedTcb)) {//没有屏蔽，有唤醒任务没有接收任务。
        /* if unblockedTcb of this signal is not setted， then set it. */
        if (!info->unblockedTcb) {
            info->unblockedTcb = tcb;
        }
        ret = OsTcbDispatch(tcb， info->sigInfo);//向任务发送信号
        OS_RETURN_IF(ret < 0， ret);
        /* set this tcb as receivedTcb */
        info->receivedTcb = tcb;//设置这个任务为接收任务
        OS_RETURN_IF(info->awakenedTcb != NULL， SIG_STOP_VISIT); /* Stop search */
    }
    return 0; /* Keep searching */
```

```
}
```

解读

- 函数的意思是，当进程中有多个任务在等待这个信号时，发送信号给第一个等待的任务 `awakenedTcb`。
- 如果没有任务在等待信号，那就从不屏蔽这个信号的任务集中随机找一个 `receivedTcb` 接受信号。
- 只要不屏蔽 `unblockedTcb` 就有值，随机的。
- 如果上面的都不满足，信号发送给 `defaultTcb`。
- 寻找发送任务的优先级是 `awakenedTcb` > `receivedTcb` > `unblockedTcb` > `defaultTcb`

信号相关函数

信号集操作函数

- sigemptyset(sigset_t *set)：信号集全部清0；
- sigfillset(sigset_t *set)： 信号集全部置1，则信号集包含linux支持的64种信号；
- sigaddset(sigset_t *set， int signum)：向信号集中加入signum信号；
- sigdelset(sigset_t *set， int signum)：向信号集中删除signum信号；
- sigismember(const sigset_t *set， int signum)：判定信号signum是否存在信号集中。

信号阻塞函数

- sigprocmask(int how， const sigset_t *set， sigset_t *oldset))； 不同how参数，实现不同功能
  - SIG_BLOCK：将set指向信号集中的信号，添加到进程阻塞信号集；
  - SIG_UNBLOCK：将set指向信号集中的信号，从进程阻塞信号集删除；
  - SIG_SETMASK：将set指向信号集中的信号，设置成进程阻塞信号集；
- sigpending(sigset_t *set))：获取已发送到进程，却被阻塞的所有信号；
- sigsuspend(const sigset_t *mask))：用mask代替进程的原有掩码，并暂停进程执行，直到收到信号再恢复原有掩码并继续执行进程。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

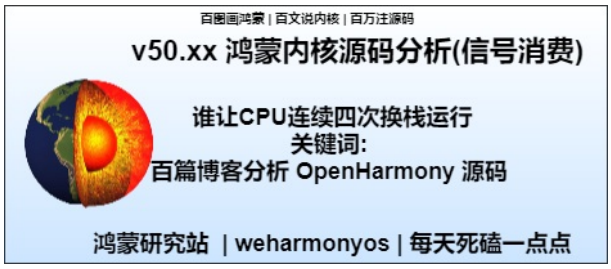
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

50_信号消费篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

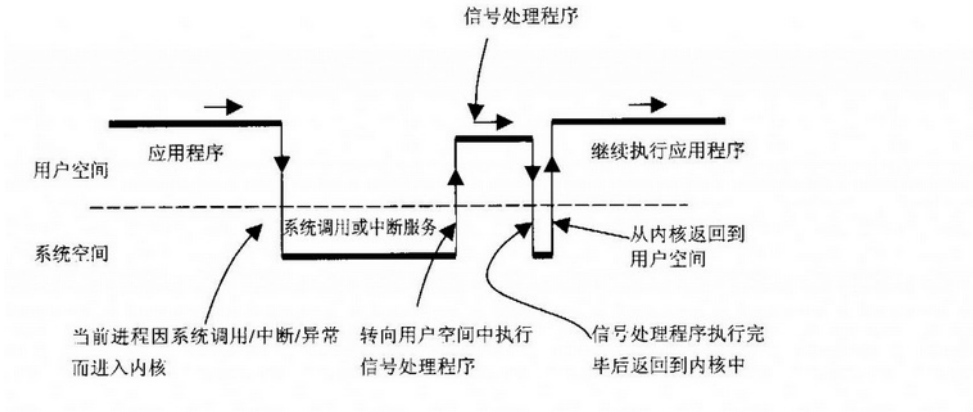
- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

信号消费

本篇为信号消费篇，读之前建议先阅读信号生产篇，信号部分姊妹篇如下:

- **v49.xx (信号生产篇) | 年过半百，依然活力十足**
- **v50.xx (信号消费篇) | 谁让CPU连续四次换栈运行**

本篇有相当的难度，涉及用户栈和内核栈的两轮切换，CPU四次换栈，寄存器改值，将围绕下图来说明。



解读

- 为本篇理解方便，把图做简化标签说明:
  - user:用户空间
  - kernel:内核空间
  - source(...):源函数

    - sighandle(...):信号处理函数，
    - syscall(...):系统调用，参数为系统调用号，如sigreturn，N(表任意)
    - user.source():表示在用户空间运行的源函数
- 系列篇已多次说过，用户态的任务有两个运行栈，一个是用户栈，一个是内核栈。栈空间分别来自用户空间和内核空间。两种空间是有严格的地址划分的，通过虚拟地址的大小就能判断出是用户空间还是内核空间。系统调用本质上是软中断，它使CPU执行指令的场地由用户栈变成内核栈。怎么变的并不复杂，就是改变(sp和cpsr寄存器的值)。sp指向哪个栈就代表在哪个栈运行， 当cpu在用户栈运行时是不能访问内核空间的，但内核态任务可以访问整个空间，而且内核态任务没有用户栈。
- 理解了上面的说明，再来说下正常系统调用流程是这样的: user.source() -> kernel。syscall(N) - > user.source() ，想要回到user.source()继续运行，就必须保存用户栈现场各寄存器的值。这些值保存在内核栈中，恢复也是从内核栈恢复。
- 信号消费的过程的上图可简化表示为: user.source() -> kernel。syscall(N) ->user。sighandle() ->kernel。syscall(sigreturn) -> user.source() 在原本要回到user.source()的中间插入了信号处理函数的调用。 这正是本篇要通过代码来说清楚的核心问题。
- 顺着这个思路可以推到以下几点，实际也是这么做的:
    - kernel。syscall(N) 中必须要再次保存user.source()的上下文`sig_switch_context`，为何已经保存了一次还要再保存一次？
    - 因为第一次是保存在内核栈中，而内核栈这部分数据会因回到用户态user。sighandle()运行而被恢复现场出栈了。保存现场/恢复现场是成双出队的好基友，注意有些文章说会把整个内核栈清空，这是不对的。
    - 第二次保存在任务结构体中，任务来源于任务池，是内核全局变量，常驻内存的。两次保存的都是user.source()运行时现场信息，再回顾下相关的结构体。关键是`sig_switch_context`

      ```
      typedef struct {
          // ...
          sig_cb  sig;//信号控制块，用于异步通信
      } LosTaskCB;
      typedef struct {//信号控制块(描述符)
          sigset_t sigFlag;  //不屏蔽的信号集
          sigset_t sigPendFlag; //信号阻塞标签集，记录那些信号来过，任务依然阻塞的集合。即:这些信号不能唤醒任务
          sigset_t sigprocmask; /* Signals that are blocked            */ //任务屏蔽了哪些信号
          sq_queue_t sigactionq; //信号捕捉队列
          LOS_DL_LIST waitList; //等待链表，上面挂的是等待信号到来的任务， 请查找 OsTaskWait(&sigcb->waitList， timeout， TRUE) 理解
          sigset_t sigwaitmask; /* Waiting for pending signals         */ //任务在等待哪些信号的到来
          siginfo_t sigunbinfo; /* Signal info when task unblocked     */ //任务解锁时的信号信息
          sig_switch_context context; //信号切换上下文， 用于保存切换现场， 比如发生系统调用时的返回，涉及同一个任务的两个栈进行切换
      } sig_cb;
      ```

    - 还必须要改变原有PC/R0/R1寄存器的值。想要执行user。sighandle()，PC寄存器就必须指向它，而R0，R1就是它的参数。
    - 信号处理完成后须回到内核态，怎么再次陷入内核态？ 答案是: `__NR_sigreturn`，这也是个系统调用。回来后还原`sig_switch_context`，即还原user.source()被打断时SP/PC等寄存器的值，使其跳回到用户栈从user.source()的被打断处继续执行。
- 有了这三个推论，再理解下面的代码就是吹灰之力了，涉及三个关键函数`OsArmA32SyscallHandle`，`OsSaveSignalContext`，`OsRestorSignalContext`本篇一一解读，彻底挖透。先看信号上下文结构体`sig_switch_context`。

sig_switch_context

```
//任务中断上下文
#define TASK_IRQ_CONTEXT \
        unsigned int R0;    \
        unsigned int R1;    \
        unsigned int R2;    \
        unsigned int R3;    \
        unsigned int R12;   \
        unsigned int USP;   \
        unsigned int ULR;   \
        unsigned int CPSR;  \
        unsigned int PC;

typedef struct {//信号切换上下文
    TASK_IRQ_CONTEXT
    unsigned int R7; //存放系统调用的ID
    unsigned int count; //记录是否保存了信号上下文
} sig_switch_context;
```

- 保存user.source()现场的结构体，`USP`，`ULR` 代表用户栈指针和返回地址。
- `CPSR` 寄存器用于设置CPU的工作模式，CPU有7种工作模式，具体可前往翻看 v36。xx (工作模式篇)谈论的用户态(`usr` 普通用户)和内核态(`sys` 超级用户)对应的只是其中的两种。二者都共用相同的寄存器。还原它就是告诉CPU内核已切到普通用户模式运行。
- 其他寄存器没有保存的原因是系统调用不会用到它们，所以不需要保存。
- `R7` 是在系统调用发生时用于记录系统调用号，在信号处理过程中，R0将获得信号编号，作为user。sighandle()的第一个参数。
- `count` 记录是否保存了信号上下文

OsArmA32SyscallHandle 系统调用总入口

```
/* The SYSCALL ID is in R7 on entry。 Parameters follow in R0。。R6 */
/*****************************************************************************
由汇编调用，见于 los_hw_exc.S    / BLX    OsArmA32SyscallHandle
SYSCALL是产生系统调用时触发的信号，R7寄存器存放具体的系统调用ID，也叫系统调用号
regs:参数就是所有寄存器
注意:本函数在用户态和内核态下都可能被调用到
//MOV     R0， SP @获取SP值，R0将作为OsArmA32SyscallHandle的参数
*****************************************************************************/
LITE_OS_SEC_TEXT UINT32 *OsArmA32SyscallHandle(UINT32 *regs)
{
    UINT32 ret;
    UINT8 nArgs;
    UINTPTR handle;
    UINT32 cmd = regs[REG_R7];//C7寄存器记录了触发了具体哪个系统调用
    if (cmd >= SYS_CALL_NUM) {//系统调用的总数
        PRINT_ERR("Syscall ID: error %d !!!\n"， cmd);
        return regs;
    }
 //用户进程信号处理函数完成后的系统调用 svc 119 #__NR_sigreturn
    if (cmd == __NR_sigreturn) {
        OsRestorSignalContext(regs);//恢复信号上下文，回到用户栈运行。
        return regs;
    }
    handle = g_syscallHandle[cmd];//拿到系统调用的注册函数，类似 SysRead
    nArgs = g_syscallNArgs[cmd / NARG_PER_BYTE]; /* 4bit per nargs */
    nArgs = (cmd & 1) ?(nArgs >> NARG_BITS):(nArgs & NARG_MASK);//获取参数个数
    if ((handle == 0) || (nArgs > ARG_NUM_7)) {//系统调用必须有参数且参数不能大于8个
        PRINT_ERR("Unsupport syscall ID: %d nArgs: %d\n"， cmd， nArgs);
        regs[REG_R0] = -ENOSYS;
        return regs;
    }
 //regs[0-6] 记录系统调用的参数，这也是由R7寄存器保存系统调用号的原因
    switch (nArgs) {//参数的个数
        case ARG_NUM_0:
        case ARG_NUM_1:
            ret = (*(SyscallFun1)handle)(regs[REG_R0]);//执行系统调用，类似 SysUnlink(pathname);
            break;
        case ARG_NUM_2://如何是两个参数的系统调用，这里传三个参数也没有问题，因被调用函数不会去取用R2值
        case ARG_NUM_3:
            ret = (*(SyscallFun3)handle)(regs[REG_R0]， regs[REG_R1]， regs[REG_R2]);//类似 SysExecve(fileName， argv， envp);
            break;
        case ARG_NUM_4:
        case ARG_NUM_5:
            ret = (*(SyscallFun5)handle)(regs[REG_R0]， regs[REG_R1]， regs[REG_R2]， regs[REG_R3]，
                                         regs[REG_R4]);
            break;
        default: //7个参数的情况
            ret = (*(SyscallFun7)handle)(regs[REG_R0]， regs[REG_R1]， regs[REG_R2]， regs[REG_R3]，
                                         regs[REG_R4]， regs[REG_R5]， regs[REG_R6]);
    }
    regs[REG_R0] = ret;//R0保存系统调用返回值
    OsSaveSignalContext(regs);//如果有信号要处理，将改写pc，r0，r1寄存器，改变返回正常用户态路径，而先去执行信号处理程序。
    /* Return the last value of curent_regs。 This supports context switches on return from the exception。
     * That capability is only used with the SYS_context_switch system call。
     */
    return regs;//返回寄存器的值
}
```

解读

- 这是系统调用的总入口，所有的系统调用都要跑这里要统一处理。通过系统号(保存在R7)，找到注册函数并回调。完成系统调用过程。
- 关于系统调用可查看 v37。xx (系统调用篇) | 系统调用到底经历了什么 本篇不详细说系统调用过程，只说跟信号相关的部分。
- `OsArmA32SyscallHandle`总体理解起来是被信号的保存和还原两个函数给包夹了。注意要在运行过程中去理解调用两个函数的过程，对于同一个任务来说，一定是先执行`OsSaveSignalContext`，第二次进入`OsArmA32SyscallHandle`后再执行`OsRestorSignalContext`。
- 看`OsSaveSignalContext`，由它负责保存user.source() 的上下文，其中改变了sp，r0/r1寄存器值，切到信号处理函数user。sighandle()运行。
- 在函数的开头，碰到系统调用号`__NR_sigreturn`，直接恢复信号上下文就退出了，因为这是要切回user.source()继续运行的操作。

```
//用户进程信号处理函数完成后的系统调用 svc 119 #__NR_sigreturn
if (cmd == __NR_sigreturn) {
    OsRestorSignalContext(regs);//恢复信号上下文，回到用户栈运行。
    return regs;
}
```

OsSaveSignalContext 保存信号上下文

有了上面的铺垫，就不难理解这个函数的作用。

```
/*****************************************************
产生系统调用时，也就是软中断时，保存用户栈寄存器现场信息
改写PC寄存器的值
*****************************************************/
void OsSaveSignalContext(unsigned int *sp)
{
    UINTPTR sigHandler;
    UINT32 intSave;
    LosTaskCB *task = NULL;
    LosProcessCB *process = NULL;
    sig_cb *sigcb = NULL;
    unsigned long cpsr;
    OS_RETURN_IF_VOID(sp == NULL);
    cpsr = OS_SYSCALL_GET_CPSR(sp);//获取系统调用时的 CPSR值
    OS_RETURN_IF_VOID(((cpsr & CPSR_MASK_MODE) != CPSR_USER_MODE));//必须工作在CPU的用户模式下，注意CPSR_USER_MODE(cpu层面)和OS_U
    SCHEDULER_LOCK(intSave);//如有不明白前往 https://my.oschina.net/weharmony 翻看工作模式/信号分发/信号处理篇
    task = OsCurrTaskGet();
    process = OsCurrProcessGet();
    sigcb = &task->sig;//获取任务的信号控制块
  //1。未保存任务上下文任务
  //2。任何的信号标签集不为空或者进程有信号要处理
    if ((sigcb->context.count == 0) && ((sigcb->sigFlag != 0) || (process->sigShare != 0))) {
        sigHandler = OsGetSigHandler();//获取信号处理函数
        if (sigHandler == 0) {//信号没有注册
            sigcb->sigFlag = 0;
            process->sigShare = 0;
            SCHEDULER_UNLOCK(intSave);
            PRINT_ERR("The signal processing function for the current process pid =%d is NULL!\n"， task->processID);
            return;
        }
        /* One pthread do the share signal */
        sigcb->sigFlag |= process->sigShare;//扩展任务的信号标签集
        unsigned int signo = (unsigned int)FindFirstSetedBit(sigcb->sigFlag) + 1;
        OsProcessExitCodeSignalSet(process， signo);//设置进程退出信号
        sigcb->context.CPSR = cpsr;  //保存状态寄存器
        sigcb->context.PC = sp[REG_PC]; //获取被打断现场寄存器的值
        sigcb->context.USP = sp[REG_SP];//用户栈顶位置，以便能从内核栈切回用户栈
        sigcb->context.ULR = sp[REG_LR];//用户栈返回地址
        sigcb->context.R0 = sp[REG_R0]; //系统调用的返回值
        sigcb->context.R1 = sp[REG_R1];
        sigcb->context.R2 = sp[REG_R2];
        sigcb->context.R3 = sp[REG_R3];
        sigcb->context.R7 = sp[REG_R7];//为何参数不用传R7，是因为系统调用发生时 R7始终保存的是系统调用号。
        sigcb->context.R12 = sp[REG_R12];//详见 https://my.oschina.net/weharmony/blog/4967613
        sp[REG_PC] = sigHandler;//指定信号执行函数，注意此处改变保存任务上下文中PC寄存器的值，恢复上下文时将执行这个函数。
        sp[REG_R0] = signo;  //参数1，信号ID
        sp[REG_R1] = (unsigned int)(UINTPTR)(sigcb->sigunbinfo。si_value。sival_ptr); //参数2
        /* sig No bits 00000100 present sig No 3， but  1<< 3 = 00001000， so signo needs minus 1 */
        sigcb->sigFlag ^= 1ULL << (signo - 1);
```

```
        sigcb->context.count++; //代表已保存
    }
    SCHEDULER_UNLOCK(intSave);
}
```

解读

- 先是判断执行条件，确实是有信号需要处理，有处理函数。自定义处理函数是由用户进程安装进来的，所有进程旗下的任务都共用，参数就是信号 `signo` ，注意可不是系统调用号，有区别的。信号编号长这样。

  ```
  #define SIGHUP    1 //终端挂起或者控制进程终止
  #define SIGINT    2 //键盘中断（ctrl + c）
  #define SIGQUIT   3 //键盘的退出键被按下
  #define SIGILL    4 //非法指令
  #define SIGTRAP   5 //跟踪陷阱（trace trap），启动进程，跟踪代码的执行
  #define SIGABRT   6 //由abort(3)发出的退出指令
  #define SIGIOT    SIGABRT //abort发出的信号
  #define SIGBUS    7 //总线错误
  #define SIGFPE    8 //浮点异常
  #define SIGKILL   9 //常用的命令 kill 9 123 | 不能被忽略、处理和阻塞
  ```

  系统调用号长这样，是不是看到一些很熟悉的函数。

  ```
  #define __NR_restart_syscall 0
  #define __NR_exit 1
  #define __NR_fork 2
  #define __NR_read 3
  #define __NR_write 4
  #define __NR_open 5
  #define __NR_close 6
  #define __NR_waitpid 7
  #define __NR_creat 8
  #define __NR_link 9
  #define __NR_unlink 10
  #define __NR_execve 11
  #define __NR_chdir 12
  #define __NR_time 13
  #define __NR_mknod 14
  #define __NR_chmod 15
  #define __NR_lchown 16
  #define __NR_break 17
  ```

- 最后是最最最关键的代码，改变pc寄存器的值，此值一变，在 `_osExceptSwiHdl` 中恢复上下文后，cpu跳到用户空间的代码段 user。sighandle(R0，R1) 开始执行，即执行信号处理函数。

  ```
  sp[REG_PC] = sigHandler;//指定信号执行函数，注意此处改变保存任务上下文中PC寄存器的值，恢复上下文时将执行这个函数。
  sp[REG_R0] = signo;  //参数1，信号ID
  sp[REG_R1] = (unsigned int)(UINTPTR)(sigcb->sigunbinfo。si_value。sival_ptr); //参数2
  ```

OsRestorSignalContext 恢复信号上下文

```
/*****************************************************
恢复信号上下文，由系统调用之__NR_sigreturn产生，这是一个内部产生的系统调用。
为什么要恢复呢？
因为系统调用的执行由任务内核态完成，使用的栈也是内核栈，CPU相关寄存器记录的都是内核栈的内容，
而系统调用完成后，需返回任务的用户栈执行，这时需将CPU各寄存器回到用户态现场
所以函数的功能就变成了还原寄存器的值
*****************************************************/
void OsRestorSignalContext(unsigned int *sp)
{
    LosTaskCB *task = NULL; /* Do not adjust this statement */
    LosProcessCB *process = NULL;
    sig_cb *sigcb = NULL;
    UINT32 intSave;
    SCHEDULER_LOCK(intSave);
```

```
    task = OsCurrTaskGet();
    sigcb = &task->sig;//获取当前任务信号控制块
    if (sigcb->context.count != 1) {//必须之前保存过，才能被恢复
        SCHEDULER_UNLOCK(intSave);
        PRINT_ERR("sig error count : %d\n"， sigcb->context.count);
        return;
    }
    process = OsCurrProcessGet();//获取当前进程
    sp[REG_PC] = sigcb->context.PC;//指令寄存器
    OS_SYSCALL_SET_CPSR(sp， sigcb->context.CPSR);//重置程序状态寄存器
    sp[REG_SP] = sigcb->context.USP;//用户栈堆栈指针， USP指的是 用户态的堆栈，即将回到用户栈继续运行
    sp[REG_LR] = sigcb->context.ULR;//返回用户栈代码执行位置
    sp[REG_R0] = sigcb->context.R0;
    sp[REG_R1] = sigcb->context.R1;
    sp[REG_R2] = sigcb->context.R2;
    sp[REG_R3] = sigcb->context.R3;
    sp[REG_R7] = sigcb->context.R7;
    sp[REG_R12] = sigcb->context.R12;
    sigcb->context.count--; //信号上下文的数量回到减少
    process->sigShare = 0; //回到用户态，信号共享清0
    OsProcessExitCodeSignalClear(process);//清空进程退出码
    SCHEDULER_UNLOCK(intSave);
}
```

解读

- 在信号处理函数完成之后，内核会触发一个 `__NR_sigreturn` 的系统调用，又陷入内核态，回到了 `OsArmA32SyscallHandle` 。
- 恢复的过程很简单，把之前保存的信号上下文恢复到内核栈sp开始位置，数据在栈中的保存顺序可查看 用栈方式篇 ，最重要的看这几句。

  ```
  sp[REG_PC] = sigcb->context.PC;//指令寄存器
  sp[REG_SP] = sigcb->context.USP;//用户栈堆栈指针， USP指的是 用户态的堆栈，即将回到用户栈继续运行
  sp[REG_LR] = sigcb->context.ULR;//返回用户栈代码执行位置
  ```

  注意这里还不是真正的切换上下文，只是改变内核栈中现有的数据。这些数据将还原给寄存器。 `USP` 和 `ULR` 指向的是用户栈的位置。一旦 `PC` ， `USP` ， `ULR` 从栈中弹出赋给寄存器。才真正完成了内核栈到用户栈的切换。回到了user.source()继续运行。

- 真正的切换汇编代码如下，都已添加注释，在保存和恢复上下文中夹着 `OsArmA32SyscallHandle`

  ```
  @ Description: Software interrupt exception handler
  _osExceptSwiHdl: @软中断异常处理，注意此时已在内核栈运行
  @保存任务上下文(TaskContext) 开始... 一定要对照TaskContext来理解
  SUB     SP， SP， #(4 * 16) @先申请16个栈空间单元用于处理本次软中断
  STMIA   SP， {R0-R12}  @Taskcontext.R[GEN_REGS_NUM] STMIA从左到右执行，先放R0 。。 R12
  MRS     R3， SPSR   @读取本模式下的SPSR值
  MOV     R4， LR    @保存回跳寄存器LR

  AND     R1， R3， #CPSR_MASK_MODE                    @ Interrupted mode 获取中断模式
  CMP     R1， #CPSR_USER_MODE                        @ User mode 是否为用户模式
  BNE     OsKernelSVCHandler                        @ Branch if not user mode 非用户模式下跳转
  @ 当为用户模式时，获取SP和LR寄出去值
  @ we enter from user mode， we need get the values of  USER mode r13(sp) and r14(lr)。
  @ stmia with ^ will return the user mode registers (provided that r15 is not in the register list)。
  MOV     R0， SP          @获取SP值，R0将作为OsArmA32SyscallHandle的参数
  STMFD   SP!， {R3}                                @ Save the CPSR 入栈保存CPSR值 => Taskcontext.regPSR
  ADD     R3， SP， #(4 * 17)                         @ Offset to pc/cpsr storage 跳到PC/CPSR存储位置
  STMFD   R3!， {R4}                                @ Save the CPSR and r15(pc) 保存LR寄存器 => Taskcontext.PC
  STMFD   R3， {R13， R14}^                           @ Save user mode r13(sp) and r14(lr) 从右向左 保存 => Taskcontext.LR和SP
  SUB     SP， SP， #4         @ => Taskcontext.resved
  PUSH_FPU_REGS R1 @保存中断模式(用户模式)
  @保存任务上下文(TaskContext) 结束
  MOV     FP， #0                                   @ Init frame pointer
  CPSIE   I @开中断，表明在系统调用期间可响应中断
  BLX     OsArmA32SyscallHandle /*交给C语言处理系统调用，参数为R0，指向TaskContext的开始位置*/
  CPSID   I @执行后续指令前必须先关中断
  @恢复任务上下文(TaskContext) 开始
  POP_FPU_REGS R1          @弹出FPU值给R1
  ADD     SP， SP，#4         @ 定位到保存旧SPSR值的位置
  ```

```
LDMFD   SP!，{R3}                       @ Fetch the return SPSR 弹出旧SPSR值
MSR     SPSR_cxsf，R3                   @ Set the return mode SPSR 恢复该模式下的SPSR值

@ we are leaving to user mode，we need to restore the values of USER mode r13(sp) and r14(lr)。
@ ldmia with ^ will return the user mode registers (provided that r15 is not in the register list)

LDMFD   SP!，{R0-R12}       @恢复R0-R12寄存器
LDMFD   SP，{R13，R14}^                 @ Restore user mode R13/R14 恢复用户模式的R13/R14寄存器
ADD     SP，SP，#(2 * 4)      @定位到保存旧PC值的位置
LDMFD   SP!，{PC}^                      @ Return to user 切回用户模式运行
@恢复任务上下文(TaskContext) 结束
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

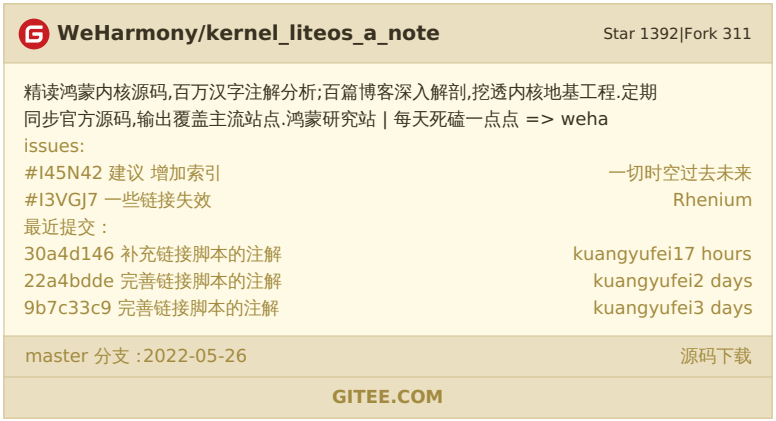
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

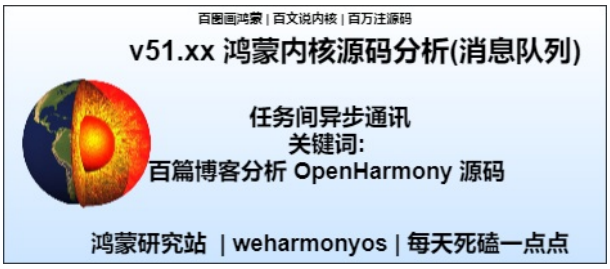


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

51_消息队列篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Litelpc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Litelpc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

本篇说清楚消息队列

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 。

基本概念

- 队列又称消息队列，是一种常用于任务间通信的数据结构。队列接收来自任务或中断的不固定长度消息，并根据不同的接口确定传递的消息是否存放在队列空间中。
- 任务能够从队列里面读取消息，当队列中的消息为空时，挂起读取任务；当队列中有新消息时，挂起的读取任务被唤醒并处理新消息。任务也能够往队列里写入消息，当队列已经写满消息时，挂起写入任务；当队列中有空闲消息节点时，挂起的写入任务被唤醒并写入消息。如果将读队列和写队列的超时时间设置为0，则不会挂起任务，接口会直接返回，这就是非阻塞模式。
- 消息队列提供了异步处理机制，允许将一个消息放入队列，但不立即处理。同时队列还有缓冲消息的作用。
- 队列用于任务间通信，可以实现消息的异步处理。同时消息的发送方和接收方不需要彼此联系，两者间是解耦的。

队列特性

- 消息以先进先出的方式排队，支持异步读写。
- 读队列和写队列都支持超时机制。
- 每读取一条消息，就会将该消息节点设置为空闲。
- 发送消息类型由通信双方约定，可以允许不同长度（不超过队列的消息节点大小）的消息。
- 一个任务能够从任意一个消息队列接收和发送消息。
- 多个任务能够从同一个消息队列接收和发送消息。
- 创建队列时所需的队列空间，默认支持接口内系统自行动态申请内存的方式，同时也支持将用户分配的队列空间作为接口入参传入的方式。

消息队列长什么样？

```
#ifndef LOSCFG_BASE_IPC_QUEUE_LIMIT
#define LOSCFG_BASE_IPC_QUEUE_LIMIT 1024 //队列个数
#endif
LITE_OS_SEC_BSS LosQueueCB *g_allQueue = NULL;//消息队列池
LITE_OS_SEC_BSS STATIC LOS_DL_LIST g_freeQueueList;//空闲队列链表，管分配的，需要队列从这里申请

typedef struct {
    UINT8 *queueHandle; /**< Pointer to a queue handle */ //指向队列句柄的指针
    UINT16 queueState; /**< Queue state */ //队列状态
    UINT16 queueLen; /**< Queue length */ //队列中消息总数的上限值，由创建时确定，不再改变
    UINT16 queueSize; /**< Node size */  //消息节点大小，由创建时确定，不再改变，即定义了每个消息长度的上限。
    UINT32 queueID; /**< queueID */   //队列ID
    UINT16 queueHead; /**< Node head */  //消息头节点位置（数组下标）
    UINT16 queueTail; /**< Node tail */  //消息尾节点位置（数组下标）
    UINT16 readWriteableCnt[OS_QUEUE_N_RW]; /**< Count of readable or writable resources， 0:readable， 1:writable */
        //队列中可写或可读消息数，0表示可读，1表示可写
    LOS_DL_LIST readWriteList[OS_QUEUE_N_RW]; /**< the linked list to be read or written， 0:readlist， 1:writelist */
        //挂的都是等待读/写消息的任务链表，0表示读消息的链表，1表示写消息的任务链表
    LOS_DL_LIST memList; /**< Pointer to the memory linked list */ //@note_why 这里尚未搞明白是啥意思 ，是共享内存吗？
} LosQueueCB;//读写队列分离
```

解读

- 和进程，线程，定时器一样，消息队列也由全局统一的消息队列池管理，池有多大？默认是1024
- 鸿蒙内核对消息的总个数有限制， queueLen 消息总数的上限，在创建队列的时候需指定，不能更改。
- 对每个消息的长度也有限制， queueSize 规定了消息的大小，也是在创建的时候指定。
- 为啥要指定总个数和每个的总大小，是因为内核一次性会把队列的总内存(queueLen * queueSize)申请下来，确保不会出现后续使用过程中内存不够的问题出现，但同时也带来了内存的浪费，因为很可能大部分时间队列并没有跑满。
- 队列的读取由 queueHead ， queueTail 管理，Head表示队列中被占用的消息节点的起始位置。Tail表示被占用的消息节点的结束位置，也是空闲消息节点的起始位置。队列刚创建时，Head和Tail均指向队列起始位置
- 写队列时，根据readWriteableCnt[1]判断队列是否可以写入，不能对已满（readWriteableCnt[1]为0）队列进行写操作。写队列支持两种写入方式：向队列尾节点写入，也可以向队列头节点写入。尾节点写入时，根据Tail找到起始空闲消息节点作为数据写入对象，如果Tail已经指向队列尾部则采用回卷方式。头节点写入时，将Head的前一个节点作为数据写入对象，如果Head指向队列起始位置则采用回卷方式。
- 读队列时，根据readWriteableCnt[0]判断队列是否有消息需要读取，对全部空闲（readWriteableCnt[0]为0）队列进行读操作会引起任务挂起。如果队列可以读取消息，则根据Head找到最先写入队列的消息节点进行读取。如果Head已经指向队列尾部则采用回卷方式。
- 删除队列时，根据队列ID找到对应队列，把队列状态置为未使用，把队列控制块置为初始状态。如果是通过系统动态申请内存方式创建的队列，还会释放队列所占内存。
- 留意readWriteList，这又是两个双向链表， 双向链表是内核最重要的结构体，牢牢的寄生在宿主结构体上。readWriteList上挂的是未来读/写消息队列的任务列表。

初始化队列

```
LITE_OS_SEC_TEXT_INIT UINT32 OsQueueInit(VOID)//消息队列模块初始化
{
    LosQueueCB *queueNode = NULL;
    UINT32 index;
    UINT32 size;

    size = LOSCFG_BASE_IPC_QUEUE_LIMIT * sizeof(LosQueueCB);//支持1024个IPC队列
    /* system resident memory， don't free */
    g_allQueue = (LosQueueCB *)LOS_MemAlloc(m_aucSysMem0， size);//常驻内存
    if (g_allQueue == NULL) {
        return LOS_ERRNO_QUEUE_NO_MEMORY;
    }
    (VOID)memset_s(g_allQueue， size， 0， size);//清0
    LOS_ListInit(&g_freeQueueList);//初始化空闲链表
    for (index = 0; index < LOSCFG_BASE_IPC_QUEUE_LIMIT; index++) {//循环初始化每个消息队列
        queueNode = ((LosQueueCB *)g_allQueue) + index;//一个一个来
        queueNode->queueID = index;//这可是 队列的身份证
        LOS_ListTailInsert(&g_freeQueueList， &queueNode->readWriteList[OS_QUEUE_WRITE]);//通过写节点挂到空闲队列链表上
    }//这里要注意是用 readWriteList 挂到 g_freeQueueList链上的，所以要通过 GET_QUEUE_LIST 来找到 LosQueueCB

    if (OsQueueDbgInitHook() != LOS_OK) {//调试队列使用的。
        return LOS_ERRNO_QUEUE_NO_MEMORY;
    }
    return LOS_OK;
```

```
}
```

解读

- 初始队列模块，对几个全局变量赋值，创建消息队列池，所有池都是常驻内存，关于池后续有专门的文章整理，到目前为止已经解除到了进程池，任务池，定时器池，队列池，==
- 将 LOSCFG_BASE_IPC_QUEUE_LIMIT 个队列挂到空闲链表 g_freeQueueList 上，供后续分配和回收。熟悉内核全局资源管理的对这种方式应该不会再陌生。

创建队列

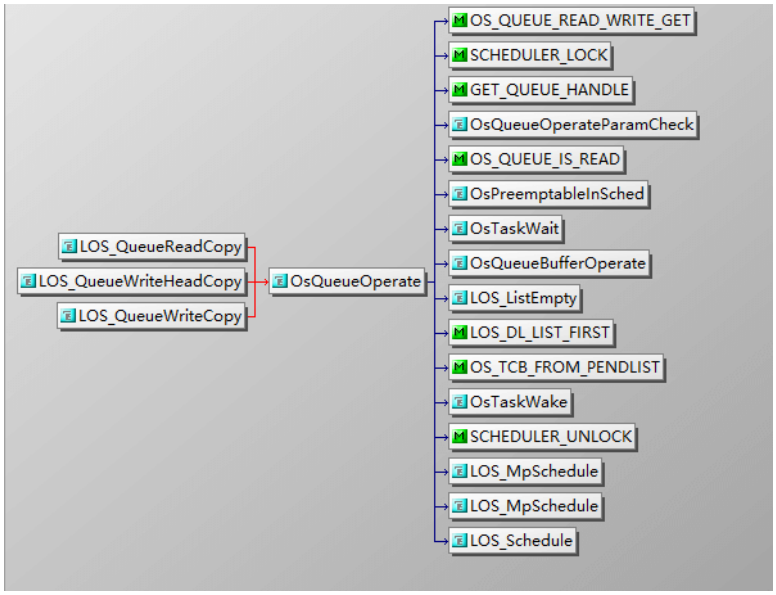
```
//创建一个队列，根据用户传入队列长度和消息节点大小来开辟相应的内存空间以供该队列使用，参数queueID带走队列ID
LITE_OS_SEC_TEXT_INIT UINT32 LOS_QueueCreate(CHAR *queueName， UINT16 len， UINT32 *queueID，
                                             UINT32 flags， UINT16 maxMsgSize)
{
    LosQueueCB *queueCB = NULL;
    UINT32 intSave;
    LOS_DL_LIST *unusedQueue = NULL;
    UINT8 *queue = NULL;
    UINT16 msgSize;

    (VOID)queueName;
    (VOID)flags;

    if (queueID == NULL) {
        return LOS_ERRNO_QUEUE_CREAT_PTR_NULL;
    }

    if (maxMsgSize > (OS_NULL_SHORT - sizeof(UINT32))) {// maxMsgSize上限 为啥要减去 sizeof(UINT32) ，因为前面存的是队列的大小
        return LOS_ERRNO_QUEUE_SIZE_TOO_BIG;
    }

    if ((len == 0) || (maxMsgSize == 0)) {
        return LOS_ERRNO_QUEUE_PARA_ISZERO;
    }

    msgSize = maxMsgSize + sizeof(UINT32);//总size = 消息体内容长度 + 消息大小(UINT32)
    /*
     * Memory allocation is time-consuming， to shorten the time of disable interrupt，
     * move the memory allocation to here。
     *///内存分配非常耗时，为了缩短禁用中断的时间，将内存分配移到此处，用的时候分配队列内存
    queue = (UINT8 *)LOS_MemAlloc(m_aucSysMem1， (UINT32)len * msgSize);//从系统内存池中分配，由这里提供读写队列的内存
    if (queue == NULL) {//这里是一次把队列要用到的所有最大内存都申请下来了，能保证不会出现后续使用过程中内存不够的问题出现
        return LOS_ERRNO_QUEUE_CREATE_NO_MEMORY;//调用处有 OsSwtmrInit sys_mbox_new DoMqueueCreate ==
    }

    SCHEDULER_LOCK(intSave);
    if (LOS_ListEmpty(&g_freeQueueList)) {//没有空余的队列ID的处理，注意软时钟定时器是由 g_swtmrCBArray统一管理的，里面有正在使用和可分配空闲的队
        SCHEDULER_UNLOCK(intSave);//g_freeQueueList是管理可用于分配的队列链表，申请消息队列的ID需要向它要
        OsQueueCheckHook();
        (VOID)LOS_MemFree(m_aucSysMem1， queue);//没有就要释放 queue申请的内存
        return LOS_ERRNO_QUEUE_CB_UNAVAILABLE;
    }

    unusedQueue = LOS_DL_LIST_FIRST(&g_freeQueueList);//找到一个没有被使用的队列
    LOS_ListDelete(unusedQueue);//将自己从g_freeQueueList中摘除， unusedQueue只是个 LOS_DL_LIST 结点。
    queueCB = GET_QUEUE_LIST(unusedQueue);//通过unusedQueue找到整个消息队列(LosQueueCB)
    queueCB->queueLen = len; //队列中消息的总个数，注意这个一旦创建是不能变的。
    queueCB->queueSize = msgSize;//消息节点的大小，注意这个一旦创建也是不能变的。
    queueCB->queueHandle = queue; //队列句柄，队列内容存储区。
    queueCB->queueState = OS_QUEUE_INUSED; //队列状态使用中
    queueCB->readWriteableCnt[OS_QUEUE_READ] = 0;//可读资源计数，OS_QUEUE_READ(0):可读。
    queueCB->readWriteableCnt[OS_QUEUE_WRITE] = len;//可些资源计数 OS_QUEUE_WRITE(1):可写， 默认len可写。
    queueCB->queueHead = 0;//队列头节点
    queueCB->queueTail = 0;//队列尾节点
    LOS_ListInit(&queueCB->readWriteList[OS_QUEUE_READ]);//初始化可读队列链表
    LOS_ListInit(&queueCB->readWriteList[OS_QUEUE_WRITE]);//初始化可写队列链表
    LOS_ListInit(&queueCB->memList);//
```

```
    OsQueueDbgUpdateHook(queueCB->queueID， OsCurrTaskGet()->taskEntry);//在创建或删除队列调试信息时更新任务条目
    SCHEDULER_UNLOCK(intSave);

    *queueID = queueCB->queueID;//带走队列ID
    return LOS_OK;
}
```

解读

- 创建和初始化一个 `LosQueueCB`
- 动态分配内存来保存消息内容， `LOS_MemAlloc(m_aucSysMem1， (UINT32)len * msgSize);`
- `msgSize = maxMsgSize + sizeof(UINT32);` 头四个字节放消息的长度，但消息最大长度不能超过 `maxMsgSize`
- `readWriteableCnt` 记录读/写队列的数量，独立计算
- `readWriteList` 挂的是等待读取队列的任务链表 将在 `OsTaskWait(&queueCB->readWriteList[readWrite]， timeout， TRUE);` 中将任务挂到链表上。

关键函数OsQueueOperate



队列的读写都要经过 `OsQueueOperate`

```
/*****************************************************
队列操作。是读是写由operateType定
本函数是消息队列最重要的一个函数，可以分析出读取消息过程中
发生的细节，涉及任务的唤醒和阻塞，阻塞链表任务的相互提醒。
*****************************************************/
UINT32 OsQueueOperate(UINT32 queueID， UINT32 operateType， VOID *bufferAddr， UINT32 *bufferSize， UINT32 timeout)
{
    LosQueueCB *queueCB = NULL;
    LosTaskCB *resumedTask = NULL;
    UINT32 ret;
    UINT32 readWrite = OS_QUEUE_READ_WRITE_GET(operateType);//获取读/写操作标识
    UINT32 intSave;

    SCHEDULER_LOCK(intSave);
    queueCB = (LosQueueCB *)GET_QUEUE_HANDLE(queueID);//获取对应的队列控制块
    ret = OsQueueOperateParamCheck(queueCB， queueID， operateType， bufferSize);//参数检查
    if (ret != LOS_OK) {
        goto QUEUE_END;
    }

    if (queueCB->readWriteableCnt[readWrite] == 0) {//根据readWriteableCnt判断队列是否有消息读/写
        if (timeout == LOS_NO_WAIT) {//不等待直接退出
            ret = OS_QUEUE_IS_READ(operateType) ? LOS_ERRNO_QUEUE_ISEMPTY : LOS_ERRNO_QUEUE_ISFULL;
            goto QUEUE_END;
        }
```

```
        if (!OsPreemptableInSched()) {//不支持抢占式调度
            ret = LOS_ERRNO_QUEUE_PEND_IN_LOCK;
            goto QUEUE_END;
        }
    //任务等待，这里很重要啊，将自己从就绪列表摘除，让出了CPU并发起了调度，并挂在readWriteList[readWrite]上，挂的都等待读/写消息的task
        ret = OsTaskWait(&queueCB->readWriteList[readWrite]， timeout， TRUE);//任务被唤醒后会回到这里执行，什么时候会被唤醒？当然是有消息的时候
        if (ret == LOS_ERRNO_TSK_TIMEOUT) {//唤醒后如果超时了，返回读/写消息失败
            ret = LOS_ERRNO_QUEUE_TIMEOUT;
            goto QUEUE_END;//
        }
    } else {
        queueCB->readWriteableCnt[readWrite]--;//对应队列中计数器--，说明一条消息只能被读/写一次
    }

    OsQueueBufferOperate(queueCB， operateType， bufferAddr， bufferSize);//发起读或写队列操作

    if (!LOS_ListEmpty(&queueCB->readWriteList[!readWrite])) {//如果还有任务在排着队等待读/写入消息(当时不能读/写的原因有可能当时队列满了==)
        resumedTask = OS_TCB_FROM_PENDLIST(LOS_DL_LIST_FIRST(&queueCB->readWriteList[!readWrite]));//取出要读/写消息的任务
        OsTaskWake(resumedTask);//唤醒任务去读/写消息啊
        SCHEDULER_UNLOCK(intSave);
        LOS_MpSchedule(OS_MP_CPU_ALL);//让所有CPU发出调度申请，因为很可能那个要读/写消息的队列是由其他CPU执行
        LOS_Schedule();//申请调度
        return LOS_OK;
    } else {
        queueCB->readWriteableCnt[!readWrite]++;//对应队列读/写中计数器++
    }

QUEUE_END:
    SCHEDULER_UNLOCK(intSave);
    return ret;
}
```

解读

- `queueID` 指定操作消息队列池中哪个消息队列
- `operateType` 表示本次是是读/写消息
- `bufferAddr` ， `bufferSize` 表示如果读操作，用buf接走数据，如果写操作，将buf写入队列。
- `timeout` 只用于当队列中没有读/写内容时的等待。
  - 当读消息时发现队列中没有可读的消息，此时timeout决定是否将任务挂入等待读队列任务链表
  - 当写消息时发现队列中没有空间用于写的消息，此时timeout决定是否将任务挂入等待写队列任务链表
- `if (!LOS_ListEmpty(&queueCB->readWriteList[!readWrite]))` 最有意思的是这行代码。
  - 在一次读消息完成后会立即唤醒写队列任务链表的任务，因为读完了就有了剩余空间，等待写队列的任务往往是因为没有空间而进入等待状态。
  - 在一次写消息完成后会立即唤醒读队列任务链表的任务，因为写完了队列就有了新消息，等待读队列的任务往往是因为队列中没有消息而进入等待状态。

编程实例

创建一个队列，两个任务。任务1调用写队列接口发送消息，任务2通过读队列接口接收消息。

- 通过LOS_TaskCreate创建任务1和任务2。
- 通过LOS_QueueCreate创建一个消息队列。
- 在任务1 send_Entry中发送消息。
- 在任务2 recv_Entry中接收消息。
- 通过LOS_QueueDelete删除队列。

```
#include "los_task.h"
#include "los_queue.h"
static UINT32 g_queue;
#define BUFFER_LEN 50
VOID send_Entry(VOID)
{
    UINT32 i = 0;
    UINT32 ret = 0;
    CHAR abuf[] = "test is message x";
    UINT32 len = sizeof(abuf);
```

```
    while (i < 5) {
        abuf[len -2] = '0' + i;
        i++;

        ret = LOS_QueueWriteCopy(g_queue , abuf , len , 0);
        if(ret != LOS_OK) {
            dprintf("send message failure , error: %x\n" , ret);
        }
        LOS_TaskDelay(5);
    }
}
VOID recv_Entry(VOID)
{
    UINT32 ret = 0;
    CHAR readBuf[BUFFER_LEN] = {0};
    UINT32 readLen = BUFFER_LEN;
    while (1) {
        ret = LOS_QueueReadCopy(g_queue , readBuf , &readLen , 0);
        if(ret != LOS_OK) {
            dprintf("recv message failure , error: %x\n" , ret);
            break;
        }
        dprintf("recv message: %s\n" , readBuf);
        LOS_TaskDelay(5);
    }
    while (LOS_OK != LOS_QueueDelete(g_queue)) {
        LOS_TaskDelay(1);
    }
    dprintf("delete the queue success!\n");
}
UINT32 Example_CreateTask(VOID)
{
    UINT32 ret = 0;
    UINT32 task1 , task2;
    TSK_INIT_PARAM_S initParam;
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)send_Entry;
    initParam.usTaskPrio = 9;
    initParam.uwStackSize = LOS_TASK_MIN_STACK_SIZE;
    initParam.pcName = "sendQueue";
#ifdef LOSCFG_KERNEL_SMP
    initParam.usCpuAffiMask = CPUID_TO_AFFI_MASK(ArchCurrCpuid());
#endif
    initParam.uwResved = LOS_TASK_STATUS_DETACHED;
    LOS_TaskLock();
    ret = LOS_TaskCreate(&task1 , &initParam);
    if(ret != LOS_OK) {
        dprintf("create task1 failed , error: %x\n" , ret);
        return ret;
    }
    initParam.pcName = "recvQueue";
    initParam.pfnTaskEntry = (TSK_ENTRY_FUNC)recv_Entry;
    ret = LOS_TaskCreate(&task2 , &initParam);
    if(ret != LOS_OK) {
        dprintf("create task2 failed , error: %x\n" , ret);
        return ret;
    }
    ret = LOS_QueueCreate("queue" , 5 , &g_queue , 0 , BUFFER_LEN);
    if(ret != LOS_OK) {
        dprintf("create queue failure , error: %x\n" , ret);
    }
    dprintf("create the queue success!\n");
    LOS_TaskUnlock();
    return ret;
}
```

结果验证

```
create the queue success!
recv message: test is message 0
```

```
recv message: test is message 1
recv message: test is message 2
recv message: test is message 3
recv message: test is message 4
recv message failure， error: 200061d
delete the queue success!
```

总结

- 消息队列解决任务间大数据的传递
- 以一种异步，解耦的方式实现任务通讯
- 全局由消息队列池统一管理
- 在创建消息队列时申请内存块存储消息内存。
- 读/写操作统一管理，分开执行，A任务 `读/写` 完消息后会立即唤醒等待 `写/读` 的B任务。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

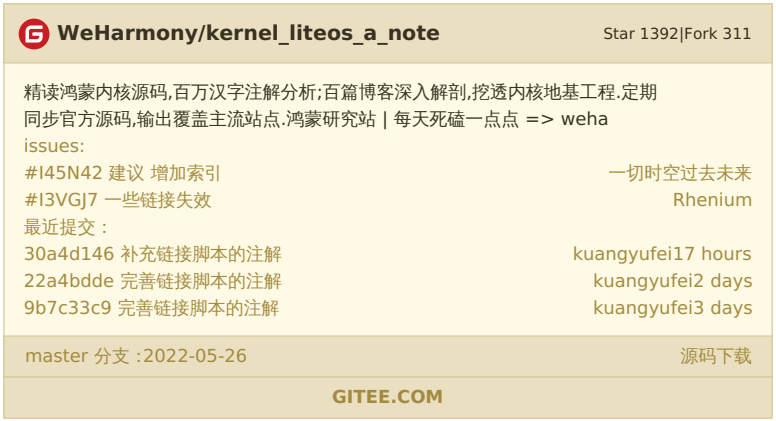
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

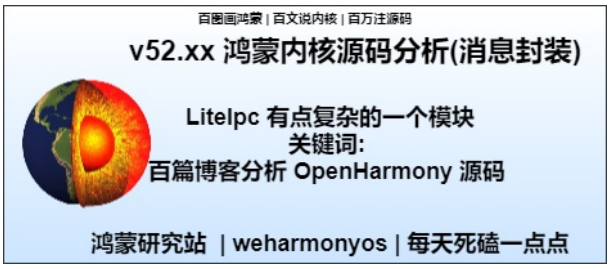


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

52_消息封装篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

基本概念

LiteIPC 是 OpenHarmony LiteOS-A 内核提供的一种新型 IPC （Inter-Process Communication，即进程间通信）机制，为轻量级进程间通信组件，为面向服务的系统服务框架提供进程间通信能力，分为内核实现和用户态实现两部分，其中内核实现完成进程间消息收发、IPC内存管理、超时通知和死亡通知等功能；用户态提供序列化和反序列化能力，并完成 IPC 回调消息和死亡消息的分发。

我们主要讲解内核态实现部分，本想一篇说完，但发现它远比想象中的复杂和重要，所以分两篇说，通讯内容和通讯机制。下篇可翻看 **鸿蒙内核源码分析(消息映射篇) | 剖析Liteipc(下)进程通讯机制** ，通讯的内容就是消息，围绕着消息展开的结构体多达 10 几个，捋不清它们之间的关系肯定是搞不懂通讯的机制，所以咱们得先搞清楚关系再说流程。下图是笔者读完 LiteIPC 模块后绘制的消息封装图，可以说 LiteIPC 是内核涉及结构体最多的模块，请消化理解，本篇将围绕它展开。

系列篇多次提过，内核的每个模块都至少围绕着一个重要结构体展开，抓住了它顺瓜摸藤就能把细节抹的清清楚楚，于 LiteIPC ，这个结构体就是 IpcMsg 。

运行机制

```
typedef struct {//IPC 消息结构体
    MsgType        type;        /**< cmd type, decide the data structure below | 命令类型，决定下面的数据结构*/
    SvcIdentity    target;      /**< serviceHandle or targetTaskId, depending on type | 因命令类型不同而异*/
    UINT32         code;       /**< service function code | 服务功能代码*/
    UINT32         flag;  ///< 标签
#if (USE_TIMESTAMP == 1)
    UINT64         timestamp; ///< 时间戳,用于验证
#endif
    UINT32         dataSz;     /**< size of data | 消息内容大小*/
    VOID           *data;  ///< 消息的内容,真正要传递的消息,这个数据内容是指spObjNum个数据的内容,定位就靠offsets
    UINT32         spObjNum; ///< 对象数量, 例如 spObjNum = 3时,offsets = [0,35,79],代表从data中读取 0 - 35给第一个对象,依次类推
    VOID           *offsets; ///< 偏移量,注意这里有多少个spObjNum就会有多少个偏移量,详见 CopyDataFromUser 来理解
    UINT32         processID;  /**< filled by kernel, processId of sender/reciever | 由内核提供,发送/接收消息的进程ID*/
    UINT32         taskID;     /**< filled by kernel, taskId of sender/reciever | 由内核提供,发送/接收消息的任务ID*/
#ifdef LOSCFG_SECURITY_CAPABILITY
```

```
    UINT32        userID;  ///< 用户ID
    UINT32        gid;   ///< 组ID
#endif
} IpcMsg;
```

解读

- 第一个 type ，通讯的本质就是你来我往，异常当然也要考虑

```
typedef enum {
    MT_REQUEST, ///< 请求
    MT_REPLY, ///< 回复
    MT_FAILED_REPLY,///< 回复失败
    MT_DEATH_NOTIFY,///< 通知死亡
    MT_NUM
} MsgType;
```

- 第二个 target ， LiteIPC 中有两个主要概念，一个是 ServiceManager ，另一个是 Service 。整个系统只能有一个 ServiceManager ，而 Service 可以有多个。 ServiceManager 有两个主要功能：一是负责 Service 的注册和注销，二是负责管理 Service 的访问权限（只有有权限的任务 Task 可以向对应的 Service 发送 IPC 消息）。首先将需要接收 IPC 消息的任务通过 ServiceManager 注册成为一个 Service ，然后通过 ServiceManager 为该 Service 任务配置访问权限，即指定哪些任务可以向该 Service 任务发送 IPC 消息。 LiteIPC 的核心思想就是在内核态为每个 Service 任务维护一个 IPC 消息队列，该消息队列通过 LiteIPC 设备文件向上层用户态程序分别提供代表收取 IPC 消息的读操作和代表发送 IPC 消息的写操作。

```
/// SVC(service)服务身份证
typedef struct {
    UINT32        handle;  //service 服务ID, 范围[0,最大任务ID]
    UINT32        token; //由应用层带入
    UINT32        cookie; //由应用层带入
} SvcIdentity;
```

- code ， timestamp 由应用层设定，用于确保回复正确有效，详见 CheckRecievedMsg
- dataSz ， data ， spObjNum ， offsets 这四个需连在一起理解，是重中之重。其实消息又分成三种类型(对象)

```
typedef enum {
    OBJ_FD,  ///< 文件句柄
    OBJ_PTR, ///< 指针
    OBJ_SVC  ///< 服务,用于设置权限
} ObjType;
typedef union {
    UINT32      fd;  ///< 文件描述符
    BuffPtr     ptr; ///< 缓存的开始地址,即:指针,消息从用户空间来时,要将内容拷贝到内核空间
    SvcIdentity  svc; ///< 服务,用于设置访问权限
} ObjContent;
typedef struct { // IpcMsg->data 包含三种子消息,也要将它们读到内核空间
    ObjType     type; ///< 类型
    ObjContent  content;///< 内容
} SpecialObj;
```

这三种对象都打包在 data 中,总长度是 dataSz ， spObjNum 表示个数， offsets 是个整型数组，标记了对应第几个对象在 data 中的位置，这样就很容易从 data 读到对象的数据。 UINT32 fd 类型对象通讯的实现是通过两个进程间共享同一个 fd 来实现通讯，具体实现函数为 HandleFd 。

```
/// 按句柄方式处理, 参数 processID 往往不是当前进程
LITE_OS_SEC_TEXT STATIC UINT32 HandleFd(UINT32 processID, SpecialObj *obj, BOOL isRollback)
{
    int ret;
    if (isRollback == FALSE) { // 不回滚
        ret = CopyFdToProc(obj->content.fd, processID);//目的是将两个不同进程fd都指向同一个系统fd,共享FD的感觉
        if (ret < 0) {//返回 processID 的 新 fd
            return ret;
        }
        obj->content.fd = ret; // 记录 processID 的新FD, 可用于回滚
    } else {// 回滚时关闭进程FD
        ret = CloseProcFd(obj->content.fd, processID);
        if (ret < 0) {
            return ret;
        }
```

```
    }
```

`SvcIdentity svc` 用于设置进程<->任务之间彼此访问权限，具体实现函数为 `HandleSvc` 。

```
/// 按服务的方式处理,此处推断 Svc 应该是 service 的简写 @note_thinking
LITE_OS_SEC_TEXT STATIC UINT32 HandleSvc(UINT32 dstTid, const SpecialObj *obj, BOOL isRollback)
{
    UINT32 taskID = 0;
    if (isRollback == FALSE) {
        if (IsTaskAlive(obj->content.svc.handle) == FALSE) {
            PRINT_ERR("Liteipc HandleSvc wrong svctid\n");
            return -EINVAL;
        }
        if (HasServiceAccess(obj->content.svc.handle) == FALSE) {
            PRINT_ERR("Liteipc %s, %d\n", __FUNCTION__, __LINE__);
            return -EACCES;
        }
        if (GetTid(obj->content.svc.handle, &taskID) == 0) {//获取参数消息服务ID所属任务
            if (taskID == OS_PCB_FROM_PID(OS_TCB_FROM_TID(taskID)->processID)->ipcInfo->ipcTaskID) {//如果任务ID一样,即任务ID为ServiceMa
                AddServiceAccess(dstTid, obj->content.svc.handle);
            }
        }
    }
    return LOS_OK;
}
```

`BuffPtr ptr` 是通过指针传值，具体实现函数为 `HandlePtr` ，对应结构体为 `BuffPtr` 。

```
typedef struct {
    UINT32        buffSz;  ///< 大小
    VOID          *buff; ///< 内容 内核需要将内容从用户空间拷贝到内核空间的动作
} BuffPtr;
/// 按指针方式处理
LITE_OS_SEC_TEXT STATIC UINT32 HandlePtr(UINT32 processID, SpecialObj *obj, BOOL isRollback)
{
    VOID *buf = NULL;
    UINT32 ret;
    if ((obj->content.ptr.buff == NULL) || (obj->content.ptr.buffSz == 0)) {
        return -EINVAL;
    }
    if (isRollback == FALSE) {
        if (LOS_IsUserAddress((vaddr_t)(UINTPTR)(obj->content.ptr.buff)) == FALSE) { // 判断是否为用户空间地址
            PRINT_ERR("Liteipc Bad ptr address\n"); //不在用户空间时
            return -EINVAL;
        }
        buf = LiteIpcNodeAlloc(processID, obj->content.ptr.buffSz);//在内核空间分配内存接受来自用户空间的数据
        if (buf == NULL) {
            PRINT_ERR("Liteipc DealPtr alloc mem failed\n");
            return -EINVAL;
        }
        ret = copy_from_user(buf, obj->content.ptr.buff, obj->content.ptr.buffSz);//从用户空间拷贝数据到内核空间
        if (ret != LOS_OK) {
            LiteIpcNodeFree(processID, buf);
            return ret;
        }//这里要说明下 obj->content.ptr.buff的变化,虽然都是用户空间的地址,但第二次已经意义变了,虽然数据一样,但指向的是申请经过拷贝后的内核空间
        obj->content.ptr.buff = (VOID *)GetIpcUserAddr(processID, (INTPTR)buf);//获取进程 processID的用户空间地址,如此用户空间操作buf其实操作
        EnableIpcNodeFreeByUser(processID, (VOID *)buf);//创建一个IPC节点,挂到可使用链表上,供读取
    } else {
        (VOID)LiteIpcNodeFree(processID, (VOID *)GetIpcKernelAddr(processID, (INTPTR)obj->content.ptr.buff));//在内核空间释放IPC节点
    }
    return LOS_OK;
}
```

- `processID` 和 `taskID` 则由内核填充，应用层是感知不到进程和任务的，暴露给它是服务ID， `SvcIdentity.handle` ，上层使用时只需向服务发送/读取消息，而服务是由内核创建，绑定在任务和进程上。所以只要有服务ID就能查询到对应的进程和任务ID。
- `userID` 和 `gid` 涉及用户和组安全模块，请查看系列相关篇。

进程和任务

再说两个结构体 `ProcIpcInfo` ， `IpcTaskInfo` `LiteIPC` 实现的是进程间的通讯，所以在进程控制块中肯定有它的位置存在，即: `ProcIpcInfo` 。

```
typedef struct {
    IpcPool pool;    ///< ipc内存池,IPC操作所有涉及内核空间分配的内存均有此池提供
    UINT32 ipcTaskID;   ///< 指定能ServiceManager的任务ID
    LOS_DL_LIST ipcUsedNodelist;///< 已使用节点链表,上面挂 IpcUsedNode 节点, 申请IpcUsedNode的内存来自内核堆空间
    UINT32 access[LOSCFG_BASE_CORE_TSK_LIMIT]; ///< 允许进程通过IPC访问哪些任务
} ProcIpcInfo;
```

而进程只是管家，真正让内核忙飞的是任务，在任务控制块中也应有 `LiteIPC` 一席之地，即: `IpcTaskInfo` 。

```
typedef struct {
    LOS_DL_LIST     msgListHead;///< 上面挂的是一个个的 ipc节点 上面挂 IpcListNode,申请IpcListNode的内存来自进程IPC内存池
    BOOL            accessMap[LOSCFG_BASE_CORE_TSK_LIMIT]; ///< 此处是不是应该用 LOSCFG_BASE_CORE_PROCESS_LIMIT ? @note_thinking
        ///< 任务是否可以给其他进程发送IPC消息
} IpcTaskInfo;
```

两个结构体不复杂，把发送/回复的消息挂到对应的链表上，并提供进程<->任务间彼此访问权限功能 `access` ， `accessMap` ，由谁来设置权限呢？上面已经说过了是 `HandleSvc` 。

IPC内存池

还有最后一个结构体 `IpcPool` ，

```
typedef struct {//用户空间和内核空间的消息传递通过偏移量计算
    VOID   *uvaddr; ///< 用户空间地址,由kvaddr映射而来的地址,这两个地址的关系一定要搞清楚,否则无法理解IPC的核心思想
    VOID   *kvaddr; ///< 内核空间地址,IPC申请的是内核空间,但是会通过 DoIpcMmap 将这个地址映射到用户空间
    UINT32 poolSize; ///< ipc池大小
} IpcPool;
```

它是 `LiteIPC` 实现通讯机制的基础，是内核设计很巧妙的地方，实现了在用户态读取内核态数据的功能。请想想它是如何做到的？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

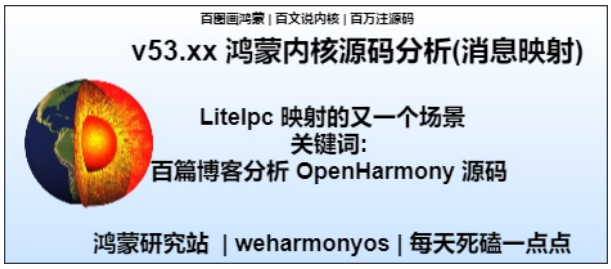
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

53_消息映射篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析Liteipc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析Liteipc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

基本概念

LiteIPC 是 OpenHarmony LiteOS-A 内核提供的一种新型 IPC （Inter-Process Communication，即进程间通信）机制，为轻量级进程间通信组件，为面向服务的系统服务框架提供进程间通信能力，分为内核实现和用户态实现两部分，其中内核实现完成进程间消息收发、IPC内存管理、超时通知和死亡通知等功能；用户态提供序列化和反序列化能力，并完成 IPC 回调消息和死亡消息的分发。

我们主要讲解内核态实现部分，本想一篇说完，但发现它远比想象中的复杂和重要，所以分通讯内容和通讯机制上下两篇说。上篇可翻看 **鸿蒙内核源码分析(消息封装篇) | 剖析Liteipc(上)进程通讯内容** ，本篇为通讯机制,介绍liteipc在内核层的实现过程。

空间映射

映射 一词在系列篇中多次出现，虚拟地址的基础就是**映射**，共享内存的实现也要靠**映射**， LiteIPC 通讯的底层实现也离不开**映射**，有意思的是将用户态的线性区和内核态的线性区进行了映射。也就是说当用户访问用户空间中的某个虚拟地址时，其实和内核空间某个虚拟地址都指向了同一个物理内存地址。可能有人会问这也行？在了解完物理地址，内核虚地址，用户虚地址之间的关系后就会明白这当然可以。

- 虚拟地址包括内核空间地址和用户进程空间地址，它们的范围对外暴露，由系统集成商设定，内核也专门提供了判断函数。即看到一个虚拟地址可以知道是内核在用还是用户(应用)进程在用。

```
/// 虚拟地址是否在内核空间
 STATIC INLINE BOOL LOS_IsKernelAddress(VADDR_T vaddr)
 {
    return ((vaddr >= (VADDR_T)KERNEL_ASPACE_BASE) &&
            (vaddr <= ((VADDR_T)KERNEL_ASPACE_BASE + ((VADDR_T)KERNEL_ASPACE_SIZE - 1))));
 }
 /// 虚拟地址是否在用户空间
 STATIC INLINE BOOL LOS_IsUserAddress(VADDR_T vaddr)
 {
    return ((vaddr >= USER_ASPACE_BASE) &&
            (vaddr <= (USER_ASPACE_BASE + (USER_ASPACE_SIZE - 1))));
 }
```

- 物理地址是由物理内存提供，系统集成商根据实际的物理内存大小来设定地址范围。至于具体的某段物理内存是给内核空间还是给用户空间使用并没有要求，所谓映射是指虚拟地址 <--> 物理地址的映射，是 N:1的关系，一个物理地址可以被多个虚拟地址映射，而一个虚拟地址只能映射到一个物理地址。
- 以上是 LiteIPC 的实现概念基础，明白后就不难理解结构体 IpcPool 存在的意义了

```
/**
* @struct IpcPool | ipc池
* @brief  LiteIPC的核心思想就是在内核态为每个Service任务维护一个IPC消息队列，该消息队列通过LiteIPC设备文件向上层
* 用户态程序分别提供代表收取IPC消息的读操作和代表发送IPC消息的写操作。
*/
typedef struct {
    VOID   *uvaddr; ///< 用户态空间地址,由kvaddr映射而来的地址,这两个地址的关系一定要搞清楚,否则无法理解IPC的核心思想
    VOID   *kvaddr; ///< 内核态空间地址,IPC申请的是内核空间,但是会通过 DoIpcMmap 将这个地址映射到用户空间
    UINT32 poolSize; ///< ipc池大小
} IpcPool;
```

文件访问

LiteIPC 的运行机制是首先将需要接收 IPC 消息的任务通过 ServiceManager 注册成为一个 Service ，然后通过 ServiceManager 为该 Service 任务配置访问权限，即指定哪些任务可以向该 Service 任务发送 IPC 消息。 LiteIPC 的核心思想就是在内核态为每个 Service 任务维护一个 IPC 消息队列，该消息队列通过 LiteIPC 设备文件向上层用户态程序分别提供代表收取 IPC 消息的读操作和代表发送 IPC 消息的写操作。 设备文件的接口层(VFS)实现为 g_liteIpcFops ，跟踪这几个函数就能够整明白整个实现 LiteIPC 过程

```
#define LITEIPC_DRIVER "/dev/lite_ipc" ///< 虚拟设备,文件访问读取
STATIC const struct file_operations_vfs g_liteIpcFops = {
    .open = LiteIpcOpen,   /* open | 创建Ipc内存池*/
    .close = LiteIpcClose,  /* close */
    .ioctl = LiteIpcIoctl,  /* ioctl | 包含读写操作 */
    .mmap = LiteIpcMmap,   /* mmap | 实现线性区映射*/
};
```

LiteIpcOpen | 创建消息内存池

```
LITE_OS_SEC_TEXT STATIC int LiteIpcOpen(struct file *filep)
{
    LosProcessCB *pcb = OsCurrProcessGet();
    if (pcb->ipcInfo != NULL) {
        return 0;
    }
    pcb->ipcInfo = LiteIpcPoolCreate();
    if (pcb->ipcInfo == NULL) {
        return -ENOMEM;
    }
    return 0;
}
///创建IPC消息内存池
LITE_OS_SEC_TEXT_INIT STATIC ProcIpcInfo *LiteIpcPoolCreate(VOID)
{
    ProcIpcInfo *ipcInfo = LOS_MemAlloc(m_aucSysMem1, sizeof(ProcIpcInfo));//从内核堆内存中申请IPC控制体
    if (ipcInfo == NULL) {
        return NULL;
    }
    (VOID)memset_s(ipcInfo, sizeof(ProcIpcInfo), 0, sizeof(ProcIpcInfo));
    (VOID)LiteIpcPoolInit(ipcInfo);
    return ipcInfo;
}
```

解读

- 进来先获取当前进程 OsCurrProcessGet() ，即为每个进程创建唯一的 IPC 消息控制体， ProcIpcInfo 在进程控制块中，负责管理 IPC 消息
- 初始化消息内存池，此处只申请结构体本身占用内存，真正的内存池在 LiteIpcMmap 中完成

LiteIpcMmap | 映射

```
///将参数线性区设为IPC专用区
LITE_OS_SEC_TEXT STATIC int LiteIpcMmap(struct file *filep, LosVmMapRegion *region)
{
    int ret = 0;
    LosVmMapRegion *regionTemp = NULL;
    LosProcessCB *pcb = OsCurrProcessGet();
    ProcIpcInfo *ipcInfo = pcb->ipcInfo;
 //被映射的线性区不能在常量和私有数据区
    if ((ipcInfo == NULL) || (region == NULL) || (region->range.size > LITE_IPC_POOL_MAX_SIZE) ||
        (!LOS_IsRegionPermUserReadOnly(region)) || (!LOS_IsRegionFlagPrivateOnly(region))) {
        ret = -EINVAL;
        goto ERROR_REGION_OUT;
    }
    if (IsPoolMapped(ipcInfo)) {//已经用户空间和内核空间之间存在映射关系了
        return -EEXIST;
    }
    if (ipcInfo->pool.uvaddr != NULL) {//ipc池已在进程空间有地址
        regionTemp = LOS_RegionFind(pcb->vmSpace, (VADDR_T)(UINTPTR)ipcInfo->pool.uvaddr);//在指定进程空间中找到所在线性区
        if (regionTemp != NULL) {
            (VOID)LOS_RegionFree(pcb->vmSpace, regionTemp);//先释放线性区
        }
  // 建议加上 ipcInfo->pool.uvaddr = NULL; 同下
    }
    ipcInfo->pool.uvaddr = (VOID *)(UINTPTR)region->range.base;//将指定的线性区和ipc池虚拟地址绑定
    if (ipcInfo->pool.kvaddr != NULL) {//如果存在内核空间地址
        LOS_VFree(ipcInfo->pool.kvaddr);//因为要重新映射,所以必须先释放掉物理内存
        ipcInfo->pool.kvaddr = NULL; //从效果上看, 这句话可以不加,但加上看着更舒服,    uvaddr 和 kvaddr 一对新人迎接美好未来
    }
    /* use vmalloc to alloc phy mem */
    ipcInfo->pool.kvaddr = LOS_VMalloc(region->range.size);//从内核动态空间中申请线性区,分配同等量的物理内存,做好 内核 <-->物理内存的映射
    if (ipcInfo->pool.kvaddr == NULL) {//申请物理内存失败, 肯定是玩不下去了.
        ret = -ENOMEM; //返回没有内存了
        goto ERROR_REGION_OUT;
    }
    /* do mmap */
    ret = DoIpcMmap(pcb, region);//对uvaddr和kvaddr做好映射关系,如此用户态下通过操作uvaddr达到操作kvaddr的目的
    if (ret) {
        goto ERROR_MAP_OUT;
    }
    /* ipc pool init */
    if (LOS_MemInit(ipcInfo->pool.kvaddr, region->range.size) != LOS_OK) {//初始化ipc池
        ret = -EINVAL;
        goto ERROR_MAP_OUT;
    }
    ipcInfo->pool.poolSize = region->range.size;//ipc池大小为线性区大小
    return 0;
ERROR_MAP_OUT:
    LOS_VFree(ipcInfo->pool.kvaddr);
ERROR_REGION_OUT:
    if (ipcInfo != NULL) {
        ipcInfo->pool.uvaddr = NULL;
        ipcInfo->pool.kvaddr = NULL;
    }
    return ret;
}
```

解读

- 这个函数一定要看明白，重要部分已加注释，主要干了两件事。
- 通过 LOS_VMalloc 向内核堆空间申请了一段物理内存，参数是线性区的大小，并做好了映射，因是内核堆空间，所以分配的虚拟地址就是个内核地址，并将这个地址赋给了 pool.kvaddr

  ```
  ipcInfo->pool.kvaddr = LOS_VMalloc(region->range.size);//从内核动态空间中申请线性区,分配同等量的物理内存,做好 内核 <-->物理内存的映射
  ```

- 通过 DoIpcMmap 将参数 pcb (用户进程)的 IPC 消息池的 pool.uvaddr 也映射到 LOS_VMalloc 分配的物理内存上,

  ```
  ret = DoIpcMmap(pcb, region);//对uvaddr和kvaddr做好映射关系,如此用户态下通过操作uvaddr达到操作
  ```

详看 DoIpcMmap 的实现,因为太重要此处代码不做删改。

```
LITE_OS_SEC_TEXT STATIC INT32 DoIpcMmap(LosProcessCB *pcb, LosVmMapRegion *region)
 {
    UINT32 i;
    INT32 ret = 0;
    PADDR_T pa;
    UINT32 uflags = VM_MAP_REGION_FLAG_PERM_READ | VM_MAP_REGION_FLAG_PERM_USER;
    LosVmPage *vmPage = NULL;
    VADDR_T uva = (VADDR_T)(UINTPTR)pcb->ipcInfo->pool.uvaddr;//用户空间地址
    VADDR_T kva = (VADDR_T)(UINTPTR)pcb->ipcInfo->pool.kvaddr;//内核空间地址
    (VOID)LOS_MuxAcquire(&pcb->vmSpace->regionMux);
    for (i = 0; i < (region->range.size >> PAGE_SHIFT); i++) {//获取线性区页数,一页一页映射
        pa = LOS_PaddrQuery((VOID *)(UINTPTR)(kva + (i << PAGE_SHIFT)));//通过内核空间查找物理地址
        if (pa == 0) {
            PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
            ret = -EINVAL;
            break;
        }
        vmPage = LOS_VmPageGet(pa);//获取物理页框
        if (vmPage == NULL) {//目的是检查物理页是否存在
            PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
            ret = -EINVAL;
            break;
        }
        STATUS_T err = LOS_ArchMmuMap(&pcb->vmSpace->archMmu, uva + (i << PAGE_SHIFT), pa, 1, uflags);//将物理页映射到用户空间
        if (err < 0) {
            ret = err;
            PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
            break;
        }
    }
    /* if any failure happened, rollback | 如果中间发生映射失败,则回滚*/
    if (i != (region->range.size >> PAGE_SHIFT)) {
        while (i--) {
            pa = LOS_PaddrQuery((VOID *)(UINTPTR)(kva + (i << PAGE_SHIFT)));//查询物理地址
            vmPage = LOS_VmPageGet(pa);//获取物理页框
            (VOID)LOS_ArchMmuUnmap(&pcb->vmSpace->archMmu, uva + (i << PAGE_SHIFT), 1);//取消与用户空间的映射
            LOS_PhysPageFree(vmPage);//释放物理页
        }
    }
    (VOID)LOS_MuxRelease(&pcb->vmSpace->regionMux);
    return ret;
 }
```

- 至次 LiteIPC 的准备工作已经完成，接下来就是操作/控制阶段了

LiteIpcIoctl | 控制

```
LITE_OS_SEC_TEXT int LiteIpcIoctl(struct file *filep, int cmd, unsigned long arg)
{
    UINT32 ret = LOS_OK;
    LosProcessCB *pcb = OsCurrProcessGet();
    ProcIpcInfo *ipcInfo = pcb->ipcInfo;
// 整个系统只能有一个ServiceManager，而Service可以有多个。ServiceManager有两个主要功能：一是负责Service的注册和注销，
// 二是负责管理Service的访问权限（只有有权限的任务（Task）可以向对应的Service发送IPC消息）。
    switch (cmd) {
        case IPC_SET_CMS:
            return SetCms(arg); //设置ServiceManager，整个系统只能有一个ServiceManager
        case IPC_CMS_CMD: // 控制命令,创建/删除/添加权限
            return HandleCmsCmd((CmsCmdContent *)(UINTPTR)arg);
        case IPC_SET_IPC_THREAD:
            return SetIpcTask();//将当前任务设置成当前进程的IPC任务ID
        case IPC_SEND_RECV_MSG://发送和接受消息,代表消息内容
            ret = LiteIpcMsgHandle((IpcContent *)(UINTPTR)arg);//处理IPC消息
            break;
    }
    return ret;
}
```

解读

- LiteIPC 中有两个主要概念，一个是 ServiceManager ，另一个是 Service 。整个系统只能有一个 ServiceManager ，而 Service 可以有多个。 ServiceManager 有两个主要功能：一是负责 Service 的注册和注销，二是负责管理 Service 的访问权限（只有有权限的任务（ Task ）可以向对应的 Service 发送 IPC 消息）。 IPC_SET_CMS 为设置 ServiceManager 命令， IPC_CMS_CMD 为对 Service 的管理命令。
- IPC_SEND_RECV_MSG 为消息的处理过程，消息的封装结合上篇理解，接收和发送消息对应的是 LiteIpcRead 和 LiteIpcWrite 两个函数。
- LiteIpcWrite 写消息指的是从用户空间向内核空间写数据，在消息内容体中已经指明这个消息时写给哪个任务的，如此达到了进程间(其实也是任务间)通讯的目的。

```
/// 写IPC消息队列,从用户空间到内核空间
LITE_OS_SEC_TEXT STATIC UINT32 LiteIpcWrite(IpcContent *content)
{
    UINT32 ret, intSave;
    UINT32 dstTid;
    IpcMsg *msg = content->outMsg;
    LosTaskCB *tcb = OS_TCB_FROM_TID(dstTid);//目标任务实体
    LosProcessCB *pcb = OS_PCB_FROM_PID(tcb->processID);//目标进程实体
    if (pcb->ipcInfo == NULL) {
        PRINT_ERR("pid %u Liteipc not create\n", tcb->processID);
        return -EINVAL;
    }
    //这里为什么要申请msg->dataSz,因为IpcMsg中的真正数据体 data是个指针,它的大小是dataSz . 同时申请存储偏移量空间
    UINT32 bufSz = sizeof(IpcListNode) + msg->dataSz + msg->spObjNum * sizeof(UINT32);//这句话是理解上层消息在内核空间数据存放的关键!!!
    IpcListNode *buf = (IpcListNode *)LiteIpcNodeAlloc(tcb->processID, bufSz);//向内核空间申请bufSz大小内存
    if (buf == NULL) {
        PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
        return -ENOMEM;
    }//IpcListNode的第一个成员变量就是IpcMsg
    ret = CopyDataFromUser(buf, bufSz, (const IpcMsg *)msg);//将消息内容拷贝到内核空间,包括消息控制体+内容体+偏移量
    if (ret != LOS_OK) {
        PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
        goto ERROR_COPY;
    }
    if (tcb->ipcTaskInfo == NULL) {//如果任务还没有IPC信息
        tcb->ipcTaskInfo = LiteIpcTaskInit();//初始化这个任务的IPC信息模块,因为消息来了要处理了
    }
    ret = HandleSpecialObjects(dstTid, buf, FALSE);//处理消息
    if (ret != LOS_OK) {
        PRINT_ERR("%s, %d\n", __FUNCTION__, __LINE__);
        goto ERROR_COPY;
    }
    /* add data to list and wake up dest task *///向列表添加数据并唤醒目标任务
    SCHEDULER_LOCK(intSave);
    LOS_ListTailInsert(&(tcb->ipcTaskInfo->msgListHead), &(buf->listNode));//把消息控制体挂到目标任务的IPC链表头上
    OsHookCall(LOS_HOOK_TYPE_IPC_WRITE, &buf->msg, dstTid, tcb->processID, tcb->waitFlag);
    if (tcb->waitFlag == OS_TASK_WAIT_LITEIPC) {//如果这个任务在等这个消息,注意这个tcb可不是当前任务
        OsTaskWakeClearPendMask(tcb);//撕掉对应标签
        OsSchedTaskWake(tcb);//唤醒任务执行,因为任务在等待读取 IPC消息
        SCHEDULER_UNLOCK(intSave);
        LOS_MpSchedule(OS_MP_CPU_ALL);//设置调度方式,所有CPU核发生一次调度,这里非要所有CPU都调度吗?
        //可不可以查询下该任务挂在哪个CPU上,只调度对应CPU呢?  注者在此抛出思考 @note_thinking
        LOS_Schedule();//发起调度
    } else {
        SCHEDULER_UNLOCK(intSave);
    }
    return LOS_OK;
ERROR_COPY:
    LiteIpcNodeFree(OS_TCB_FROM_TID(dstTid)->processID, buf);//拷贝发生错误就要释放内核堆内存,那可是好大一块堆内存啊
    return ret;
}
```

  - 大概流程就是从 LiteIpc 内存池中分配内核空间装用户空间的数据，注意一定要从 LiteIpcNodeAlloc 分配，原因代码中也已注明。
  - 有数据了就将数据挂到目标任务的 IPC 双向链表上，如果任务在等待读取消息(OS_TASK_WAIT_LITEIPC)则唤醒目标任务执行，并发起调度 LOS_Schedule 。
- LiteIpcRead 和 LiteIpcWrite 是遥相呼应，读消息指将内核空间数据读到用户空间处理。

```
/// 读取IPC消息
LITE_OS_SEC_TEXT STATIC UINT32 LiteIpcRead(IpcContent *content)
```

```
{
    UINT32 intSave, ret;
    UINT32 selfTid = LOS_CurTaskIDGet();//当前任务ID
    LOS_DL_LIST *listHead = NULL;
    LOS_DL_LIST *listNode = NULL;
    IpcListNode *node = NULL;
    UINT32 syncFlag = (content->flag & SEND) && (content->flag & RECV);//同步标签
    UINT32 timeout = syncFlag ? LOS_MS2Tick(LITEIPC_TIMEOUT_MS) : LOS_WAIT_FOREVER;
    LosTaskCB *tcb = OS_TCB_FROM_TID(selfTid);//获取当前任务实体
    if (tcb->ipcTaskInfo == NULL) {//如果任务还没有赋予IPC功能
        tcb->ipcTaskInfo = LiteIpcTaskInit();//初始化任务的IPC
    }
    listHead = &(tcb->ipcTaskInfo->msgListHead);//获取IPC信息头节点
    do {//注意这里是个死循环
        SCHEDULER_LOCK(intSave);
        if (LOS_ListEmpty(listHead)) {//链表为空 ?
            OsTaskWaitSetPendMask(OS_TASK_WAIT_LITEIPC, OS_INVALID_VALUE, timeout);//设置当前任务要等待的信息
            OsHookCall(LOS_HOOK_TYPE_IPC_TRY_READ, syncFlag ? MT_REPLY : MT_REQUEST, tcb->waitFlag);//向hook模块输入等待日志信息
            ret = OsSchedTaskWait(&g_ipcPendlist, timeout, TRUE);//将任务挂到全局链表上,任务进入等IPC信息,等待时间(timeout),此处产生调度,将切换
            //如果一个消息在超时前到达,则任务会被唤醒执行,返回就不是LOS_ERRNO_TSK_TIMEOUT
            if (ret == LOS_ERRNO_TSK_TIMEOUT) {//如果发生指定的时间还没有IPC到达时
                OsHookCall(LOS_HOOK_TYPE_IPC_READ_TIMEOUT, syncFlag ? MT_REPLY : MT_REQUEST, tcb->waitFlag);//打印任务等待IPC时发生 回复
                SCHEDULER_UNLOCK(intSave);
                return -ETIME;
            }
            if (OsTaskIsKilled(tcb)) {//如果发生任务被干掉了的异常
                OsHookCall(LOS_HOOK_TYPE_IPC_KILL, syncFlag ? MT_REPLY : MT_REQUEST, tcb->waitFlag);//打印任务在等待IPC期间被干掉了的
                SCHEDULER_UNLOCK(intSave);
                return -ERFKILL;
            }
            SCHEDULER_UNLOCK(intSave);
        } else {//有IPC节点数据时
            listNode = LOS_DL_LIST_FIRST(listHead);//拿到首个节点
            LOS_ListDelete(listNode);//从链表上摘掉节点,读后即焚
            node = LOS_DL_LIST_ENTRY(listNode, IpcListNode, listNode);//获取节点实体
            SCHEDULER_UNLOCK(intSave);
            ret = CheckRecievedMsg(node, content, tcb);//检查收到的信息
            if (ret == LOS_OK) {//信息没问题
                break;
            }
            if (ret == -ENOENT) { /* It means that we've recieved a failed reply | 收到异常回复*/
                return ret;
            }
        }
    } while (1);
    node->msg.data = (VOID *)GetIpcUserAddr(LOS_GetCurrProcessID(), (INTPTR)(node->msg.data));//转成用户空间地址
    node->msg.offsets = (VOID *)GetIpcUserAddr(LOS_GetCurrProcessID(), (INTPTR)(node->msg.offsets));//转成用户空间的偏移量
    content->inMsg = (VOID *)GetIpcUserAddr(LOS_GetCurrProcessID(), (INTPTR)(&(node->msg)));//转成用户空间数据结构
    EnableIpcNodeFreeByUser(LOS_GetCurrProcessID(), (VOID *)node);//创建一个空闲节点,并挂到进程IPC已使用节点链表上
    return LOS_OK;
}
```

- 调度到目标任务后，将切到 `LiteIpcRead` 执行，此时读函数正在经历一个do .. while(1) 死循环等待消息到来。 `LiteIpcRead` 的最后是内核地址和用户地址的转换，这也是 `LiteIpc` 最精彩的部分，它们指向同一块数据。
- 当 `LiteIpcRead` 读取不到消息时，即当前任务的消息链表为空时，任务会设置一个等待标签 `OS_TASK_WAIT_LITEIPC` ，并将自己挂起，由 `OsSchedTaskWait` 让出 `CPU` 给其他任务继续执行，请反复理解读写函数。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

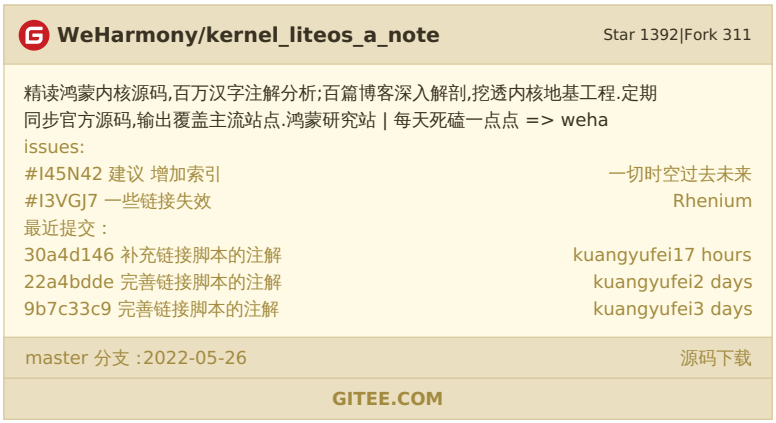
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

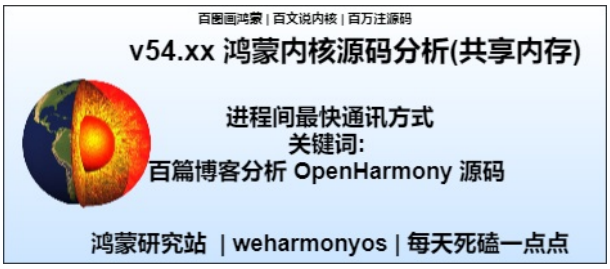


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

54_共享内存篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

通讯机制相关篇为:

- v41.04 鸿蒙内核源码分析(通讯总览) | 内核跟人一样都喜欢八卦
- v42.08 鸿蒙内核源码分析(自旋锁) | 死等丈夫归来的贞洁烈女
- v43.05 鸿蒙内核源码分析(互斥锁) | 有你没她 相安无事
- v44.02 鸿蒙内核源码分析(快锁使用) | 用户态负责快锁逻辑
- v45.02 鸿蒙内核源码分析(快锁实现) | 内核态负责快锁调度
- v46.01 鸿蒙内核源码分析(读写锁) | 内核如何实现多读单写
- v47.05 鸿蒙内核源码分析(信号量) | 谁在解决任务间的同步
- v48.07 鸿蒙内核源码分析(事件机制) | 多对多任务如何同步
- v49.05 鸿蒙内核源码分析(信号生产) | 年过半百 活力十足
- v50.03 鸿蒙内核源码分析(信号消费) | 谁让CPU连续四次换栈运行
- v51.03 鸿蒙内核源码分析(消息队列) | 进程间如何异步传递大数据
- v52.02 鸿蒙内核源码分析(消息封装) | 剖析LiteIpc(上)进程通讯内容
- v53.01 鸿蒙内核源码分析(消息映射) | 剖析LiteIpc(下)进程通讯机制
- v54.01 鸿蒙内核源码分析(共享内存) | 进程间最快通讯方式

运行机制

共享好端端的一词，近些年被玩坏了，共享单车,共享充电宝,共享办公室，共享雨伞... 甚至还有共享女朋友，真是人有多大胆，共享有多大产。但凡事太尽就容易恶心到人，自己也一度被 **共享内存** 恶心到了，一直不想碰它，拖到了现在才写。

共享内存的原理简单，目的是为了进程间通讯，方法是通过映射到同一块物理内存。它是一种稀缺资源由内核按资源池方式管理,数量有限,默认是 `192` 个，用资源ID唯一标识，用户进程需要时通过系统调用向内核申请共享内存大小,管理器从资源池中分配一个可用资源ID,并向物理内存申请对应的物理页框。

如何使用共享内存就涉及到了内存模块最重要的概念 **映射**，不清楚的可以翻看系列相关篇。有共享需求的进程在各自的进程空间中划出一个线性区映射到共享内存段，那如何找到这个共享内存段呢？由系统调用提供操作接口，简单说是先通过参数 `key` 创建共享资源ID(`shmid`)，再由 `shmid` 来连接/删除/控制 共享内存。详见本篇末尾的 `4` 个系统调用 `Shm***` 。

如何实现?

这是笔者看完内核共享内存模块画出来的图，尽量用一张图表达一个模块的内容，因为百文是在给源码注释的过程中产生的，所以会画出这种比较怪异的图，有代码，也有模型，姑且称之为 **代码模型图**:

图分 **管理** 和 **映射使用** 两部分解读。 为了精简，代码展示只留下骨干，删除了判断，检查的代码。

管理部分

- 初始化共享内存,共享内存是以资源池的方式管理的,上来就为全局变量 `g_shmSegs` 向内核堆空间申请了 `g_shmInfo.shmmni` 个 `struct shmIDSource`

```
#define SHM_MNI 192 //共享内存总数 默认192
// 共享内存模块设置信息
struct shminfo {
  unsigned long shmmax, shmmin, shmmni, shmseg, shmall, __unused[4];
};
STATIC struct shminfo g_shmInfo = { //描述共享内存范围的全局变量
```

```
    .shmmax = SHM_MAX,//共享内存单个上限 4096页 即 16M
    .shmmin = SHM_MIN,//共享内存单个下限 1页 即:4K
    .shmmni = SHM_MNI,//共享内存总数 默认192
    .shmseg = SHM_SEG,//每个用户进程可以使用的最多的共享内存段的数目 128
    .shmall = SHM_ALL,//系统范围内共享内存的总页数，4096页
    };
  //共享内存初始化
    UINT32 ShmInit(VOID)
    {
        // ..
        ret = LOS_MuxInit(&g_sysvShmMux, NULL);//初始化互斥
        g_shmSegs = LOS_MemAlloc((VOID *)OS_SYS_MEM_ADDR, sizeof(struct shmIDSource) * g_shmInfo.shmmni);//分配shm段数组
        (VOID)memset_s(g_shmSegs, (sizeof(struct shmIDSource) * g_shmInfo.shmmni),
                0, (sizeof(struct shmIDSource) * g_shmInfo.shmmni));//数组清零
        for (i = 0; i < g_shmInfo.shmmni; i++) {
            g_shmSegs[i].status = SHM_SEG_FREE;//节点初始状态为空闲
            g_shmSegs[i].ds.shm_perm.seq = i + 1;//struct ipc_perm shm_perm;系统为每一个IPC对象保存一个ipc_perm结构体,结构说明了IPC对象的权限
            LOS_ListInit(&g_shmSegs[i].node);//初始化节点
        }
        g_shmUsedPageCount = 0;
        return LOS_OK;
    }
```

- 系列篇多次提过，每个功能模块都至少有一个核心结构体来支撑模块的运行，进程是 `PCB` ，任务是 `TCB` ，而共享内存就是 `shmIDSource`

```
struct shmIDSource {//共享内存描述符
    struct shmid_ds ds; //是内核为每一个共享内存段维护的数据结构
    UINT32 status; //状态 SHM_SEG_FREE ...
    LOS_DL_LIST node; //节点,挂VmPage
  #ifdef LOSCFG_SHELL
    CHAR ownerName[OS_PCB_NAME_LEN];
  #endif
  };
```

首先**shmid_ds**是真正描述共享内存信息的结构体,记录了本次共享内存由谁创建,大小,用户/组,访问时间等等。

```
//每个共享内存段在内核中维护着一个内部结构shmid_ds
  struct shmid_ds {
    struct ipc_perm shm_perm;///< 操作许可，里面包含共享内存的用户ID、组ID等信息
    size_t shm_segsz; ///< 共享内存段的大小，单位为字节
    time_t shm_atime; ///< 最后一个进程访问共享内存的时间
    time_t shm_dtime;  ///< 最后一个进程离开共享内存的时间
    time_t shm_ctime;  ///< 创建时间
    pid_t shm_cpid;  ///< 创建共享内存的进程ID
    pid_t shm_lpid;  ///< 最后操作共享内存的进程ID
    unsigned long shm_nattch; ///< 当前使用该共享内存段的进程数量
    unsigned long __pad1; //保留扩展用
    unsigned long __pad2;
  };
//内核为每一个IPC对象保存一个ipc_perm结构体，该结构说明了IPC对象的权限和所有者
  struct ipc_perm {
    key_t __ipc_perm_key; //调用shmget()时给出的关键字
    uid_t uid;    //共享内存所有者的有效用户ID
    gid_t gid;    //共享内存所有者所属组的有效组ID
    uid_t cuid;    //共享内存创建 者的有效用户ID
    gid_t cgid;    //共享内存创建者所属组的有效组ID
    mode_t mode;   //权限 + SHM_DEST / SHM_LOCKED /SHM_HUGETLB 标志位
    int __ipc_perm_seq;  //序列号
    long __pad1;   //保留扩展用
    long __pad2;
  };
```

status 表示这段共享内存的状态,因为是资源池的方式，只有 `SHM_SEG_FREE` 的状态才可供分配，进程池和任务池也是这种管理方式。

```
  #define SHM_SEG_FREE    0x2000 //空闲未使用
  #define SHM_SEG_USED    0x4000 //已使用
  #define SHM_SEG_REMOVE  0x8000 //删除
```

node双向链表上挂的是一个个的物理页框`VmPage`，这是核心属性，数据将被存在这一个个物理页框中。`ShmAllocSeg`为具体的分配函数

```
STATIC INT32 ShmAllocSeg(key_t key, size_t size, INT32 shmflg)
 {
    // ...
    count = LOS_PhysPagesAlloc(size >> PAGE_SHIFT, &seg->node);//分配共享页面,函数内部把node都挂好了.
    if (count != (size >> PAGE_SHIFT)) {//当未分配到足够的内存时,处理方式是:不稀罕给那么点,舍弃!
        (VOID)LOS_PhysPagesFree(&seg->node);//释放节点上的物理页框
        seg->status = SHM_SEG_FREE;//共享段变回空闲状态
        return -ENOMEM;
    }
    ShmSetSharedFlag(seg);//将node的每个页面设置为共享页
    g_shmUsedPageCount += size >> PAGE_SHIFT;
    seg->status |= SHM_SEG_USED; //共享段贴上已在使用的标签
    seg->ds.shm_perm.mode = (UINT32)shmflg & ACCESSPERMS;
    seg->ds.shm_perm.key = key;//保存参数key,如此 key 和 共享ID绑定在一块
    seg->ds.shm_segsz = size; //共享段的大小
    seg->ds.shm_perm.cuid = LOS_GetUserID(); //设置用户ID
    seg->ds.shm_perm.uid = LOS_GetUserID();  //设置用户ID
    seg->ds.shm_perm.cgid = LOS_GetGroupID(); //设置组ID
    seg->ds.shm_perm.gid = LOS_GetGroupID(); //设置组ID
    seg->ds.shm_lpid = 0; //最后一个操作的进程
    seg->ds.shm_nattch = 0; //绑定进程的数量
    seg->ds.shm_cpid = LOS_GetCurrProcessID(); //获取进程ID
    seg->ds.shm_atime = 0; //访问时间
    seg->ds.shm_dtime = 0; //detach 分离时间 共享内存使用完之后，需要将它从进程地址空间中分离出来；将共享内存分离并不是删除它，只是使该共享内
    seg->ds.shm_ctime = time(NULL);//创建时间
 #ifdef LOSCFG_SHELL
    (VOID)memcpy_s(seg->ownerName, OS_PCB_NAME_LEN, OsCurrProcessGet()->processName, OS_PCB_NAME_LEN);
 #endif
    return segNum;
 }
```

映射使用部分

- **第一步: 创建共享内存** 要实现共享内存,首先得创建一个内存段用于共享，干这事的是`ShmGet`

```
/*!
 * @brief ShmGet
 * 得到一个共享内存标识符或创建一个共享内存对象
 * @param key 建立新共享内存对象 标识符是IPC对象的内部名。为使多个合作进程能够在同一IPC对象上汇聚，需要提供一个外部命名方案。
        为此，每个IPC对象都与一个键（key）相关联，这个键作为该对象的外部名,无论何时创建IPC结构（通过msgget、semget、shmget创建），
        都应给IPC指定一个键, key_t由ftok创建,ftok当然在本工程里找不到,所以要写这么多.
 * @param shmflg IPC_CREAT IPC_EXCL
          IPC_CREAT： 在创建新的IPC时，如果key参数是IPC_PRIVATE或者和当前某种类型的IPC结构无关，则需要指明flag参数的IPC_CREAT标志位，
                则用来创建一个新的IPC结构。（如果IPC结构已存在，并且指定了IPC_CREAT，则IPC_CREAT什么都不做，函数也不出错）
          IPC_EXCL： 此参数一般与IPC_CREAT配合使用来创建一个新的IPC结构。如果创建的IPC结构已存在函数就出错返回，
                返回EEXIST（这与open函数指定O_CREAT和O_EXCL标志原理相同）
 * @param size 新建的共享内存大小，以字节为单位
 * @return
 *
 * @see
 */
INT32 ShmGet(key_t key, size_t size, INT32 shmflg)
 {
    SYSV_SHM_LOCK();
    if (key == IPC_PRIVATE) {
        ret = ShmAllocSeg(key, size, shmflg);
    } else {
        ret = ShmFindSegByKey(key);//通过key查找资源ID
        ret = ShmAllocSeg(key, size, shmflg);//分配一个共享内存
    }
    SYSV_SHM_UNLOCK();
    return ret;
 }
```

- **第二步: 进程线性区绑定共享内存** shmat()函数的作用就是用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。
，`ShmAt`的第一个参数其实是`ShmGet`成功时的返回值 ， `ShmatVmmAlloc`负责分配一个可用的线性区并和共享内存映射好

```
/*!
```

```
 * @brief ShmAt
 * 用来启动对该共享内存的访问，并把共享内存连接到当前进程的地址空间。
 * @param shm_flg 是一组标志位，通常为0。
 * @param shmaddr 指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。
 * @param shmid 是shmget()函数返回的共享内存标识符
 * @return
 * 如果shmat成功执行，那么内核将使与该共享存储相关的shmid_ds结构中的shm_nattch计数器值加1
shmid 就是个索引,就跟进程和线程的ID一样 g_shmSegs[shmid] shmid > 192个
 * @see
 */
VOID *ShmAt(INT32 shmid, const VOID *shmaddr, INT32 shmflg)
{
    struct shmIDSource *seg = NULL;
    LosVmMapRegion *r = NULL;
    ret = ShmatParamCheck(shmaddr, shmflg);//参数检查
    SYSV_SHM_LOCK();
    seg = ShmFindSeg(shmid);//找到段
    ret = ShmPermCheck(seg, acc_mode);
    seg->ds.shm_nattch++;//ds上记录有一个进程绑定上来
    r = ShmatVmmAlloc(seg, shmaddr, shmflg, prot);//在当前进程空间分配一个线性区并映射到共享内存
    r->shmid = shmid;//把ID给线性区的shmid
    r->regionFlags |= VM_MAP_REGION_FLAG_SHM;//这是一个共享线性区
    seg->ds.shm_atime = time(NULL);//访问时间
    seg->ds.shm_lpid = LOS_GetCurrProcessID();//进程ID
    SYSV_SHM_UNLOCK();
    return (VOID *)(UINTPTR)r->range.base;
}
```

- **第三步: 控制/使用 共享内存**，这才是目的，前面的都是前戏

```
/*!
 * @brief ShmCtl
 * 此函数可以对shmid指定的共享存储进行多种操作（删除、取信息、加锁、解锁等）
 * @param buf 是一个结构指针，它指向共享内存模式和访问权限的结构。
 * @param cmd command是要采取的操作，它可以取下面的三个值 ：
    IPC_STAT：把shmid_ds结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖shmid_ds的值。
    IPC_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为shmid_ds结构中给出的值
    IPC_RMID：删除共享内存段
 * @param shmid 是shmget()函数返回的共享内存标识符
 * @return
 *
 * @see
 */
INT32 ShmCtl(INT32 shmid, INT32 cmd, struct shmid_ds *buf)
{
    SYSV_SHM_LOCK();
    switch (cmd) {
        case IPC_STAT:
        case SHM_STAT://取段结构
            ret = LOS_ArchCopyToUser(buf, &seg->ds, sizeof(struct shmid_ds));//把内核空间的共享页数据拷贝到用户空间
            if (cmd == SHM_STAT) {
                ret = (unsigned int)((unsigned int)seg->ds.shm_perm.seq << 16) | (unsigned int)((unsigned int)shmid & 0xffff); /* 16: use the seq
            }
            break;
        case IPC_SET://重置共享段
            ret = ShmPermCheck(seg, SHM_M);
            //从用户空间拷贝数据到内核空间
            ret = LOS_ArchCopyFromUser(&shm_perm, &buf->shm_perm, sizeof(struct ipc_perm));
            seg->ds.shm_perm.uid = shm_perm.uid;
            seg->ds.shm_perm.gid = shm_perm.gid;
            seg->ds.shm_perm.mode = (seg->ds.shm_perm.mode & ~ACCESSPERMS) |
                                    (shm_perm.mode & ACCESSPERMS);//可访问
            seg->ds.shm_ctime = time(NULL);
#ifdef LOSCFG_SHELL
            (VOID)memcpy_s(seg->ownerName, OS_PCB_NAME_LEN, OS_PCB_FROM_PID(shm_perm.uid)->processName,
                    OS_PCB_NAME_LEN);
#endif
            break;
        case IPC_RMID://删除共享段
            ret = ShmPermCheck(seg, SHM_M);
```

```
            seg->status |= SHM_SEG_REMOVE;
            if (seg->ds.shm_nattch <= 0) {//没有任何进程在使用了
                ShmFreeSeg(seg);//释放 归还内存
            }
            break;
        case IPC_INFO://把内核空间的共享页数据拷贝到用户空间
            ret = LOS_ArchCopyToUser(buf, &g_shmInfo, sizeof(struct shminfo));
            ret = g_shmInfo.shmmni;
            break;
        case SHM_INFO:
            shmInfo.shm_rss = 0;
            shmInfo.shm_swp = 0;
            shmInfo.shm_tot = 0;
            shmInfo.swap_attempts = 0;
            shmInfo.swap_successes = 0;
            shmInfo.used_ids = ShmSegUsedCount();//在使用的seg数
            ret = LOS_ArchCopyToUser(buf, &shmInfo, sizeof(struct shm_info));//把内核空间的共享页数据拷贝到用户空间
            ret = g_shmInfo.shmmni;
            break;
        default:
            VM_ERR("the cmd(%d) is not supported!", cmd);
            ret = EINVAL;
            goto ERROR;
    }
    SYSV_SHM_UNLOCK();
    return ret;
}
```

- **第四步: 完事了解绑/删除**，好聚好散还有下次，在 `ShmDt` 中主要干了解除映射 `LOS_ArchMmuUnmap` 这件事，没有了映射就不再有关系了，并且会检测到最后一个解除映射的进程时,会彻底释放掉这段共享内存 `ShmFreeSeg`

```
/**
 * @brief 当对共享存储的操作已经结束时，则调用shmdt与该存储段分离
    如果shmat成功执行，那么内核将使与该共享存储相关的shmid_ds结构中的shm_nattch计数器值减1
 * @attention 注意：这并不从系统中删除共享存储的标识符以及其相关的数据结构。共享存储的仍然存在，
    直至某个进程带IPC_RMID命令的调用shmctl特地删除共享存储为止
 * @param shmaddr
 * @return INT32
 */
INT32 ShmDt(const VOID *shmaddr)
{
    LosVmSpace *space = OsCurrProcessGet()->vmSpace;//获取进程空间
    (VOID)LOS_MuxAcquire(&space->regionMux);
    region = LOS_RegionFind(space, (VADDR_T)(UINTPTR)shmaddr);//找到线性区
    shmid = region->shmid;//线性区共享ID
    LOS_RbDelNode(&space->regionRbTree, &region->rbNode);//从红黑树和链表中摘除节点
    LOS_ArchMmuUnmap(&space->archMmu, region->range.base, region->range.size >> PAGE_SHIFT);//解除线性区的映射
    (VOID)LOS_MuxRelease(&space->regionMux);
    /* free it */
    free(region);//释放线性区所占内存池中的内存
    SYSV_SHM_LOCK();
    seg = ShmFindSeg(shmid);//找到seg,线性区和共享段的关系是 1:N 的关系,其他空间的线性区也会绑在共享段上
    ShmPagesRefDec(seg);//页面引用数 --
    seg->ds.shm_nattch--;//使用共享内存的进程数少了一个
    if ((seg->ds.shm_nattch <= 0) && //无任何进程使用共享内存
        (seg->status & SHM_SEG_REMOVE)) {//状态为删除时需要释放物理页内存了,否则其他进程还要继续使用共享内存
        ShmFreeSeg(seg);//释放seg 页框链表中的页框内存,再重置seg状态
    } else {
    seg->ds.shm_dtime = time(NULL);//记录分离的时间
    seg->ds.shm_lpid = LOS_GetCurrProcessID();//记录操作进程ID
    }
    SYSV_SHM_UNLOCK();
```

总结

看到这里你应该不会问共享内存的作用和为啥它是最快的进程间通讯方式了,如果还有这两个问题说明还要再看一遍 :P ,另外细心的话会发现共享内存会有个小缺点,就是同时访问的问题,所以需要使用互斥锁来保证同时只有一个进程在使用, `SYSV_SHM_LOCK` 和 `SYSV_SHM_UNLOCK` 在以上的四个步骤中都有出现。

```
STATIC LosMux g_sysvShmMux; //互斥锁,共享内存本身并不保证操作的同步性,所以需用互斥锁
/* private macro */
#define SYSV_SHM_LOCK()     (VOID)LOS_MuxLock(&g_sysvShmMux, LOS_WAIT_FOREVER) //申请永久等待锁
#define SYSV_SHM_UNLOCK()   (VOID)LOS_MuxUnlock(&g_sysvShmMux) //释放锁
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

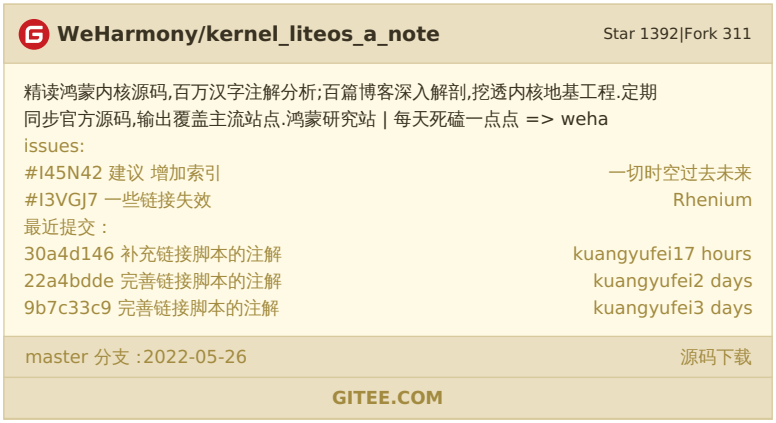
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断

编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

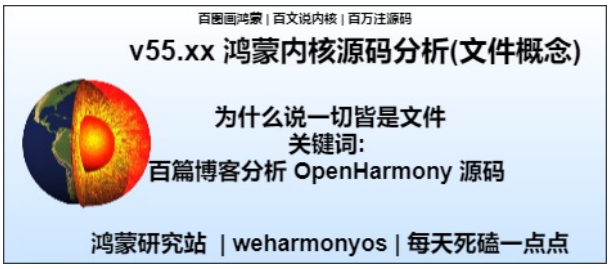


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

55_文件概念篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到`/`谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

本篇开始说文件系统，它是内核五大模块之一，甚至有Linux的设计哲学是"一切皆文件"的说法。所以其重要性不言而喻。搞不清楚文件系统，内核肯定是没整明白。文件系统相关概念巨多，后续将结合内核源码详细阐述，本篇先说清楚源头概念:**文件**。

什么是文件

- 不说清楚什么是文件就说不清楚文件系统，更说不清楚内核是如何管理和为什么要这么来管理文件的。
- 现代操作系统为解决信息能独立于进程之外被长期存储引入了文件，将文件抽象成一个宽泛的概念，把文档、目录（文件夹）、键盘、监视器、硬盘、可移动媒体设备、打印机、调制解调器、虚拟终端，还有进程间通信（`IPC`）和网络通信等输入/输出资源都看成文件来统一操作。
- 因为它们都具有共同的**读**和**写**共性，一旦具有普适性，可以抽象出理想的模型，通过这个模型，设计工作就会变得简单而有序，API的设计可以化繁为简。用户可以使用通用的方式去访问任何资源，使他们被处理时均可统一使用字节流方式，而差异部分则由相应的中间件做好对底层的适配。
- 不准确但是形象的例子 Linux 系统把硬件设备映射成文件，例如将摄像头映射为 /dev/video，然后就可以使用基本的函数操作它。用 open() 函数连接设备，再用 read() 函数读取图像，最后用 write() 函数保存图像。 而在声卡设备中，read() 函数会变为录音功能，write() 函数变为播放功能。

文件类型

从内核视角将文件分成七种类型:

- **普通文件 （- regular file）**

  大家普遍理解的文件属于此类，(如:图片，视频，mp3，ppt，zip ==)，这类文件也叫正则文件，当然是无处不在。

  ```
  turing@ubuntu:/home/tools$ ls -hil
  total 12M
  1083954 -rwxrwxr-x 1 turing turing 2.3M Feb 18 18:55 gn
  1083803 -rw-r--r-- 1 root   root   9.4M Nov 25  2020 hapsigntoolv2.jar
  1083802 -rw-r--r-- 1 root   root    58K Nov 25  2020 hmos_app_packing_tool.jar
  ```

- **目录文件（d，directory file）**

  就是目录或者说文件夹，能用`cd`命令进入的。它同样无处不在

  ```
  turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ls -lhi
  ```

```
total 68K
1202976 drwxr-xr-x  3 turing turing 4.0K Jun 21 02:38 applications
1173738 drwxr-xr-x 10 turing turing 4.0K Jun 21 02:38 base
1106153 drwxr-xr-x  3 turing turing 4.0K Jun 21 02:38 build
```

- **块设备文件（b，block device）**

  就是存储数据以供系统存取的接口设备，简单而言就是硬盘。例如一号硬盘的代码是 /dev/hda1等文件。属性为 [b]：`block device`，通常在 `/dev` 目录下能看到它。

```
turing@ubuntu:/dev$ ls -lhi
total 0
210 brwxr-xr-x  2 root   root        420 Jul 23 18:59 block
337 brwxr-xr-x  2 root   root         80 Jul 23 18:05 bsg
```

- **字符设备（c，char device）**

  字符设备文件：即串行端口的接口设备，例如键盘、鼠标等等。通常在 `/dev` 目录下能看到它

```
turing@ubuntu:/dev$ ls -lhi
total 0
124 crw-------  1 root   root     10，  175 Jul 23 18:05 agpgart
373 crw-r--r--  1 root   root     10，  235 Jul 23 18:05 autofs
```

- **套接字文件（s，socket）**

  这类文件通常用在网络数据连接。可以启动一个程序来监听客户端的要求，客户端就可以通过套接字来进行数据通信，最常在 /var/run目录中看到这种文件类型。

```
turing@ubuntu:/var/run$ ls -lhi
690 srw-rw-rw-  1 root            root     0 Jul 23 18:05 snapd-snap.socket
689 srw-rw-rw-  1 root            root     0 Jul 23 18:05 snapd.socket
```

- **管道文件（p，pipe）**

  管道文件主要用于进程间通讯。比如使用 `mkfifo` 命令可以创建一个 `FIFO` 文件，启用一个进程 `A` 从 `FIFO` 文件里读数据，启动进程 `B` 往 `FIFO` 里写数据，先进先出，随写随读。

```
turing@ubuntu:/var/run$ ls -lhi
269 prw-------  1 root            root     0 Jul 23 18:05 initctl
```

- **符号链接文件（l，symbolic link）**

  这里说的链接指的是软链接，类似Windows下面的快捷方式。，这类文件非常多，尤其 `/bin` ， `/usr/bin` 目录下最多。

```
turing@ubuntu:/bin$ ls -lhi
143828 lrwxrwxrwx 1 root root      29 Jul 14 21:51  rmiregistry -> /etc/alternatives/rmiregistry
132128 lrwxrwxrwx 1 root root       4 Jul 14 19:10  rnano -> nano
132131 lrwxrwxrwx 1 root root      29 Jul 14 19:10  rrsync -> ../share/rsync/scripts/rrsync
132132 lrwxrwxrwx 1 root root      21 Jul 14 19:10  rsh -> /etc/alternatives/rsh
```

文件属性

文件属性，简单的说，有这么几种

- 权限
- 所属者
- 所属组

```
1173738    drwxr-xr--  10     turing  turing  4.0K      Jun 21 02:38  base
[   0 ]   [    1   ] [ 2 ] [   3 ][  4 ] [    5   ] [    6    ] [  7  ]
[vnode编号] [  权限  ]  [硬链接][拥有者][群组]  [文件容量]  [  修改日期 ] [ 文件名]
```

[vnode编号] `vnode` 是文件系统非常重要的一个概念，后续有专门的篇幅结合源码详细说明，每个文件都有唯一的一个编号，跟身份证号一样，全国有100万人叫 `李伟` ，大家沟通都是叫 `李伟` ，不会喊身份证，并不影响沟通，但到了公安局就只认身份证，只要敢犯罪保准一逮一个准。所以视角不同，关注的点是不一样的。文件管理的机制是一模一样的，普通用户只需记住高清大片放在 `C:\xx\xx\xx\xx\xxx\xxx\xx.avi` 下就可以了，不管埋的多深都能翻出来。根本不需要知道 `vnode.id` 是多少。但到了内核层面，它操作的都是 `vnode.id`

[权限] 对于多用户多群组的系统，就必须有权限来加持文件操作，该栏可以分成以下4个小组

```
d， rwx， r-x， r-x
```

- 第一个字符 `d` 单独成组，这个表示文件类型，这里表示是个**目录文件（d，directory file）**
- 剩下的三个主要由[rwx]组成，r-read， w-write， x-execute， [-]表示占位符，即没权限。
  - 第二组为『文件拥有者的权限』， `rwx` 表示文件拥有者可读可写可执行
  - 第三组为『同群组的权限』； `r-x` 文件所属组可读可执行但不可写
  - 第四组为『其他非本群组的权限』， `r--` 其他人可读
- 权限除了字母表示外还可以用数字表示

```
r=100=4， w=010=2， x=001=1， -=0
rwxr-xr-- 可表示为
111101100 = 754
```

`chmod [-R] xxx` 文件或目录 ：改变文件拥有者 有两种方法可改变文件的权限

- 数字法 : chmod -R 777 ohos_config.json

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ls -hli
1103292 -rw-r--r--  1 turing root    350 Jul 21 00:17 ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ chmod -R 777 ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ls -hli
1103292 -rwxrwxrwx  2 turing root    350 Jul 21 00:17 ohos_config.json
```

`777` = `(111)(111)(111)` = `(rwx)(rwx)(rwx)`

- 字母法 :

```
     u   +(加入)    r
chmod   g   -(除去)    w   文件或目录
     o   =(设定)    x
     a
(u)user (g)group (o)others (a)all

chmod  u=rwx，go=rx   ohos_config.json 结果: rwxr-xr-x
chmod  a+w ohos_config.json          结果: rwxrwxrwx
chmod  u-r+wx ohos_config.json       结果: -wxrwxrwx
```

[链接] 一栏代表的是硬链接的数量，有硬链接就会有软链接，有什么区别呢。 先说清楚为什么会有**链接** ？ 原因是因为同一个文件往往需要被同一个用户或多个用户同时使用，好东西要懂得分享，好人一生平安，大片怎能独享。做个小实验看下二者的区别

```
#对ohos_config.json 创建硬链接和软链接
#创建硬链接命令 ln ohos_config.json hard_link
#创建软链接命令 ln -s ohos_config.json hard_link
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ls -hli
1103292 -rw-r--r--  1 turing root    350 Jul 21 00:17 ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ln ohos_config.json hard_link
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ln -s ohos_config.json soft_link
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ ls -hli
1103292 -rw-r--r--  2 turing root    350 Jul 21 00:17 hard_link
1103292 -rw-r--r--  2 turing root    350 Jul 21 00:17 ohos_config.json
1086100 lrwxrwxrwx  1 turing root     16 Jul 29 01:06 soft_link -> ohos_config.json
```

- **硬链接:** 记录大片被分享的次数， `hard_link` 和 `ohos_config.json` 的 `vnode_id` 都是 `1103292` ，二者内容一模一样。但是和没创建之前的区别是

[链接]数，也叫引用数，由1变成了2 。而新增加的这次引用数就是 `hard_link` 导致的

- **软链接:** 是单独创建了另一个文件，有独立的 `vnode_id` ，只是这个文件的内容指向了 `ohos_config.json` 而已，这样做有什么好处呢？ 举个例子就明白了。
- 某酒店301房住着一位美女，想进房间就需要有钥匙，只要去敲门就给你一把钥匙，离开了钥匙归还，硬链接就是301房间发出去的钥匙数量。那软链接是什么呢？ 是旁边的302房间， 进入302房间里面只有一张纸条上面写着"去301房间敲门，你懂的"。明白了吗？ 虽然绕了个湾，但换来的是非常灵活的操作，公安来了怎么办？ 只需把纸条内容改成 "去404房间敲门。 `404` 房间在开民主生活会，啥问题也没有。
- 在应用层是大量的软链接在被使用，比如 版本切换，升级软链接就非常的方便。

[拥有者]

`chown` ：改变文件拥有者 `chown [-R]` 账号名称 文件或目录 `chown [-R]` 账号名称:用户组名称 文件或目录

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ll
-rw-r--r--  2 root   root    350 Jul 21 00:17 ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$sudo chown -R turing:turing ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ll
-rw-r--r--  2 turing   turing    350 Jul 21 00:17 ohos_config.json
```

[群组]

`chgrp [-R]` 用户组名称 文件或目录

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ll
-rw-r--r--  2 root   root    350 Jul 21 00:17 ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$sudo chgrp -R turing ohos_config.json
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ll
-rw-r--r--  2 root   turing    350 Jul 21 00:17 ohos_config.json
```

[修改日期]

用 `stat` 命令可以查看一个文件的信息

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS$ stat ohos_config.json
File: ohos_config.json
Size: 350       Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d Inode: 1103292     Links: 2
Access : (0644/-rw-r--r--)  Uid : ( 1000/  turing)   Gid : (    0/    root)
Access: 2021-07-24 02:07:21.683190622 -0700
Modify: 2021-07-21 00:17:34.733766830 -0700
Change: 2021-07-29 01:20:14.314343117 -0700
Birth: -
```

- `mtime(modify time)` :修改时间是文件内容最后一次被修改的时间。比如： `vim` 操作后保存文件。 `ls -l` 列出的就是这个时间
- `atime(access time)` :访问时间是读一次文件的内容，这个时间就会更新。比如 `more` 、 `cat` 等命令。 `ls` 、 `stat` 命令不会修改 atime
- `ctime(change time)` :状态改动时间，是该文件的 `vnode` 节点最后一次被修改的时间，通过 `chmod` 、 `chown` 命令修改一次文件属性，这个时间就会更新

了解了文件后，再看文件系统。

文件系统

什么是文件系统？ 看看维基百科的解释:

- 计算机的文件系统是一种存储和组织计算机数据的方法，它使得对其访问和查找变得容易，文件系统使用文件和树形目录的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的那个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。
- 文件系统通常使用硬盘和光盘这样的存储设备，并维护文件在设备中的物理位置。但是，实际上文件系统也可能仅仅是一种访问资料的界面而已，实际的数据是通过网络协议（如NFS、SMB、9P等）提供的或者内存上，甚至可能根本没有对应的文件（如proc文件系统）。
- 严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型。

简单地说，文件系统是操作系统中负责管理持久数据的子系统，基本数据单位是文件，它的目的是对磁盘上的文件进行组织管理，组织的方式不同，就会形成不同的文件系统。

计算机文件系统很像我们大学的智能图书管理系统，你去图书馆借书，只需在屏幕上选中要借的书本列表提交后会自动把书提取出来放到你的面前，你并不需要知道书本是如何被检测出来，它真实的摆放在几号馆的几号书架的第几排。每个大学都有一套独立的图书管理方式，有的按分类，有的按科目，有的按地域，有的按时间。 即便都按分类来的，分类的方法也会不一样，而且管理1万册很高效的方法却不一定对1000万册也同样高效，但衡量方法的好坏无非是看以下几个要素:

- 增删改查的速度要快，
- 存储空间要小，空间回收算法要好，
- 安全机制，什么人有什么权限对这本书执行什么操作。
- 各种操作记录，书的入库时间，最后的借阅时间，修改时间等等都要记录在案。

计算机的文件系统因为技术的更新，因为各个公司的利益保护等等诸多原因，肯定也是百花齐放的，跟计算机语言一样，绝大多数语言的发明只是为了解决某个实验室或者某个公司当下遇到的问题，很多压根没想那么远，标准和规范那都是后话，取决于你的市场规模和背后的金主。统一标准是好，但真的很难。世界语出现很多年了，但又有几个人去学。联合国存在也很多年了，就不听你，不交会费，你能有啥办法。经济基础决定上层建筑，这句话初中政治就反复讲，当初不理解，现在是彻底明白了，终究是要靠实力说话的。

所以不要去奇怪为什么会有这么多语言要学，这么多前端，后台框架要搞，互联网技术版图还处于群雄争霸时代，巨头林立，身处其中的码农都是绞肉机里的肉。而且这种分裂的趋势会愈演愈烈， `PC` 时代 `Windows` 一统天下， 手机时代 苹果， `Android` 楚汉相争，万物互联时代 鸿蒙，苹果， `Fuchsia` 很可能是三分天下。新生代不断崛起，老贵族不下牌桌。

文件系统按类型可分成以下四种:

- **磁盘文件系统：**是一种设计用来利用数据存储设备来保存计算机文件的文件系统，最常用的数据存储设备是磁盘驱动器，可以直接或者间接地连接到计算机上。例如： `FAT` 、 `exFAT` 、 `NTFS` 、 `HFS` 、 `HFS+` 、 `ext2` 、 `ext3` 、 `ext4` 、 `ODS-5` 、 `btrfs` 、 `XFS` 、 `UFS` 、 `ZFS` 。有些文件系统是行程文件系统（也有译作日志文件系统）或者追踪文件系统。
- **闪存文件系统：**闪存文件系统是一种设计用来在闪存上储存文件的文件系统。随着移动设备的普及和闪存容量的增加，这类文件系统越来越流行。尽管磁盘文件系统也能在闪存上使用，但闪存文件系统是闪存设备的首选，理由如下：
  - 擦除区块：闪存的区块在重新写入前必须先进行擦除。擦除区块会占用相当可观的时间。因此，在设备空闲的时候擦除未使用的区块有助于提高速度，而写入数据时也可以优先使用已经擦除的区块。
  - 随机访问：由于在磁盘上寻址有很大的延迟，磁盘文件系统有针对寻址的优化，以尽量避免寻址。但闪存没有寻址延迟。
  - 写入平衡（Wear levelling）：闪存中经常写入的区块往往容易损坏。闪存文件系统的设计可以使数据均匀地写到整个设备。 日志文件系统具有闪存文件系统需要的特性，这类文件系统包括 `JFFS2` 和 `YAFFS` 。也有为了避免日志频繁写入而导致闪存寿命衰减的非日志文件系统，如 `exFAT` 。

  `JFFS2` （全称： Journalling Flash File System Version2），是 `Redhat` 公司开发的闪存文件系统，其前身是 `JFFS` ，最早只支持 `NOR Flash` ， 自2.6版以后开始支持 `NAND Flash` ， 适合使用于嵌入式系统。

  `YAFFS` （全称：Yet Another Flash File System）是由 `Aleph One` 公司所发展出来的 `NAND Flash` 嵌入式文件系统。
- **伪文件系统：**启动时动态生成的文件系统，包含有关当前正在运行的内核的许多信息、配置和日志，由于它们放置在易失性存储器中，因此它们仅在运行时可用，而在关闭时消失。这些伪文件常挂载到以下目录: `sysfs (/sys)` ， `procfs (/proc)` ， `debugfs (/sys/kernel/debug)` ， `configfs (/sys/kernel/config)` ， `tracefs (/sys/kernel/tracing)` ， `tmppfs` (`/dev/shm` ， `/run` ， `/sys/fs/cgroup` ， `/tmp/` ， `/var/volatile` ， `/run/user/<id>`)， `devtmpfs` (`/dev`)
  - `procfs` 是 进程文件系统 (file system) 的缩写，用于通过内核访问进程信息。这个文件系统通常被挂载到 `/proc` 目录。由于 `/proc` 不是一个真正的文件系统，它也就不占用存储空间，只是占用有限的内存。
  - `tmpfs` （temporary file system） 是类Unix系统上暂存档存储空间的常见名称，通常以挂载文件系统方式实现，并将资料存储在易失性存储器而非永久存储设备中。所有在tmpfs上存储的资料在理论上都是暂时借放的，那也表示说，文件不会创建在硬盘上面。一旦重启，所有在tmpfs里面的资料都会消失不见。
  - `Sysfs` 是Linux 2。6所提供的一种虚拟文件系统。这个文件系统不仅可以把设备（ `devices` ）和驱动程序（ `drivers` ）的信息从内核输出到用户空间，也可以用来对设备和驱动程序做设置。 `sysfs` 的目的是把一些原本在 `procfs` 中的，关于设备的部分，独立出来，以‘设备层次结构架构’（device tree）的形式呈现。
  - `devtmpfs` 是在 `Linux` 核心启动早期建立一个初步的 `/dev` ，令一般启动程序不用等待 `udev` ，缩短 GNU/Linux 的开机时间。将设备也看成为文件，突出了 `Linux` 文件系统的特点：一切皆文件或目录。
- **网络文件系统:** `NFS` ，(Network File System）是一种将远程主机上的分区（目录）经网络挂载到本地系统的一种机制，是一种分布式文件系统，力求客户端主机可以访问服务器端文件，并且其过程与访问本地存储时一样，它由Sun公司开发，于1984年发布。它的特点是将网络也看出了文件，再次体现一切皆文件的说法。

对于鸿蒙内核， `JFFS2` ， `YAFFS` ， `tmpfs` ， `procfs` ， `FAT` ， `NTFS` ， `ZFS` 将是后续章节的重点。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很

重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。

- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

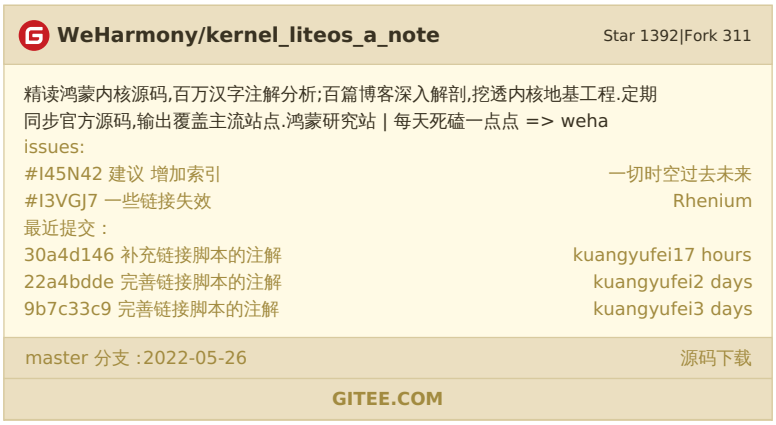
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

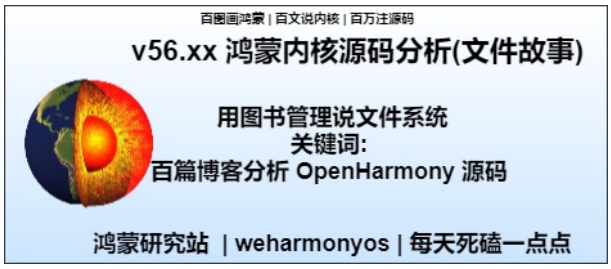


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

56_文件故事篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到`/`谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

本篇讲一个大型图书馆的管理方案，来说清楚计算机文件系统是如何管理的。如果读懂了这个方案，就基本了解了文件系统最底层的运行机制。

如何建图书馆

假如给你一个100*100米，高10米的场地用于建图书馆，放置全世界的图书档案，有以下几个运营要求:

- 图书有大有小，差异巨大，比如一本大英百科全书，估计有几百万页，同时也有只有薄薄一页的一封电报。
- 图书不是一次性给齐，后续不断有新增的，修改的，删除的，比如第一次给的24史不全只有12本，后面陆陆续续补上了。
- 书的规模是千万级的。必须要在最短的时间内找到书，新书也要在最短的时间内找到地方搁置。
- 每一次对书的操作都要记录下来，比如何时入库，何时修改，何时被借阅了。
- 权限验证，不同的人对书有不同的权限，比如有的人可以在书上涂鸦，但有的人只能看。
- 图书馆必须安全，不能因为局部失火导致整个图书馆不能正常运营。
- 可以存放目录信息，每个人都可以建自己的书籍目录，要求这些目录信息也能被保存。
- 让借书，还书，拿目录信息变简单，每个人只凭条形码操作。

请问如果是你会如何设计这个图书馆？并让它即安全又高效的运行。

小易的解决方案

有个叫小易的小伙子提出了一种解决方案:

- 全仓库建大小相同的格子，这种格子统一叫单元格。甭管是什么内容最后都是放到格子里，若每个格子按 `0.25*0.25*0.25` 米算， 整个场地可建设成 640万个单元格。单元格有唯一且统一的编号。从 `0` 一直编到 `640万-1` 。
- 因为单元格太多，管理非常复杂，所以将场地分成大A区，大B区，大C区，.....N区，比如分成8大区，每个区分配80万个单元格。 大A区编号[0 - 799999]，...依次类推。
- 每个大区又划分成统计区，目录区，图书区。
  - 统计区是描述整个图书馆和各分区的信息数据，占用1000个单元格
  - 目录区是为管理图书区而产生的信息，占19000个单元格，分成三块:
    - 索引表块(占18900个单元格):小易规定后续将用一页纸来记录书的索引信息，并把这页纸叫索引页，将索引页装订成一本索引表书，这本书有连续的统一的页编号(也叫条形码)。 像大英百科全书这样有几百万页的一本书，不管后续在图书区里占用多少单元格，但其在索引表书中就是一张纸，这张纸有固定格式，记录书的名称，权限，修改时间等等信息。

    - 索引页位图块(占10个单元格):记录索引页的使用情况， `0|1` 代表 `未使用|已使用` 。
    - 图书区单元格位图块(:占90个单元格):记录图书区单元格的使用情况， `0|1` 代表 `未使用|已使用` 。
- 图书区里放的是真正要管理的书籍，按单元格的容量来放，大点的书会分成多个单元格来存放。共占78万个单元格。

将以上信息简化成树形图表示如下:

```
└图书馆  => 共 640万个单元格，平均被分成八大战区
  ├大A区 => 80万单元格
  │ ├图书区 => 78万个单元格
  │ ├目录区 => 19000个单元格
  │ │ ├图书区单元格位图块 => 90个单元格
  │ │ ├索引表块 => 18900 个单元格
  │ │ │ ├A文件索引页 => 信息登记表，占一页纸，描述一本书的名称，权限，时间 ==
  │ │ │ ├...
  │ │ │ └B目录索引页
  │ │ └索引页位图块 => 10个单元格
  │ └统计区 (1000) => 记录图书馆和各分区的全局信息，使用频率高
  └大B区
    ├图书区
    ├目录区
    │ ├图书区单元格位图块
    │ ├索引表块
    │ │ ├A文件索引页
    │ │ ├...
    │ │ └B目录索引页
    │ └索引页位图块
    └统计区
  ...
```

请务必理解这些概念关系，在脑海中形成脑图，这是后续理解整个文件系统如何运作和管理的最关键底层逻辑。以下一一展开说明这些概念。

单元格

因为需求是图书大小没有限制，差别极大，有的书大到上千万页，有的小到只有一页。是个开放问题，需要在空间和时间上进行取舍，不想浪费时间就得浪费空间。没有边界就不方便做特殊处理，这需要标准化的统一管理。如何标准化？ 答案是: **建相同的格子** 至于是大格子还是小格子可以灵活，但必须是一样尺寸的。小易建了 `0.25*0.25*0.25` 米的标准格，可放1000页(1K页)书的，所有图书都是统一按页数放，放满1000页就换个格子放剩下的。 格子是图书的关系是(N:1)的关系，即一本书可以分多个格子放，但一个格子中不能放两本不一样的书。 没错，其中就会存在浪费的问题，但浪费就是浪费了。 表现如下:

- 如果一本书只有一页，那也要占一个格子，等于浪费掉999页的空间，注意不再往里面放其他书，否则管理会非常的麻烦。
- 而99999页的大英百科，将被分成100份，最后的999页也占个格子，等于浪费掉一页的空间。
- 当然，如果已知该图书馆将放置的基本都是10K页厚的书就可以按10K页的格子来建设，这样格子就少了，节省了查找的时间。 但如果基本都是10页的书就可以按10页来划格子，节省了空间但换来更多的格子。总之要在时间和空间上取一个平衡。你想如果将10000页的书放在10页单元格的图书馆中意味着要将书本切成1000份存放，后续维护是非常的耗时的。

统计区

统计区用于统计整个图书馆和各大区的全局信息，这种信息非常的重要，被使用频率极高，就像问我们国家有多少人口一样，马上就要答出来，而不是让各省逐级汇报下加起来再回复。

```
整个图书馆的全局信息包括:

战区数: 8
战区名称: A区，B区 ...
单元格的总数: 640W
已用单元格: 330W
剩余单元格: 310W
战区单元格范围: A区(0 ~ 799999)，B区(800000 ~ ...) ....
索引页编号:A区(0~99999)，B区(100000 ~ ...)
图书馆的创建时间:1921年 xx月
图书馆的名称: 世界图书馆
历史大事件:  1949年....
          1956年....
列表描述各大分区基本信息
  A分区 基本信息...
  B分区 基本信息...
```

```
C分区 基本信息...
```

- 整个图书馆全局信息为何不单独于各分区放置，而要在每个分区都保存一份呢？ 原因是因为防止数据被损坏，这么重要的数据只有一份如果哪天图书馆着火了放置那部分单元格被火烧掉了，那就歇菜了，整个图书馆都不能正常运作，所以用多份存档的方式是为了数据的安全。烧了东边还有西边撒，虽然会牺牲空间，但图书馆的安全健壮性却上了台阶。全局也会保存各分区的基本信息。
- 对分区更详细的信息会在分区的全局信息块中描述。这相当于广东省总人口会在中央登记下，但具体下属市，区，街道办的人口数据由广东省自己维护了。

```
各分区全局信息包括

战区名称: A区
战区范围：(0 ~ 799999)
单元格的总数: 80W
已用单元格: 20W
剩余单元格: 60W
数据块总数:78W
索引块总数:19000
....
```

统计区的信息会非常的多，可以理解为也是一本本的书，有固定的格式，同样放在单元格中。 再次说明下本篇说的任何信息最后都是放在单元格中的，理解这点很关键!

目录区

这个区和统计区一样，是因为要高效的管理图书区而衍生出来的区。目录区和图书区所分配单元格数量是在图书馆开业那天就定下来了，填满图书区单元格的真正的图书，而谁也不知道会有些什么图书要进进出出，图书大小决定了单元格的使用情况，每本书会占用一张索引页，所以最后一定会出现两种情况:

- 索引页用完了，但图书区还有格子没被填满。这种情况出现在大量的小而多图书，因为多占用的索引页就会多，因为小占用的图书区单元格就会少。
- 反之，另一种情况是索引页没用完，但图书区没单元格了。这种情况出现在大量的大而少图书，因为大占用图书区单元格就会多，因为少索引页就会少。
- 这就是为什么有时看着明明磁盘还有很大空间，但就是存储进去的原因所在。因为该分区小文件太多了，试着删除那些很小而不用的文件试试。

索引表

完全可以把索引表看成是一本书，书的内容是一页一页的索引页，每一页记录一本书的索引信息，1000页就可以记录1000本书的索引信息，这本书也是要装到格子里的。装这本书的单元格叫索引表块。 如果按1000万本书计算 就会生成1000万张索引页， 每个格子1000页，那么 1000万/1000 = 1万 ，也就是说 索引表这本书，需要1万个单元格来存放。 索引页也有全局唯一且统一的编号，注意这个编号和单元格的编号是两码事，这是很多人搞不明白文件系统的关键所在，二者编号范围在统计区有保存。

大分区中三个分区(统计区，目录区，图书区)的排列格式是固定的。所以很容易计算出某个分区下索引区块的开始和结束位置。这里假定索引表块在分区相对位置的第3000个单元格开始。

如何借书

此时外部人员可凭条形码来取书。流程如下:

- 技术部的屌丝小王拿着 `5300` 这个数字(条形码)来取书。
- 管理员需先锁定这个条形码在哪个大分区，在统计区一查发现在大A区， `索引页编号:A区(0~99999)`
- 管理员立即计算索引页存放的格子位置(3000+5300/1000)=3006号格子，搬梯子手机拍照第 `300` 页的内容。内容如下:

  ```
  名称: 编程珠玑
  大小: 14284页       占用图书区总格子数: 15        单元格大小: 1000页   属于普通文件
    条形码: 5300      捆绑数: 2人
  权限：(0644/-rw-r-----)  用户ID：( 10/  小张)   组ID：(2/ 技术部)
  访问时间: 2021-07-24 02:07:21.683190622 -0700
  修改书籍时间: 2021-07-21 00:17:34.733766830 -0700
  修改索引时间: 2021-07-29 01:20:14.314343117 -0700
  图书区存放位置:12，32，45，....980
  ```

  - 管理员先验权限和所属，表上已经说的很清楚，这本书主人是技术部的小张。
  - 小张本人 `rw-` 对这本书可以读，可以修改。

- 技术部同学 `r--` 对这本书只能读，屌丝小王属于技术部，所以可以借，但不能在上面涂鸦。
- 非技术部同学 `---` 说明没有任何权限。
- 索引表上清晰的记录了书本的名称，总页数，格子的大小，条形码。
- 数据块号: 15 代表编程珠玑放在大A区图书区的15个格子里， `12，32，45 ....` 给出了格子的编号，编程珠玑是按编号顺序存放的。
- 有了图书区存放位置管理员再依次把书拷贝一份出来，按顺序叠放好，形成完整的编程珠玑交给屌丝小王，同时将借阅数: 2人 改成3人，把访问时间改成当前时间。

两个位图块

如果屌丝小王也想像小张一样，将爱书 `论程序员的自我修养` 捐给图书馆，流程又是怎样的呢？

很明显需要增加两部分内容:

- 索引页 ， `论程序员的自我修养` 将在索引表中有张单独的索引页记录信息，这需要一张干净的索引页。
- 图书区单元格， 实际存放 `论程序员的自我修养` 内容的单元格，占用多少单元格取决于书的大小。这需要一批干净的单元格。

注意虽然索引页编号是按顺序编的，一个号接一个号的在索引表这本书里。但是图书馆运营久了有些书是会被销毁的，例如: 条形码为200的 `delphi程序设计` 这本书太老太久没人借站着位置就被销毁了，但擦涂的是第200索引页上的记录， 第200页这张纸还是存在的，一直在 199 - 201 页之间。擦洗干净了谁管你以前是干什么的， 又可用于记录新书的索引信息。那么如何能快速的知道哪些索引页和图书区单元格没有被使用呢？ 答案是: **索引页位图块 和 图书区单元格位图块**

可以把索引页位图看成是一本书，书的内容是一页一页的位图页，存放这本书的单元格叫 **索引页位图块**。 将 **位图页** 画成 `100*100` 的如下格子

```
01010101101011010101010111010101010110101 1
10101101010101011010110101101010101010101 0
11010101010110101101011010101011010110101 1
01011010110101010101100111010110101101011 0
10101101010101011010110101101010101010101 0
11010101010110101101011010101011010110101 1
01111010110101010101010111010110101101011 0
01011010110101010101100111010110101101011 0
10101101010101011010110101101010101010101 0
....
```

每一位代表一个索引页的使用情况，上面说了1000万本图书对应就会有1000万张索引页，而一张位图页能标识100*100=1万张索引页的使用情况。总的计算公式就是: 1000万索引页/(100*100)=1千张位图页/1000=1个索引页位图块 也就是说只需要一个格子就能装下1000万的索引占用情况。

同样的道理适用于 **图书区单元格位图块**，它记录的是图书区单元格的使用情况。位图是最简单最高效的记录两种状态是否变更的方法。

如何捐书

有了以上的基础，小王捐书的流程就简单了。

- 从索引页位图中找一个没有被使用 `0` 的索引页，同时在页中标记为 `1` ，比如 条形码为 `9527` 的可用
- 根据书本的大小来计算需要多少个数据块来存放 `论程序员的自我修养` ，因为程序眼屌丝的书都很厚，例如 9888页，一个单元格放1000页，就需要10个单元格。
- 再从数据块位图中找到没有被使用 `0` 的10个单元格，同时在页中标记为 `1` ，比如数据块编号 `3，89，765，...` ，因不断的变动，很大概率是找不到连续的单元格。
- 这时就可以创建属于 `论程序员的自我修养` 这本书的索引页了。如下

```
名称: 论程序员的自我修养
大小: 9888页       数据块号: 10        单元格大小: 1000页   属于普通文件
  条形码: 9527     硬链接数: 1人
权限 : (0644/-rw-r-----) 用户ID : ( 4/  屌丝小王)  组ID : (2/ 技术部)
访问时间: 2021-08-04 03:07:21.683190622 -0700
修改书籍时间: 2021-08-04 00:45:34.733766830 -0700
修改索引时间: 2021-08-04 01:20:14.314343117 -0700
数据块位置:3，89，765，...
```

目录项

能否用小易的方案记录以下这种信息关系？

```
├── 金庸小说全集 (条形码:322 )
│   ├── 射雕英雄传 (条形码:1245 )
│   ├── 神雕侠侣  (条形码:23456 )
│   ├── 鹿鼎记    (条形码:34567 )
```

其实也是可以的`金庸小说全集`虽看似一个目录，但他们在索引区没有太多的区别，金庸小说全集目录同样有一张索引页，内容如下:

```
名称: 金庸小说全集/
大小: 1页       数据块号: 1         单元格大小: 1000页   目录
条形码: 322     捆绑数: 17
权限 : (0755/drwxr-xr-x) 用户ID: ( 10/  小张)  组ID : (2/ 技术部)
Access: 2021-08-03 22:11:49.021942010 -0700
Modify: 2021-07-23 18:53:38.656550199 -0700
Change: 2021-07-23 18:53:38.656550199 -0700
数据块位置:15
```

- 唯一的差别是索引页中挑明了它是个目录。这个等于告诉了管理员如何取数据块的数据。目录中的内容如射雕英雄传的索引信息并不在金庸小说全集的索引页中体现。因为索引页的大小是有限的，不能承载太多的内容，不确定的因素都移到了数据块区。
- 数据块位置:`15`中存的是`射雕英雄传`等的条形码，`1245，23456，34567`，有了条形码就能找到索引页，找到索引页就找到了一切

映射关系

小易的方案基本是文件系统的底层实现。理解了这套方案对后续基于源码理解鸿蒙文件系统的实现会变得简单， 图书馆系统和计算机文件系统概念映射关系如下

```
小易方案        - > ext 文件系统
画格子画分区过程 - > 格式化(formate)
图书馆营业       - > 挂载(mount)
大A区           - > 块组(group)
统计区          - > 超级块(super block)
分区描述列表     - > 块组描述符(GDT)
索引表块        - > 索引表(index table)
索引页          - > 索引节点(inode)
条形码          - > 索引编号(inode.id)
图书区位图块     - > 数据块位图(Blocks bitmap)
索引页位图块     - > 索引位图(inode bitmap)
单元格          - > 逻辑块(Blocks)
目录区          - > 索引块(inode Blocks)
图书区          - > 数据块(date Blocks)
```

问题

思考一个问题

- 如果本市有好几个图书馆，并且各自采用了不同的管理方案，又该如何统一管理呢？ 比如要将A图书馆的书移动到B图书馆该如何实现呢？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

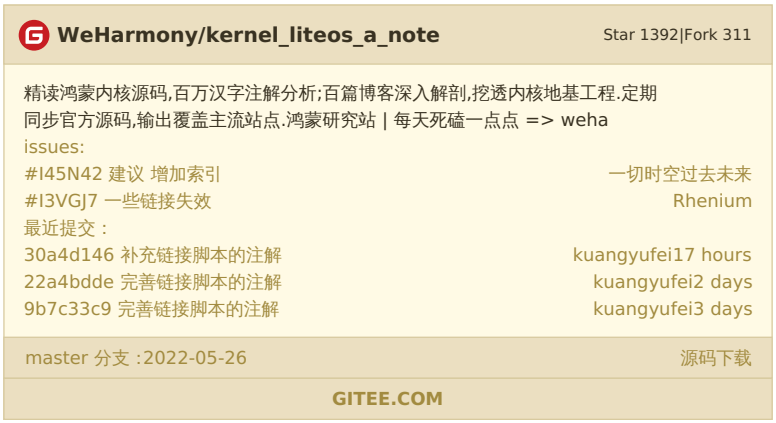
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

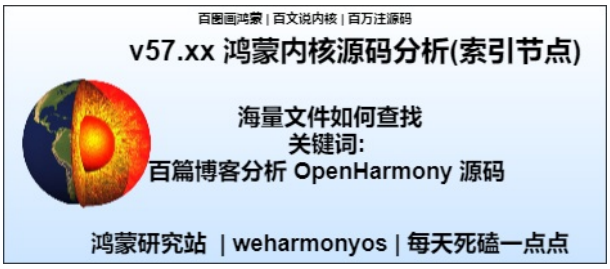


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

57_索引节点篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

读懂鸿蒙内核的关键线索是 LOS_DL_LIST (双向链表)，它是系列篇开篇的内容。 而读懂文件系统的关键线索是 vnode (索引节点)， vnode 在文件系统中起承上启下的关键点。 vnode 是 BSD 的叫法，鸿蒙沿用了 BSD 的称呼， linux 的叫法是 inode ，关于 vnode 有翻译成虚拟节点，但系列篇还是统一翻译成索引节点。

什么是 vnode

先看大佬们对其的定义

OpenBSD 定义

> A vnode is an object in kernel memory that speaks the UNIX file interface (open， read， write， close， readdir， etc.). Vnodes can represent files， directories， FIFOs， domain sockets， block devices， character devices.

> vnode 是内核内存中的一个对象，它使用 UNIX 文件接口（打开、读取、写入、关闭、readdir 等）. Vnodes 可以代表文件、目录、管道、套接字、块设备、字符设备。

freeBSD 定义

> vnode -- internal representation of a file or directory . The vnode is the focus of all file activity in UNIX. A vnode is described by struct vnode. There is a unique vnode allocated for each active file， each current directory， each mounted-on file， text file， and the root.

> vnode -- 文件或目录的内部表示。 vnode 是 UNIX 中所有文件活动的焦点。 vnode 由 struct vnode 描述。 为每个活动文件、每个当前目录、每个挂载文件、文本文件和根分配了一个唯一的 vnode。

linux 定义

> The inode (index node) is a data structure in a Unix-style file system that describes a file-system object such as a file or a directory. Each inode stores the attributes and disk block locations of the object's data.[1] File-system object attributes may include metadata (times of last change， access， modification)， as well as owner and permission data.

> inode（索引节点）是 Unix 风格的文件系统中的一种数据结构，用于描述文件系统对象，例如文件或目录。 每个 inode 存储对象数据的属性和磁盘块位置。 文件系统对象属性可能包括元数据（上次更改、访问、修改的时间），以及所有者和权限数据。

综上所述，发现木有，这说的可不就是 [v63.xx 鸿蒙内核源码分析(文件系统篇) | 用图书管理说文件系统] 中的索引页吗？ 没读过的建议先阅读后再继续。对于在硬盘中的 vnode ，在系统启动后 vnode 会被加载到内存管理，但因内存问题并不会全部加载。

vnode 长啥样

Vnode 是具体文件或目录在VFS层的抽象封装，它屏蔽了不同文件系统的差异，实现资源的统一管理。Vnode 通过哈希以及LRU机制进行管理。当系统启动后，对文件或目录的访问会优先从哈希链表中查找 Vnode 缓存，若缓存没有命中，则并从对应文件系统中搜索目标文件或目录，创建并缓存对应的 Vnode 。当 Vnode 缓存数量达到上限时，将淘汰长时间未访问的 Vnode ，其中挂载点 Vnode 与设备节点 Vnode 不参与淘汰。 Vnode 节点主要有以下几种类型：

- 挂载点：挂载具体文件系统，如 / 、 /storage
- 设备节点： /dev 目录下的节点，对应于一个设备，如 /dev/mmcblk0
- 文件/目录节点：对应于具体文件系统中的文件 / 目录，如 /bin/init

节点创建流程如图

本篇主要围绕 vnode 结构体来说，说透说烂这个文件系统最关键的节点。

```
struct IATTR { //此结构用于记录 vnode 的属性
  /* This structure is used for record vnode attr. */
  unsigned int attr_chg_valid;//节点改变有效性 (CHG_MODE | CHG_UID | ... )
  unsigned int attr_chg_flags;//额外的系统与用户标志（flag），用来保护该文件
  unsigned attr_chg_mode; //确定了文件的类型，以及它的所有者、它的group、其它用户访问此文件的权限 (S_IWUSR | ...)
  unsigned attr_chg_uid; //用户ID
  unsigned attr_chg_gid; //组ID
  unsigned attr_chg_size; //节点大小
  unsigned attr_chg_atime; //节点最近访问时间
  unsigned attr_chg_mtime; //节点对应的文件内容被修改时间
  unsigned attr_chg_ctime; //节点自身被修改时间
};
// 对IATTR的修改最终将落到 vnode->vop->Chattr(vnode， attr);
enum VnodeType {//节点类型
    VNODE_TYPE_UNKNOWN，       /* unknown type */ //未知类型
    VNODE_TYPE_REG，           /* regular file */ //vnode代表一个正则文件(普通文件)
    VNODE_TYPE_DIR，           /* directory */    //vnode代表一个目录
    VNODE_TYPE_BLK，           /* block device */ //vnode代表一个块设备
    VNODE_TYPE_CHR，           /* char device */  //vnode代表一个字符设备
    VNODE_TYPE_BCHR，          /* block char mix device *///块和字符设备混合
    VNODE_TYPE_FIFO，          /* pipe */   //vnode代表一个管道
    VNODE_TYPE_LNK，           /* link */   //vnode代表一个符号链接
};
struct Vnode {//vnode并不包含文件名，因为 vnode和文件名是 1:N 的关系
    enum VnodeType type;               /* vnode type */ //节点类型 (文件|目录|链接...)
    int useCount;                      /* ref count of users *///节点引用(链接)数，即有多少文件名指向这个vnode，即上层理解的硬链接数
    uint32_t hash;                     /* vnode hash */ //节点哈希值
    uint uid;                         /* uid for dac */ //文件拥有者的User ID
    uint gid;                         /* gid for dac */ //文件的Group ID
    mode_t mode;                          /* mode for dac */ //chmod 文件的读、写、执行权限
    LIST_HEAD parentPathCaches;         /* pathCaches point to parents */ //指向父级路径缓存，上面的都是当了爸爸节点
    LIST_HEAD childPathCaches;          /* pathCaches point to children */ //指向子级路径缓存，上面都是当了别人儿子的节点
    struct Vnode *parent;              /* parent vnode */ //父节点
    struct VnodeOps *vop;               /* vnode operations */ //相当于指定操作Vnode方式 (接口实现|驱动程序)
    struct file_operations_vfs *fop;    /* file operations */ //相当于指定文件系统
    void *data;                       /* private data */  //文件数据block的位置，指向每种具体设备私有的成员，例如 ( drv_data | nfsnode | ....)
    uint32_t flag;                     /* vnode flag */  //节点标签
    LIST_ENTRY hashEntry;              /* list entry for bucket in hash table */ //通过它挂入哈希表 g_vnodeHashEntrys[i]， i:[0，g_vnodeHashMask]
    LIST_ENTRY actFreeEntry;           /* vnode active/free list entry */ //通过本节点挂到空闲链表和使用链表上
    struct Mount *originMount;         /* fs info about this vnode */ //自己所在的文件系统挂载信息
    struct Mount *newMount;            /* fs info about who mount on this vnode */ //其他挂载在这个节点上文件系统信息
};
```

解读

- VnodeType 即七种文件类型，鸿蒙增加了一种 VNODE_TYPE_BCHR ，去掉了 socket 类型，没搞懂为什么。
- useCount 代表硬链接数，任何目录下都会有 . ， .. 两个文件， 前者指向当前目录，后者指向父目录。这样做的好处是由索引页指向的数据块中(目录项)存有父目录和当前目录的索引号，有了索引号就能很快的找到对应的索引页。例如当外部使用 cd ../../../ 这样的命令时，只需在当前目录(inode)所指向的目录项中查找 .. 的索引号。这样是非常的快捷和方便的，用自己勤劳的双手就能解决的困扰何必去麻烦别人呢。因为被下级留有记录所以硬链接数会增加。会增加多少呢？ 举例说明， stat 命令用于查看索引节点信息

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a$ stat kernel
File: kernel
Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 1099218   Links: 7
Access : (0755/drwxr-xr-x)  Uid : ( 1000/  turing)   Gid : ( 1000/  turing)
```

注意 `Inode: 1099218` ，而 `Links: 7` 代表 `kernel` 被七个地方所关联，除了自己应该还有六个，在哪呢？ 用 `ll -a` 命令展开 `kernel` 看看

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a/kernel$ ll -a
total 36
drwxr-xr-x  7 turing turing 4096 Jun 21 02:38 ./
drwxr-xr-x 21 turing turing 4096 Jul 23 19:45 ../
drwxr-xr-x 11 turing turing 4096 Jun 21 02:38 base/
-rwxr-xr-x  1 turing turing 2214 Jun 21 02:38 BUILD.gn*
drwxr-xr-x  3 turing turing 4096 Jun 21 02:38 common/
drwxr-xr-x  9 turing turing 4096 Jun 21 02:38 extended/
drwxr-xr-x  2 turing turing 4096 Jun 21 02:38 include/
-rwxr-xr-x  1 turing turing 2864 Jun 21 02:38 Kconfig*
drwxr-xr-x  4 turing turing 4096 Jun 21 02:38 user/
```

发现包括 `.` ， `..` 在内有七个目录 `d` 代表的是目录，但是注意其中的 `../` 并不指向 `kernel` 而是指向它的父级 `liteos_a` ，其余的 `./` ， `base/..` ， `common/..` 六个刚好指向 `kernel` ，可以验证下它们的 inode 信息就知道了。

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a$ stat ./kernel/.
File: ./kernel/.
Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 1099218   Links: 7
Access : (0755/drwxr-xr-x)  Uid : ( 1000/  turing)   Gid : ( 1000/  turing)
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a$ stat ./kernel/base/..
  File: ./kernel/base/..
  Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 1099218   Links: 7
Access : (0755/drwxr-xr-x)  Uid : ( 1000/  turing)   Gid : ( 1000/  turing)
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a$ stat ./kernel/..
File: ./kernel/..
Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 1099213   Links: 21
Access : (0755/drwxr-xr-x)  Uid : ( 1000/  turing)   Gid : ( 1000/  turing)
```

会发现 `./kernel/.` 和 `./kernel/base/..` 的 `Inode` 都是 `1099218` ，而 `./kernel/..` 的为 `1099213` 是不一样。

- 正常情况下一个目录的 `.` ， `..` 是不一样的，但只有一个目录例外，就是 `/`

```
turing@ubuntu:/$ stat /.
File: /.
Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 2         Links: 20
turing@ubuntu:/$ stat /..
File: /..
Size: 4096      Blocks: 8          IO Block: 4096   directory
Device: 805h/2053d Inode: 2         Links: 20
```

其 `inode` 结果都是一样的， `Inode: 2` ， `inode` 号对应什么文件可以使用 "find / -inum NUM" 来查看。同时请思考两个问题。

  - 为什么 `/` 的 `inode` 的编号一定是2 ？ `inode` 为 `0` 和 `1` 的节点又去哪了呢？
  - `inode` 编号真的是唯一的吗？ 不同的文件系统可以有相同编号的 `inode` 吗？ 如果可以有，那上层又是如何确保全局唯一的呢？

- `uid` ， `gid``mode` 代表文件所属用户/用户组和权限。discretionary access control (DAC) 自主访问控制。在计算机安全中，自主访问控制 (DAC) 是一种由可信计算机系统评估标准定义的访问控制“作为一种根据对象所属的主体和组的身份限制对对象的访问的手段。 控制方式是自由的，因为具有特定访问权限的主体能够将该权限（可能是间接地）传递给任何其他主体（除非受到强制访问控制的约束）。与其对应的是 mandatory access control (MAC) 强制访问控制。

- `parentPathCaches``childPathCaches` 路径缓存链表，用户快速查找父子信息。

- `parent` 指向父节点，父节点不管是什么内容，一样都是文件，都用 `Vnode` 描述。

- `VnodeOps *vop` 这是对 `vnode` 的操作， `vnode` 本身也是数据，存储在索引表中，记录了用户，用户组，权限，时间等信息，这部分信息是可以修改的，就需要接口来维护，便是 `VnodeOps` 。

```
struct VnodeOps {
    int (*Create)(struct Vnode *parent， const char *name， int mode， struct Vnode **vnode);//创建节点
  int (*Lookup)(struct Vnode *parent， const char *name， int len， struct Vnode **vnode);//查询节点
  //Lookup向底层文件系统查找获取inode信息
  int (*Open)(struct Vnode *vnode， int fd， int mode， int flags);//打开节点
  int (*Close)(struct Vnode *vnode);//关闭节点
  int (*Reclaim)(struct Vnode *vnode);//回收节点
  int (*Unlink)(struct Vnode *parent， struct Vnode *vnode， const char *fileName);//取消硬链接
  int (*Rmdir)(struct Vnode *parent， struct Vnode *vnode， const char *dirName);//删除目录节点
  int (*Mkdir)(struct Vnode *parent， const char *dirName， mode_t mode， struct Vnode **vnode);//创建目录节点
  int (*Readdir)(struct Vnode *vnode， struct fs_dirent_s *dir);//读目录节点
  int (*Opendir)(struct Vnode *vnode， struct fs_dirent_s *dir);//打开目录节点
  int (*Rewinddir)(struct Vnode *vnode， struct fs_dirent_s *dir);//定位目录节点
  int (*Closedir)(struct Vnode *vnode， struct fs_dirent_s *dir);//关闭目录节点
  int (*Getattr)(struct Vnode *vnode， struct stat *st);//获取节点属性
  int (*Setattr)(struct Vnode *vnode， struct stat *st);//设置节点属性
  int (*Chattr)(struct Vnode *vnode， struct IATTR *attr);//改变节点属性(change attr)
  int (*Rename)(struct Vnode *src， struct Vnode *dstParent， const char *srcName， const char *dstName);//重命名
   ....
}
```

看到没有里面的所有方法都是对索引节点(索引页)的增删改查操作，并不操作索引节点指向的数据块(图书区)内容。 各个文件系统都要去实现这些接口。

```
//文件系统(fat)实现对索引节点的操作
struct VnodeOps fatfs_vops = {
    /* file ops */
    .Getattr = fatfs_stat，
    .Chattr = fatfs_chattr，
    .Lookup = fatfs_lookup，
    .Rename = fatfs_rename，
    .Create = fatfs_create，
    .Unlink = fatfs_unlink，
    .Reclaim = fatfs_reclaim，
    .Truncate = fatfs_truncate，
    .Truncate64 = fatfs_truncate64，
    /* dir ops */
    .Opendir = fatfs_opendir，
    .Readdir = fatfs_readdir，
    .Rewinddir = fatfs_rewinddir，
    .Closedir = fatfs_closedir，
    .Mkdir = fatfs_mkdir，
    .Rmdir = fatfs_rmdir，
    .Fscheck = fatfs_fscheck，
    .Symlink = fatfs_symlink，
    .Readlink = fatfs_readlink，
};
```

- 那么对数据块(图书区)的修改用什么方法呢？ 答案是: `file_operations_vfs` 。

```
//该结构由设备在向系统注册时提供，它用于回调以执行特定于设备的操作。
struct file_operations_vfs
{
  int     (*open)(struct file *filep);
  int     (*close)(struct file *filep);
  ssize_t (*read)(struct file *filep， char *buffer， size_t buflen);
  ssize_t (*write)(struct file *filep， const char *buffer， size_t buflen);
  off_t   (*seek)(struct file *filep， off_t offset， int whence);
  int     (*ioctl)(struct file *filep， int cmd， unsigned long arg);
  int     (*mmap)(struct file* filep， struct VmMapRegion *region);
};
struct file_operations_vfs fatfs_fops = {
    .open = fatfs_open，
    .read = fatfs_read，
```

```
    .write = fatfs_write ,
    .seek = fatfs_lseek ,
    .close = fatfs_close ,
    .mmap = OsVfsFileMmap ,
    .fallocate = fatfs_fallocate ,
    .fallocate64 = fatfs_fallocate64 ,
    .fsync = fatfs_fsync ,
    .ioctl = fatfs_ioctl ,
};
```

`file_operations_vfs` 看参数就知道，很是给 `vnode` 的上层的使用的，它是夹在应用层和 `vnode` 中间的一层，是 `vnode` 起承上启下作用的上层，具体为什么要有 `file` 存在后续会详细说明，总之通过 `file` 找到 `vnode` ，从而对 `vnode` 指向的内容区进行修改。我们在应用层比如修改一个 `ppt` ，创建一个 `word` 文档这些操作就是通过 `file_operations_vfs` 。 一定要搞清楚 `VnodeOps` 和 `file_operations_vfs` 二者的区别，一个是对索引页的操作，一个是对索引页指向的内容的操作。

- `data` 使用了一个 `void` 类型，这是私有格式数据，说明运行时才知道是什么类型，就像一个没有任何提示信息的私人密码箱一样，是打不开的，不知道顺序乱开只会毁掉数据，只有密码箱那边派人来了才能开，而这人就是各种不同的文件系统。每种文件系统如何读取数据的方式是不同的，差异化的就有接口内部来实现了。对外是相同的，无非都是读读写写。
- `hashEntry` 使用哈希算法来检索 `vnode`
- `actFreeEntry` :这个就不用介绍了，双向链表是内核最重要的结构体，通过它挂到全局空闲链表和使用链表上。
- `originMount` 和 `newMount` 是挂载相关的，任何文件系统都需要先挂载到根文件系统下才能使用。关于挂载后续有详细介绍。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

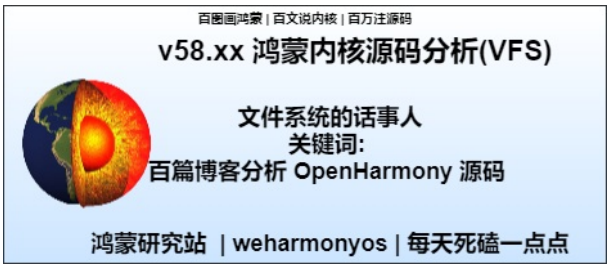
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

58_VFS篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

基本概念 | 官方定义

VFS （Virtual File System）是文件系统的虚拟层，它不是一个实际的文件系统，而是一个异构文件系统之上的软件粘合层，为用户提供统一的类Unix文件操作接口。由于不同类型的文件系统接口不统一，若系统中有多个文件系统类型，访问不同的文件系统就需要使用不同的非标准接口。而通过在系统中添加VFS层，提供统一的抽象接口，屏蔽了底层异构类型的文件系统的差异，使得访问文件系统的系统调用不用关心底层的存储介质和文件系统类型，提高开发效率。

OpenHarmony 内核中， VFS 框架是通过在内存中的树结构来实现的，树的每个结点都是一个 Vnode 结构体，父子结点的关系以 PathCache 结构体保存。 VFS 最主要的两个功能是：

- 查找节点。
- 统一调用（标准）。

VFS 层具体实现包括四个方面:

- 通过三大函数指针操作接口，实现对不同文件系统类型调用不同接口实现标准接口功能；
- 通过 Vnode 与 PathCache 机制，提升路径搜索以及文件访问的性能；
- 通过挂载点管理进行分区管理；
- 通过FD管理进行进程间FD隔离等。

三大操作接口

VFS 层通过函数指针的形式，将统一调用按照不同的文件系统类型，分发到不同文件系统中进行底层操作。各文件系统的各自实现一套Vnode操作(VnodeOps)、挂载点操作(MountOps)以及文件操作接口(file_operations_vfs)，并以函数指针结构体的形式存储于对应 Vnode 、挂载点、 File 结构体中，实现 VFS 层对下访问。这三个接口分别为:

VnodeOps | 操作 Vnode 节点

```
struct VnodeOps {
    int (*Create)(struct Vnode *parent， const char *name， int mode， struct Vnode **vnode);//创建节点
    int (*Lookup)(struct Vnode *parent， const char *name， int len， struct Vnode **vnode);//查询节点
    //Lookup向底层文件系统查找获取inode信息
    int (*Open)(struct Vnode *vnode， int fd， int mode， int flags);//打开节点
    int (*Close)(struct Vnode *vnode);//关闭节点
    int (*Reclaim)(struct Vnode *vnode);//回收节点
```

```
    int (*Unlink)(struct Vnode *parent， struct Vnode *vnode， const char *fileName);//取消硬链接
    int (*Rmdir)(struct Vnode *parent， struct Vnode *vnode， const char *dirName);//删除目录节点
    int (*Mkdir)(struct Vnode *parent， const char *dirName， mode_t mode， struct Vnode **vnode);//创建目录节点
    /*
    创建一个目录时，实际做了3件事：在其“父目录文件”中增加一个条目；分配一个inode；再分配一个存储块，
    用来保存当前被创建目录包含的文件与子目录。被创建的“目录文件”中自动生成两个子目录的条目，名称分别是：“.”和“..”。
    前者与该目录具有相同的inode号码，因此是该目录的一个“硬链接”。后者的inode号码就是该目录的父目录的inode号码。
    所以，任何一个目录的"硬链接"总数，总是等于它的子目录总数（含隐藏目录）加2。即每个“子目录文件”中的“..”条目，
    加上它自身的“目录文件”中的“.”条目，再加上“父目录文件”中的对应该目录的条目。
    */
    int (*Readdir)(struct Vnode *vnode， struct fs_dirent_s *dir);//读目录节点
    int (*Opendir)(struct Vnode *vnode， struct fs_dirent_s *dir);//打开目录节点
    int (*Rewinddir)(struct Vnode *vnode， struct fs_dirent_s *dir);//定位目录节点
    int (*Closedir)(struct Vnode *vnode， struct fs_dirent_s *dir);//关闭目录节点
    int (*Getattr)(struct Vnode *vnode， struct stat *st);//获取节点属性
    int (*Setattr)(struct Vnode *vnode， struct stat *st);//设置节点属性
    int (*Chattr)(struct Vnode *vnode， struct IATTR *attr);//改变节点属性(change attr)
    int (*Rename)(struct Vnode *src， struct Vnode *dstParent， const char *srcName， const char *dstName);//重命名
    int (*Truncate)(struct Vnode *vnode， off_t len);//缩减或扩展大小
    int (*Truncate64)(struct Vnode *vnode， off64_t len);//缩减或扩展大小
    int (*Fscheck)(struct Vnode *vnode， struct fs_dirent_s *dir);//检查功能
    int (*Link)(struct Vnode *src， struct Vnode *dstParent， struct Vnode **dst， const char *dstName);
    int (*Symlink)(struct Vnode *parentVnode， struct Vnode **newVnode， const char *path， const char *target);
    ssize_t (*Readlink)(struct Vnode *vnode， char *buffer， size_t bufLen);
};
```

MountOps | 挂载点操作

```
//挂载操作
struct MountOps {
    int (*Mount)(struct Mount *mount， struct Vnode *vnode， const void *data);//挂载
    int (*Unmount)(struct Mount *mount， struct Vnode **blkdriver);//卸载
    int (*Statfs)(struct Mount *mount， struct statfs *sbp);//统计文件系统的信息，如该文件系统类型、总大小、可用大小等信息
};
```

file_operations_vfs | 文件操作接口

```
struct file_operations_vfs
{
  int     (*open)(struct file *filep); //打开文件
  int     (*close)(struct file *filep); //关闭文件
  ssize_t (*read)(struct file *filep， char *buffer， size_t buflen); //读文件
  ssize_t (*write)(struct file *filep， const char *buffer， size_t buflen);//写文件
  off_t   (*seek)(struct file *filep， off_t offset， int whence);//寻找，检索 文件
  int     (*ioctl)(struct file *filep， int cmd， unsigned long arg);//对文件的控制命令
  int     (*mmap)(struct file* filep， struct VmMapRegion *region);//内存映射实现<文件/设备 - 线性区的映射>
  /* The two structures need not be common after this point */

#ifndef CONFIG_DISABLE_POLL
  int     (*poll)(struct file *filep， poll_table *fds); //轮询接口
#endif
  int     (*stat)(struct file *filep， struct stat* st); //统计接口
  int     (*fallocate)(struct file* filep， int mode， off_t offset， off_t len);
  int     (*fallocate64)(struct file *filep， int mode， off64_t offset， off64_t len);
  int     (*fsync)(struct file *filep);
  ssize_t (*readpage)(struct file *filep， char *buffer， size_t buflen);
  int     (*unlink)(struct Vnode *vnode);
};
```

PathCache | 路径缓存

PathCache 是路径缓存，它通过哈希表存储，利用父节点 Vnode 的地址和子节点的文件名，可以从 PathCache 中快速查找到子节点对应的 Vnode 。当前 PageCache 仅支持缓存二进制文件，在初次访问文件时通过 mmap 映射到内存中，下次再访问时，直接从 PageCache 中读取，可以提升对同一个文件的读写速度。另外基于 PageCache 可实现以文件为基底的进程间通信。下图展示了文件/目录的查找流程。

```
LIST_HEAD g_pathCacheHashEntrys[LOSCFG_MAX_PATH_CACHE_SIZE]; //路径缓存哈希表项
struct PathCache {//路径缓存
    struct Vnode *parentVnode;    /* vnode points to the cache */
    struct Vnode *childVnode;     /* vnode the cache points to */
    LIST_ENTRY parentEntry;       /* list entry for cache list in the parent vnode */
    LIST_ENTRY childEntry;        /* list entry for cache list in the child vnode */
    LIST_ENTRY hashEntry;         /* list entry for buckets in the hash table */
    uint8_t nameLen;              /* length of path component */
#ifdef LOSCFG_DEBUG_VERSION
    int hit;                      /* cache hit count*/
#endif
    char name[0];                 /* path component name */
};
//路径缓存初始化
int PathCacheInit(void)
{
    for (int i = 0; i < LOSCFG_MAX_PATH_CACHE_SIZE; i++) {
        LOS_ListInit(&g_pathCacheHashEntrys[i]);
    }
    return LOS_OK;
}
```

挂载点管理

当前OpenHarmony内核中，对系统中所有挂载点通过链表进行统一管理。挂载点结构体中，记录了该挂载分区内的所有Vnode。当分区卸载时，会释放分区内的所有Vnode。

```
static LIST_HEAD *g_mountList = NULL;//挂载链表，上面挂的是系统所有挂载点
struct Mount {
    LIST_ENTRY mountList;              /* mount list */    //通过本节点将Mount挂到全局Mount链表上
    const struct MountOps *ops;        /* operations of mount */ //挂载操作函数
    struct Vnode *vnodeBeCovered;      /* vnode we mounted on */ //要被挂载的节点 即 /bin1/vs/sd 对应的 vnode节点
    struct Vnode *vnodeCovered;        /* syncer vnode */   //要挂载的节点 即/dev/mmcblk0p0 对应的 vnode节点
    struct Vnode *vnodeDev;            /* dev vnode */
    LIST_HEAD vnodeList;               /* list of vnodes */  //链表表头
    int vnodeSize;                     /* size of vnode list */ //节点数量
    LIST_HEAD activeVnodeList;         /* list of active vnodes */ //激活的节点链表
    int activeVnodeSize;               /* szie of active vnodes list *///激活的节点数量
    void *data;                        /* private data */ //私有数据，可使用这个成员作为一个指向它们自己内部数据的指针
    uint32_t hashseed;                 /* Random seed for vfs hash */ //vfs 哈希随机种子
    unsigned long mountFlags;          /* Flags for mount */ //挂载标签
    char pathName[PATH_MAX];           /* path name of mount point */ //挂载点路径名称  /bin1/vs/sd
    char devName[PATH_MAX];            /* path name of dev point */  //设备名称 /dev/mmcblk0p0
};
//分配一个挂载点
struct Mount* MountAlloc(struct Vnode* vnodeBeCovered ,  struct MountOps* fsop)
{
    struct Mount* mnt = (struct Mount*)zalloc(sizeof(struct Mount));//申请一个mount结构体内存，小内存分配用 zalloc
    if (mnt == NULL) {
        PRINT_ERR("MountAlloc failed no memory!\n");
        return NULL;
    }

    LOS_ListInit(&mnt->activeVnodeList);//初始化激活索引节点链表
    LOS_ListInit(&mnt->vnodeList);//初始化索引节点链表

    mnt->vnodeBeCovered = vnodeBeCovered;//设备将装载到vnodeBeCovered节点上
    vnodeBeCovered->newMount = mnt;//该节点不再是虚拟节点，而作为 设备结点
#ifdef LOSCFG_DRIVERS_RANDOM //随机值 驱动模块
    HiRandomHwInit();//随机值初始化
    (VOID)HiRandomHwGetInteger(&mnt->hashseed);//用于生成哈希种子
    HiRandomHwDeinit();//随机值反初始化
#else
    mnt->hashseed = (uint32_t)random(); //随机生成哈子种子
#endif
    return mnt;
}
```

fd管理 | 两种描述符/句柄的关系

Fd（File Descriptor）是描述一个打开的文件/目录的描述符。当前OpenHarmony内核中，fd总规格为896，分为三种类型：

- 普通文件描述符，系统总数量为512。

```
#define CONFIG_NFILE_DESCRIPTORS    512 // 系统文件描述符数量
```

- Socket描述符，系统总规格为128。

```
#define LWIP_CONFIG_NUM_SOCKETS         128 //socket链接数量
#define CONFIG_NSOCKET_DESCRIPTORS  LWIP_CONFIG_NUM_SOCKETS
```

- 消息队列描述符，系统总规格为256。

```
#define CONFIG_NQUEUE_DESCRIPTORS    256
```

请记住，在OpenHarmony内核中，在不同的层面会有两种文件句柄::

- 系统文件描述符(sysfd)，由内核统一管理，和进程描述符形成映射关系，一个 sysfd 可以被多个 profd 映射，也就是说打开一个文件只会占用一个 sysfd ，但可以占用多个 profd ，即一个文件被多个进程打开。
- 进程文件描述符(profd)，由进程管理的叫进程文件描述符，内核对不同进程中的 fd 进行隔离，即进程只能访问本进程的 fd 。举例说明之间的关系:

```
文件            sysfd     profd
吃个桃桃.mp4        10    13(A进程)
吃个桃桃.mp4        10    3(B进程)
容嬷嬷被冤枉.txt    12    3(A进程)
容嬷嬷被冤枉.txt    12    3(C进程)
```

- 不同进程的相同 fd 往往指向不同的文件，但有三个 fd 例外
  - STDIN_FILENO(fd = 0) 标准输入 接收键盘的输入
  - STDOUT_FILENO(fd = 1) 标准输出 向屏幕输出
  - STDERR_FILENO(fd = 2) 标准错误 向屏幕输出 sysfd 和所有的 profd 的(0，1，2)号都是它们。熟知的 printf 就是向 STDOUT_FILENO 中写入数据。
- 具体涉及结构体

```
struct file_table_s {//进程fd <--> 系统FD绑定
    intptr_t sysFd; /* system fd associate with the tg_filelist index */
};//sysFd的默认值是-1
struct fd_table_s {//进程fd表结构体
    unsigned int max_fds;//进程的文件描述符最多有256个
    struct file_table_s *ft_fds; /* process fd array associate with system fd *///系统分配给进程的FD数组 ，fd 默认是 -1
    fd_set *proc_fds; //进程fd管理位，用bitmap管理FD使用情况，默认打开了 0，1，2        (stdin，stdout，stderr)
    fd_set *cloexec_fds;
    sem_t ft_sem; /* manage access to the file table */ //管理对文件表的访问的信号量
};
struct files_struct {//进程文件表结构体
    int count;          //持有的文件数量
    struct fd_table_s *fdt; //持有的文件表
    unsigned int file_lock; //文件互斥锁
    unsigned int next_fd;   //下一个fd
#ifdef VFS_USING_WORKDIR
    spinlock_t workdir_lock; //工作区目录自旋锁
    char workdir[PATH_MAX];  //工作区路径，最大 256个字符
#endif
};
typedef struct ProcessCB {
#ifdef LOSCFG_FS_VFS
    struct files_struct *files;        /**< Files held by the process */ //进程所持有的所有文件，注者称之为进程的文件管理器
#endif //每个进程都有属于自己的文件管理器，记录对文件的操作。 注意:一个文件可以被多个进程操作
```

```
}
```

解读

- 鸿蒙的每个进程`ProcessCB`都有属于自己的进程的文件描述符`files_struct`，该进程和文件系统有关的信息都由它表达。
- 搞清楚`files_struct`，`fd_table_s`，`file_table_s`三个结构体的关系就明白了进度描述符和系统描述符的关系。
- `fd_table_s`是由`alloc_fd_table`分配的一个结构体数组，用于存放进程的文件描述符

```
//分配进程文件表，初始化 fd_table_s 结构体中每个数据，包括系统FD(0，1，2)的绑定
static struct fd_table_s * alloc_fd_table(unsigned int numbers)
{
  struct fd_table_s *fdt;
  void *data;
  fdt = LOS_MemAlloc(m_aucSysMem0， sizeof(struct fd_table_s));//申请内存
  if (!fdt)
    {
      goto out;
    }
  fdt->max_fds = numbers;//最大数量
  if (!numbers)
    {
      fdt->ft_fds = NULL;
      fdt->proc_fds = NULL;
      return fdt;
    }
  data = LOS_MemAlloc(m_aucSysMem0， numbers * sizeof(struct file_table_s));//这是和系统描述符的绑定
  if (!data)
    {
      goto out_fdt;
    }
  fdt->ft_fds = data;//这其实是个 int[] 数组，
  for (int i = STDERR_FILENO + 1; i < numbers; i++)
    {
       fdt->ft_fds[i].sysFd = -1;//默认的系统描述符都为-1，即还没有和任何系统文件描述符绑定
    }
  data = LOS_MemAlloc(m_aucSysMem0， sizeof(fd_set));//管理FD的 bitmap
  if (!data)
    {
      goto out_arr;
    }
  (VOID)memset_s(data， sizeof(fd_set)， 0， sizeof(fd_set));
  fdt->proc_fds = data;
  alloc_std_fd(fdt);//分配标准的0，1，2系统文件描述符，这样做的结果是任务进程都可以写系统文件(0，1，2)
  (void)sem_init(&fdt->ft_sem， 0， 1);//互斥量初始化
  return fdt;
out_arr:
  (VOID)LOS_MemFree(m_aucSysMem0， fdt->ft_fds);
out_fdt:
  (VOID)LOS_MemFree(m_aucSysMem0， fdt);
out:
  return NULL;
}
```

- `file_table_s`记录`sysfd`和`profd`的绑定关系. `fdt->ft_fds[i].sysFd`中的`i`就是`profd`

百文说内核 | 抓住主脉络

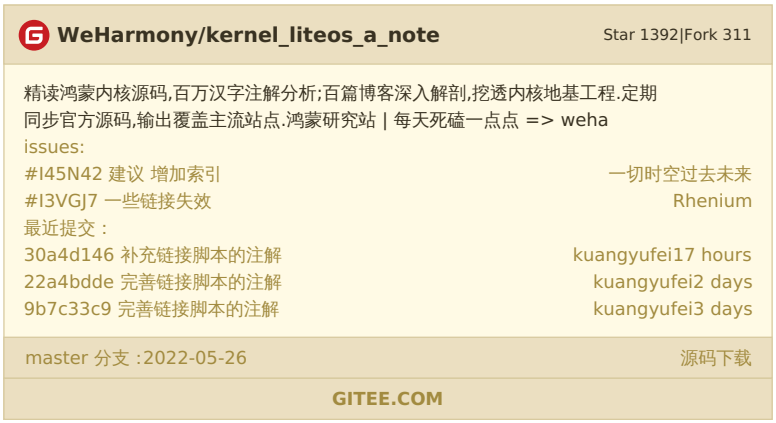
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

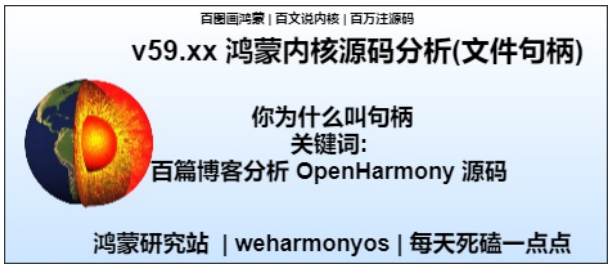


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

59_文件句柄篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

句柄 | handle

```
int open(const char* pathname，int flags);
ssize_t read(int fd， void *buf， size_t count);
ssize_t write(int fd， const void *buf， size_t count);
int close(int fd);
```

只要写过应用程序代码操作过文件不会陌生这几个函数，文件操作的几个关键步骤嘛，跟把大象装冰箱分几步一样。先得把冰箱门打开，再把大象放进去，再关上冰箱门。其中最重要的一个参数就是 fd ，应用程序所有对文件的操作都基于它。 fd 可称为文件描述符，或者叫文件句柄(handle)，个人更愿意称后者。 因为更形象， handle 英文有手柄的意思，跟开门一样，握住手柄才能开门，手柄是进门关门的抓手。映射到文件系统， fd 是应用层出入内核层的抓手。句柄是一个数字编号， open | creat 去申请这个编号，内核会创建文件相关的一系列对象，返回编号，后续通过编号就可以操作这些对象。原理就是这么的简单，本篇将从 fd 入手，跟踪文件操作的整个过程。

请记住，鸿蒙内核中，在不同的层面会有两种文件句柄:

- 系统文件句柄(sysfd)，由内核统一管理，和进程文件句柄形成映射关系，一个 sysfd 可以被多个 profd 映射，也就是说打开一个文件只会占用一个 sysfd ，但可以占用多个 profd ，即一个文件被多个进程打开。
- 进程文件句柄(profd)，由进程管理的叫进程文件句柄，内核对不同进程中的 fd 进行隔离，即进程只能访问本进程的 fd 。举例说明之间的关系:

```
文件           sysfd    profd
吃个桃桃.mp4      10    13(A进程)
吃个桃桃.mp4      10    3(B进程)
容嬷嬷被冤枉.txt   12    3(A进程)
容嬷嬷被冤枉.txt   12    3(C进程)
```

进程文件句柄

在鸿蒙一个进程默认最多可以有 256 个 fd ，即最多可打开256个文件。文件也是资源的一种，系列篇多次说过进程是管理资源的，所以在进程控制块中能看到文件的影子 files_struct 。 files_struct 可理解为进程的文件管理器，里面只放和本进程相关的文件，线程则共享这些文件。另外子进程也会拷贝一份父进程的 files_struct 到自己的 files_struct 上，在父子进程篇中也讲过 fork 的本质就是拷贝资源，其中就包括了文件内容。

```
//进程控制块
typedef struct ProcessCB {
```

```
    //..
    #ifdef LOSCFG_FS_VFS
        struct files_struct *files;        /**< Files held by the process */ //进程所持有的所有文件，注者称之为进程的文件管理器
    #endif //每个进程都有属于自己的文件管理器，记录对文件的操作。 注意:一个文件可以被多个进程操作
} LosProcessCB;
struct files_struct {//进程文件表结构体
    int count;    //持有的文件数量
    struct fd_table_s *fdt; //持有的文件表
    unsigned int file_lock; //文件互斥锁
    unsigned int next_fd; //下一个fd
#ifdef VFS_USING_WORKDIR
    spinlock_t workdir_lock; //工作区目录自旋锁
    char workdir[PATH_MAX];  //工作区路径，最大 256个字符
#endif
};
```

`fd_table_s` 为 `files_struct` 的成员，负责记录所有进程文件句柄的信息，个人觉得鸿蒙这块的实现有点乱，没有封装好。

```
struct fd_table_s {//进程fd表结构体
    unsigned int max_fds;//进程的文件描述符最多有256个
    struct file_table_s *ft_fds; /* process fd array associate with system fd *///系统分配给进程的FD数组 ，fd 默认是 -1
    fd_set *proc_fds; //进程fd管理位，用bitmap管理FD使用情况，默认打开了 0，1，2        (stdin，stdout，stderr)
    fd_set *cloexec_fds;
    sem_t ft_sem; /* manage access to the file table */ //管理对文件表的访问的信号量
};
```

`file_table_s` 记录进程 `fd` 和系统 `fd` 之间的绑定或者说映射关系

```
struct file_table_s {//进程fd <--> 系统fd绑定
    intptr_t sysFd; /* system fd associate with the tg_filelist index */
};
```

`fd_set` 实现了进程 `fd` 按位图管理，系列操作为 `FD_SET` ， `FD_ISSET` ， `FD_CLR` ， `FD_ZERO` 除以 `8` 是因为 `char` 类型占 `8` 个 `bit` 位。请尝试去理解下按位操作的具体实现。

```
typedef struct fd_set
{
  unsigned char fd_bits [(FD_SETSIZE+7)/8];
} fd_set;
#define FD_SET(n， p)  FDSETSAFESET(n， (p)->fd_bits[((n)-LWIP_SOCKET_OFFSET)/8] = (u8_t)((p)->fd_bits[((n)-LWIP_SOCKET_OFFSET)/8] |  (1 <
#define FD_CLR(n， p)  FDSETSAFESET(n， (p)->fd_bits[((n)-LWIP_SOCKET_OFFSET)/8] = (u8_t)((p)->fd_bits[((n)-LWIP_SOCKET_OFFSET)/8] & ~(1
#define FD_ISSET(n，p) FDSETSAFEGET(n， (p)->fd_bits[((n)-LWIP_SOCKET_OFFSET)/8] &   (1 << (((n)-LWIP_SOCKET_OFFSET) & 7)))
#define FD_ZERO(p)    memset((void*)(p)， 0， sizeof(*(p)))
```

vfs_procfd.c 为进程文件句柄实现文件，每个进程的 `0` ， `1` ， `2` 号 `fd` 是由系统占用并不参与分配，即为大家熟知的:

- `STDIN_FILENO(fd = 0)` 标准输入 接收键盘的输入
- `STDOUT_FILENO(fd = 1)` 标准输出 向屏幕输出
- `STDERR_FILENO(fd = 2)` 标准错误 向屏幕输出

```
/* minFd should be a positive number，and 0，1，2 had be distributed to stdin，stdout，stderr */
    if (minFd < MIN_START_FD) {
        minFd = MIN_START_FD;
    }
//分配进程文件句柄
static int AssignProcessFd(const struct fd_table_s *fdt， int minFd)
{
    if (fdt == NULL) {
        return VFS_ERROR;
    }
    if (minFd >= fdt->max_fds) {
        set_errno(EINVAL);
        return VFS_ERROR;
    }
 //从表中搜索未使用的 fd
```

```
    /* search unused fd from table */
    for (int i = minFd; i < fdt->max_fds; i++) {
        if (!FD_ISSET(i， fdt->proc_fds)) {
            return i;
        }
    }
    set_errno(EMFILE);
    return VFS_ERROR;
}
//释放进程文件句柄
void FreeProcessFd(int procFd)
{
    struct fd_table_s *fdt = GetFdTable();

    if (!IsValidProcessFd(fdt， procFd)) {
        return;
    }
    FileTableLock(fdt);
    FD_CLR(procFd， fdt->proc_fds); //相应位清0
    FD_CLR(procFd， fdt->cloexec_fds);
    fdt->ft_fds[procFd].sysFd = -1; //解绑系统文件描述符
    FileTableUnLock(fdt);
}
```

- 分配和释放的算法很简单，由位图的相关操作完成。
- `fdt->ft_fds[i].sysFd` 中的 `i` 代表进程的 `fd`， `-1` 代表没有和系统文件句柄绑定。
- 进程文件句柄和系统文件句柄的意义和关系在 (VFS篇)中已有说明，此处不再赘述，请自行前往翻看。

系统文件句柄

系统文件句柄的实现类似，但它并不在鸿蒙内核项目中，而是在 `NuttX` 项目的 fs_files.c 中， 因鸿蒙内核项目中使用了其他第三方的项目，所以需要加进来一起研究才能看明白鸿蒙整个内核的完整实现。具体涉及的子系统仓库如下:

- **子系统注解仓库**

  在给鸿蒙内核源码加注过程中发现仅仅注解内核仓库还不够，因为它关联了其他子系统，若对这些子系统不了解是很难完整的注解鸿蒙内核，所以也对这些关联仓库进行了部分注解，这些仓库包括:

  - **编译构建子系统 | build_lite**
  - **协议栈 | lwip**
  - **文件系统 | NuttX**
  - **标准库 | musl**

- 同样由位图来管理系统文件句柄，具体相关操作如下

```
//用 bitmap 数组来记录文件描述符的分配情况，一位代表一个SYS FD
static unsigned int bitmap[CONFIG_NFILE_DESCRIPTORS / 32 + 1] = {0};
//设置指定位值为 1
static void set_bit(int i， void *addr)
{
  unsigned int tem = (unsigned int)i >> 5; /* Get the bitmap subscript */
  unsigned int *addri = (unsigned int *)addr + tem;
  unsigned int old = *addri;
  old = old | (1UL << ((unsigned int)i & 0x1f)); /* set the new map bit */
  *addri = old;
}
//获取指定位，看是否已经被分配
bool get_bit(int i)
{
  unsigned int *p = NULL;
  unsigned int mask;

  p = ((unsigned int *)bitmap) + (i >> 5); /* Gets the location in the bitmap */
  mask = 1 << (i & 0x1f); /* Gets the mask for the current bit int bitmap */
  if (!(~(*p) & mask)){
    return true;
  }
  return false;
```

```
}
```

- tg_filelist 是全局系统文件列表，统一管理系统 fd ，其中的关键结构体是 file ，这才是内核对文件对象描述的实体，是本篇最重要的内容。

```
#if CONFIG_NFILE_DESCRIPTORS > 0
struct filelist tg_filelist; //全局统一管理系统文件句柄
#endif
struct filelist
{
 sem_t   fl_sem;              /* Manage access to the file list */
 struct file fl_files[CONFIG_NFILE_DESCRIPTORS];
};
struct file
{
 unsigned int        f_magicnum;  /* file magic number */
 int                 f_oflags;    /* Open mode flags */
 struct Vnode        *f_vnode;    /* Driver interface */
 loff_t              f_pos;       /* File position */
 unsigned long       f_refcount;  /* reference count */
 char                *f_path;     /* File fullpath */
 void                *f_priv;     /* Per file driver private data */
 const char          *f_relpath;  /* realpath */
 struct page_mapping  *f_mapping;  /* mapping file to memory */
 void                *f_dir;      /* DIR struct for iterate the directory if open a directory */
 const struct file_operations_vfs *ops;
 int fd;
};
```

  - f_magicnum 魔法数字，每种文件格式不同魔法数字不同， gif 是 47 49 46 38 ， png 是 89 50 4e 47
  - f_oflags 操作文件的权限模式，读/写/执行
  - f_vnode 对应的 vnode
  - f_pos 记录操作文件的当前位置
  - f_refcount 文件被引用的次数，即文件被所有进程打开的次数。
  - f_priv 文件的私有数据
  - f_relpath 记录文件的真实路径
  - f_mapping 记录文件和内存的映射关系，这个在文件映射篇中有详细介绍。
  - ops 对文件内容的操作函数
  - fd 文件句柄编号，系统文件句柄是唯一的，一直到申请完为止，当 f_refcount 为0时，内核将回收 fd 。

open | creat | 申请文件句柄

通过文件路径名 pathname 获取文件句柄，鸿蒙实现过程如下

```
SysOpen //系统调用
    AllocProcessFd  //分配进程文件句柄
    do_open //向底层打开文件
        fp_open //vnode 层操作
            files_allocate
            filep->ops->open(filep) //调用各文件系统的函数指针
    AssociateSystemFd //绑定系统文件句柄
```

建一个 file 对象， i 即为分配到的系统文件句柄。

```
//创建系统文件对象及分配句柄
int files_allocate(struct Vnode *vnode_ptr， int oflags， off_t pos， void *priv， int minfd)
  //...
  while (i < CONFIG_NFILE_DESCRIPTORS)//系统描述符
    {
      p = ((unsigned int *)bitmap) + (i >> 5); /* Gets the location in the bitmap */
      mask = 1 << (i & 0x1f); /* Gets the mask for the current bit int bitmap */
      if ((~(*p) & mask))//该位可用于分配
        {
          set_bit(i， bitmap);//占用该位
          list->fl_files[i].f_oflags   = oflags;
          list->fl_files[i].f_pos      = pos;//偏移位
```

```
        list->fl_files[i].f_vnode    = vnode_ptr;//vnode
        list->fl_files[i].f_priv     = priv;//私有数据
        list->fl_files[i].f_refcount = 1; //引用数默认为1
        list->fl_files[i].f_mapping  = NULL;//暂无映射
        list->fl_files[i].f_dir      = NULL;//暂无目录
        list->fl_files[i].f_magicnum = files_magic_generate();//魔法数字
        process_files = OsCurrProcessGet()->files;//获取当前进程文件管理器
        return (int)i;
      }
    i++;
  }
  // ...
}
```

read | write

```
SysRead   //系统调用|读文件:从文件中读取nbytes长度的内容到buf中(用户空间)
 fd = GetAssociatedSystemFd(fd); //通过进程fd获取系统fd
 read(fd， buf， nbytes);  //调用系统fd层的读函数
  fs_getfilep(fd， &filep);  //通过系统fd获取file对象
  file_read(filep， buf， nbytes) //调用file层的读文件
    ret = (int)filep->ops->read(filep， (char *)buf， (size_t)nbytes);//调用具体文件系统的读操作
```

```
SysWrite   //系统调用|写文件:将buf中(用户空间)nbytes长度的内容写到文件中
 fd = GetAssociatedSystemFd(fd); //通过进程fd获取系统fd
 write(sysfd， buf， nbytes);  //调用系统fd层的写函数
  fs_getfilep(fd， &filep);  //通过系统fd获取file对象
  file_seek64
  file_write(filep， buf， nbytes);//调用file层的写文件
    ret = filep->ops->write(filep， (const char *)buf， nbytes);//调用具体文件系统的写操作
```

此处仅给出 **file_write** 的实现

```
ssize_t file_write(struct file *filep， const void *buf， size_t nbytes)
{
  int ret;
  int err;

  if (buf == NULL)
    {
      err = EFAULT;
      goto errout;
    }

  /* Was this file opened for write access？ */

  if ((((unsigned int)(filep->f_oflags)) & O_ACCMODE) == O_RDONLY)
    {
      err = EACCES;
      goto errout;
    }

  /* Is a driver registered？ Does it support the write method？ */

  if (!filep->ops || !filep->ops->write)
    {
      err = EBADF;
      goto errout;
    }

  /* Yes， then let the driver perform the write */

  ret = filep->ops->write(filep， (const char *)buf， nbytes);
  if (ret < 0)
    {
      err = -ret;
```

```
      goto errout;
    }

  return ret;

errout:
  set_errno(err);
  return VFS_ERROR;
}
```

close

```
//关闭文件句柄
int SysClose(int fd)
{
    int ret;

    /* Process fd convert to system global fd */
    int sysfd = DisassociateProcessFd(fd);//先解除关联

    ret = close(sysfd);//关闭文件，个人认为应该先 close - > DisassociateProcessFd
    if (ret < 0) {//关闭失败时
        AssociateSystemFd(fd， sysfd);//继续关联
        return -get_errno();
    }
    FreeProcessFd(fd);//释放进程fd
    return ret;
}
```

- 解除进程 `fd` 和系统 `fd` 的绑定关系
- `close` 时会有个判断，这个文件的引用数是否为 `0` ，只有为 `0` 才会真正的执行 `_files_close`

```
  int files_close_internal(int fd， LosProcessCB *processCB)
  {
    //...
    list->fl_files[fd].f_refcount--;
    if (list->fl_files[fd].f_refcount == 0)
      {
  #ifdef LOSCFG_KERNEL_VM
        dec_mapping_nolock(filep->f_mapping);
  #endif
        ret = _files_close(&list->fl_files[fd]);
        if (ret == OK)
          {
            clear_bit(fd， bitmap);
          }
      }
    // ...
  }
  static int _files_close(struct file *filep)
  {
    struct Vnode *vnode = filep->f_vnode;
    int ret = OK;

    /* Check if the struct file is open (i.e.， assigned an vnode) */
    if (filep->f_oflags & O_DIRECTORY)
      {
        ret = closedir(filep->f_dir);
        if (ret != OK)
          {
            return ret;
          }
      }
    else
      {
        /* Close the file， driver， or mountpoint. */
        if (filep->ops && filep->ops->close)
          {
```

```
        /* Perform the close operation */

        ret = filep->ops->close(filep);
        if (ret != OK)
          {
            return ret;
          }
      }
    VnodeHold();
    vnode->useCount--;
    /* Block char device is removed when close */
    if (vnode->type == VNODE_TYPE_BCHR)
      {
        ret = VnodeFree(vnode);
        if (ret < 0)
          {
            PRINTK("Removing bchar device %s failed\n"， filep->f_path);
          }
      }
    VnodeDrop();
  }

  /* Release the path of file */

  free(filep->f_path);

  /* Release the file descriptor */

  filep->f_magicnum = 0;
  filep->f_oflags   = 0;
  filep->f_pos      = 0;
  filep->f_path     = NULL;
  filep->f_priv     = NULL;
  filep->f_vnode    = NULL;
  filep->f_refcount = 0;
  filep->f_mapping  = NULL;
  filep->f_dir      = NULL;

  return ret;
}
```

- 最后 `FreeProcessFd` 负责释放该文件在进程层面占用的资源

百文说内核 | 抓住主脉络

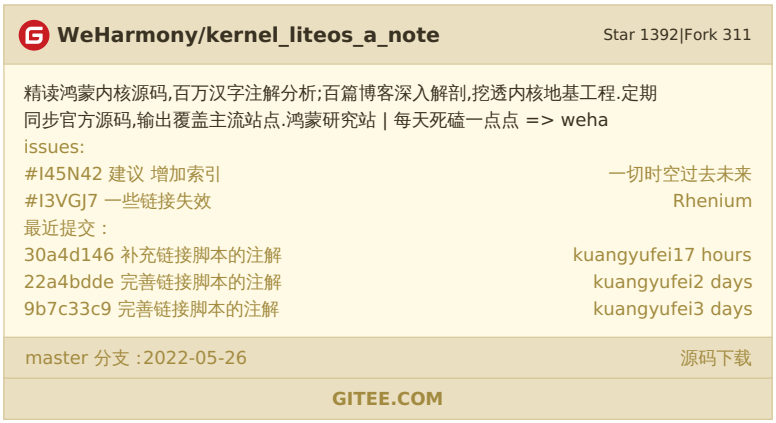
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

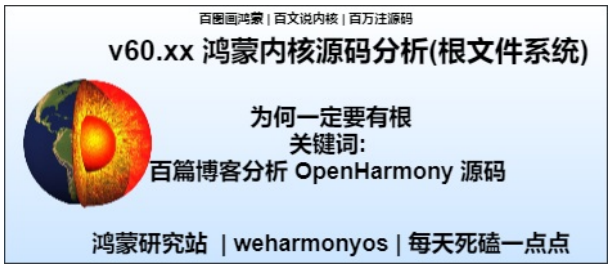


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

60_根文件系统

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

FHS | 文件系统层次结构标准

- 在 [挂载目录篇] 中提到内核为了兼容文件系统的差异性，引出了目录树的概念，目录树是由各个文件系统像搭积木一样拼接起来的，任何文件系统只需要挂载到一个目录上就能对接进来，内核抽象出统一的挂载接口，各文件系统自己实现这些接口就行。 既然目录如此重要， 就需要规范管理， 类Unix都遵循 FHS 规范，鸿蒙同样遵循。

- 文件系统层次结构标准（英语：Filesystem Hierarchy Standard，FHS）定义了Linux操作系统中的主要目录及目录内容。FHS由Linux基金会维护。 当前版本为3.0版，于2015年发布。基本目录如下:

```
/      根目录
/home   用户主文件夹
/etc    系统主要的配置文件几乎都放置在这个目录内
/root   系统管理员（root）的主文件夹
/bin    可以被 root 与一般账号所使用
/sbin   这些命令只有 root 才能够利用来“设置”系统
/lib    放置的则是在开机时会用到的函数库
/opt    用于安装第三方应用程序的
/dev    任何设备与接口设备都是以文件的形式存在于这个目录当中
/proc   一个虚拟的文件系统， 它放置的数据都是在内存当中
/sys    也是一个虚拟的文件系统，主要也是记录与内核相关的信息
/media  放置的就是可删除的设备
/mnt    暂时挂载某些额外的设备
/srv    一些网络服务启动之后，这些服务所需要取用的数据目录
/tmp    正在执行的程序暂时放置文件的地方， 系统会不定期删除

/usr    “UNIX 操作系统软件资源” 所放置的目录
    /usr/bin/：   绝大部分的用户可使用命令都放在这里
     /usr/include/：C/C++等程序语言的头文件 header 与包含文件include放置处
     /usr/lib/：   包含各应用软件的函数库、目标文件以及一些不被一般用户惯用的执行文件或脚本
     /usr/local/： 系统管理员在本机自行安装下载的软件建议安装到此目录
     /usr/sbin/：  非系统正常运行所需的系统命令
     /usr/share/： 放置共享文件的地方
     /usr/src/：   一般源码建议放置到这里

/var    该目录主要针对常态性可变动文件
   /var/cache/： 应用程序本身运行过程中会产生的一些暂存文件
```

```
/var/lib/ :   程序本身执行的过程中，需要的数据文件放置的目录
/var/lock/ :  目录下的文件资源一次只能被一个应用程序所使用
/var/log/ :   放置登录文件的目录
/var/mail/ :  放置个人电子邮件信箱的目录
/var/run/ :   某些程序或服务启动后的PID目录
/var/spool/ : 放置排队等待其他应用程程序使用的数据
```

什么是根文件系统

看网上有很多的文章，但基本全是一大抄，说是内核启动时所mount的第一个文件系统，这话固然是没错， 但想重新定义下这个概念， 所谓 **根文件系统** 就是先挂到根目录 `/` 上的文件系统。 核心是根目录 `/` 。 `/` 目录并不必先属于哪个文件系统，否则就是先有蛋还是先有鸡的问题，所以别被蒙圈了，它跟其他文件系统没有任何区别，只是它先来，把坑 `/` 给占了，后续来的只能挂到它下面的目录上，最终形成整颗目录树。

理解了上面，就容易明白以下几个问题:

- 一个系统可以存在多个不同的文件系统，谁做根文件系统只决于内核在启动阶段想让谁做。
- 文件系统可存在于诸多介质上，例如:硬盘(mmc)，闪存(flash)，内存(RAM)。每种介质上有其最合适配套的文件系统，mmc一般是(fat，ext)，flash包括(jffs2)，内存(proc，sys，tmpfs，ramfs)
- 文件系统可简单，可复杂，只要能实现内核定义的三类接口就可以称之为文件系统，哪三类接口:
  - 挂载接口: `MountOps ops`
  - 操作 `inode` 节点接口: `VnodeOps *vop`
  - 操作 `file` 接口: `file_operations_vfs *fop` ，这个接口底层实际操作的是 `inode` 所指向的数据块。
- 不管怎样，内核启动后必须得有一个文件系统用于挂载到 `/` 下。 鸿蒙根文件系统目录结构如下:

```
.
├── app
├── bin
│   ├── init
│   ├── shell
│   └── tftp
├── data
│   └── system
│       └── param
├── etc
├── lib
│   ├── libc++.so
│   └── libc.so
├── system
│   ├── external
│   └── internal
└── usr
    ├── bin
    └── lib
```

这些数据是怎么来的呢 ？ 比如: `libc.so` 这种C库函数，启动后就马上需要使用的， 这需要先外部制作好，烧录到flash的指定位置。 同时注意鸿蒙制作的根文件系统并没有 `/dev` 目录，这个在 **设备文件篇** 中详细说明。

根文件系统制作过程

以 `liteos_a` 内核为例，其提供了制作根文件系统的方法:

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a$ make help
-------------------------------------------------------------
1.====make help:    get help infomation of make
2.====make:         make a debug version based the .config
3.====make debug:   make a debug version based the .config
4.====make release: make a release version for all platform
5.====make release PLATFORM=xxx:  make a release version only for platform xxx
6.====make rootfsdir: make a original rootfs dir
7.====make rootfs FSTYPE=***: make a original rootfs img
8.====make test: make the testsuits_app and put it into the rootfs dir
9.====make test_apps FSTYPE=***: make a rootfs img with the testsuits_app in it
xxx should be one of (hi3516cv300 hi3516ev200 hi3556av100/cortex-a53_aarch32 hi3559av100/cortex-a53_aarch64)
```

```
*** should be one of (jffs2)
```

其中第七项 `make rootfs` ， `FSTYPE` 支持 `jffs2` ，`vfat` 文件格式系统

本篇跟踪敲下 `make rootfs FSTYPE=jffs2` 后发生了什么

查看 **kernel/liteos_a/Makefile**

```
#执行 make rootfs FSTYPE=jffs2 一切从这里开始
$(ROOTFS): $(ROOTFSDIR) #依赖于 ROOTFSDIR
 $(HIDE)$(LITEOSTOPDIR)/tools/scripts/make_rootfs/rootfsimg.sh $(ROOTFS_DIR) $(FSTYPE) #制作镜像文件
 $(HIDE)cd $(ROOTFS_DIR)/.. && zip -r $(ROOTFS_ZIP) $(ROOTFS) #打rootfs.zip包
```

解读

- 编译整个内核的目标文件

```
 $(LITEOS_TARGET): $(__LIBS) sysroot
    $(HIDE)touch $(LOSCFG_ENTRY_SRC)
    #逐个编译子目录中的 makefile
    $(HIDE)for dir in $(LITEOS_SUBDIRS); \
    do $(MAKE) -C $$dir all || exit 1; \
    done
    # 生成 liteos.map
    $(LD) $(LITEOS_LDFLAGS) $(LITEOS_TABLES_LDFLAGS) $(LITEOS_DYNLDFLAGS) -Map=$(OUT)/$@.map -o $(OUT)/$@ --start-group $(LITE
 # $(SIZE) -t --common $(OUT)/lib/*.a >$(OUT)/$@.objsize
    $(OBJCOPY) -O binary $(OUT)/$@ $(LITEOS_TARGET_DIR)/$@.bin #生成 liteos.bin 文件
    $(OBJDUMP) -t $(OUT)/$@ |sort >$(OUT)/$@.sym.sorted #生成 liteos.sym.sorted 文件
    $(OBJDUMP) -d $(OUT)/$@ >$(OUT)/$@.asm # 生成 liteos.asm文件
```

- 使用 `$(APPS)` 编译 **kernel/liteos_a/apps** 目录下的各个 `APP` 如(`init` ， `shell` ， `tftp`)，这些APP也称为内置到内核的APP，

```
# 编译多个应用程序
 $(APPS): $(LITEOS_TARGET) sysroot #依赖于 LITEOS_TARGET ， sysroot
    $(HIDE)$(MAKE) -C apps all #执行apps目录下Makefile 的all目标， -C代表进入apps目录，
```

- 使用 **tools/scripts/make_rootfs/rootfsdir.sh** 创建根系统下的各个目录(`/bin` ， `/app` ， `/lib`)。

```
#创建根文件系统的各个目录
 mkdir -p ${ROOTFS_DIR}/bin ${ROOTFS_DIR}/lib ${ROOTFS_DIR}/usr/bin ${ROOTFS_DIR}/usr/lib ${ROOTFS_DIR}/etc \
 ${ROOTFS_DIR}/app ${ROOTFS_DIR}/data ${ROOTFS_DIR}/proc ${ROOTFS_DIR}/dev ${ROOTFS_DIR}/data/system ${ROOTFS_DIR}/data/s
 ${ROOTFS_DIR}/system ${ROOTFS_DIR}/system/internal ${ROOTFS_DIR}/system/external ${OUT_DIR}/bin ${OUT_DIR}/libs
 if [ -d "${BIN_DIR}" ] && [ "$(ls -A "${BIN_DIR}")" != "" ]; then
    cp -f ${BIN_DIR}/* ${ROOTFS_DIR}/bin
    if [ -e ${BIN_DIR}/shell ] && [ "${BIN_DIR}/shell" != "${OUT_DIR}/bin/shell" ]; then
       cp -f ${BIN_DIR}/shell ${OUT_DIR}/bin/shell #拷贝 shell 到根文件系统的 /bin下
    fi
    if [ -e ${BIN_DIR}/tftp ] && [ "${BIN_DIR}/tftp" != "${OUT_DIR}/bin/tftp" ]; then
       cp -f ${BIN_DIR}/tftp ${OUT_DIR}/bin/tftp   #拷贝 tftp 到根文件系统的 /bin下
    fi
 fi
 cp -f ${LIB_DIR}/* ${ROOTFS_DIR}/lib    #将c/c++ .so 库拷贝到根文件系统的 /lib
 cp -f ${LIB_DIR}/* ${OUT_DIR}/libs
```

- 使用 `prepare` 创建 `musl` 目录，并将 c/c++ 库拷贝到该目录下

```
prepare: #准备工作，创建 musl 目录，用于拷贝 c/c++ .so库
  $(HIDE)mkdir -p $(OUT)/musl
  ifeq ($(LOSCFG_COMPILER_CLANG_LLVM)， y) #使用clang-9 ，鸿蒙默认用这个编译
    $(HIDE)cp -f $$($(CC) --target=$(LLVM_TARGET) --sysroot=$(SYSROOT_PATH) $(LITEOS_CFLAGS) -print-file-name=libc.so) $(OUT)/musl #
    $(HIDE)cp -f $$($(GPP) --target=$(LLVM_TARGET) --sysroot=$(SYSROOT_PATH) $(LITEOS_CXXFLAGS) -print-file-name=libc++.so) $(OUT)
  else
    $(HIDE)cp -f $(LITEOS_COMPILER_PATH)/target/usr/lib/libc.so $(OUT)/musl
    $(HIDE)cp -f $(LITEOS_COMPILER_PATH)/arm-linux-musleabi/lib/libstdc++.so.6 $(OUT)/musl
    $(HIDE)cp -f $(LITEOS_COMPILER_PATH)/arm-linux-musleabi/lib/libgcc_s.so.1 $(OUT)/musl
```

```
    $(STRIP) $(OUT)/musl/*
endif
```

- tools/scripts/make_rootfs/rootfsimg.sh 生成镜像文件 rootfs_jffs2.img ， 调用 mkfs.jffs2 来制作 jffs2 文件格式的镜像。

```
ROOTFS_IMG=${ROOTFS_DIR}"_"${FSTYPE}".img"
JFFS2_TOOL=mkfs.jffs2 #linux 下 制作 jffs2镜像文件的工具
WIN_JFFS2_TOOL=mkfs.jffs2.exe #windows 下 制作 jffs2镜像文件的工具
chmod -R 755 ${ROOTFS_DIR}
if [ -f "${ROOTFS_DIR}/bin/init" ]; then
    chmod 700 ${ROOTFS_DIR}/bin/init 2> /dev/null
fi
if [ -f "${ROOTFS_DIR}/bin/shell" ]; then
    chmod 700 ${ROOTFS_DIR}/bin/shell 2> /dev/null
fi

if [ "${FSTYPE}" = "jffs2" ]; then
    if [ "${system}" != "Linux" ] ; then
        tool_check ${WIN_JFFS2_TOOL}
        ${WIN_JFFS2_TOOL} -q -o ${ROOTFS_IMG} -d ${ROOTFS_DIR} --pagesize=4096
    else
        tool_check ${JFFS2_TOOL}
        ${JFFS2_TOOL} -q -o ${ROOTFS_IMG} -d ${ROOTFS_DIR} --pagesize=4096
    fi
elif [ "${FSTYPE}" = "yaffs2" ]; then
    # to do
fi
```

- 最后用 zip 命令将 rootfs 打包成 rootfs.zip ，至此完成了鸿蒙根系统的制作过程。 **将增加了一个 out 目录，内容如下:**

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/kernel/liteos_a/out/hi3518ev300$ ls
bin  lib  liteos  liteos.asm  liteos.bin  liteos.map  liteos.sym.sorted  musl  obj  rootfs  rootfs_jffs2.img  rootfs.zip
```

  - rootfs 便为制作的鸿蒙根文件系统
  - rootfs_jffs2.img 为镜像文件，可以烧到 flash 中。

启动过程

这里列出启动根文件系统的相关代码

```
STATIC UINT32 OsSystemInitTaskCreate(VOID)
{
    UINT32 taskID;
    TSK_INIT_PARAM_S sysTask;

    (VOID)memset_s(&sysTask， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));
    sysTask.pfnTaskEntry = (TSK_ENTRY_FUNC)SystemInit;
    sysTask.uwStackSize = LOSCFG_BASE_CORE_TSK_DEFAULT_STACK_SIZE;
    sysTask.pcName = "SystemInit";
    sysTask.usTaskPrio = LOSCFG_BASE_CORE_TSK_DEFAULT_PRIO;
    sysTask.uwResved = LOS_TASK_STATUS_DETACHED;
#if (LOSCFG_KERNEL_SMP == YES)
    sysTask.usCpuAffiMask = CPUID_TO_AFFI_MASK(ArchCurrCpuid());
#endif
    return LOS_TaskCreate(&taskID， &sysTask);
}
```

解读

- 首先内核开了一个叫 SystemInit 任务来处理系统初始化代码，任务入口函数为 SystemInit
- SystemInit 层层调用到 MountPartitions ，挂载分区。

```
SystemInit(void)
  ...
```

```
OsMountRootfs()
  AddPartitions //注册分区驱动程序
  MountPartitions()
     #define ROOT_DEV_NAME        "/dev/spinorblk0"
     #define ROOT_DIR_NAME        "/"
     ret = mount(ROOT_DEV_NAME， ROOT_DIR_NAME， fsType， mountFlags， NULL);//
```

在 **设备文件篇** 中将详细说明 `/dev/spinorblk0` 的来源，简单的说就根文件系统烧录在 `nor flash` 介质设备的第一个分区上，分区名称 /dev/spinorblk0 只是表示一个“虚”的设备文件名而已，其背后是个实实在在的文件系统。现将它挂到 `/` 上，结果是 `nor flash` 的第一个分区成了根文件系统。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

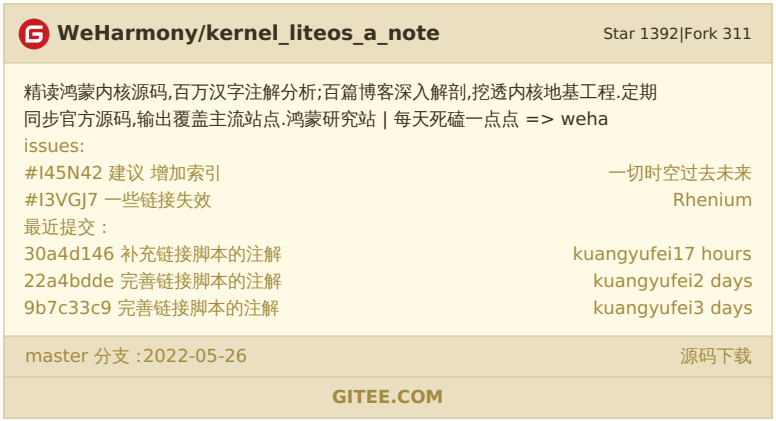
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

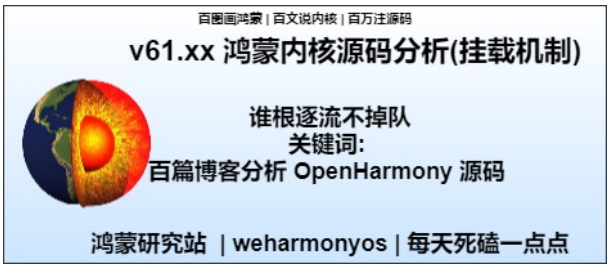


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

61_挂载机制篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

关于文件系统的介绍已经写了三篇，但才刚刚开始，其中的 [文件系统篇] 一定要阅读，用生活中的场景去解释计算机各模块设计的原理和运行机制是整个系列篇最大的特点，计算机文件系统相关概念是非常的多的，若不还原其本质，不跳出这些概念去看问题是很难理解它为什么要弄这么些东东出来让你头大。 反之，如果搞明白了这些概念背后的真相你想忘记它们都很难，问题是经不起追问的， 多追问几个为什么就会离本源越来越近。

前几篇中追问了以下几个问题:

- 对内核来说 inode 真的是唯一的吗？ 答案是否定的，使用电脑的经验告诉我们，当把电脑硬盘拆下来挂到其他电脑上时，里面的数据一样能访问，并没有让你一切重来，而 inode 是存放硬盘上的，你没有办法让已编好序号的 inode 按你的逻辑重排，这不合理更不科学。 所以结论是 inode 的全局唯一性不是不想做，而是压根臣妾做不到啊。 inode 唯一性仅限于某个文件系统的内部。
- 经验还告诉我们硬盘可以有多个分区，每个分区可以被格式化成不同的文件系统。(例如:C盘: NTF ，D盘 FAT32 ，E盘 ext)，数据可以相互拷贝，毫无障碍。不同的文件系统是如何实现文件迁移到呢 ？ 具体实现细节是怎样的 ？

如果想明白了这些问题， 就能逆向倒推为什么要有目录，为什么需要挂载使用， 为什么需要根文件系统。一切将是水到渠成 。 先说目录，从内核视角看目录可不能像普通老百姓从用户视角去看，目录是为了屏蔽文件系统之间的差异而设计出来的概念，也就说必须在 inode 的局部唯一性之上存在一个全局唯一性才能解决统一性问题。目录从更大尺度上去兼容并蓄各文件系统。

那它是如何解决的呢？

- 首先各个文件系统记录了自己内部目录层级关系的，这个在 文件系统篇 | 目录项 中已经说过了。 这种关系是绝对的但也是相对的，绝对是对内，相对是对外。 例如: A文件系统内部如下:

```
├─古龙系列 inode id : 789
│  ├─小李飞刀 inode id : 56
│  ├─楚留香 inode id : 342
│  └─陆小凤 inode id : 432
└─金庸系列 inode id : 5567
   ├─倚天屠龙记 inode id : 89
   ├─射雕英雄传 inode id : 1212
   └─笑傲江湖 inode id : 567843
```

B文件系统内部如下:

```
├─席绢系列 ：inode id : 87
│  ├─上错花轿嫁对郎：inode id : 89
│  ├─吻上你的心：inode id : 789
```

```
  │ └─红袖招 : inode id : 56
  └─琼瑶系列 : inode id : 321
    ├─在水一方 : inode id : 234
    ├─梅花三弄 : inode id : 5678
    ├─烟雨濛濛 : inode id : 987
    └─还珠格格 : inode id : 23
```

其中 `789` ， `89` 两个文件系统中都用到了，但它们在内部是唯一的。在A文件系统中通过 `789` 就能找到 `56` ， `342` ， `432` ，并且能得到相对路径: 古龙系列/小李飞刀，古龙系列/楚留香 。也就是说拿着 `inode` 只要进入了本文件系统地盘，那都不叫事，事都能给你办的妥妥的。 那如何才能进入而且不会搞错呢？

挂载目录

答案就是: 挂载目录，也叫挂载点，集体统一指挥的前提是需要先回归集体。如果已经有一颗目录树，将你们的目录树挂上来形成一颗更大的树不就统一了吗？ 例如已有:

```
├─小说系列 inode id : 2
│ ├─武侠小说 inode id : 13
│ ├─言情小说 inode id : 14
```

其实它也是个文件系统，叫根文件系统， 它的 `inode` 也是独立的， 并且能得到相对路径 `小说系列/武侠小说` ， `小说系列/武侠小说` 通过两个 `mount` 动作， 将它变成如下所示

```
├─小说系列 (根文件系统)
  ├─武侠小说 (根文件系统)
  │ ├─古龙系列 (A文件系统)
  │ │ ├─小李飞刀
  │ │ ├─楚留香
  │ │ └─陆小凤
  │ └─金庸系列 (A文件系统)
  │   ├─倚天屠龙记
  │   ├─射雕英雄传
  │   └─笑傲江湖
  └─言情小说 (根文件系统)
    ├─席绢系列 (B文件系统)
    │ ├─上错花轿嫁对郎
    │ ├─吻上你的心
    │ └─红袖招
    └─琼瑶系列 (B文件系统)
      ├─在水一方
      ├─梅花三弄
      ├─烟雨濛濛
      └─还珠格格
```

哦，原来整颗目录树是由这三个文件系统像搭积木一样拼接起来。而两个文件系统的衔接点，必然会产生一个新的概念出来， 这个概念就是 **挂载点**，也叫 **挂载目录**

Mount

可以猜测到的是**挂载点**的描述结构体中必有两个文件系统接驳点 `inode` 的信息，挂钩和脱钩的操作也只属于它专有。具体如下:

```
//挂载操作
struct MountOps {
    int (*Mount)(struct Mount *mount， struct Vnode *vnode， const void *data);//挂载
    int (*Unmount)(struct Mount *mount， struct Vnode **blkdriver);//卸载
    int (*Statfs)(struct Mount *mount， struct statfs *sbp);//统计文件系统的信息，如该文件系统类型、总大小、可用大小等信息
};
struct Mount {
    LIST_ENTRY mountList;             /* mount list */    //通过本节点将Mount挂到全局Mount链表上
    const struct MountOps *ops;       /* operations of mount */ //挂载操作函数
    struct Vnode *vnodeBeCovered;     /* vnode we mounted on */ //要被挂载的节点 即 /bin1/vs/sd 对应的 vnode节点
    struct Vnode *vnodeCovered;       /* syncer vnode */   //要挂载的节点 即/dev/mmcblk0p0 对应的 vnode节点
    LIST_HEAD vnodeList;              /* list of vnodes */  //链表表头
    int vnodeSize;                    /* size of vnode list */ //节点数量
    LIST_HEAD activeVnodeList;        /* list of active vnodes */ //激活的节点链表
    int activeVnodeSize;              /* szie of active vnodes list *///激活的节点数量
```

```
    void *data;                     /* private data */ //私有数据，可使用这个成员作为一个指向它们自己内部数据的指针
    uint32_t hashseed;              /* Random seed for vfs hash */ //vfs 哈希随机种子
    unsigned long mountFlags;       /* Flags for mount */ //挂载标签
    char pathName[PATH_MAX];        /* path name of mount point */ //挂载点路径名称  /bin1/vs/sd
    char devName[PATH_MAX];         /* path name of dev point */  //设备名称 /dev/mmcblk0p0
};
```

解读

- `mountList` ：挂载点由双向链表全局统一管理
- `vnodeBeCovered`：，记录挂到根文件系统的哪个节点上。
- `vnodeCovered`：设备也是一种文件，也被统一管理，统一在`/dev`目录下，内核会给设备的每个分区分配一个`vnode`节点，一个分区对应一个文件系统，设备文件后续有专门的介绍，此处不展开。
- `vnodeList`：指的是A/B文件系统的节点链表，由挂载点结构体记录。
- `activeVnodeList`：`A` 文件系统节点的使用情况，统一由双向链表管理。
- `activeVnodeSize`：`A` 文件系统已被使用的节点数
- `data` 这是文件系统的私有数据，跟 索引节点篇 | Vnode -> data 一样理解。
- `pathName` :这个很重要，记录了`小说系列/武侠小说`，因为文件的绝对路径是拼接起来的，以`小说系列/武侠小说/古龙系列/小李飞刀`这个完整的路径来说，它是由`小说系列/武侠小说`(根文件系统提供) + `古龙系列/小李飞刀`(A文件系统提供) 这两部分拼成的。
- `devName` :一般名称类似于`mmcblk0p0` = `mmc` + `block0` + `Partition0`
  - `mmc`：`MultiMediaCard` 可理解为硬盘
  - `block0`：0号块设备
  - `Partition` :0号分区，一个分区上安装一个文件系统。
- `MountOps ops`：每个文件系统挂载方式是不用的，都需要实现这几个接口(挂载，卸载，统计)。

```
// 文件系统 proc 对 MountOps 接口实现
const struct MountOps procfs_operations = {
    .Mount = VfsProcfsMount，//装载
    .Unmount = NULL，
    .Statfs = VfsProcfsStatfs，//统计信息
};
//文件系统 fat 对MountOps 接口实现
struct MountOps fatfs_mops = {
    .Mount = fatfs_mount，
    .Unmount = fatfs_umount，
    .Statfs = fatfs_statfs，
};
//文件系统 jffs 对MountOps 接口实现
const struct MountOps jffs_operations = {
  .Mount = VfsJffs2Bind，
  .Unmount = VfsJffs2Unbind，
  .Statfs = VfsJffs2Statfs，
};
```

问题

上面提到 挂载就需要一个已经存在的文件系统提供目录，也就是根文件系统，但根文件系统又是怎么来的呢？

百文说内核 | 抓住主脉络

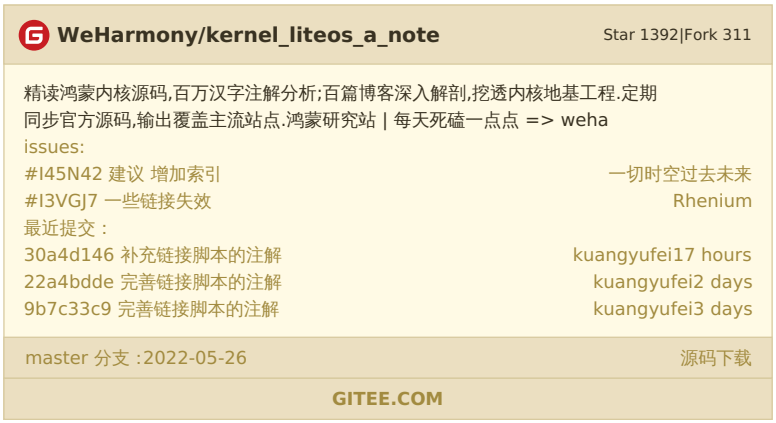
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

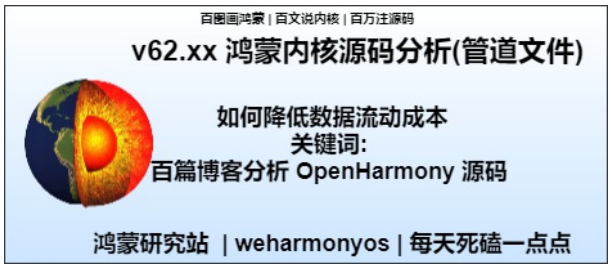


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

62_管道文件篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

什么是管道

- 管道 | pipes 最早最清晰的陈述来源于 McIlroy 由 1964 年写的一份内部文件。这份文件提出像花园的水管那样把程序连接在一起。文档全文内容如下:

> Summary--what's most important.
> To put my strongest concerns into a nutshell:
> 1. We should have some ways of coupling programs like
> garden hose--screw in another segment when it becomes when
> it becomes necessary to massage data in another way.
> This is the way of IO also.
> 2. Our loader should be able to do link-loading and
> controlled establishment.
> 3. Our library filing scheme should allow for rather
> general indexing， responsibility， generations， data path
> switching.
> 4. It should be possible to get private system components
> (all routines are system components) for buggering around with.
>
> M. D. McIlroy
> October 11， 1964

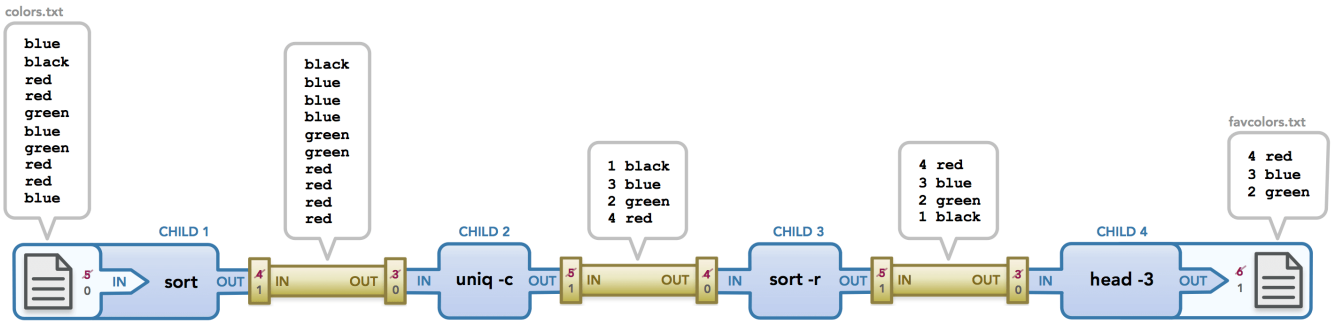
- Unix 的缔造者 肯.汤普森 只花了一个小时就在操作系统中实现了管道的系统调用。他自己说这是简直小菜一碟，因为I/O的重定向机制是管道的实现基础，但效果确是很震撼。管道的本质是 I/O 的重定向，是对数据的不断编辑，不断流动，只有这样的数据才有价值。

- 在文件概念篇中提到过， Unix "一切皆文件"的说法是源于输入输出的共性，只要涵盖这两个特性都可以也应当被抽象成文件统一管理和流动。拿跟城市的发展来举例，越是人口流动和资金流动频繁的城市一定是越发达的城市。这个道理请仔细品，城市的规划应该让流动的成本变低，时间变短，而不是到处查身份证，查户口本。对内核设计者来说也是一样，能让数据流动的成本变得极为简单，方便的系统也一定是好的架构， Unix 能做到多年强盛不衰其中一个重要原因是它仅用一个 | 符号实现了文件之间的流动性问题。这是一种伟大的创举，必须用专门的章篇对其大书特书。

管道符号 |

管道符号是两个命令之间的一道竖杠 | ，简单而优雅，例如， ls 用于显示某个目录中文件， wc 用于统计行数。 ls | wc 则代表统计某个目录下的文件数量 再看个复杂的:

```
$ < colors.txt sort | uniq -c | sort -r | head -3 > favcolors.txt
```



- colors.txt 为原始的文件内容，输出给 sort 处理
- sort 对 colors.txt 内容进行顺序编辑后输出给 uniq 处理
- uniq 对 内容进行去重编辑后输出给 sort -r 处理
- sort -r 对内容进行倒序编辑后输出给 head -3 处理
- head -3 对 内容进行取前三编辑后输出到 favcolors.txt 文件保存。
- 最后 cat favcolors.txt 查看结果

```
$ cat favcolors.txt
   4 red
   3 blue
   2 green
```

经典管道案例

以下是 linux 官方对管道的经典案例。 查看 pipe

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(int argc， char *argv[])
{
   int pipefd[2];
   pid_t cpid;
   char buf;

   if (argc != 2) {
      fprintf(stderr， "Usage: %s <string>\n"， argv[0]);
      exit(EXIT_FAILURE);
   }

   if (pipe(pipefd) == -1) {
      perror("pipe");
      exit(EXIT_FAILURE);
   }

   cpid = fork();
   if (cpid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
   }

   if (cpid == 0) {    /* Child reads from pipe */
      close(pipefd[1]);          /* Close unused write end */
```

```
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

解读

- `pipe(pipefd)` 为系统调用，申请了两个文件句柄，并对这两个文件进行了管道绑定。 在鸿蒙管道的系统调用为 `SysPipe` ，具体实现往下看。
- `main` 进程 `fork()` 了一个子进程，具体的 `fork` 过程请前往 [v45.xx (Fork篇) | 一次调用，两次返回] 翻看。子进程将复制父进程的文件资源。所以子进程 `cpid` 也拥有了 `pipefd` 两个句柄，背后的含义就是可以去操作 `pipefd` 对应的文件
- `if (cpid == 0)` 代表的是子进程的返回，
  - `close(pipefd[1])` 关闭了 `pipefd[1]` 文件句柄，因为程序设计成子进程负责读文件操作，它并不需要操作 `pipefd[1]`
  - `while (read(pipefd[0], &buf, 1)` 子进程不断的读取文件 `pipefd[0]` 的内容。
  - 按理说能不断的读取 `pipefd[0]` 数据说明有进程在不断的往 `pipefd[0]` 中写入数据。但管道的思想是往 `pipefd[1]` 中写入数据，数据却能跑到 `pipefd[0]` 中。
- `(cpid > 0)` 也就是代码中的 `} else {` 代表的是父进程 `main` 的返回。
  - `close(pipefd[0])` 关闭了 `pipefd[0]` 文件句柄，因为程序设计成父进程负责写文件，它并不需要操作 `pipefd[0]`
  - `write(pipefd[1], argv[1], strlen(argv[1]))` 父进程往 `pipefd[1]` 中写入数据。数据将会出现在 `pipefd[0]` 中供子进程读取。

鸿蒙实现

管道的实现函数级调用关系如下:

```
SysPipe //系统调用
    AllocProcessFd  //分配两个进程描述符
    pipe    //底层管道的真正实现
        pipe_allocate   //分配管道
        "/dev/pipe%d"   //生成创建管道文件路径，用于创建两个系统文件句柄
        pipecommon_allocdev //分配管道共用的空间
        register_driver //注册管道设备驱动程序
        open    //打开两个系统文件句柄
        fs_getfilep //获取两个系统文件句柄的实体对象 `file`
    AssociateSystemFd //进程和系统文件句柄的绑定
```

其中最关键的是 `pipe` ，它才是真正实现管道思想的落地代码，代码稍微有点多，但看明白了这个函数就彻底明白了管道是怎么回事了，看之前先建议看文件系统相关篇幅，有了铺垫再看代码和解读就很容易明白。

```
int pipe(int fd[2])
{
  struct pipe_dev_s *dev = NULL;
  char devname[16];
  int pipeno;
  int errcode;
  int ret;
  struct file *filep = NULL;
  size_t bufsize = 1024;

  /* Get exclusive access to the pipe allocation data */

  ret = sem_wait(&g_pipesem);
  if (ret < 0)
    {
      errcode = -ret;
      goto errout;
    }
```

```
/* Allocate a minor number for the pipe device */

pipeno = pipe_allocate();
if (pipeno < 0)
  {
    (void)sem_post(&g_pipesem);
    errcode = -pipeno;
    goto errout;
  }

/* Create a pathname to the pipe device */

snprintf_s(devname ,  sizeof(devname) ,  sizeof(devname) - 1 ,  "/dev/pipe%d" ,  pipeno);

/* No.. Allocate and initialize a new device structure instance */

dev = pipecommon_allocdev(bufsize ,  devname);
if (!dev)
  {
    (void)sem_post(&g_pipesem);
    errcode = ENOMEM;
    goto errout_with_pipe;
  }

dev->d_pipeno = pipeno;

/* Check if the pipe device has already been created */

if ((g_pipecreated & (1 << pipeno)) == 0)
  {
    /* Register the pipe device */

    ret = register_driver(devname ,  &pipe_fops ,  0660 ,  (void *)dev);
    if (ret != 0)
      {
        (void)sem_post(&g_pipesem);
        errcode = -ret;
        goto errout_with_dev;
      }

    /* Remember that we created this device */

     g_pipecreated |= (1 << pipeno);
  }
else
  {
     UpdateDev(dev);
  }
(void)sem_post(&g_pipesem);

/* Get a write file descriptor */

fd[1] = open(devname ,  O_WRONLY);
if (fd[1] < 0)
  {
    errcode = -fd[1];
    goto errout_with_driver;
  }

/* Get a read file descriptor */

fd[0] = open(devname ,  O_RDONLY);
if (fd[0] < 0)
  {
    errcode = -fd[0];
    goto errout_with_wrfd;
  }

ret = fs_getfilep(fd[0] ,  &filep);
filep->ops = &pipe_fops;
```

```
  ret = fs_getfilep(fd[1] ， &filep);
  filep->ops = &pipe_fops;

  return OK;

errout_with_wrfd:
  close(fd[1]);

errout_with_driver:
  unregister_driver(devname);

errout_with_dev:
  if (dev)
    {
      pipecommon_freedev(dev);
    }

errout_with_pipe:
  pipe_free(pipeno);

errout:
  set_errno(errcode);
  return VFS_ERROR;
}
```

解读

- 在鸿蒙管道多少也是有限制的，也由位图来管理，最大支持32个，用一个32位的变量 `g_pipeset` 就够了，位图如何管理请自行翻看位图管理篇。要用就必须申请，由 `pipe_allocate` 负责。

```
    #define MAX_PIPES 32    //最大支持管道数
    static sem_t  g_pipesem       = {NULL};
    static uint32_t g_pipeset     = 0;  //管道位图管理器
    static uint32_t g_pipecreated = 0;

    static inline int pipe_allocate(void)
    {
    int pipeno;
    int ret = -ENFILE;

    for (pipeno = 0; pipeno < MAX_PIPES; pipeno++)
       {
       if ((g_pipeset & (1 << pipeno)) == 0)
          {
          g_pipeset |= (1 << pipeno);
          ret = pipeno;
          break;
          }
       }
    return ret;
    }
```

- 管道对外表面上看似对两个文件的操作，其实是对一块内存的读写操作。操作内存就需要申请内存块，鸿蒙默认用了 `1024 | 1K` 内存，操作文件就需要文件路径 `/dev/pipe%d` 。

```
  size_t bufsize = 1024;
  snprintf_s(devname ， sizeof(devname) ， sizeof(devname) - 1 ， "/dev/pipe%d" ， pipeno);
  dev = pipecommon_allocdev(bufsize ， devname);
```

- 紧接着就是要提供操作文件 `/dev/pipe%d` 的 `VFS` ，即注册文件系统的驱动程序，上层的读写操作，到了底层真正的读写是由 `pipecommon_read` 和 `pipecommon_write` 落地。

```
  ret = register_driver(devname ， &pipe_fops ， 0660 ， (void *)dev);
  static const struct file_operations_vfs pipe_fops =
  {
```

```
.open = pipecommon_open ,       /* open */
.close = pipe_close ,           /* close */
.read = pipecommon_read ,       /* read */
.write = pipecommon_write ,     /* write */
.seek = NULL ,                  /* seek */
.ioctl = NULL ,                 /* ioctl */
.mmap = pipe_map ,              /* mmap */
.poll = pipecommon_poll ,       /* poll */
#ifndef CONFIG_DISABLE_PSEUDOFS_OPERATIONS
.unlink = pipe_unlink ,         /* unlink */
#endif
};
```

`pipecommon_read` 代码有点多，此处不放出来，代码中加了很多的信号量，目的就是确保对这块共享内存能正常操作。

- 要操作两个文件句柄就必须都要打开文件，只不过打开方式一个是读，一个是写，`pipe` 默认是对 `fd[1]` 为写入，`fd[0]` 为读取，这里可翻回去看下经典管道案例的读取过程。

```
fd[1] = open(devname， O_WRONLY);
fd[0] = open(devname， O_RDONLY);
```

- 最后绑定 `file` 的文件接口操作，在文件句柄篇中已详细说明，应用程序操作的是 `fd | 文件句柄` ，到了内核是需要通过 `fd` 找到 `file` ，再找到 `file->ops` 才能真正的操作文件。

```
ret = fs_getfilep(fd[0]， &filep);
filep->ops = &pipe_fops;

ret = fs_getfilep(fd[1]， &filep);
filep->ops = &pipe_fops;
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

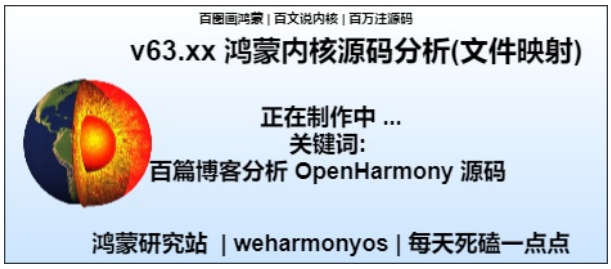
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

63_文件映射篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

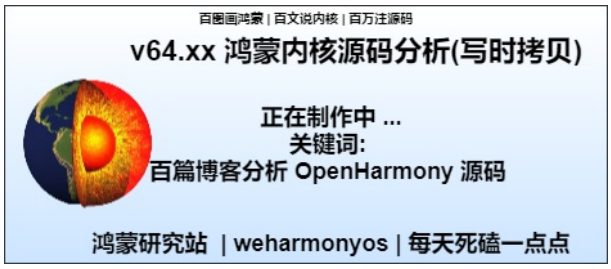
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

64_写时拷贝篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

文件系统相关篇为:

- v55.02 鸿蒙内核源码分析(文件概念) | 为什么说一切皆是文件
- v56.04 鸿蒙内核源码分析(文件故事) | 用图书管理说文件系统
- v57.06 鸿蒙内核源码分析(索引节点) | 谁是文件系统最重要的概念
- v58.02 鸿蒙内核源码分析(VFS) | 文件系统的话事人
- v59.04 鸿蒙内核源码分析(文件句柄) | 你为什么叫句柄
- v60.07 鸿蒙内核源码分析(根文件系统) | 谁先挂到 / 谁就是老大
- v61.05 鸿蒙内核源码分析(挂载机制) | 谁根逐流不掉队
- v62.05 鸿蒙内核源码分析(管道文件) | 如何降低数据流动成本
- v63.03 鸿蒙内核源码分析(文件映射) | 正在制作中 ...
- v64.01 鸿蒙内核源码分析(写时拷贝) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

65_芯片模式

本篇关键词：、、、

下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

进口最多的产品

- 我国每年进口最多的商品不是石油而是**芯片**，2021年我国进口集成电路6355亿个，同比增长16.9%，而进口集成电路的金额约4400亿美元（2.8万亿），同比增长25.6%左右。比同年中国进口的石油和大豆加起来都多。每年进口集成电路的金额占中国所有进口金额的16%，也就是说每进口6块钱的商品中，就有1块钱是芯片。而我国芯片的自给率只有26.6%，其中汽车芯片的自给率更是低至5% 。

整合元件制造商 | IDM 模式

整合元件制造商（IDM，Integrated Device Manufacturer）从IC设计、制造、封装、测试到销售都一手包办，但需要雄厚的运营资本才能支撑此营运模式，最典型的是英特尔。另一方面，三星电子虽然具有晶圆厂，能制造自己设计的芯片，但因为建厂成本太高，它同时提供代工服务。原本为IDM的AMD则在2009年将晶圆制造业务独立为格芯（GlobalFoundries），而转型为无厂半导体公司；

全球主要IDM企业有：

- **英特尔 （Intel | 美国）** ：首家推出x86架构中央处理器的公司，总部位于美国加利福尼亚州圣克拉拉。由罗伯特·诺伊斯、高登·摩尔、安迪·葛洛夫，以“集成电子”（Integrated Electronics）之名在1968年7月18日共同创办公司，将高端芯片设计能力与领导业界的制造能力结合在一起。英特尔也有开发主板芯片组、网卡、闪存、绘图芯片、嵌入式处理器，与对通信与运算相关的产品等。“Intel Inside”的广告标语与Pentium系列处理器在1990年代间非常成功地打响英特尔的品牌名号。

- **三星 （Samsung | 韩国）** ：三星电子（朝鲜语：삼성전자／三星電子，英语：Samsung Electronics），是三星集团旗下的子公司，韩国最大的消费电子产品及电子组件制造商，亦是全球最大的信息技术公司。目前主要的半导体业务是DRAM、NAND等闪存、微控制器以及影像感测器。近年来为了扩大市场，也进入了微处理器以及晶圆代工。苹果公司iPhone的主要零件：闪存、多层陶瓷电容器、电池和软性电路板均为三星所代工和研发制造的。三星在DRAM领域居产业霸主，让海力士、尔必达、美光等公司的市占率都在三星之下。

- **SK海力士（SK Hynix | 韩国）** ：SK海力士前身为1983年成立的现代电子产业株式会社，2012年被SK集团收购以后正式更名为SK海力士株式会社。SK海力士致力于生产DRAM、NAND Flash和CIS非存储器为主的半导体产品。作为全球领先的半导体制造商，当前在韩国利川和清州、 中国无锡和重庆设有四个生产基地，并在全球16个国家和地区设立了销售、研发等基地。基于过去三十多年的半导体生产运营经验， SK海力士致力于实现持续的研发与投资，增强技术与成本竞争力，引领全球半导体市场。

- **美光（Micron | 美国）** ：美光科技公司（英语：Micron Technology, Inc.，NASDAQ：MU），简称美光科技，是美国一家总部位于爱达荷州波夕的半导体制造公司，于1978年由Ward Parkinson、Joe Parkinson、Dennis Wilson和Doug Pitman创立。其主要业务为生产多种形式的半导体器件，包括动态随机存取存储器，闪存和固态驱动器；其主要产品包括DRAM、NAND快闪存储器、CMOS影像感测器、其它半导体元件和内存模组。美光科技目前旗下品牌包括Crucial（英睿达）、Ballistix（铂胜）和2006年购并而来的Lexar（雷克沙）。

- **德州仪器 （TI | 美国）** ：德州仪器（英语：Texas Instruments, TI）是一家位于美国德克萨斯州达拉斯的跨国公司，以开发、制造、销售半导体和计算器技术闻名于世，主要从事数字信号处理与模拟电路方面的研究、制造和销售。它在25个国家有制造、设计或者销售机构。根据2021年IC Insights统计，德州仪器是世界第九大半导体制造商；曾经是移动电话的第二大芯片供应商，仅次于高通；同时也是在世界范围内的第一大数

字信号处理器（DSP）和模拟半导体组件的制造商，其产品还包括计算器、微控制器以及多核处理器。德州仪器居世界半导体公司20强。

- **恩智浦&飞思卡尔（NXP&Freescale）**：恩智浦半导体（英语：NXP Semiconductors），前身为飞利浦半导体，由荷兰企业飞利浦在1953年创立，公司总部位于荷兰埃因霍温。2006年8月31日，该公司首席执行官万豪敦在柏林向客户和员工宣布公司的新名称。恩智浦半导体目前可以提供半导体、系统和软件解决方案；使用在汽车、手机、智能识别应用、电视、机上盒以及其他电子设备上。
  - 2015年3月3日，恩智浦半导体以现金加股票收购同业飞思卡尔（Freescale）
  - 2018年7月25日，高通宣布放弃收购恩智浦，并向恩智浦支付20亿美元的分手费。
  - 2019年5月29日恩智浦半导体斥资17.6亿美元现金收购美满电子科技的无线连接组合，包括美满电子科技的WiFi连接业务部门、蓝牙技术组合和相关资产。
  - 恩智浦主要致力于 智慧识别、汽车电子、家庭娱乐、多重市场半导体、恩智浦软件市场。
- **东芝 （Toshiba | 日本）**：东芝（日语：株式会社東芝／かぶしきがいしゃとうしば ，英语：Toshiba Corporation）是一家总部位于东京港区的日本跨国企业集团，为电力开发、工业生产、环境保护、商业办公、半导体等领域提供设备制造和解决方案，曾为全球最大的个人电脑、家用电器和医疗设备制造商之一，也是闪存的发明者，其消费电子产品业务已悉数分拆出售，白色家电业务售予中国美的集团，电视机业务售予中国海信集团，个人电脑业务售予台湾鸿海集团子公司夏普，电脑内存业务被剥离为后来的铠侠。业务范围涵盖：
  - 电力：东芝是全球重要的电力开发和传输设备制造商，为火力发电、水力发电、氢能发电等提供技术和解决方案，为变电站提供各种输变电设备。
  - 工业：为电动机、变频器、工业CT设备、高压传动系统、动力电池等工业设备制造商。
  - 环保：为自来水处理系统、污水处理系统提供控制技术。
  - 办公系统：制造各类打印机、复印机、传真机、复合机、触摸互动一体机等办公自动化器材。
  - 电梯：为专业电梯制造商，制造各类乘客电梯、货梯、电动扶梯、观光电梯。
  - 空调：生产家用及商用空调。
  - 电子组件：生产各类半导体器件以及移动硬盘、机械硬盘等存储产品。
- **英飞凌（Infineon | 德国）**：英飞凌科技股份有限公司（Infineon Technologies，FWB：IFX）总部位于德国慕尼黑，主力提供半导体和系统解决方案，解决在高能效、移动性和安全性方面带来的挑战（而主要业务亦包括为关连公司西门子交通集团生产铁路机车车辆牵引系统内IGBT-VVVF之半导体组件）。英飞凌前身是西门子集团旗下子公司西门子半导体（Siemens Semiconductor），于1999年独立，2000年上市。中文名曾被称为亿恒科技，2002年起更至现名。其无线解决方案部门在2010年8月售给英特尔。
- **意法半导体（ST | 意法）**：意法半导体（英语：STMicroelectronics）是一家国际性的半导体生产商，总部位于瑞士日内瓦。意法半导体集团在1987年由意大利的Società Generale Semiconduttori (SGS) Microelettronica与法国汤姆逊（Thomson）公司的半导体分部Thomson Semiconducteurs两家半导体公司合并而成，该公司自1998年5月汤姆逊撤股后由SGS-THOMSON更名为意法半导体（STMicroelectronics）。与所谓的无厂半导体公司不同，意法半导体拥有并营运自己的半导体晶圆厂。意法半导体由五个产品团队组成，每个团队由数个部门或业务单位组成，每个部门均负责设计、工业化、生产（使用意法半导体的制造工厂）与销售自己的产品，业务透过中央研发组织与地区的营业部支持。产品部分为：
  - 家用个人通信团队：消费、多媒体、无线及有线产品
  - 存储器产品团队：独立记忆芯片（EEPROM、闪存（NAND与NOR）、串行闪存与智能卡）
  - 汽车产品团队：关系汽车产业的芯片（车身、传动系、安全设备等）
  - 微型、功率与模拟团队：模拟与功率集成电路及单片机
  - 电脑周边团队：电脑周边用芯片（硬盘控制器、打印机等）
  - 前端技术与生产：研究与开发，该公司总体上拥有16个研究与开发单位及39个设计与应用中心

无厂半导体 | Fabless 模式

无厂半导体公司（英语：fabless semiconductor company）是指只从事硬件芯片的电路设计，后再交由晶圆代工厂制造，并负责销售的公司。由于半导体器件制造耗资极高，将集成电路产业的设计和制造两大部分分开，使得无厂半导体公司可以将精力和成本集中在市场研究和电路设计上。而专门从事晶圆代工的公司则可以同时为多家无厂半导体公司制造生产，尽可能提高其生产线的利用率

全球主要IC设计企业有：

- **高通 （Qualcomm | 美国）**：高通公司（英语：Qualcomm）是位于美国加州圣地亚哥的无线电通信技术研发公司，由加州大学圣地亚哥分校教授厄文·马克·雅各布和安德鲁·维特比创建，于1985年成立。
  - 2017年11月6日，博通有限（Broadcom）计划以60美元现金加10美元股份的配比，合共每股70美元，涉资超过1300亿美元收购高通。
  - 2017年11月13日，高通全体董事一致正式拒绝博通收购提案，该公司董事会一致认为，以高通在行动技术的领导地位，以及未来成长潜力来看，博通的提议明显低估高通的价值。
  - 2018年3月12日，美国总统特朗普发布行政命令，以国家安全为由阻挡这起并购案。特朗普声明表示：“依据可信的证据，我认为博通并购高通可能妨碍美国国家安全。”
- **博通 （Broadcom | 新加坡）**：博通有限公司（英语：Broadcom Limited，NASDAQ：AVGO），前身为安华高科技（Avago Technologies Limited），美国的无厂半导体公司，其业务内容广泛，在美国圣荷西设立总部。
  - 安华高科技创立于1961年，在2016年1月完成对博通的收购后，改名博通有限公司。安华高科技于1961年成立，原为惠普公司之下的半导体部门。

  - 1999年，HP公司分拆出安捷伦公司。2005年，安捷伦公司将其I/O solutions部门分拆出售。
  - 2015年5月29日，安华高科技公司收购博通公司。2016年完成收购，改名博通有限公司。在2018年总部搬迁至美国。
- **联发科 （MediaTek | 台湾）** ：联发科技（英语：MediaTek Inc.，有时非正式缩写作MTK），简称联发科，是台湾一家为无线通信、高清电视设计系统芯片的无厂半导体公司。公司成立于1997年，总部位于新竹科学园区，在全球设有25个分公司和办事处，2013年成为全球第四大无晶圆厂IC设计商，2016年成为全球第三大，2020年凭借天玑系列芯片成为全球市场占有率第一大。
- **英伟达（NVIDIA | 美国）** ：黄仁勋、克里斯·马拉科夫斯基和卡蒂斯·普里姆于1993年4月美国加州创办了NVIDIA 。作为一家无晶圆IC半导体设计公司，NVIDIA于自己的实验室研发芯片，但将芯片制造工序分包给晶圆代工场。NVIDIA的产品组合包括绘图处理器、个人电脑平台（主板逻辑核心）芯片组和数字媒体播放器的软件。
- **AMD （美国）** ：超微半导体公司（英语：Advanced Micro Devices, Inc.；缩写：AMD、超微，或译“超威”），创立于1969年，是一家专注于微处理器及相关技术设计的跨国公司，总部位于美国加州旧金山湾区硅谷内的森尼韦尔市。最初，超微拥有晶圆厂来制造其设计的芯片，自2009年超微将自家晶圆厂拆分为现今的格芯以后，成为无厂半导体公司，仅负责硬件集成电路设计及产品销售业务。现时，超微的主要产品是中央处理器（包括嵌入式平台）、图形处理器、主板芯片组以及电脑存储器。AMD 是目前除了英特尔以外，最大的x86架构微处理器供应商，自收购冶天科技以后，则成为除了英伟达和将发布独立显卡的英特尔以外仅有的独立图形处理器供应商，自此成为一家同时拥有中央处理器和图形处理器技术的半导体公司，也是唯一可与英特尔和英伟达匹敌的厂商。
- **海思（Hisilicon | 大陆）** ：海思半导体（英语：Hisilicon），中国大陆芯片设计公司，属于华为集团，于2004年4月创建，总部位于中国大陆广东省深圳，现为中国大陆最大的无晶圆厂芯片设计公司。主要产品为无线通信芯片，包括拥有WCDMA、LTE等功能的手机系统单片机。海思半导体的前身为创建于1991年的华为集成电路设计中心。2020年第一季度，华为海思的智能手机处理器出货量首次在中国大陆市场超过高通，位居第一。
  - 麒麟系列产品：是面向手机、平板电脑等终端设备设计的SoC。如 990系列、9000系列

9000系列

型号	工艺	CPU				GPU		内存支持			卫星定位	通讯技术支持			样品发布时间	采用产品
		指令集 (ARM)	微架构	核心数	频率(GHz)	微架构	频率 (MHz)	样式	总线带宽(bit)	带宽		类型	总线宽度 (bit)	带宽 (GB/s)		
Kirin 9000E	TSMC 5 nm FinFET (EUV)	ARMv8.2-A	Cortex-A77 Cortex-A55 DynamIQ	(1+3)+4	3.13 (A77 H) 2.54 (A77 L) 2.05 (A55)	Mali-G78 MP22	?	LPDDR4X-2133 LPDDR5-2750	64-bit(4x16-bit) Quad-channel	34.1 (LPDDR4X) 44 (LPDDR5)	Galileo	Balong 5000 (Sub-6-GHz only; NSA & SA), Balong 4G	不适用	不适用	Q4 2020	列表 华为Mate 40系列
Kirin 9000 5G/4G						Mali-G78 MP24	?						不适用	不适用		列表 华为Mate 40 Pro 华为 Mate 40 Pro+ 华为Mate 40 RS Porsche Design

    - 巴龙系列产品（Modem）：是终端设备的基带处理器，搭载在华为手机、随身WiFi和CPE等设备上。如 巴龙5G01、巴龙5000
    - 可穿戴设备 SoC ：用于真无线耳机、智能眼镜和智能手表等可穿戴设备的SoC。如 麒麟 A1
    - 服务器处理器：基于ARM架构的服务器处理器 SoC。如 鲲鹏916、鲲鹏920
    - 人工智能处理器：基于自研达芬奇架构设计的人工智能处理器。如 昇腾310、昇腾910
- **苹果 （Apple | 美国）** ：全球市值最高的公司，太有名了。苹果标志的来由多被误解为“图灵自杀时吃了一口的氰化物溶液苹果”。这个传闻来源为2001年的英国电影《Enigma》，在该部电影中虚构了前述有关图灵自杀与苹果公司标志关系的情节，被部分公众以及媒体讹传。而苹果标志的设计师在一次采访中亲自证实这个标志与图灵（或者其它的猜测，比如被夏娃咬的那个苹果）无关。乔布斯也曾说“被咬掉一口的设计只是为了让它看起来不像樱桃”。
- **美满电子(Marvell | 美国)** ：美满电子科技公司（Marvell Technology Group）是一家美国芯片制造商，专门制造存储、通讯以及消费性电子产品芯片。公司由周秀文（英语：Sehat Sutardja）（Sehat Sutardja）博士、妻戴伟立（Weili Dai）、弟周秀武三人共同创立于1995年，总部在美国硅谷，在中国上海设有研发中心，亚太地区包括北京、韩国、台湾这些地方都有技术支持团队，现在的员工大约5000人并在不断扩张中。其是美国纳斯达克的上市公司，市值大约是110亿美元。目前Marvell是全球十大无芯片工厂半导体设计公司。Marvell现在每天要运送超过100万片基于ARM架构的处理器，这不单单包括手机应用处理器，同时也包括通信处理器、存储、WiFi等芯片。
- **赛灵思（Xilinx | 美国）** 是一家位于美国的可编程逻辑器件生产商。1984年创建于美国加利福尼亚州的硅谷，该公司发明了现场可编程逻辑门阵列，并由此成名。赛灵思还是第一个无晶圆厂半导体公司。赛灵思是FPGA、可编程SoC及ACAP的发明者， 其高度灵活的可编程芯片由一系列先进的软件和工具提供支持，可推动跨行业和多种技术的快速创新 - 从消费电子类到汽车类再到云端。
  - 2020年10月27日，赛灵思正式被AMD收购。
- **紫光展锐 (unisoc | 大陆)** ：紫光展锐（上海）科技有限公司是我国集成电路设计企业的龙头企业。以生态为核心战略，高举5G和AI两面技术旗帜，以价值、未来、服务为三个指向，为个人与社会的智能化服务。紫光展锐是全球少数全面掌握2G/3G/4G/5G、Wi-Fi、蓝牙、电视调频、卫星通信等全场景通信技术的企业之一，并具备稀缺的大型芯片集成及套片能力。产品包括移动通信中央处理器，基带芯片，AI芯片，射频前端芯片，射频芯片等各类通信、计算及控制芯片。

晶圆代工 | Foundry 模式

晶圆代工或晶圆专工（Foundry），是半导体产业的一种商业模式，指接受其他无厂半导体公司（Fabless）委托、专门从事晶圆成品的加工而制造集成电路，并不自行从事产品设计与后端销售的公司。在纯晶圆代工公司出现之前，芯片设计公司只能向整合元件制造厂购买空闲的晶圆产能，产量与生产排程都受到非常大的限制，不利于大规模量产产品。 1987年创立的台积电是第一家专门从事晶圆代工的厂商，此后联电亦转型为专门晶圆代工公司；以及有中芯国际、世界先进、格罗方德等公司接连成立，目前全世界有十余家提供晶圆代工服务的公司。此外即使是垂直整合制造的半导体公司，如三星电子、英特尔等，也有晶圆代工的业务。晶圆代工服务使得专业芯片设计的商业模式也得以实现，推升了芯片设计公司的蓬勃发展。

全球主要晶圆代工企业有：

- **台积电（TSMC | 台湾）** ：台湾积体电路制造公司（英语：Taiwan Semiconductor Manufacturing），是台湾一家从事晶圆代工的半导体制造厂，总部位于新竹科学园区。1986年，(现中国台湾省)政府为培植半导体产业，由工研院主导与荷兰飞利浦共同签约并成立一家半导体制造公司，交由时任工研院院长的 张忠谋 带着一群以出身工研院为主的工程师一同筹办，张忠谋遂出任台湾积体电路制造股份有限公司董事长兼总

裁。

- **格罗方德（Global Foundries | 美国）**：格罗方德半导体股份有限公司(Global Foundries)是一家总部位于美国加州硅谷桑尼维尔市的半导体晶圆代工厂商, 成立于2009年3月 。格罗方德半导体股份有限公司由AMD拆分而来、与阿联酋阿布扎比先进技术投资公司(ATIC)和穆巴达拉发展公司(Mubadala)联合投资成立的半导体制造企业。
- **联电（UMC | 台湾）**：联华电子股份有限公司，简称联电，英文全名“United Microelectronics Corporation”，英文缩写“UMC”，(现中国台湾省)政府出资创立于1980年，为台湾国际半导体晶圆专工业界的领导者，提供高品质的晶圆制造服务，专注于逻辑及特殊技术，为跨越电子行业的各项主要应用产品生产芯片。联电完整的制程技术及制造解决方案包括逻辑/混合信号、嵌入式高压解决方案、嵌入式非挥发性内存、RFSOI及BCD。
- **中芯国际（SMIC | 大陆）**：中芯国际集成电路制造有限公司（简称中芯国际，港交所：981、NYSE：SMI)于2000年4月3日在开曼群岛注册成立，总部位于中国大陆上海。公司的创立者之一为曾在台积电任职过的张汝京。是世界领先的集成电路晶圆代工企业之一，也是中国大陆规模最大、技术最先进的集成电路芯片制造企业。中芯国际向全球客户提供0.35微米到14纳米晶圆代工与技术服务，包括逻辑芯片，混合信号/射频收发芯片，耐高压芯片，系统芯片，闪存芯片，EEPROM芯片，图像传感器芯片及LCoS微型显示器芯片，电源管理，微型机电系统等。同时7纳米已经进入到客户风险量产阶段。
- **力晶科技（Powerchip | 台湾**：力晶积成电子制造股份有限公司（英语：Powerchip Semiconductor Manufacturing Corporation），简称力积电、PSMC，业务范围涵盖动态随机存取存储器（DRAM）、非易失性存储器（Flash）制造及晶圆代工两大类别。总公司位于台湾新竹市新竹科学工业园区，创办人为黄崇仁。
- **高塔半导体（Tower jazz | 以色列）**：高塔半导体有限公司（英语：Tower Semiconductor Ltd.）是以色列的一家半导体专业代工厂，总部在以色列的米格达勒埃梅克。起始于1993年购并了美国国家半导体的150mm芯片制造设备，并在1994年成为上市公司。
- **世界先进(VIS | 台湾)**：世界先进积体电路股份有限公司（简称「世界先进」）于1994年12月5日在新竹科学园区设立。自成立以来，公司在制程技术及生产效能上不断精进，并持续提供最具成本效益的完整解决方案及高附加价值的服务予客户，成为「特殊积体电路制造服务」的领导厂商。世界先进目前拥有五座八吋晶圆厂，分别位于台湾与新加坡。2021年平均月产能约24.1万片八吋晶圆。世界先进及子公司共有近6,000名员工，坚持以「客户服务为导向」的经营理念，持续加强对特殊积体电路晶圆代工客户的专业服务。
- **华虹（大陆）**：华虹集团是中国目前拥有先进芯片制造主流工艺技术的8+12寸芯片制造企业。集团旗下业务包括集成电路研发制造、电子元器件分销、智能化系统应用等板块，其中芯片制造核心业务分布在浦东金桥、张江、康桥和江苏无锡四个基地，目前运营3条8英寸生产线、3条12英寸生产线。量产工艺制程覆盖1微米至28纳米各节点。二十多年来，集团在致力于发展自主可控集成电路产业的征程上取得了多个行业第一和唯一：率先建成了中国大陆第一条8英寸集成电路生产线，建设了本土企业第一条全自动的12英寸生产线；具有唯一一家国家级集成电路研发中心；成为业界第一家，也是唯一一家，连续两年建设并投产运营两条12英寸生产线的企业。

封测 | OSAT 模式

这个类比软件开发中的测试岗，相对来说没太多技术含量，所以国内企业占有较多席位。封测（OSAT）主要是封装测试代工厂，系统级封装（System in Package, SiP）为一种集成电路封装的概念，是将一个系统或子系统的全部或大部分电子功能配置在集成型衬底内，而芯片以2D、3D的方式接合到集成型衬底的封装方式。SIP不仅可以组装多个芯片，还可以作为一个专门的处理器、DRAM、闪存与被动组件结合电阻器和电容器、连接器、天线等，全部安装在同一衬底上。这意味着，一个完整的功能单位可以建在一个多芯片封装，因此，需要添加少量的外部组件，使其工作，大概有图中测试内容

全球主要封测代工企业有：

- **日月光（ASE | 台湾）** ：日月光是最大的半导体委外封装和测试（OSAT）供应商，占有19％的市场份额。 日月光为全球90％以上的电子公司提供半导体封装和测试服务。创办人张姚宏影为星云法师弟子，日月光名称由星云法师命名：星云大师觉得电和光代表快、代表速度，可以照耀万千人，可以永久、可以大，取名“日月光”。同时，星云大师也为日月光下了“日月无私照、光明有佛心”的对联。
- **安靠（Amkor | 美国）** ：艾克爾國際科技（英语：Amkor Technology，Inc.）是一家美国半导体产品封装和测试服务提供商。该公司成立于1969年，自2005年总部从美国宾夕法尼亚州威彻斯特搬迁至亚利桑那州坦佩。截至2017年，艾克爾國際科技在全球拥有约29,300名员工，销售额为41.9亿美元。艾克爾國際科技在中国大陆、日本、韩国、马来西亚、菲律宾、葡萄牙和台湾设有工厂。它为芯片制造商提供封装和测试集成电路（IC）服务。
- **长电科技（大陆）** ：长电科技是全球领先的集成电路制造和技术服务提供商，提供全方位的芯片成品制造一站式服务，包括集成电路的系统集成、设计仿真、技术开发、产品认证、晶圆中测、晶圆级中道封装测试、系统级封装测试、芯片成品测试并可向世界各地的半导体客户提供直运服务。通过高集成度的晶圆级封装（WLP）、2.5D/3D封装、系统级封装（SiP）、高性能倒装芯片封装和先进的引线键合技术，长电科技的产品、服务和技术涵盖了主流集成电路系统应用，包括网络通讯、移动终端、高性能计算、车载电子、大数据存储、人工智能与物联网、工业智造等领域。长电科技在全球拥有23000多名员工，在中国、韩国和新加坡设有六大生产基地和两大研发中心，在逾23个国家和地区设有业务机构，可与全球客户进行紧密的技术合作并提供高效的产业链支持。
- **矽品（SPIL | 台湾）** ：
- **力成科技（台湾）** ：
- **华天科技（大陆）** ：华天科技成立于2003年12月25日，主要从事半导体集成电路、半导体元器件的封装测试业务。作为全球半导体封测知名企业，华天科技为客户提供封装设计、封装仿真、引线框封装、基板封装、晶圆级封装、晶圆测试及功能测试、 物流配送等一站式服务。凭借先进的技术能力，系统级生产和质量把控，已成为半导体封测业务首选品牌。
- **通富微电（大陆）** ：公司成立于1997年10月，通富微电专业从事集成电路封装测试，总部位于江苏南通，拥有崇川总部工厂、南通通富微电子有限公司（南通通富）、合肥通富微电子有限公司（合肥通富）、厦门通富微电子有限公司（厦门通富）、苏州通富超威半导体有限公司（TF-AMD苏州）、TF AMD Microelectronics (Penang) Sdn. Bhd.（TF-AMD槟城）六大生产基地。通过自身发展与并购，公司已成为本土半导体跨国集团公司、中国集成电路封装测试领军企业，集团员工总数超1.5万人。
- **京元电子（台湾）** ：
- **南茂科技（新加坡）** ：
- **联合科技（台湾）** ：

IP核 | ARM 模式

只负责设计电路，不负责芯片设计、不负责制造、销售的公司则称为IP核公司。

- IP核，全称知识产权核（英语：Semiconductor intellectual property core），是在集成电路的可重用设计方法学中，指某一方提供的、形式为逻辑单元、芯片设计的可重用模块。
- IP核通常已经通过了设计验证，设计人员以IP核为基础进行设计，可以缩短设计所需的周期。IP核可以通过协议由一方提供给另一方，或由一方独自占有。IP核的概念源于产品设计的专利证书和源代码的著作权等。设计人员能够以IP核为基础进行专用集成电路或现场可编程逻辑门阵列的逻辑设计，以减少设计周期。
- IP核分为软核、硬核和固核。软核通常是与工艺无关、具有寄存器传输级硬件描述语言描述的设计代码，可以进行后续设计；硬核是前者通过逻辑综合、布局、布线之后的一系列表征文件，具有特定的工艺形式、物理实现方式；固核则通常介于上面两者之间，它已经通过功能验证、时序分析等过程，设计人员可以以逻辑门级网表的形式获取。
- **ARM（英国）**：安谋控股公司（英语：ARM Holdings plc.，写作arm），又称ARM公司，是软银集团旗下的半导体设计与软件公司，全球总部位于英国剑桥，北美总部位于美国圣何塞，亦是一年一度的ARM技术大会（Arm TechCon）举办地。主要的产品是ARM架构处理器及相关外围组件的电路设计方案，产品以知识产权核授权的形式与相应的软件开发工具一起向客户销售。
  - 以设计 **ARM处理器架构** 闻名于世，技术具有性能高、成本低和能耗省的特点，产品已遍及工业控制、消费类电子产品、通信系统、网络系统。ARM自己不制造芯片，而将其技术知识产权（IP核）授权给世界上许多著名的半导体厂，其中包括Intel、IBM、LG半导体、NEC、SONY、飞利浦、Atmel、Broadcom、Cirrus Logic、Freescale、Actions等。
  - 2010年6月中，苹果公司向ARM董事会表示有意以85亿美元的价格收购ARM公司，但遭到ARM董事会的拒绝。称ARM公司作为独立公司更具价值，“买家展开收购的唯一理由是消灭竞争对手。”
  - 2016年7月18日，日本软银集团同意以243亿英镑（约309亿美元）全现金方式收购ARM公司。交易于2016年9月5日完成，成为软银集团旗下的全资子公司，同时软银集团表示不干预亦不影响ARM现在及未来的商业计划和决策。
  - 2020年9月14日，NVIDIA宣布出价400亿美元（这项交易包含215亿美元Nvidia股票及120亿美元现金，软银将持有Nvidia最多8.1%流通股。此外，根据合约中的额外对价条款，软银可能还可再获得50亿美元的现金或股票。）从日本软银集团手中收购安谋控股。收购完成后软银将保留安谋控股不到10%的股份。此项交易仍须获得英国、中国、欧盟和美国相关机构的批准，监管审批可能需要长达一年半的时间。然而，尽管NVIDIA一再表示收购完成后ARM仍然保持开源模式和对客户采取中立态度，此收购案却遭到监管机构和大型企业（例：高通、微软、Google等）的一致反对。他们担心一旦收购完成后，将会造成严重的竞争问题，使ARM无法保持中立性。
  - 最终，2022年2月7日，NVIDIA和软银宣布已同意终止双方此前达成的ARM股份交易协议，收购案正式宣告失败，同时软银将推动ARM上市。

几点感想

看完之后站长有两点感想

- 芯片产业体量巨大，足以卷入庞大的技术人才诞生更多伟大的产品。韩国这么点人口都能在半导体行业产生像三星这般伟大的公司，所以我们真的是缺人才吗? 我们缺平台，缺载体，缺组织者，缺切入点。
- 鸿蒙生当其时，填补了这些缺口，鸿蒙撬动的是软硬件的上下游技术，以此为载体，形势一片大好。我辈应当努力，合力围拱，奋力追赶，弯道超车。加油 !!!

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

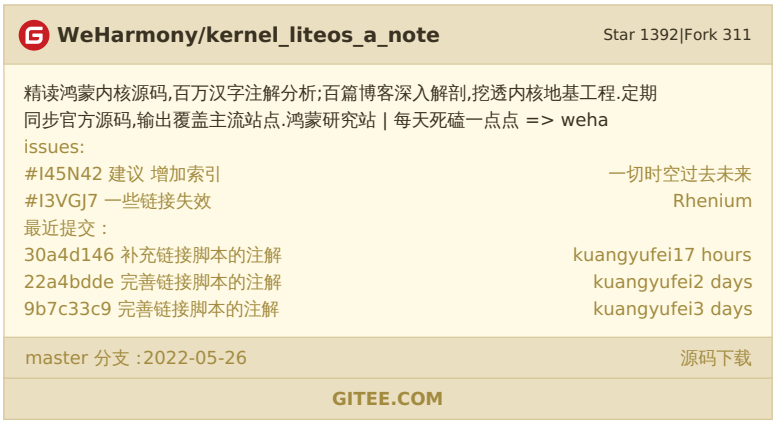
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

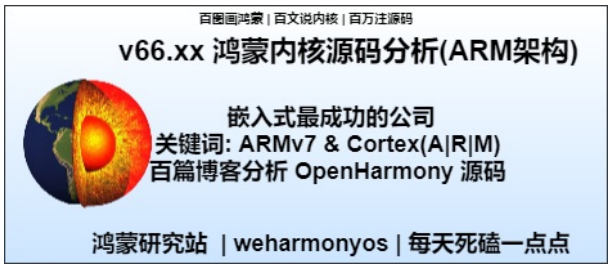


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

66_ARM架构篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

ARM模式

- ARM 公司本身并不靠自有的设计来制造或出售 CPU ，而是将处理器架构授权给有兴趣的厂家。提供了 ARM 内核的集成硬件叙述，包含完整的软件开发工具（编译器、 debugger 、 SDK ），以及针对内含 ARM CPU 硅芯片的销售权。
- 许多半导体公司持有 ARM 授权： Atmel 、 Broadcom 、 Cirrus Logic 、 Freescale （于2004从摩托罗拉公司独立出来）、富士通、英特尔（借由和Digital的控诉调停）、 IBM 、 NVIDIA 、台湾新唐科技（Nuvoton Technology）、英飞凌、任天堂、恩智浦半导体（于2006年从飞利浦独立出来）、冲电气、三星电子、苹果、夏普、意法半导体、德州仪器和VLSI等许多这些公司均拥有各个不同形式的 ARM 授权。虽然 ARM 的授权项目由保密合约所涵盖，在知识产权工业， ARM 是广为人知最昂贵的 CPU 内核之一。单一的客户产品包含一个基本的 ARM 内核可能就需索取一次高达美金 20万 的授权费用。而若是牵涉到大量架构上修改，则费用就可能超过千万美元。

处理器时间轴 | Cortex | 2006

年份	经典核心					Cortex核心			
	ARM7	ARM8	ARM9	ARM10	ARM11	微控制器	实时	应用 (32位)	应用 (64位)
1993	ARM700								
1994	ARM710 ARM7DI ARM7TDMI								
1995	ARM710a								
1996		ARM810							
1997	ARM710T ARM720T ARM740T								
1998			ARM9TDMI ARM940T						
1999			ARM9E-S ARM966E-S						
2000			ARM920T ARM922T ARM946E-S	ARM1020T					
2001	ARM7TDMI-S ARM7EJ-S		ARM9EJ-S ARM926EJ-S	ARM1020E ARM1022E					
2002				ARM1026EJ-S	ARM1136J(F)-S				
2003			ARM968E-S		ARM1156T2(F)-S ARM1176JZ(F)-S				
2004						Cortex-M3			
2005					ARM11MPCore			Cortex-A8	
2006			ARM996HS						
2007						Cortex-M1		Cortex-A9	
2008									
2009						Cortex-M0		Cortex-A5	
2010						Cortex-M4(F)		Cortex-A15	
2011							Cortex-R4 Cortex-R5 Cortex-R7	Cortex-A7	
2012						Cortex-M0+			Cortex-A53 Cortex-A57
2013								Cortex-A12	
2014						Cortex-M7(F)		Cortex-A17	
2015									Cortex-A35 Cortex-A72
2016						Cortex-M23 Cortex-M33(F)	Cortex-R8 Cortex-R52	Cortex-A32	Cortex-A73
2017									Cortex-A55 Cortex-A75
2018						Cortex-M35P			Cortex-A76

从图中可以看出 ARM 有 **经典(Classic)** 和 **Cortex** 两个核心系列，太老的历史就不去翻了， Cortex 就是 ARM 公司一个系列处理器的名称。比如英特尔旗下处理器有酷睿，奔腾，赛扬。 ARM 在最初的处理器型号都用数字命名，最后一个是 ARM11 系列，在 ARM11 以后的产品改用 Cortex 命名，时间分割线是 2006年 前后，并分成 A 、 R 和 M 三类，有意思的是这三个系列也暗合了 ARM 这个名字，为各种不同的市场提供服务。

- **应用类**（Application）：简称 Cortex-A 系列，面向尖端的基于虚拟内存的操作系统和用户应用。
- **嵌入式类**（Real-time）：简称 Cortex-R 系列，针对实时系统。
- **微处理器类**（Micro-controller）：简称 Cortex-M 系列，对微控制器和低成本应用提供优化。
- 鸿蒙内核分成 **轻量型**(基于 LiteOS_M) 和 **小型**(基于 LiteOS_A) 说的就是分别基于 Cortex-A/R 和 Cortex-M 的内核实现。

指令集时间轴 | RISC | ARMv7

看完处理器时间轴，再看 ARM **指令集架构**历史，很多人分不清指令集和处理器的区别，指令集是处理器使用的指令编码方式，指令集的命名方式为 armv + version，目前是 armv1 ~ armv8，数字越大表示指令集越先进，对于不同的处理器， arm 公司设计的处理器采用了不同的指令集。

- 精简指令集计算机（英语：reduced instruction set computer，缩写：**RISC**）或简译为 **精简指令集**，是计算机中央处理器的一种设计模式。特点是指令数目少，每条指令都采用标准字长、执行时间短。 RISC 处理器每条指令执行一个动作，只需一个周期即可完成，优化了操作执行时间。使用固定长度的指令，所以流水线更容易。并且由于它缺乏复杂的指令解码逻辑，它支持更多的寄存器并且花费更少的时间将值加载和存储到内存中。总结下来这样的好处是非常的省电，对于手持设备来说这是巨大的优势，不用抖音没滑几下就要到处找充电宝，所以这也是在移动互联网时代 ARM 芯片大行其道的最底层原因，通常被认为是当今可用的最高效的 CPU 架构技术。目前使用**RISC**的微处理器包括 DEC Alpha 、 ARC 、 ARM 、 AVR 、 MIPS 、 PA-RISC 、 Power ISA （包括PowerPC、PowerXCell）、 RISC-V 和 SPARC 等。
- 与之对应的是复杂指令集计算机（英文：Complex Instruction Set Computer；缩写：**CISC**）或简译为 **复杂指令集**，是一种微处理器指令集架构，每个指令可执行若干低端操作，诸如从存储器读取、存储、和计算操作，全部集于单一指令之中。特点是指令数目多而复杂，每条指令字长并不相等，电脑必须加以判读，并为此付出了性能的代价。
- 指令集是标准，基于标准可以设计无数的处理器型号。这并不难理解，跟我们手机充电线一样 TYPE-C 是目前大部分安卓手机的标准，但设计充电线的公司可以有很多， ARM 就是设计充电线的公司。 指令集：处理器 = 1：N (1对多) 关系，指令集向下兼容，指令集的设计原则是 **开闭原则**，对扩展是开放的，但是对于修改是封闭的。但注意 **RISC** 是一套标准，可不是 ARM 公司的私有财产。 ARM 公司在使用这个标准的时候为了方便和效率肯定会在内部对其命名。如下表所示：

指令集架构	**处理器家族**
ARMv1	ARM1
ARMv2	ARM2、ARM3
ARMv3	ARM6、ARM7
ARMv4	StrongARM、ARM7TDMI、ARM9TDMI
ARMv5	ARM7EJ、ARM9E、ARM10E、XScale
ARMv6	ARM11、ARM Cortex-M
ARMv7	ARM Cortex-A、ARM Cortex-M、ARM Cortex-R
ARMv8	Cortex-A35、Cortex-A50系列[18]、Cortex-A70系列、Cortex-X1
ARMv9	Cortex-A510、Cortex-A710、Cortex-X2
表中不难发现 **ARMv7** 是个转折点，**Cortex** 三个应用场景产品正是基于它横空出世，鸿蒙内核源码分析系列篇的 ARM文档基础 《ARM体系架构参考手册》背景为 **ARMv7** ，其提供了关于 ARM 处理器架构和指令集，区分接口，所有的ARM处理器的支持（如指令语义）的实现细节等等，可在QQ群中下载名称为：**ARM体系架构参考手册(ARMv7-A/R).pdf** ，关于指令集的介绍具体的翻看系列篇 **(指令集篇)**。	

- ARM 架构详细历史

Cores [edit]

Main article: List of ARM microarchitectures

v65.xx 鸿蒙内核源码分析（ARM 架构篇）

Architecture	Core bit-width	Cores		Profile	References
		Arm Ltd.	Third-party		
ARMv1	32	ARM1		Classic	[a 1]
ARMv2	32	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[60]	Classic	[a 1]
ARMv3	32	ARM6, ARM7		Classic	[a 2]
ARMv4	32	ARM8	StrongARM, FA526, ZAP Open Source Processor Core	Classic	[a 2] [61]
ARMv4T	32	ARM7TDMI, ARM9TDMI, SecurCore SC100		Classic	[a 2]
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	Classic	
ARMv6	32	ARM11		Classic	
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller	
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300	Apple M7	Microcontroller	
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller	
ARMv8-M	32	ARM Cortex-M23,[62] ARM Cortex-M33[63]		Microcontroller	[64]
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time	
ARMv8-R	32	ARM Cortex-R52		Real-time	[65][66][67]
	64	ARM Cortex-R82		Real-time	
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Scorpion/Krait, PJ4/Sheeva, Apple Swift (A6, A6X)	Application	
ARMv8-A	32	ARM Cortex-A32[68]		Application	
	64/32	ARM Cortex-A35,[69] ARM Cortex-A53, ARM Cortex-A57,[70] ARM Cortex-A72,[71] ARM Cortex-A73[72]	X-Gene, Nvidia Denver 1/2, Cavium ThunderX, AMD K12, Apple Cyclone (A7)/Typhoon (A8, A8X)/Twister (A9, A9X)/Hurricane+Zephyr (A10, A10X), Qualcomm Kryo, Samsung M1/M2 ("Mongoose") /M3 ("Meerkat")	Application	[73][74][75][76][77][78]
	64	ARM Cortex-A34[79]		Application	
ARMv8.1-A	64/32	TBA	Cavium ThunderX2	Application	[80]
ARMv8.2-A	64/32	ARM Cortex-A55,[81] ARM Cortex-A75,[82] ARM Cortex-A76,[83] ARM Cortex-A77, ARM Cortex-A78, ARM Cortex-X1, ARM Neoverse N1	Nvidia Carmel, Samsung M4 ("Cheetah"), Fujitsu A64FX (ARMv8 SVE 512-bit)	Application	[84][85][86]
	64	ARM Cortex-A65, ARM Neoverse E1 with simultaneous multithreading (SMT), ARM Cortex-A65AE[87] (also having e.g. ARMv8.4 Dot Product; made for safety critical tasks such as advanced driver-assistance systems (ADAS))	Apple Monsoon+Mistral (A11) (September 2017)	Application	
ARMv8.3-A	64/32	TBA		Application	
	64	TBA	Apple Vortex+Tempest (A12, A12X, A12Z), Marvell ThunderX3 (v8.3+)[88]	Application	
ARMv8.4-A	64/32	TBA		Application	
	64	TBA	Apple Lightning+Thunder (A13), Apple Firestorm+Icestorm (A14), Apple Firestorm+Icestorm (M1)	Application	
ARMv8.5-A	64/32	TBA		Application	
	64	TBA	Apple Avalanche+Blizzard (A15)	Application	
ARMv8.6-A	64	TBA		Application	
ARMv8.7-A	64	TBA		Application	[89]
ARMv9-A	64	ARM Cortex-A510, ARM Cortex-A710, ARM Cortex-X2, ARM Neoverse N2		Application	[90][91]

八种CPU模式

CPU ARM架构指定了以下的CPU模式。在任何时刻，CPU只可处于某一种模式，但可由于外部事件（中断）或编程方式进行模式切换。具体翻看 **(工作模式篇)** 结合代码详细说明。

- **用户模式** ：仅非特权模式。
- **系统模式** ：仅无需例外进入的特权模式。仅以执行明确写入CPSR的模式位的指令进入。
- **Supervisor (svc) 模式** ：在CPU被重置或者SWI指令被执行时进入的特权模式。
- **Abort 模式** ：预读取中断或数据中断异常发生时进入的特权模式。
- **未定义模式** ：未定义指令异常发生时进入的特权模式。
- **中断模式** ：处理器接受一条IRQ中断时进入的特权模式。
- **快中断模式**： 处理器接受一条IRQ中断时进入的特权模式。
- **Hyp 模式**： armv-7a为cortex-A15处理器提供硬件虚拟化引进的管理模式。

寄存器

寄存器，对于所有CPU模式

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

图表解读

- 寄存器 `R0-R7` 对于所有 CPU 模式都是相同的，它们不会被分块。
- `R8-R12` ， `R8_fiq-R12_fiq` ，其实是不同的寄存器，前缀一样是为了管理和识别方便，在实际代码中当切到**快中断模式**后，使用 `R8` 其实在 CPU 内部用的是 `R8_fiq` 寄存器，此处暂且记下，具体在 **(中断切换篇)** 中结合源码详细说明。
- 对于所有的特权 CPU 模式，除了系统 CPU 模式(与用户模式共用)之外， `R13` 和 `R14` 都是分块的。也就是说，每个因为一个异常（exception）而进入的模式，有其自己的 `R13` 和 `R14` 。这些寄存器通常分别包含堆栈指针和函数调用的返回地址。
  - `R13` 也被指为 SP（Stack Pointer）
  - `R14` 也被指为 LR（Link Register）
  - `R15` 也被指为 PC（Program Counter） 由此也能推出这些特权模式有自己独立的运行栈。具体在 **(寄存器篇)** 结合源码详细说明

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

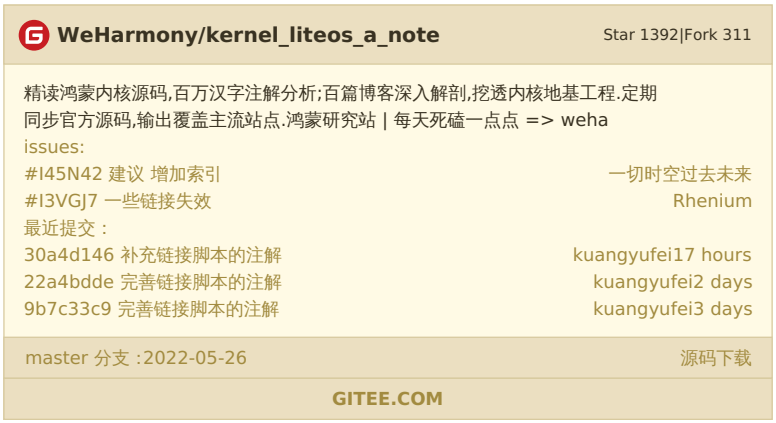
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

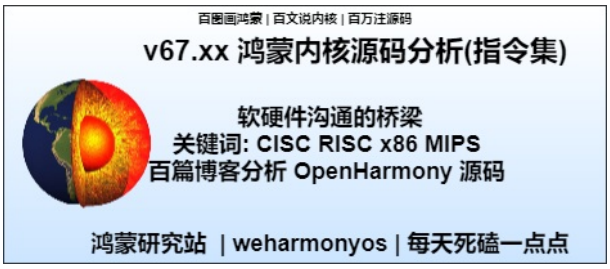


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

67_指令集篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

指令集

- 在计算机中，指示计算机硬件执行某种运算、处理功能的命令称为指令，它是计算机运行的最小的功能单位，而硬件的作用是完成每条指令规定的功能。
- 每一种处理器都有自己可以识别的一整套指令，称为指令集，它是硬件和软件之间沟通的桥梁。是处理器提供给软件控制它的语言，处理器执行指令时，根据不同的指令采取不同的动作，完成不同的功能，既可以改变自己内部的工作状态，也能控制其它外围电路的工作状态。按 CPU 的设计理念，将软硬件沟通方式分成复杂指令集(CISC)和精简指令集(RISC)，不同的时期流行不同的方式，取决于当时的沟通成本/性价比。
- 早期的 CPU 全部是 CISC 架构，受限于软件和编译器技术不发达，设计方向是要用最少的机器语言指令来完成所需的计算任务，需更多的从硬件角度去考虑 CPU 的设计，能用硬件完成的就尽量不用软件去做。其结果导致硬件较复杂，功耗大，成本高。
- 后期因软件和编译器技术快速崛起，相比于硬件呈指数级的增长，原来的指令集架构就不再适用新时代， RISC 的优势就凸显出来，设计方向是降低硬件的大小，减少处理器的功耗，从而硬件成本低廉，省电价格又便宜自然就占据了市场的主导权，尤其在嵌入式领域这种优势就更加的明显。

对比项	CISC（复杂指令集） Complex Instruction Set Computer	RISC（精简指令集） reduced instruction set computer
目的	增强原有指令的功能，设置更为复杂的新指令实现软件功能的硬化	减少指令种类和简化指令功能，提高指令的执行速度
指令系统	复杂、庞大	简单、精简
指令数目	一般大于200条	一般小于100条
指令字长	不固定	固定
可访存指令	不加限制	只有 LOAD/STORE指令
各指令的执行时长	指令间执行时长差距很大	绝大多数在1个周期内完成
各种指令的使用频度	指令间频度差距很大（二八原则）	指令都比较常用
通用寄存器数量	较少	多

中断	机器是在一条指令执行结束后响应中断	机器在一条指令执行的适当地方可响应中断
单元电路（功耗）	包含有丰富的电路单元，因而功能强、面积大、功耗大	包含有较少的单元电路，因而面积小、功耗低
目标代码的执行效率	难以用优化编译生成高效的目标代码程序	采用优化的编译程序，生成代码相对较为高效
控制方式	绝大多数为微程序控制	绝大多数为组合逻辑控制
指令流水线	可以通过一定方式实现	必须实现
微处理器的设计周期	微处理器结构复杂，设计周期长	微处理器结构简单，布局紧凑，设计周期短，且易于采用最新技术
适用场景	适用于【通用型机器】；功能强大，易于/利于实现、处理特殊功能，因有专用指令来完成特定功能	适用于【专用型机器】，因RISC指令系统的确定与特定的应用领域有关；指令规整，性能容易把握，易学易用
被应用的指令集(架构)	X86指令集(架构)	MIPS、ARM、RISC-V、Power-PC[IBM]、SPARC、AArch64(基于ARMv8架构的、分离出来的64位的执行状态) 指令集(架构)

x86

- x86 泛指一系列基于 Intel 8086 且向后兼容的中央处理器指令集架构。最早的 8086 处理器于 1978 年由 Intel 推出，为 16位 微处理器。 由于以 86 作为结尾(但不知为何以 86 数字结尾)，因此其架构被称为 x86 。
- 1985 年，英特尔发布第三代 32 位 CPU 架构，但由于数字并不能作为注册商标，英特尔将其称为 IA-32 ，全名为“Intel Architecture, 32-bit”。同时 CPU 型号到了 intel 80386 ，所以也称为 i386 ，叫法一直延续至今，现在说的 x86 = i386 = IA-32 指代 32 位的架构。
- 2001 年，英特尔原本已经决定在 64 位时代推出新的架构 IA-64 技术的 Itanium 处理器产品线来接替取代 x86 ，但它与 x86 的软件天生不兼容，因此想了各种办法来运行 x86 的软件，但结果效率十分低下，加之处理器本身和软件移植的成本难以控制，因此这个项目最终告吹。
- 2003 年，英特尔的竞争对手 AMD 公司自行把 32 位 x86 （或称为 IA-32 ）拓展为 64 位，并命名为 x86-64 或 Hammer 架构，而后更名为 AMD64 架构，将架构打上了自己的 Flag ，由于 AMD64 处理器产品线首先进入市场，微软先推出了基于 AMD64 的操作系统版本。就不愿意为英特尔 IA-64 再开发另一个 64 位版本，英代尔被迫采纳 AMD64 架构且增加某些新的扩展到他们自己的产品，显然他们不想承认这些指令集是来自它的主要对手，便命名为 EM64T 架构，后正式更名为 Intel 64 也叫 x64 。
- 所以在 PC 机时代，你会很容易看到 x86 ， x86_64 ， x64 ， i386 ， IA32 ， IA64 ， amd64 这些玩意，大概知道表示什么，但很烦就不能简单点吗？其背后是硬件公司竞争对手之间，软硬件公司之间激烈的 PK ，这样的案例比比皆是，屡见不鲜。

MIPS

- MIPS（Microprocessor without Interlocked Pipeline Stages），是一种采取精简指令集（RISC）的指令集架构（ISA），由美国MIPS计算机系统公司开发，现为美普思科技。 MIPS 广泛被使用在许多电子产品、网络设备、个人娱乐设备与商业设备上。最早的 MIPS 架构是 32 位，最新的版本已经变成 64 位。商业市场主要竞争对手为 ARM 与 RISC-V 。
- 在一些大学和技术学校中计算机架构的课程上，学生们通常会学习 MIPS 架构。这个架构极大地影响了后来的精简指令集架构，如 Alpha 。
- MIPS本来就没有一个较为统一的生态。命运多舛，被转卖多次、很多相关专利已经被卖掉了、碎片化问题非常严重。 2021 年 3 月， MIPS 宣布 MIPS 架构的开发已经结束，因为该公司正在向 RISC-V 过渡。

arm

- 最著名的是 **armv7指令集** 支持两种指令集，ARM 指令集和 Thumb 指令集，从功耗上来说，thumb 指令集的功耗要低于 arm 指令集。
- ARM公司的商业模式: IP(Intellectual Property,知识产权)授权模式(ARM架构授权、IP核授权、使用级授权);
- 主要业务范围: ARM指令集、ARM微架构、ARM芯片(不自己造)

RISC-V

- RISC-V（发音为“risk-five”）号称芯片设计领域的 linux ，是一个基于精简指令集（RISC）原则的开源指令集架构（ISA），目标是成为一个通用的指令集架构（ISA），RISC-V的不同寻常不仅在于它是一个最近诞生的指令集架构（它诞生于最近十年，而大多数其他指令集都诞生于20世纪70到80年代），而且在于它是一个开源的指令集架构。与几乎所有的旧架构不同，它的未来不受任何单一公司的浮沉或一时兴起的决定的影响（这一点让许多过去的指令集架构都遭了殃）。它属于一个开放的，非营利性质的基金会。RISC-V基金会的目标是保持RISC-V的稳定性，仅仅出于技术原因缓慢而谨慎地发展它，并力图让它之于硬件如同Linux之于操作系统一样受欢迎。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

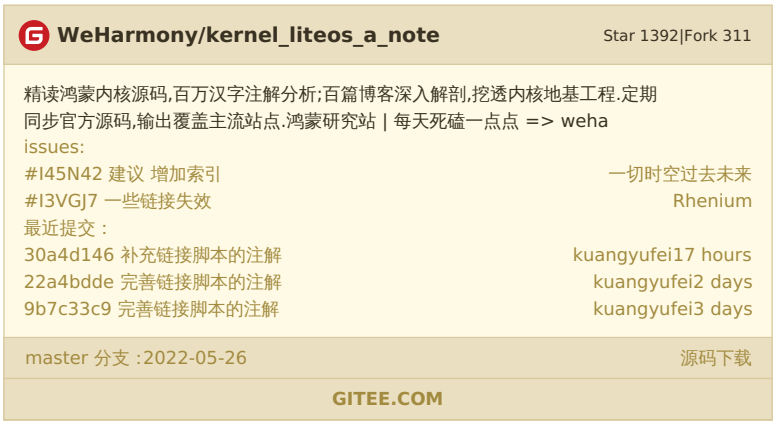
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

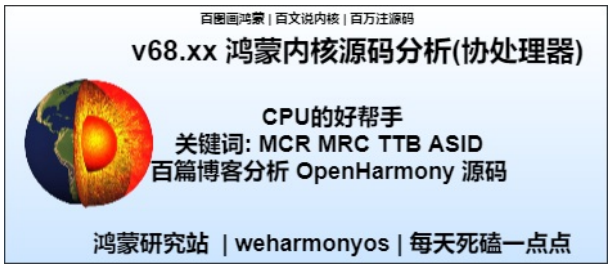


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

68_协处理器篇

本篇关键词：CP15 、MCR、MRC、ASID、MMU



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

本篇很重要，对 `CP15` 协处理所有 `16` 个寄存器一一介绍，可能是全网介绍 `CP15` 最全面的一篇，鸿蒙内核的汇编部分(尤其开机启动)中会使用，熟练掌握后看汇编代码将如虎添翼。

协处理器

协处理器 (co-processor) 顾名思义是协助主处理器完成工作，例如浮点、图像、音频处理这一类外围工作。角色相当于老板的助理/秘书，咱皇上身边的人，专干些咱皇上又不好出面的脏活累活，您可别小看了这个角色，权利不大但能力大，是能通天的人，而且老板越大，身边这样的人还不止一个。

在 `arm` 的协处理器设计中，最多可以支持 `16` 个协处理器，通常被命名为 `cp0` ~ `cp15` ，本篇主要说第 `16` 号协处理器 `cp15`

CP15

关于 `cp15` 详细介绍见于 **<< ARM体系架构参考手册(ARMv7-A/R).pdf >>** 的 **B3.17**。 `cp15` 一共有 `16` 个 `32` 位的寄存器，其编号为 `C0 ~ C15` ，用来控制 cache 、 TCM 和存储器管理。 `cp15` 寄存器都是复合功能寄存器，不同功能对应不同的内存实体，全由访问指令的参数来决定，对于 `armv7` 架构而言， `A` 系列和 `R` 系列是统一设计的， `A` 系列带有 `MMU` 相关的控制，而 `R` 系列带有 `MPU` 相关控制，针对不同的功能需要做区分，同时又因为协处理器 `cp15` 只支持 `16` 个寄存器，而需要支持的功能较多，所以通过同一寄存器不同功能的方式来满足需求。

mcr | mrc 指令

armv7 中对于协处理器的访问， `CP15` 的寄存器只能被 `MRC` 和 `MCR` （Move to Coprocessor from ARM Register ）指令访问。 `MCR` 表示将 `arm` 核心寄存器中的值的写到 `cp15` 寄存器中， `MRC` 从 `cp15` 寄存器中读到 `arm` 核心寄存器中，大部分指令都需要在 `PL1` 以及更高的特权级下才能正常执行，这是因为 `cp15` 协处理器大多都涉及到系统和内存的设置， `user` 模式没有操作权限， `user` 模式仅能访问 `cp15` 中有限的几个寄存器比如：ISB、DSB、DMB、TPIDRURW、TPIDRURO 寄存器。

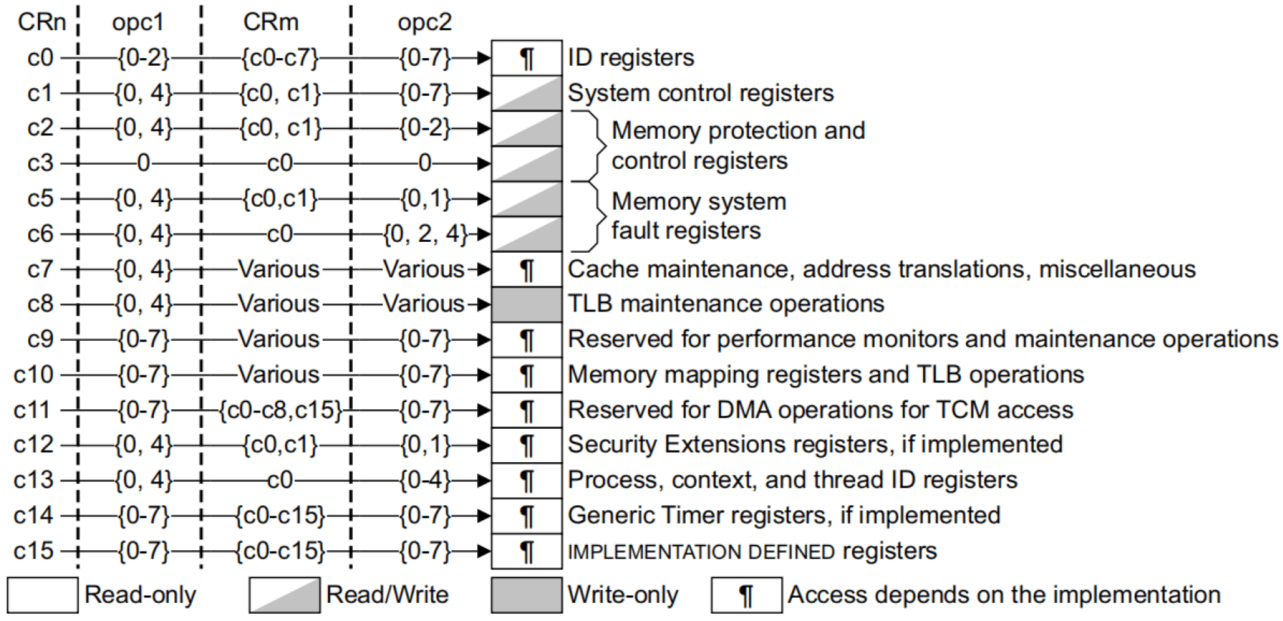
```
从 `cp**` 寄存器中读到 `arm` 核心寄存器中
MRC<cond> <coproc>, <opc1>, <Rt>, <CRn>, <CRm>{, <opc2>}
```

- **cond** : 指令后缀，表示条件执行，关于条件执行可以参考 arm状态寄存器
- **coproc** :协处理器的名称，cp0～cp15 分别对应名称 p0～p15
- **opc1** :对于 cp15 而言，这一个参数一般为0。
- **Rt** :arm 的通用寄存器
- **CRn** :与 arm 核心寄存器交换数据的核心寄存器名，c0～c15
- **CRm** :需要额外操作的协处理器的寄存器名，c0～c15，针对多种功能的 cp15 寄存器，需要使用 CRm 和 opc2 来确定 CRn 对应哪个寄存器实

体。

- opc2 ：可选，与 CRm搭配使用，同样是决定多功能寄存器中指定实体。

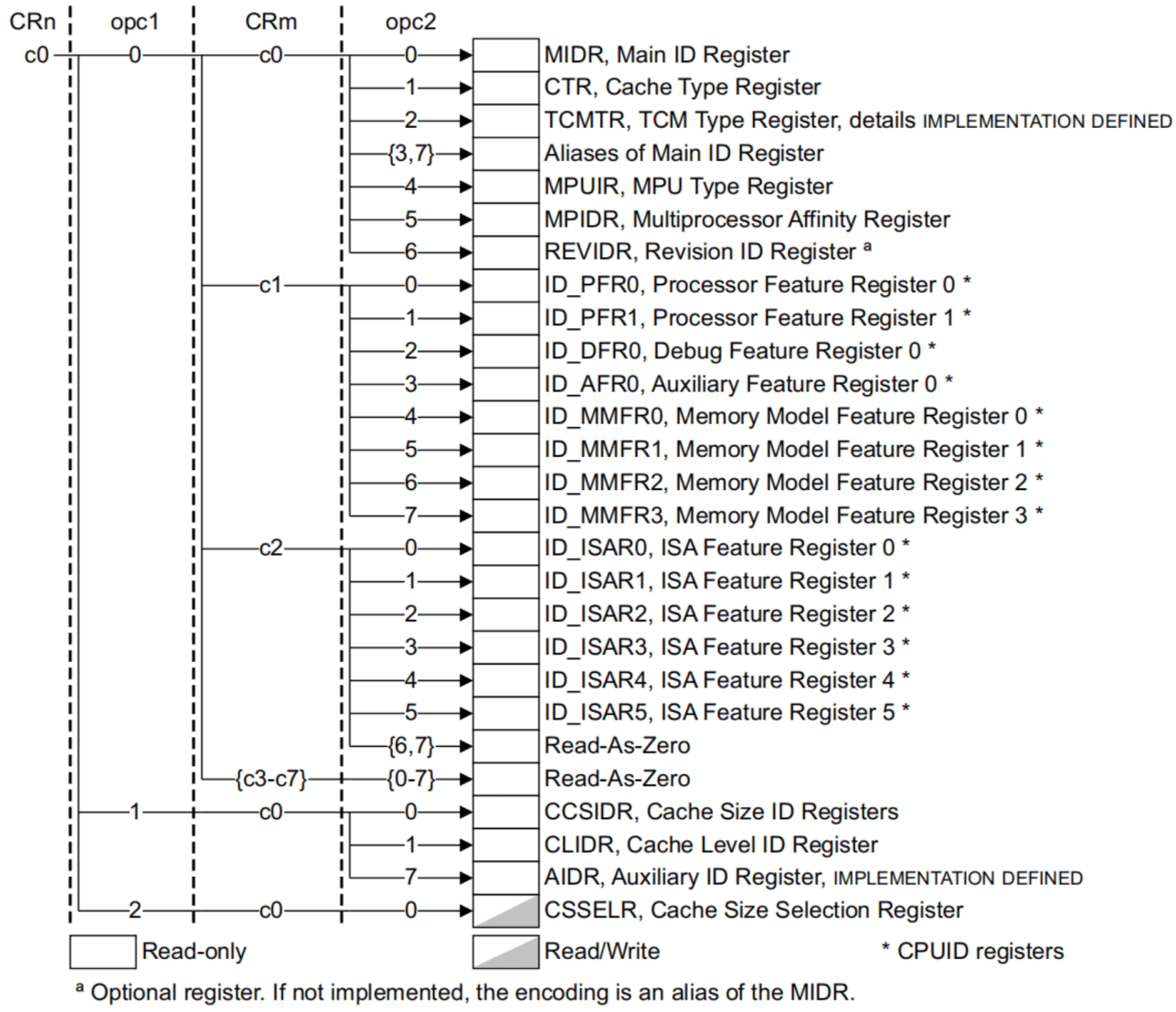
啥玩意，太抽象没看懂，后面直接上内核代码就懂了，先看16个寄存器的功能介绍表



c0 寄存器

c0 寄存器提供处理器和特征识别 ，内核宏定义为，可参考图理解

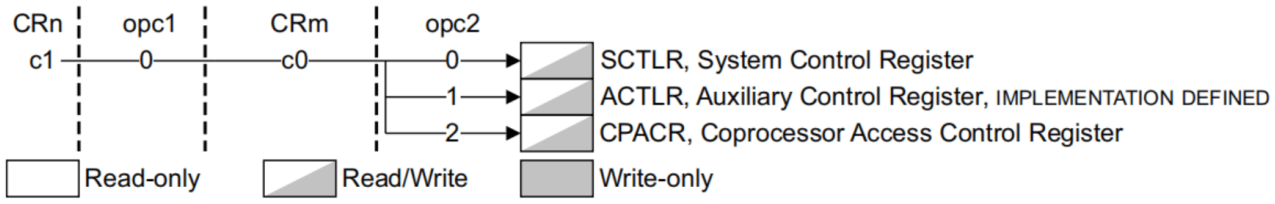
```
/*!
 * Identification registers (c0) | c0 - 身份寄存器
 */
#define MIDR              CP15_REG(c0, 0, c0, 0)    /*! Main ID Register | 主ID寄存器 */
#define MPIDR             CP15_REG(c0, 0, c0, 5)    /*! Multiprocessor Affinity Register | 多处理器关联寄存器给每个CPU制定一个逻辑地址*/
#define CCSIDR            CP15_REG(c0, 1, c0, 0)    /*! Cache Size ID Registers | 缓存大小ID寄存器*/
#define CLIDR             CP15_REG(c0, 1, c0, 1)    /*! Cache Level ID Register | 缓存登记ID寄存器*/
#define VPIDR             CP15_REG(c0, 4, c0, 0)    /*! Virtualization Processor ID Register | 虚拟化处理器ID寄存器*/
#define VMPIDR            CP15_REG(c0, 4, c0, 5)    /*! Virtualization Multiprocessor ID Register | 虚拟化多处理器ID寄存器*/
```



c1 寄存器

c1 为系统控制寄存器

```
/*!
 * System control registers (c1) | c1 - 系统控制寄存器 各种控制位（可读写）
 */
#define SCTLR           CP15_REG(c1, 0, c0, 0)    /*! System Control Register | 系统控制寄存器*/
#define ACTLR           CP15_REG(c1, 0, c0, 1)    /*! Auxiliary Control Register | 辅助控制寄存器*/
#define CPACR           CP15_REG(c1, 0, c0, 2)    /*! Coprocessor Access Control Register | 协处理器访问控制寄存器*/
```




```
/// 读取CP15的系统控制寄存器到 R0寄存器
STATIC INLINE UINT32 OsArmReadSctlr(VOID)
```

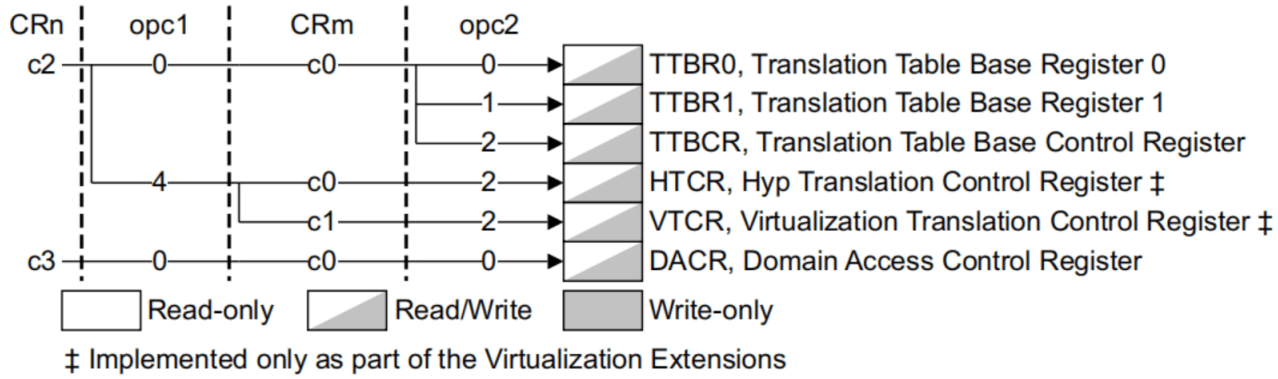
```
{
    UINT32 val;
    __asm__ volatile("mrc p15, 0, %0, c1,c0,0" : "=r"(val));
    return val;
}
/// R0寄存器写入CP15的系统控制寄存器
STATIC INLINE VOID OsArmWriteSctlr(UINT32 val)
{
    __asm__ volatile("mcr p15, 0, %0, c1,c0,0" ::"r"(val));
    __asm__ volatile("isb" ::: "memory");
}
```

解读

- 从图中找到 `c1-0-c0-0` 行，后边的备注是 **SCTLR, System Control Register** 系统控制寄存器，其操作模式是支持 **Read/Write**
- `%0` 表示 **r0** 寄存器，注意这个寄存器是 `CPU` 的寄存器，: "=r"(val) 意思向编译器声明，会修改 `R0` 寄存器的值，改之前提前打好招呼，都是绅士文明人。其实编译器的功能是非常强大的，不仅仅是大家普遍认为的只是编译代码的工具而已。 `OsArmReadSctlr` 的含义就是读取CP15的系统控制寄存器到R0寄存器。
- `volatile` 的意思还告诉编译器，不要去优化这段代码，原封不动的生成目标指令。
- "isb" ::: "memory" 还是告诉编译器内存的内容要被更改了，需要无效所有 `Cache` ，并访问实际的内容，而不是 `Cache！`
- **CRn | CRm | opc2** 是一套组合拳， `c7-0-c10-4` `c7-0-c10-5` 都表示不同的功能含义

c2、c3 寄存器

```
/*!
 * Memory protection and control registers (c2 & c3) | c2 - 传说中的TTB寄存器，主要是给MMU使用 c3 - 域访问控制位
 */
#define TTBR0           CP15_REG(c2, 0, c0, 0)    /*! Translation Table Base Register 0 | 转换表基地址寄存器0*/
#define TTBR1           CP15_REG(c2, 0, c0, 1)    /*! Translation Table Base Register 1 | 转换表基地址寄存器1*/
#define TTBCR           CP15_REG(c2, 0, c0, 2)    /*! Translation Table Base Control Register | 转换表基地址控制寄存器*/
#define DACR            CP15_REG(c3, 0, c0, 0)    /*! Domain Access Control Register | 域访问控制寄存器*/
```



看段代码

```
STATIC INLINE UINT32 OsArmReadTtbr0(VOID)
{
    UINT32 val;
    __asm__ volatile("mrc p15, 0, %0, c2,c0,0" : "=r"(val));
    return val;
}
STATIC INLINE VOID OsArmWriteTtbr0(UINT32 val)
{
    __asm__ volatile("mcr p15, 0, %0, c2,c0,0" ::"r"(val));
    __asm__ volatile("isb" ::: "memory");
}
STATIC INLINE UINT32 OsArmReadTtbr1(VOID)
{
    UINT32 val;
    __asm__ volatile("mrc p15, 0, %0, c2,c0,1" : "=r"(val));
    return val;
```

```
}
STATIC INLINE VOID OsArmWriteTtbr1(UINT32 val)
{
    __asm__ volatile("mcr p15, 0, %0, c2,c0,1" ::"r"(val));
    __asm__ volatile("isb" ::: "memory");
}
```

c2 寄存器负责存页表的基地址，即一级映射描述符表的基地址。还记得吗？每个进程的页表都是独立的！c2 值一变，当前使用的页表就发生了变化，页表变化意味着虚拟地址和物理地址的映射关系发生了变化。那么什么情况下会修改里面的值呢？很容易想到只有在进程切换时发生的 mmu 上下文切换，直接看代码吧！

```
/// mmu 上下文切换
VOID LOS_ArchMmuContextSwitch(LosArchMmu *archMmu)
{
    UINT32 ttbr;
    UINT32 ttbcr = OsArmReadTtbcr();//读取TTB寄存器的状态值
    if (archMmu) {
        ttbr = MMU_TTBRx_FLAGS | (archMmu->physTtb);//进程TTB物理地址值
        /* enable TTBR0 */
        ttbcr &= ~MMU_DESCRIPTOR_TTBCR_PD0;//使能TTBR0
    } else {
        ttbr = 0;
        /* disable TTBR0 */
        ttbcr |= MMU_DESCRIPTOR_TTBCR_PD0;
    }
#ifdef LOSCFG_KERNEL_VM
    /* from armv7a arm B3.10.4, we should do synchronization changes of ASID and TTBR. */
    OsArmWriteContextidr(LOS_GetKVmSpace()->archMmu.asid);//这里先把asid切到内核空间的ID
    ISB; //指令必须同步 ，清楚流水线中未执行指令
#endif
    OsArmWriteTtbr0(ttbr);//通过r0寄存器将进程页面基址写入TTB
    ISB; //指令必须同步
    OsArmWriteTtbcr(ttbcr);//写入TTB状态位
    ISB; //指令必须同步
#ifdef LOSCFG_KERNEL_VM
    if (archMmu) {
        OsArmWriteContextidr(archMmu->asid);//通过R0寄存器写入进程标识符至C13寄存器
        ISB;
    }
#endif
}
```

至于具体内核哪些地方会触发到 mmu 的切换，可前往翻看 **（进程切换篇）**

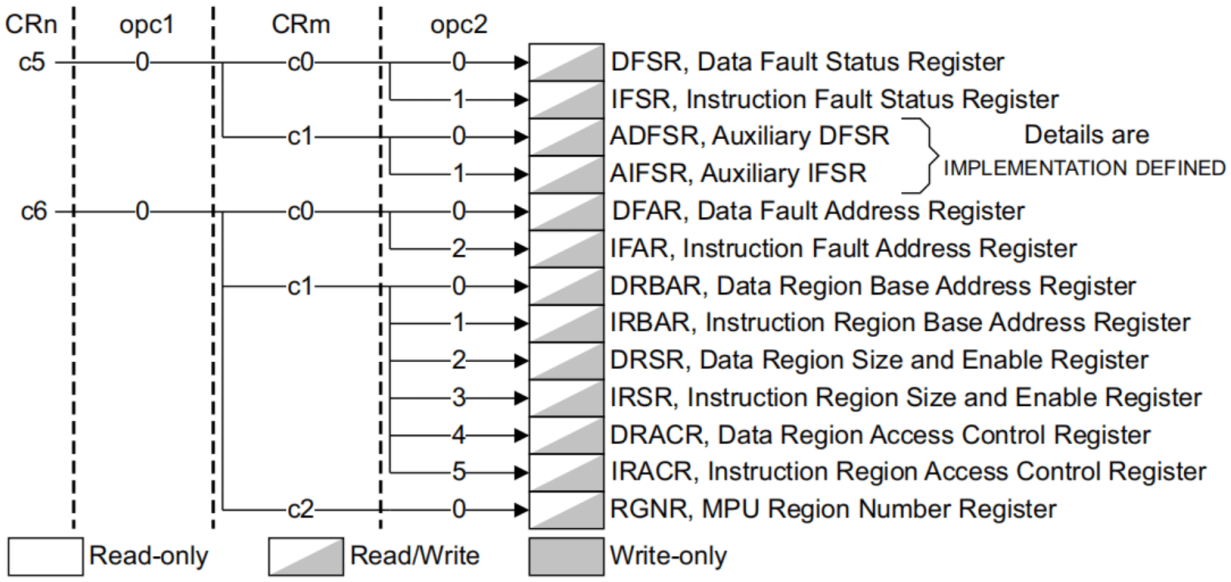
c4 寄存器

c4 没有用于任何 **ARMv7** 实现，这么不待见4，难道原因跟中国人一样觉得数字不吉利 ，但老师教的老外是不喜欢 13 啊 ， 但c13确很重要

c5 c6 寄存器

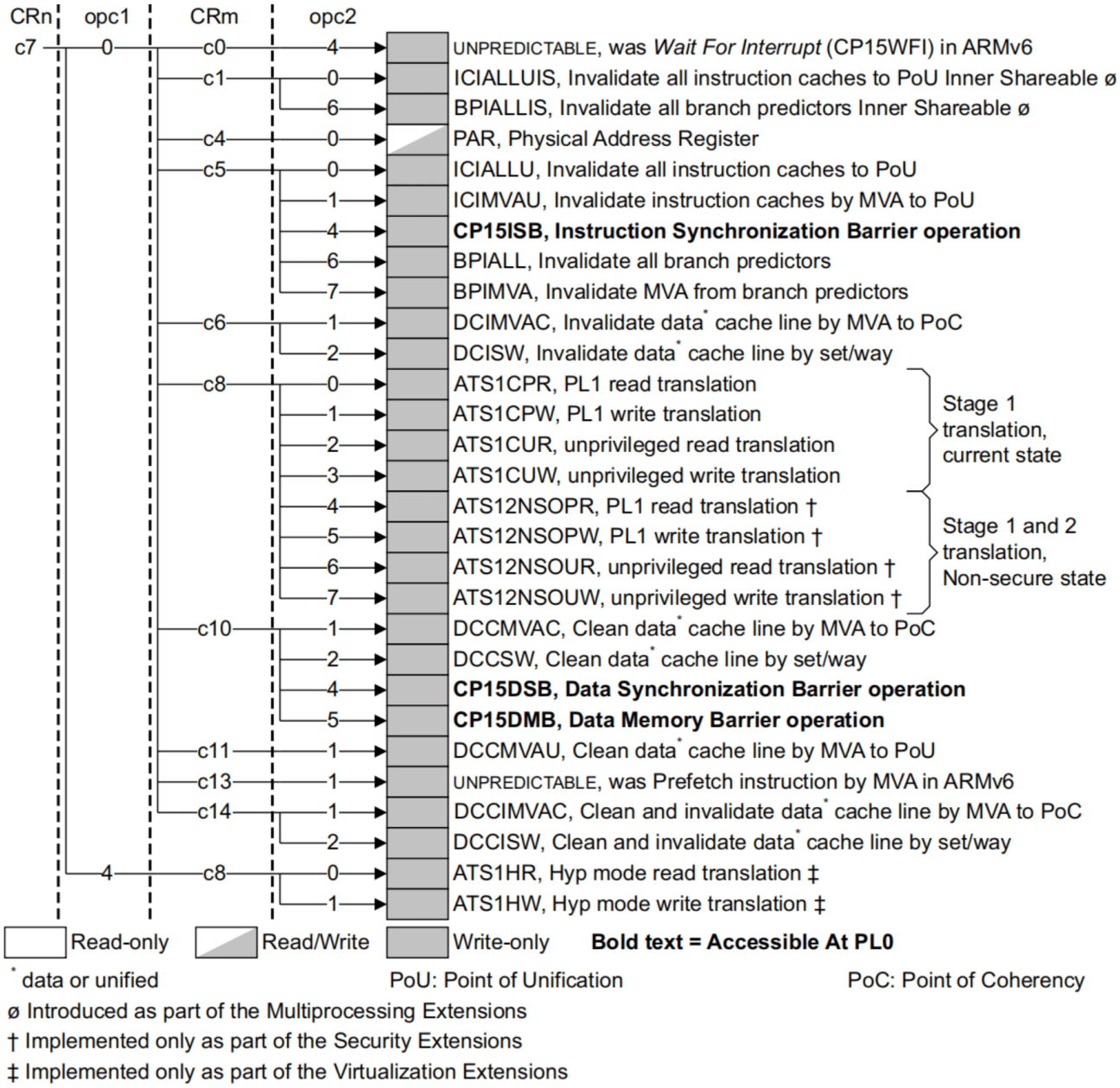
c5和c6寄存器提供内存系统故障报告。此外，c6还提供了MPU区域寄存器。这一类寄存器在软件排错时可以提供非常大的帮助，比如通过 DFSR(数据状态寄存器)、IFSR(指令状态寄存器) 的 status bits 可以查到系统 abort 类型，内核中的缺页异常就是通过该寄存器传递异常地址，从而分配页面的。

```
/*!
 * Memory system fault registers (c5 & c6) | c5 - 内存失效状态 c6 - 内存失效地址
 */
#define DFSR            CP15_REG(c5, 0, c0, 0)    /*! Data Fault Status Register | 数据故障状态寄存器 */
#define IFSR            CP15_REG(c5, 0, c0, 1)    /*! Instruction Fault Status Register | 指令故障状态寄存器*/
#define DFAR            CP15_REG(c6, 0, c0, 0)    /*! Data Fault Address Register | 数据故障地址寄存器*/
#define IFAR            CP15_REG(c6, 0, c0, 2)    /*! Instruction Fault Address Register | 指令错误地址寄存器*/
```



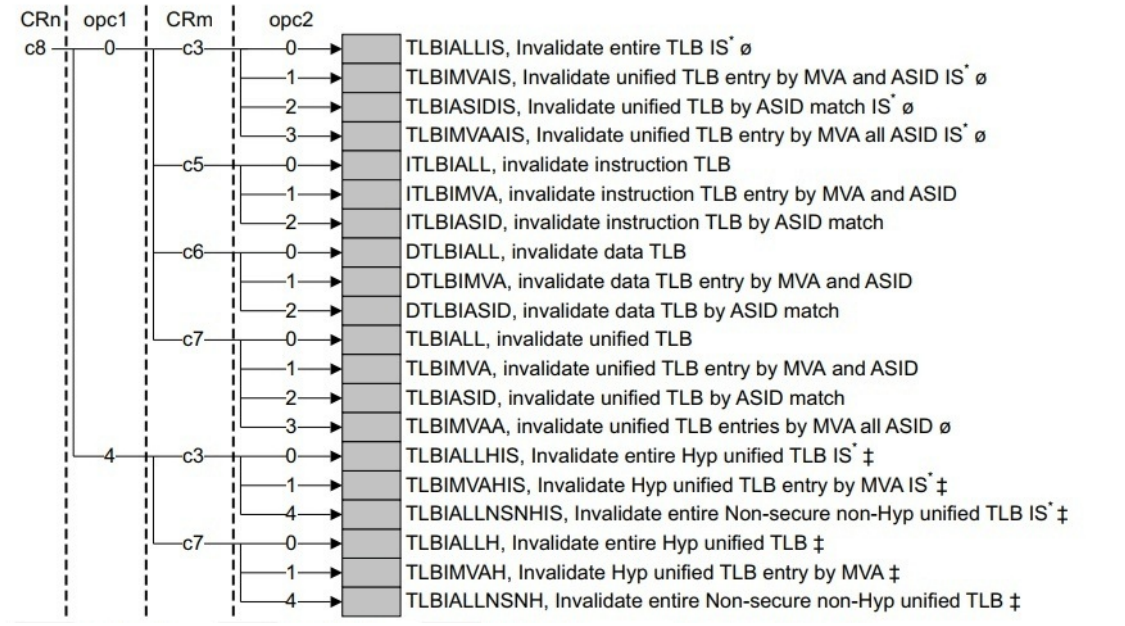
c7 寄存器

c7寄存器提供高速缓存维护操作和内存屏障操作。

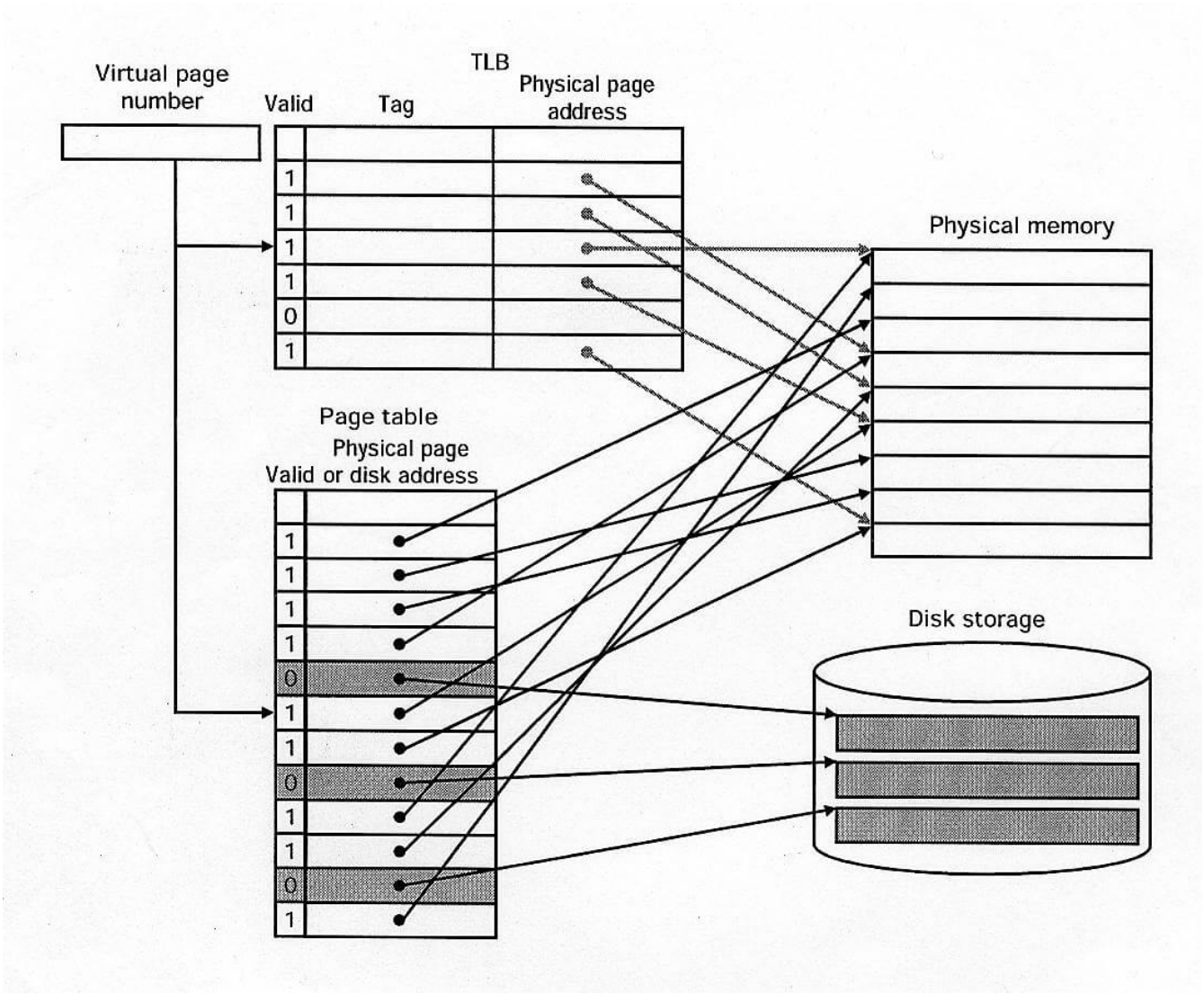


c8 寄存器

c8 寄存器提供 TLB 维护功能



TLB 是硬件上的一个 cache ，因为页表一般都很大，并且存放在内存中，所以处理器引入 MMU 后，读取指令、数据需要访问两次内存：首先通过查询页表得到物理地址，然后访问该物理地址读取指令、数据。为了减少因为MMU导致的处理器性能下降，引入了 TLB ，可翻译为“地址转换后援缓冲器”，也可简称为“快表”。简单地说， TLB 就是页表的 Cache ，其中存储了当前最可能被访问到的页表项，其内容是部分页表项的一个副本。只有在 TLB 无法完成地址翻译任务时，才会到内存中查询页表，这样就减少了页表查询导致的处理器性能下降。详细看

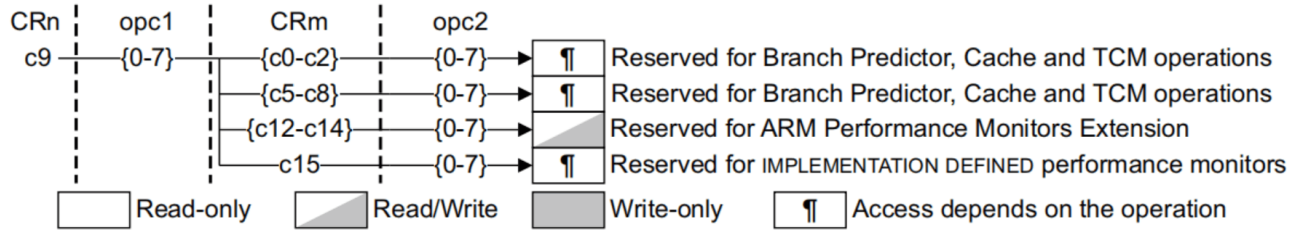


照着图说吧，步骤是这样的。

- 图中的 page table 的基地址就是上面 TTB 寄存器值，整个 page table 非常大，有多大接下来会讲，所以只能存在内存里， TTB 中只是存一个开始位置而已。
- 虚拟地址是程序的地址逻辑地址，也就是喂给 CPU 的地址，必须经过 MMU 的转换后变成物理内存才能取到真正的指令和数据。
- TLB 是 page table 的迷你版， MMU 先从 TLB 里找物理页，找不到了再从 page table 中找，从 page table 中找到后会放入 TLB 中，注意这一步非常非常的关键。因为 page table 是属于进程的会有很多个，而 TLB 只有一个，不放入就会出现多个进程的 page table 都映射到了同一个物理页框而不自知。一个物理页同时只能被一个 page table 所映射。但除了 TLB 的唯一性外，要做到不错乱还需要了一个东西，就是进程在映射层面的唯一标识符 - asid ，具体可前往翻看 **（进程切换篇）** 有详细说明。

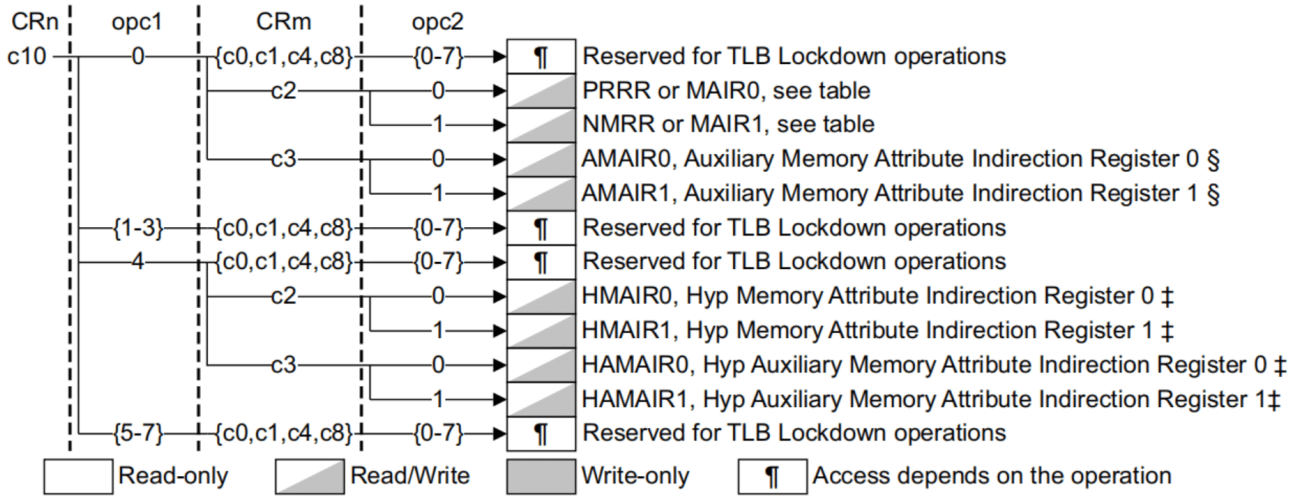
c9 寄存器

c9 寄存器主要为 cache、分之预测 和 tcm 保留功能，这些保留功能由处理的实现决定



c10 寄存器

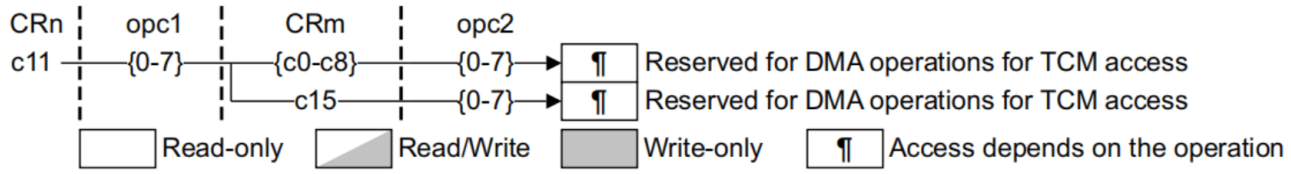
c10 寄存器主要提供内存重映射和 TLB 控制功能



Without Large Physical Address Extension	With Large Physical Address Extension
PRRR, Primary Region Remap Register	MAIR0, Memory Attribute Indirection Register 0
NMRR, Normal Memory Remap Register	MAIR1, Memory Attribute Indirection Register 1

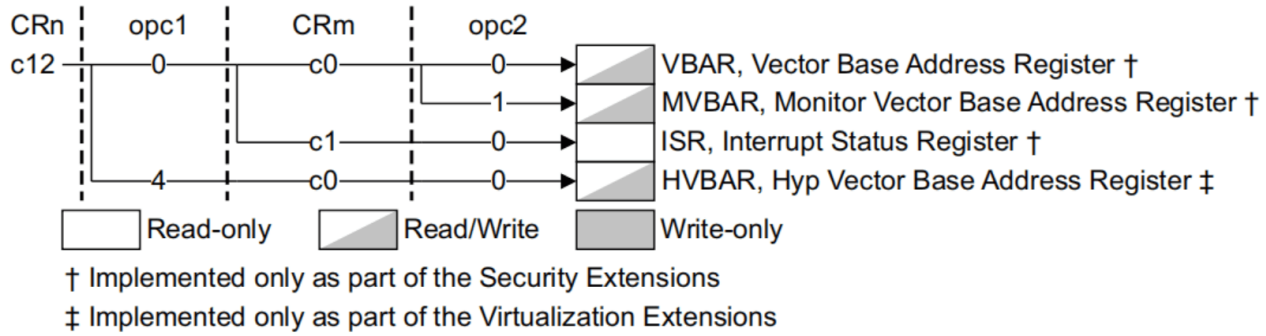
c11 寄存器

c11 寄存器主要提供 TCM 和 DMA 的保留功能，这些保留功能由处理的实现决定



c12 寄存器

c12 安全扩展寄存器

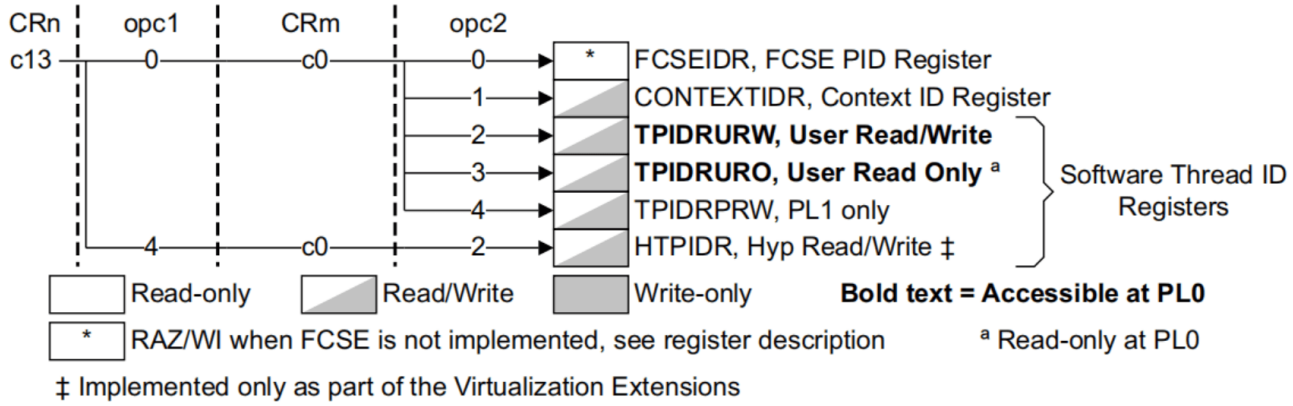


c13 寄存器

c13 寄存器提供进程、上下文以及线程ID处理功能

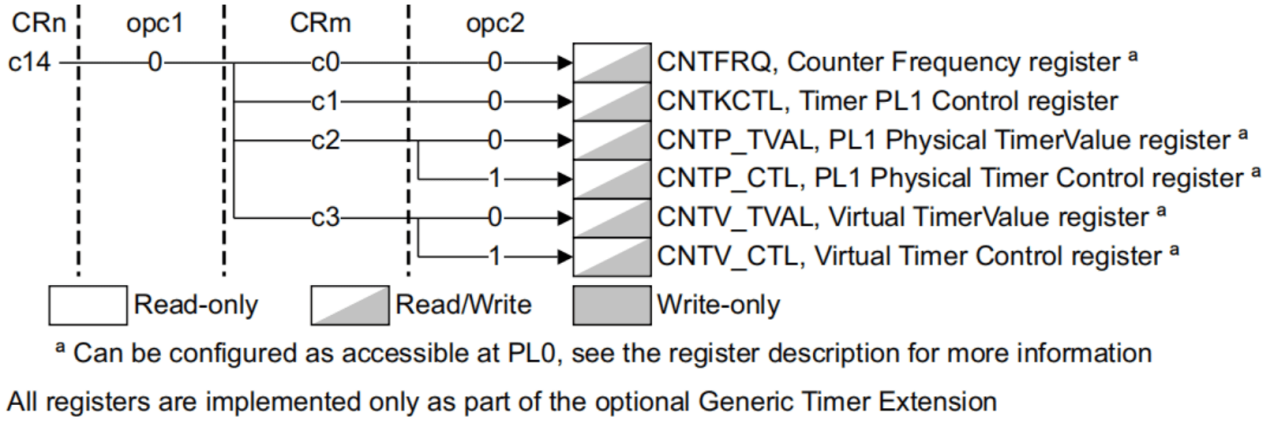
```
/*!
 * Process, context and thread ID registers (c13) | c13 - 进程标识符
 */
#define FCSEIDR            CP15_REG(c13, 0, c0, 0)    /*! FCSE Process ID Register | FCSE（Fast Context Switch Extension，快速上下文切换）进程ID寄
#define CONTEXTIDR         CP15_REG(c13, 0, c0, 1)    /*! Context ID Register | 上下文ID寄存器*/
```

```
#define TPIDRURW        CP15_REG(c13, 0, c0, 2)   /*! User Read/Write Thread ID Register | 用户读/写线程ID寄存器*/
#define TPIDRURO        CP15_REG(c13, 0, c0, 3)   /*! User Read-Only Thread ID Register | 用户只读写线程ID寄存器*/
#define TPIDRPRW        CP15_REG(c13, 0, c0, 4)   /*! PL1 only Thread ID Register | 仅PL1线程ID寄存器*/
```



c14 寄存器

c14 寄存器提供通用定时器扩展的保留功能



c15 寄存器

ARMv7 保留 c15 用于实现定义的目的，并且不对 c15 编码的使用施加任何限制。 意思就是可以将他当通用寄存器来使用 语法: `c15 0-7 c0-c15 0-7`

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

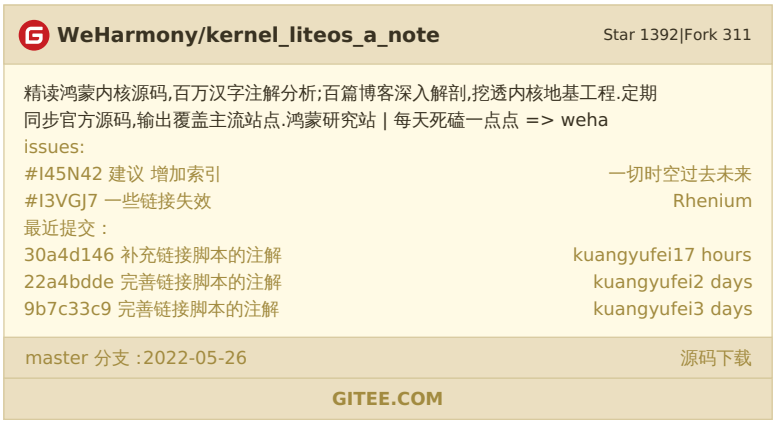
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

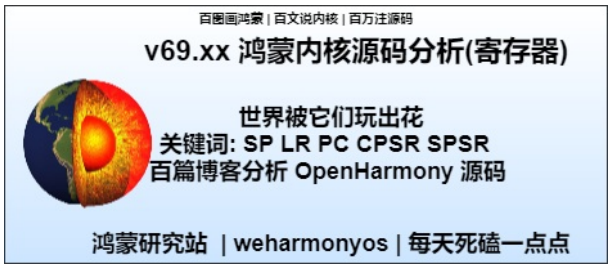


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

69_工作模式篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

本篇需结合 << ARM体系架构参考手册(ARMv7-A/R).pdf >> 阅读。

本篇说清楚CPU的工作模式

工作模式(Working mode) 也叫操作模式（Operating mode）又叫处理器模式（Processor mode），是 CPU 运行的重要参数，决定着处理器的工作方式，比如如何裁决特权级别和报告异常等。 系列篇为方便理解，统一叫工作模式，CPU的工作模式。

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 其他篇。

正如一个互联网项目的后台管理系统有权限管理一样，CPU工作是否也有权限(模式)？ 一个成熟的软硬件架构，肯定会有这些设计，只是大部分人不知道，也不需要知道，老百姓就干好老百姓的活就行了，有工作能吃饱饭就知足了，宫的事你管那么多干嘛，你也管不了。

应用程序就只关注应用功能，业务逻辑相关的部分就行了，底层实现对应用层屏蔽的越干净系统设计的就越优良。

但鸿蒙内核源码分析系列篇的定位就是要把整个底层解剖，全部掰开，看看宫里究竟发生了么事。从本篇开始要接触大量的汇编的代码，将鸿蒙内核的每段汇编代码一一说明白。如此才能知道最开始的开始发生了什么，最后的最后又发生了什么。

七种模式

本篇需结合 << ARM体系架构参考手册(ARMv7-A/R).pdf >> 阅读。

在ARM体系中，CPU很像有七个老婆的韦小宝，工作在以下七种模式中:

The ARM720T supports seven modes of operation as listed in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution state
FIQ	fiq	Designed to support a data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

- **用户模式（usr）**：该模式是用户程序的工作模式，它运行在操作系统的用户态，它没有权限去操作其它硬件资源，只能执行处理自己的数据，也不能切换到其它模式下，要想访问硬件资源或切换到其它模式只能通过软中断或产生异常。
- **快速中断模式（fiq）**：快速中断模式是相对一般中断模式而言的，用来处理高优先级中断的模式，处理对时间要求比较紧急的中断请求，主要用于高速数据传输及通道处理中。
- **普通中断模式（irq）**：一般中断模式也叫普通中断模式，用于处理一般的中断请求，通常在硬件产生中断信号之后自动进入该模式，该模式可以自由访问系统硬件资源。
- **管理模式（svc）**：操作系统保护模式，CPU上电复位和当应用程序执行 SVC 指令调用系统服务时也会进入此模式，操作系统内核的普通代码通常工作在这个模式下。
- **终止模式（abt）**：当数据或指令预取终止时进入该模式，中止模式用于支持虚拟内存或存储器保护，当用户程序访问非法地址，没有权限读取的内存地址时，会进入该模式，
- **系统模式（sys）**：供操作系统使用的高特权用户模式，与用户模式类似，但具有可以直接切换到其他模式等特权，用户模式与系统模式两者使用相同的寄存器，都没有SPSR（Saved Program Statement Register，已保存程序状态寄存器），但系统模式比用户模式有更高的权限，可以访问所有系统资源。
- **未定义模式（und）**：未定义模式用于支持硬件协处理器的软件仿真，CPU在指令的译码阶段不能识别该指令操作时，会进入未定义模式。

除用户模式外，其余6种工作模式都属于特权模式

- 特权模式中除了系统模式以外的其余5种模式称为异常模式
- 大多数程序运行于用户模式
- 进入特权模式是为了处理中断、异常、或者访问被保护的系统资源
- 硬件权限级别：系统模式 > 异常模式 > 用户模式
- 快中断(fiq)与慢中断(irq)区别：快中断处理时禁止中断

每种模式都有自己独立的入口和独立的运行栈空间。 **系列篇之CPU篇** 已介绍过只要提供了入口函数和运行空间，CPU就可以干活了。入口函数解决了指令来源问题，运行空间解决了指令的运行场地问题。 而且在多核情况下，每个CPU核的每种特权模式都有自己独立的栈空间。注意是特权模式下的栈空间，用户模式的栈空间是由用户(应用)程序提供的。

如何让这七种模式能流畅的跑起来呢？ 至少需要以下解决三个基本问题。

- 栈空间是怎么申请的？申请了多大？
- 被切换中的模式代码放在哪里？谁来安排它们放在哪里？
- 模式之间是怎么切换的？状态怎么保存？

本篇代码来源于鸿蒙内核源码之reset_vector_mp.S，点击查看 这个汇编文件大概 500多行，非常重要，本篇受限于篇幅只列出一小部分，说清楚以上三个问题。系列其余篇中将详细说明每段汇编代码的作用和实现，可前往查阅。

1.异常模式栈空间怎么申请？

鸿蒙是如何给异常模式申请栈空间的

```
#define CORE_NUM              LOSCFG_KERNEL_SMP_CORE_NUM //CPU 核数
#ifdef LOSCFG_GDB
#define OS_EXC_UNDEF_STACK_SIZE  512
#define OS_EXC_ABT_STACK_SIZE    512
#else
#define OS_EXC_UNDEF_STACK_SIZE  40
#define OS_EXC_ABT_STACK_SIZE    40
#endif
#define OS_EXC_FIQ_STACK_SIZE    64
#define OS_EXC_IRQ_STACK_SIZE    64
#define OS_EXC_SVC_STACK_SIZE    0x2000 //8K
#define OS_EXC_STACK_SIZE       0x1000 //4K

@六种特权模式申请对应的栈运行空间
__undef_stack:
    .space OS_EXC_UNDEF_STACK_SIZE * CORE_NUM
__undef_stack_top:

__abt_stack:
    .space OS_EXC_ABT_STACK_SIZE * CORE_NUM
__abt_stack_top:

__irq_stack:
    .space OS_EXC_IRQ_STACK_SIZE * CORE_NUM
__irq_stack_top:

__fiq_stack:
    .space OS_EXC_FIQ_STACK_SIZE * CORE_NUM
__fiq_stack_top:

__svc_stack:
    .space OS_EXC_SVC_STACK_SIZE * CORE_NUM
__svc_stack_top:

__exc_stack:
    .space OS_EXC_STACK_SIZE * CORE_NUM
__exc_stack_top:
```

代码解读

- 六种异常模式都有自己独立的栈空间
- 每种模式的 OS_EXC_***_STACK_SIZE 栈大小都不一样，最大是管理模式（svc）8K，最小的只有40个字节。 svc模式为什么要这么大呢？ 因为开机代码和系统调用代码的运行都在管理模式，系统调用的函数实现往往较复杂，最大不能超过8K。 例如:某个系统调用中定义一个8K的局部变量，内核肯定立马闪蹦。因为栈将溢出，处理异常的程序出现了异常，后面就再也没人兜底了，只能是死局。
- 鸿蒙是支持多核处理的， CORE_NUM 表明，每个CPU核的每种异常模式都有自己的独立栈空间。注意理解这个是理解内核代码的基础。否则会一头雾水。

2。异常模式入口地址在哪？

本篇需结合 << ARM体系架构参考手册(ARMv7-A/R).pdf >> 阅读。

Table 2-4 Exception vector addresses

High address	Low address	Exception	Mode on entry
0xFFFF0000	0x00000000	Reset	Supervisor
0xFFFF0004	0x00000004	Undefined instruction	Undefined
0xFFFF0008	0x00000008	Software interrupt	Supervisor
0xFFFF000C	0x0000000C	Abort (prefetch)	Abort
0xFFFF0010	0x00000010	Abort (data)	Abort
0xFFFF0014	0x00000014	Reserved	Reserved
0xFFFF0018	0x00000018	IRQ	IRQ
0xFFFF001C	0x0000001C	FIQ	FIQ

这就是一切一切的开始，指定所有异常模式的入口地址表，这就是规定，没得商量的。在低地址情况下。开机代码就是放在 0x00000000的位置，触发开机键后，硬件将PC寄存器置为0x00000000，开始了万里长征的第一步。在系统运行过程中就这么来回跳。

```
b  reset_vector          @开机代码
b  _osExceptUndefInstrHdl  @异常处理之CPU碰到不认识的指令
b  _osExceptSwiHdl   @异常处理之:软中断
b  _osExceptPrefetchAbortHdl @异常处理之:取指异常
b  _osExceptDataAbortHdl  @异常处理之:数据异常
b  _osExceptAddrAbortHdl  @异常处理之:地址异常
b  OsIrqHandler   @异常处理之:硬中断
b  _osExceptFiqHdl   @异常处理之:快中断
```

以上是各个异常情况下的入口地址，在reset_vector_mp.S中都能找到，经过编译链接后就会变成

```
b  0x00000000     @开机代码
b  0x00000004     @异常处理之CPU碰到不认识的指令
b  0x00000008  @异常处理之:软中断
b  0x0000000C     @异常处理之:取指异常
b  0x00000010  @异常处理之:数据异常
b  0x00000014  @异常处理之:地址异常
b  0x00000018  @异常处理之:硬中断
b  0x0000001C  @异常处理之:快中断
```

不管是主动切换的异常，还是被动切换的异常，都会先跳到对应的入口去处理。每个异常的代码都起始于汇编，处理完了再切回去。举个例子: 某个应用程序调用了系统调用(比如创建定时器)，会经过以下大致过程:

- swi指令将用户模式切换到管理模式（svc）
- 在管理模式中先保存用户模式的现场信息(R0-R15寄存器值入栈)
- 获取系统调用号，知道是调用了哪个系统调用
- 查询系统调用对应的注册函数
- 执行真正的创建定时器函数
- 执行完成后，恢复用户模式的现场信息(R0-R15寄存器值出栈)
- 跳回用户模式继续执行

各异常处理代码很多，不一一列出，本篇只列出开机代码，请尝试读懂鸿蒙内核开机代码，后续讲详细说明每行代码的用处。

开机代码

```
reset_vector:   //开机代码
/* clear register TPIDRPRW */
mov     r0 ， #0     @r0 = 0
mcr     p15 ， 0 ， r0 ， c13 ， c0 ， 4  @c0，c13 = 0 ， C13为进程标识符
/* do some early cpu setup: i/d cache disable ， mmu disabled */ @禁用MMU ， i/d缓存
mrc     p15 ， 0 ， r0 ， c1 ， c0 ， 0   @r0 = c1 ，c1寄存器详细解释见第64页
bic     r0 ， #(1<<12)    @位清除指令，清除r0的第11位
bic     r0 ， #(1<<2 | 1<<0)  @清除第0和2位 ，禁止 MMU和缓存 0位:MMU enable/disable 2位:Cache enable/disable
mcr     p15 ， 0 ， r0 ， c1 ， c0 ， 0  @c1=r0

/* r11: delta of physical address and virtual address */@物理地址和虚拟地址的增量
adr     r11 ， pa_va_offset @将基于PC相对偏移的地址pa_va_offset值读取到寄存器R11中
ldr     r0 ， [r11]    @将R11的值给r0
sub     r11 ， r11 ， r0   @r11 = r11 - r0

mrc     p15 ， 0 ， r12 ， c0 ， c0 ， 5             /* r12: get cpuid */ @获取CPUID
and     r12 ， r12 ， #MPIDR_CPUID_MASK @r12经过掩码过滤
cmp     r12 ， #0 @当前是否为0号CPU
bne     secondary_cpu_init @不是0号主CPU则调用secondary_cpu_init

/* if we need to relocate to proper location or not */
adr     r4 ， __exception_handlers          /* r4: base of load address */ @r4获得加载基地址
ldr     r5 ， =SYS_MEM_BASE                 /* r5: base of physical address */@r5获得物理基地址
subs    r12 ， r4 ， r5                     /* r12: delta of load address and physical address */ @r12=r4-r5 加载地址和物理地址的增量
beq     reloc_img_to_bottom_done            /* if we load image at the bottom of physical address */

/* we need to relocate image at the bottom of physical address */
ldr     r7 ， =__exception_handlers         /* r7: base of linked address (or vm address) */
ldr     r6 ， =__bss_start                  /* r6: end of linked address (or vm address) */
sub     r6 ， r7                            /* r6: delta of linked address (or vm address) */
add     r6 ， r4                            /* r6: end of load address */
```

异常的优先级

当同时出现多个异常时，该响应哪一个呢？这涉及到了异常的优先级，顺序如下

- 1.Reset (highest priority).
- 2.data Abort.
- 3.FIQ.
- 4.IRQ.
- 5.Prefetch Abort.
- 6.Undefined Instruction ， SWI (lowest priority).

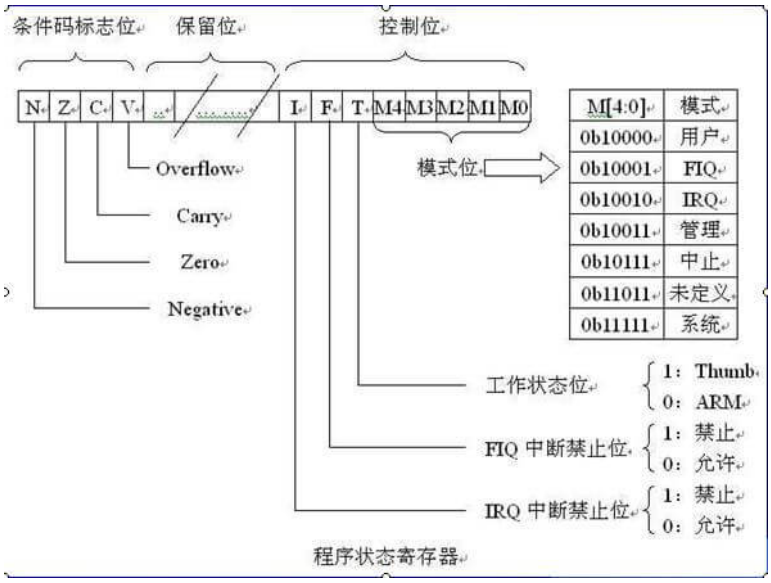
可以看出swi的优先级最低，swi就是软中断，系统调用就是通过它来实现的。

3。异常模式怎么切换？

写应用程序经常会用到状态，来记录各种分支逻辑，传递参数。这么多异常模式，相互切换，中间肯定会有很多的状态需要保存。比如:如何能知道当前运行在哪种模式下？怎么查？去哪里查呢？ 答案是: CPSR(一个) 和 SPSR(5个) 这些寄存器：

- 保存有关最近执行的ALU操作的信息
- 控制中断的启用和禁用
- 设置处理器操作模式

CPSR 寄存器



CPSR(current program status register)当前程序的状态寄存器

CPSR有4个8位区域：标志域（F）、状态域（S）、扩展域（X）、控制域（C） 32 位的程序状态寄存器可分为4 个域：

-
  - 位[31：24]为条件标志位域，用f 表示；
-
  - 位[23：16]为状态位域，用s 表示；
-
  - 位[15：8]为扩展位域，用x 表示；
-
  - 位[7：0]为控制位域，用c 表示；

CPSR和其他寄存器不一样，其他寄存器是用来存放数据的，都是整个寄存器具有一个含义。 而CPSR寄存器是按位起作用的，也就是说，它的每一位都有专门的含义，记录特定的信息。

CPSR的低8位（包括I、F、T和M[4：0]）称为控制位，程序无法修改， 除非CPU运行于特权模式下，程序才能修改控制位

N、Z、C、V均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变， 并且可以决定某条指令是否被执行!意义重大!

- CPSR的第31位是 N，符号标志位。它记录相关指令执行后，其结果是否为负。 如果为负 N = 1，如果是非负数 N = 0。
- CPSR的第30位是Z，0标志位。它记录相关指令执行后，其结果是否为0。 如果结果为0。那么Z = 1。如果结果不为0，那么Z = 0。
- CPSR的第29位是C，进位标志位(Carry)。一般情况下，进行无符号数的运算。 加法运算：当运算结果产生了进位时（无符号数溢出），C=1，否则C=0。 减法运算（包括CMP）：当运算时产生了借位时（无符号数溢出），C=0，否则C=1。
- CPSR的第28位是V，溢出标志位(Overflow)。在进行有符号数运算的时候， 如果超过了机器所能标识的范围，称为溢出。

MSR{条件} 程序状态寄存器(CPSR 或SPSR)_<域>，操作数 MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中 示例如下：

```
MSR CPSR，R0  @传送R0 的内容到CPSR
MSR SPSR，R0  @传送R0 的内容到SPSR
MSR CPSR_c，R0 @传送R0 的内容到CPSR，但仅仅修改CPSR中的控制位域
```

MRS{条件} 通用寄存器，程序状态寄存器(CPSR 或SPSR) MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下两种情况： 1) 当需要改变程序状态寄存器的内容时，可用MRS 将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。 2) 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。 示例如下：

```
MRS R0，CPSR  @传送CPSR 的内容到R0
MRS R0，SPSR  @传送SPSR 的内容到R0
        @MRS指令是唯一可以直接读取CPSR和SPSR寄存器的指令
```

SPSR 寄存器

SPSR（saved program status register）程序状态保存寄存器。五种异常模式下一个状态寄存器SPSR，用于保存CPSR的状态，以便异常返回后恢复异常发生时的工作状态。

- 1、SPSR 为 CPSR 中断时刻的副本，退出中断后，将SPSR中数据恢复到CPSR中。
- 2、用户模式和系统模式下SPSR不可用，所以SPSR寄存器只有5个

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

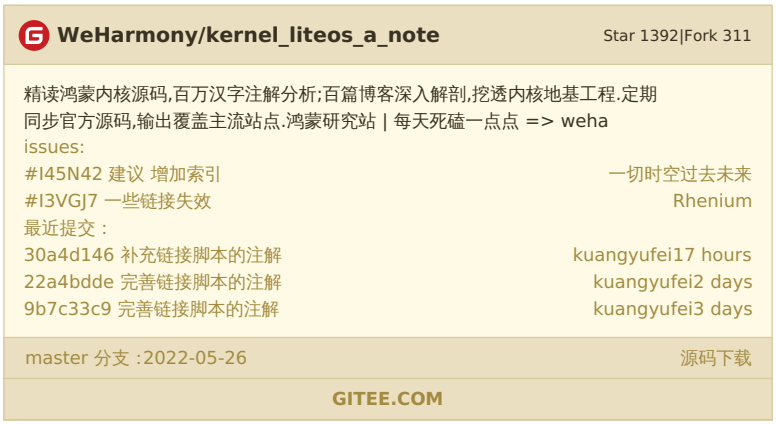
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

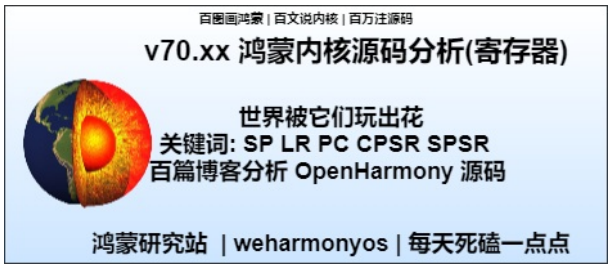


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

70_寄存器篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

本篇说清楚寄存器

本篇需结合 << **ARM体系架构参考手册(ARMv7-A/R).pdf** >> 阅读。

寄存器的本质

寄存器从大一的计算机组成原理就开始听到它，感觉很神秘，如梦如雾多年。揭开本质后才发现，寄存器就是一个32位的存储空间，一个int变量而已(其背后的硬件原理是D触发器)，但它的厉害之处在于极高频率的使用，让人不敢相信是怎么做到的，不管再复杂再牛牛的应用程序，电商也好，游戏，直播也罢，到了它这里都变成了有限的十几个寄存器在玩，简直太神奇了。 本篇将清楚说明寄存器的数量和功能，至于它是如何把复杂的上层程序变成了这十几个寄存器来玩？这是编译器的事情，不在讨论范围之内。

在 32 位的 ARM 架构中，核心寄存器（core register）的数量一般有 37 个或者更多，视处理器实现的功能多少而定。所谓核心寄存器就是指 ARM 处理器内核执行常规指令时使用的寄存器，不包括用于浮点计算和 SIMD 技术的特殊寄存器，也可以理解为是 ARM 的核心处理器单元（PE）中的寄存器，不包括外围的协处理器中的寄存器。

ARM7的37个寄存器，具体看图说明:

ARM state general registers and program counter

31个通用寄存器,r13_*这些是单独算的,r13,r14各六个　来源: 鸿蒙内核源码分析

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

注意: r15(pc寄存器)七种工作模式通用, 因为代码是共用的, 所以可以通用.

6个状态寄存器,系统和用户模式寄存器共用　详见: weharmony.gitee.io

ARM state program status registers weharmony.github.io

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

这些寄存器不能同时显示，处理器指令状态和工作模式指定哪些寄存器可供使用，图中一一对应。

- 其中31个通用32位寄存器，系统和用户模式全程复用寄存器，而其余5中异常(或叫特权)模式从R8_* ~ R14_* 的寄存器叫模式专属寄存器。这种特征的寄存器有个专门的称呼，叫 Banked register.Bank 本意是银行和存款的意思，在这里的意思是"有备份的"。
- 注意 r8 和 r8_fiq是两个不同的寄存器，名字前缀是为了好记，管理方便，以示同级概念理解。如此凑成了31个寄存器。
- 其中r13寄存器用于SP寄存器，始终指向栈顶，因为每种工作模式都有独立的运行栈，所以有独立的寄存器去记住各自的栈顶。
- 同理r14寄存器用于LR寄存器，用于保存模式切换时的切换位置，也是独立存在，说明模式间回跳时并不需要重新给r14_*赋值，只需在跳出去的时候保存即可。
- 系统和用户模式共用r13(sp)和r14(lr)寄存器，所以在每个子函数的栈帧中都要保存上一个调用它函数的SP和LR值，自己执行完成后要从栈帧中恢复这两个寄存器的值，否则无法界定回去后从哪里开始，从哪里计算偏移位置。
- r15(pc)寄存器是指向代码段的，所有模式复用的原因是它是共用的，一份代码，你运行与不运行，代码段就在哪里，不增不减。
- 6个状态寄存器，其中CPSR(1个)和SPSR_*(5个)，它们主要用于自运行或发生模式切换后的各种状态保存。
- CPSR:程序状态寄存器(current program status register) (当前程序状态寄存器)，在任何处理器模式下被访问。
- SPSR:程序状态保存寄存器（saved program status register），每一种处理器模式下都有一个状态寄存器SPSR，SPSR用于保存CPSR的状态，以便异常返回后恢复异常发生时的工作状态。当特定 的异常中断发生时，这个寄存器用于存放当前程序状态寄存器的内容。在异常中断退出时，可以用SPSR来恢复CPSR。

七种工作模式

关于工作模式在本文末尾对应篇中有详细介绍，可自行前往查看。此处只简单说明下。 本篇需结合 << **ARM体系架构参考手册(ARMv7-A/R).pdf** >> 阅读。

在ARM体系中，CPU工作在以下七种模式中:

The ARM720T supports seven modes of operation as listed in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution state
FIQ	fiq	Designed to support a data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

- **用户模式（usr）**：该模式是用户程序的工作模式，它运行在操作系统的用户态，它没有权限去操作其它硬件资源，只能执行处理自己的数据，也不能切换到其它模式下，要想访问硬件资源或切换到其它模式只能通过软中断或产生异常。
- **快速中断模式（fiq）**：快速中断模式是相对一般中断模式而言的，用来处理高优先级中断的模式，处理对时间要求比较紧急的中断请求，主要用于高速数据传输及通道处理中。
- **普通中断模式（irq）**：一般中断模式也叫普通中断模式，用于处理一般的中断请求，通常在硬件产生中断信号之后自动进入该模式，该模式可以自由访问系统硬件资源。
- **管理模式（svc）**：操作系统保护模式，CPU上电复位和当应用程序执行 SVC 指令调用系统服务时也会进入此模式，操作系统内核的普通代码通常工作在这个模式下。
- **终止模式（abt）**：当数据或指令预取终止时进入该模式，中止模式用于支持虚拟内存或存储器保护，当用户程序访问非法地址，没有权限读取的内存地址时，会进入该模式，
- **系统模式（sys）**：供操作系统使用的高特权用户模式，与用户模式类似，但具有可以直接切换到其他模式等特权，用户模式与系统模式两者使用相同的寄存器，都没有SPSR（Saved Program Statement Register，已保存程序状态寄存器），但系统模式比用户模式有更高的权限，可以访问所有系统资源。
- **未定义模式（und）**：未定义模式用于支持硬件协处理器的软件仿真，CPU在指令的译码阶段不能识别该指令操作时，会进入未定义模式。

除用户模式外，其余6种工作模式都属于特权模式

- 特权模式中除了系统模式以外的其余5种模式称为异常模式
- 大多数程序运行于用户模式
- 进入特权模式是为了处理中断、异常、或者访问被保护的系统资源
- 硬件权限级别：系统模式 > 异常模式 > 用户模式
- 快中断(fiq)与慢中断(irq)区别：快中断处理时禁止中断

每种模式都有自己独立的入口和独立的运行栈空间。 **系列篇之CPU篇** 已介绍过只要提供了入口函数和运行空间，CPU就可以干活了。入口函数解决了指令来源问题，运行空间解决了指令的运行场地问题。 而且在多核情况下，每个CPU核的每种特权模式都有自己独立的栈空间。注意是特权模式下的栈空间，用户模式的栈空间是由用户(应用)程序提供的。

R0~R7 寄存器

这 8 个寄存器是最普通的，所有模式都可以访问和使用。 尤其是 `R0` 是寄存器中的王牌，被称为头号寄存器，通用寄存器中它用的最高频，随便翻段汇编代码都能看到它的影子。鸿蒙开机第一跳指令就是 r0 = 0

```
reset_vector: //鸿蒙开机代码
   /* clear register TPIDRPRW */
   mov     r0 ， #0     @r0 = 0
   mcr     p15 ， 0 ， r0 ， c13 ， c0 ， 4  @c0 , c13 = 0 ， C13为进程标识符
   /* do some early cpu setup: i/d cache disable ， mmu disabled */ @禁用MMU ， i/d缓存
   mrc     p15 ， 0 ， r0 ， c1 ， c0 ， 0   @r0 = c1 ，c1寄存器详细解释见第64页
   bic     r0 ， #(1<<12)    @位清除指令，清除r0的第11位
   bic     r0 ， #(1<<2 | 1<<0)  @清除第0和2位 ，禁止 MMU和缓存 0位:MMU enable/disable 2位:Cache enable/disable
   mcr     p15 ， 0 ， r0 ， c1 ， c0 ， 0  @c1=r0
```

再看拿自旋锁的汇编代码，这些代码都在系列篇中详细讲解过，可前往 v08.xx 鸿蒙内核源码分析(总目录) 自行查看。

```
FUNCTION(ArchSpinLock) @非要拿到锁
 mov  r1 ， #1  @r1=1
1:      @循环的作用，因SEV是广播事件。不一定lock->rawLock的值已经改变了
 ldrex r2 ， [r0] @r0 = &lock->rawLock ， 即 r2 = lock->rawLock
 cmp  r2 ， #0  @r2和0比较
 wfene    @不相等时，说明资源被占用，CPU核进入睡眠状态
 strexeq r2 ， r1 ， [r0]@此时CPU被重新唤醒，尝试令lock->rawLock=1，成功写入则r2=0
 cmpeq r2 ， #0  @再来比较r2是否等于0，如果相等则获取到了锁
 bne  1b   @如果不相等，继续进入循环
 dmb     @用DMB指令来隔离，以保证缓冲中的数据已经落实到RAM中
 bx  lr   @此时是一定拿到锁了，跳回调用ArchSpinLock函数
```

`R0` 被潜规则的干了两件事，突出了它的重要性:

- 第一个参数 由R0保管，当然第二个参数就给R1保管
- 函数的返回值统一交给R0保管， 例如 a -> b ，b执行完会把返回值给r0，回到a后，a从r0取值，不管取到什么，它就认为这是b的返回值，默认都只认r0保存了返回值，这就是规定。

具体看一个C函数和它的汇编，在系列篇也已经讲过，可自行翻看。

```
//++++++++++++ square(c -> 汇编)++++++++++++++++++++++++++
int square(int a , int b){
   return a*b;
}
square(int ， int):
       sub     sp ， sp ， #8     @sp减去8，意思为给square分配栈空间，只用2个栈空间完成计算
       str     r0 ， [sp ， #4]   @第一个参数入栈
       str     r1 ， [sp]       @第二个参数入栈
       ldr     r1 ， [sp ， #4]   @取出第一个参数给r1
       ldr     r2 ， [sp]       @取出第二个参数给r2
       mul     r0 ， r1 ， r2     @执行a*b给R0，返回值的工作一直是交给R0的
       add     sp ， sp ， #8     @函数执行完了，要释放申请的栈空间
       bx      lr            @子程序返回，等同于mov pc，lr，即跳到调用处
//++++++++++++ fp(c -> 汇编)++++++++++++++++++++++++++
int fp(int b)
{
   int a = 1;
   return square(a+b , a+b);
}
fp(int):
       push    {r11 ， lr}      @r11(fp)/lr入栈，保存调用者main的位置
       mov     r11 ， sp        @r11用于保存sp值，函数栈开始位置
       sub     sp ， sp ， #8     @sp减去8，意思为给fp分配栈空间，只用2个栈空间完成计算
       str     r0 ， [sp ， #4]   @先保存参数值，放在SP+4，此时r0中存放的是参数
       mov     r0 ， #1         @r0=1
       str     r0 ， [sp]       @再把1也保存在SP的位置
       ldr     r0 ， [sp]       @把SP的值给R0
       ldr     r1 ， [sp ， #4]   @把SP+4的值给R1
       add     r1 ， r0 ， r1     @执行r1=a+b
       mov     r0 ， r1         @r0=r1，用r0，r1传参
       bl      square(int ， int)@先mov lr ， pc 再mov pc square(int ， int)
       mov     sp ， r11        @函数执行完了，要释放申请的栈空间
       pop     {r11 ， lr}      @弹出r11和lr，lr是专用标签，弹出就自动复制给lr寄存器
       bx      lr            @子程序返回，等同于mov pc，lr，即跳到调用处
```

这段代码同样适用于理解以下的各个寄存器。R0的作用相当于 x86 的 EAX

R7 寄存器

为啥要单独讲R7寄存器，因为它偶尔作为特殊寄存器在使用。内核对上层应用提供了数百个系统调用功能，当发生系统调用时，在CPU工作模式切换过程中，系统调用号是一直保存在R7寄存器中的，通过系统调用号就能查询到对应的注册函数。具体在 系统调用篇中有详细的过程说明，这里只列出部分代码

```
//4个参数的系统调用时底层处理
static inline long __syscall4(long n， long a， long b， long c， long d)
{
 register long a7 __asm__("a7") = n; //将系统调用号保存在R7寄存器
 register long a0 __asm__("a0") = a; //R0
 register long a1 __asm__("a1") = b; //R1
 register long a2 __asm__("a2") = c; //R2
 register long a3 __asm__("a3") = d; //R3
 __asm_syscall("r"(a7)， "0"(a0)， "r"(a1)， "r"(a2)， "r"(a3))
}
//切换到SVC模式后，由汇编代码调用由C语言实现的系统调用统一入口
LITE_OS_SEC_TEXT UINT32 *OsArmA32SyscallHandle(UINT32 *regs)
{
   UINT32 ret;
   UINT8 nArgs;
   UINTPTR handle;
   UINT32 cmd = regs[REG_R7];// 从R7寄存器中取出系统调用号
   handle = g_syscallHandle[cmd];//查询系统调用的注册函数
   //...
}
```

fp(R11) 寄存器

R11：可以用作通用寄存器，在开启特定编译选项时可以用作帧指针寄存器FP，用来实现栈回溯功能。 GNU编译器（gcc）默认将R11作为存储变量的通用寄存器，因而默认情况下无法使用FP的栈回溯功能。为支持调用栈解析功能，需要在编译参数中添加 -fno-omit-frame-pointer 选项，提示编译器将R11作为FP使用。

FP寄存器（Frame Point），帧指针寄存器，指向当前函数的父函数的栈帧起始地址。利用该寄存器可以得到父函数的栈帧，从栈帧中获取父函数的FP，就可以得到祖父函数的栈帧，以此类推，可以追溯程序调用栈，得到函数间的调用关系。

在鸿蒙内核R11是当FP寄存器使用。

SP(R13) 寄存器

SP:栈指针寄存器(stack pointer)，它也是 `banked register`，而且所有模式都有一份，总共有 6 个（有虚拟化支持时再多一个），分别用于用户、 `IRQ`、 `FIQ`、 未定义、中止和管理员模式。在 ARM 手册，有时用 `SP_usr`、`SP_svc` 这样的写法来表示不同模式下的 SP 寄存器。

SP指向函数栈的栈顶，如此 `fp` 和 `sp` 就划定了函数栈的范围，函数在运行期间除了动态申请的内存要跑出去玩，其余就在这块空间里玩。

在鸿蒙内核R13是当SP寄存器使用。

LR(R14) 寄存器

又叫 Link Register，简称 `LR`，在主动调用子函数时， `ARM` 处理器会自动将子函数的返回地址放到这个寄存器中。 另外在异常发生的被动阶段，会导致程序正常运行的被打断， 并将控制流转移到相应的异常处理（异常响应），有些异常（ `fiq`、 `irq` ）事件处理后，系统还希望能回 到当初异常发生时被打断的源程序断点处继续完成源程序的执行（异常返回），这就需要一种解决方案， 用于记录源程序的断点位置，以便正确的异常返回。 类似的还有子程序的调用和 返回。在主程序中（通过子程序调用指令）调用子程序时，也需要记录下主程序中的调用点位置，以便将来的子程序的返回。

LR:链接寄存器(linked pointer)，就是用来解决上述问题的，ARM处理器中使用 R14实现对断点和调用点的记录，即R14用作返回连接寄存器（LR），确保回来知道自己从哪个位置中断，以便继续执行。

在鸿蒙内核 `R14` 是当 `LR` 寄存器使用。

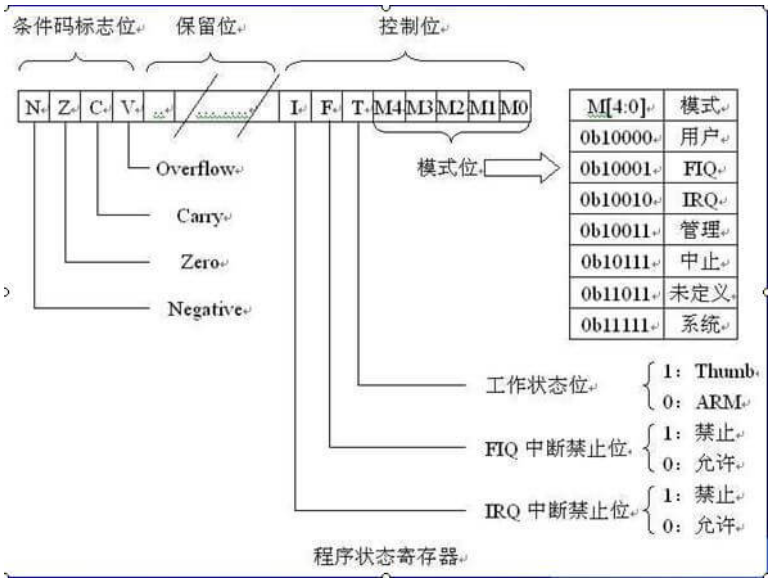
PC(R15) 寄存器

简称 PC（Program Counter）。当执行 ARM 指令（每条指令 4 字节），它的值为当前指令的地址加 8，当执行 `Thumb` 指令时，它的值为当前指令的地址加 4，其设计原则是让 `PC` 指向当前指令后面的第二条指令。

PC寄存器涉及到arm的流水线结构设计，具体在后续流水线篇中详细说明，敬请关注。

在鸿蒙内核 `R15` 是当 `PC` 寄存器使用。

CPSR 寄存器



CPSR(current program status register)当前程序的状态寄存器 CPSR有4个8位区域：标志域（F）、状态域（S）、扩展域（X）、控制域（C） 32 位的程序状态寄存器可分为4 个域：

-
  - 位[31：24]为条件标志位域，用f 表示；
-
  - 位[23：16]为状态位域，用s 表示；
-
  - 位[15：8]为扩展位域，用x 表示；
-
  - 位[7：0]为控制位域，用c 表示；

CPSR和其他寄存器不一样，其他寄存器是用来存放数据的，都是整个寄存器具有一个含义。 而CPSR寄存器是按位起作用的，也就是说，它的每一位都有专门的含义，记录特定的信息。

CPSR的低8位（包括I、F、T和M[4：0]）称为控制位，程序无法修改， 除非CPU运行于特权模式下，程序才能修改控制位

N、Z、C、V均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行!意义重大!

- CPSR的第31位是 N，符号标志位。它记录相关指令执行后，其结果是否为负。 如果为负 N = 1，如果是非负数 N = 0。
- CPSR的第30位是Z，0标志位。它记录相关指令执行后，其结果是否为0。 如果结果为0。那么Z = 1。如果结果不为0，那么Z = 0。
- CPSR的第29位是C，进位标志位(Carry)。一般情况下，进行无符号数的运算。 加法运算：当运算结果产生了进位时（无符号数溢出），C=1，否则C=0。 减法运算（包括CMP）：当运算时产生了借位时（无符号数溢出），C=0，否则C=1。
- CPSR的第28位是V，溢出标志位(Overflow)。在进行有符号数运算的时候， 如果超过了机器所能标识的范围，称为溢出。

MSR{条件} 程序状态寄存器(CPSR 或SPSR)_<域>，操作数 MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中 示例如下：

```
MSR CPSR，R0  @传送R0 的内容到CPSR
MSR SPSR，R0  @传送R0 的内容到SPSR
MSR CPSR_c，R0 @传送R0 的内容到CPSR，但仅仅修改CPSR中的控制位域
```

MRS{条件} 通用寄存器，程序状态寄存器(CPSR 或SPSR) MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用在以下两种情况： 1) 当需要改变程序状态寄存器的内容时，可用MRS 将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。 2) 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。 示例如下：

```
MRS R0，CPSR  @传送CPSR 的内容到R0
MRS R0，SPSR  @传送SPSR 的内容到R0
```

```
@MRS指令是唯一可以直接读取CPSR和SPSR寄存器的指令
```

SPSR 寄存器

`SPSR`（saved program status register）程序状态保存寄存器。五种异常模式下一个状态寄存器`SPSR`，用于保存`CPSR`的状态，以便异常返回后恢复异常发生时的工作状态。

- 1、SPSR 为 CPSR 中断时刻的副本，退出中断后，将SPSR中数据恢复到CPSR中。
- 2、用户模式和系统模式下SPSR不可用，所以SPSR寄存器只有5个

留个问题

从R11 ~ R15 寄存器除了R12都用着专用寄存器，用作为特殊用途，单独独R12夹在中间不上不下的，这又是为什么呢？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

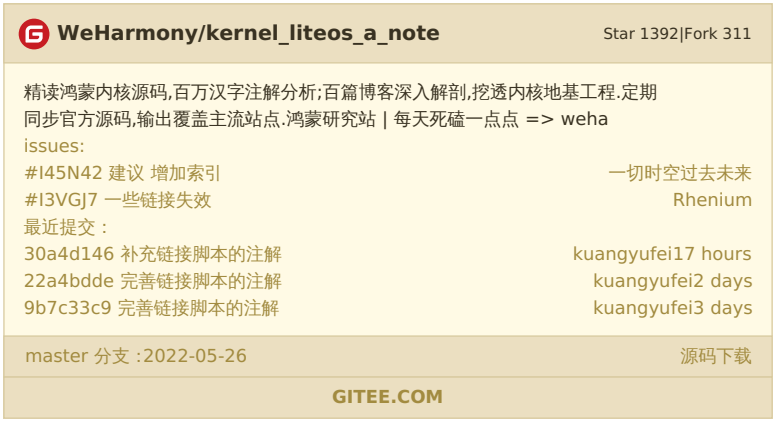
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

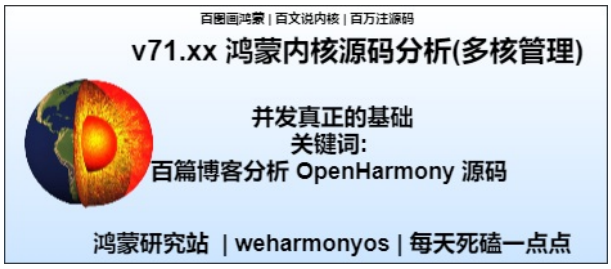


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

71_多核管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

本篇说清楚CPU

读本篇之前建议先读 v08.xx 鸿蒙内核源码分析(总目录) 进程/线程篇。

- 指令是稳定的，但指令序列是变化的，只有这样计算机才能够实现用计算来解决一切问题这个目标。计算是稳定的，但计算的数据是多变的，多态的，地址是数据，控制信号也是数据。指令集本身也是数据(固定的数据)。只有这样才能够让计算机不必修改基础架构却可以适应不断发展变化的技术革命。
- `cpu` 是负责执行指令的，谁能给它指令？是线程(也叫任务)， 任务是内核的调度单元，调度到哪个任务CPU就去执行哪个任务的指令。 要执行指令就要有个取指令的开始地址。 开始地址就是大家所熟知的main函数。一个程序被加载解析后内核会在ELF中找到main函数的位置，并自动创建一个线程，指定线程的入口地址为main函数的地址，由此开始了取指，译指，执指之路。
- 多线程内核是怎么处理的？ 一样的， 以JAVA举例，对内核来说 new thread中的run() 函数 和 main() 并没有区别。 都是一个线程(任务)的执行入口。 注意在系列篇中反复的说任务就是线程，线程就是任务，它们是一个东西在不同层面上的描述。对应用层说线程，对内核层说任务。 有多少个线程就会有多少个入口，它们统一接受调度算法的调度， 调度算法只认优先级的高低，不会管你是main() 还是 run() 而区别对待。
- 定时器的实现也是通过任务实现的，只不过是个系统任务 `OsSwtmrTaskCreate` ，优先级最高，和入口地址 `OsSwtmrTask` 由系统指定。
- 所以理解CPU就要先理解任务，任务是理解内核的主线，把它搞明白了分析内核就轻轻松松，事半功倍了。看似高深的CPU只不过是搂草打兔子。不相信？那就看看内核对CPU是怎么描述的吧。本篇就围绕这个结构体展开说。

Percpu

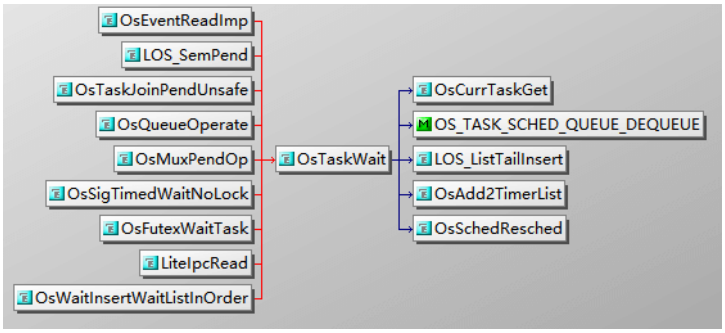
percpu变量，顾名思义，就是对于同一个变量，每个cpu都有自己的一份，它可以被用来存放一些cpu独有的数据，比如cpu的id，cpu上正在运行的任务等等。

```
Percpu g_percpu[LOSCFG_KERNEL_CORE_NUM];//CPU核描述符，描述每个CPU的信息。
typedef struct {//内核对cpu的描述
    SortLinkAttribute taskSortLink;           /* task sort link */ //挂等待和延时的任务
    SortLinkAttribute swtmrSortLink;          /* swtmr sort link */ //挂定时器
    UINT32 idleTaskID;                      /* idle task id */  //空闲任务ID 见于 OsIdleTaskCreate
    UINT32 taskLockCnt;                     /* task lock flag */ //任务锁的数量，当 > 0 的时候，需要重新调度了
    UINT32 swtmrHandlerQueue;                   /* software timer timeout queue id */ //软时钟超时队列句柄
    UINT32 swtmrTaskID;                     /* software timer task id */ //软时钟任务ID
    UINT32 schedFlag;                       /* pending scheduler flag */ //调度标识 INT_NO_RESCH INT_PEND_RESCH
#if (LOSCFG_KERNEL_SMP == YES)
    UINT32 excFlag;                     /* cpu halt or exc flag */ //CPU处于停止或运行的标识
```

```
#endif
} Percpu;
```

至于 g_percpu 的值怎么来的，因和编译过程相关，将在后续编译篇中说明。 Percpu 结构体不复杂，但很重要，一个一个掰开了说。

- taskSortLink 是干什么用的？ 一个任务在运行过程中，经常会主动或被动停止，而进入等待状态。
  - 主动停止情况， 例如:主动delay300毫秒，这是应用层很常见的操作。
  - 被动停止情况， 例如:申请互斥锁失败，等待某个事件发生。 发生这些情况时任务将被挂到 taskSortLink 上。这些任务可能来自不同的进程，但都是因为在被这个CPU执行时停下来了，等着再次被它执行。下图很清晰的看出在哪种情况下会被记录在案。




```
UINT32 OsTaskWait(LOS_DL_LIST *list， UINT32 timeout， BOOL needSched)
{
   LosTaskCB *runTask = NULL;
   LOS_DL_LIST *pendObj = NULL;
   runTask = OsCurrTaskGet();//获取当前任务
   OS_TASK_SCHED_QUEUE_DEQUEUE(runTask， OS_PROCESS_STATUS_PEND);//将任务从就绪队列摘除，并变成阻塞状态
   pendObj = &runTask->pendList;
   runTask->taskStatus |= OS_TASK_STATUS_PEND;//给任务贴上阻塞任务标签
   LOS_ListTailInsert(list， pendObj);//将阻塞任务挂到list上，，这步很关键，很重要!
   if (timeout != LOS_WAIT_FOREVER) {//非永远等待的时候
      runTask->taskStatus |= OS_TASK_STATUS_PEND_TIME;//阻塞任务再贴上在一段时间内阻塞的标签
      OsAdd2TimerList(runTask， timeout);//把任务加到定时器链表中
   }
   if (needSched == TRUE) {//是否需要调度
      OsSchedResched();//申请调度，里面直接切换了任务上下文，至此任务不再往下执行了。
      if (runTask->taskStatus & OS_TASK_STATUS_TIMEOUT) {//这条语句是被调度再次选中时执行的，和上面的语句可能隔了很长时间，所以很可能
         runTask->taskStatus &= ~OS_TASK_STATUS_TIMEOUT;//如果任务有timeout的标签，那么就去掉那个标签
         return LOS_ERRNO_TSK_TIMEOUT;
      }
   }
   return LOS_OK;
}
LITE_OS_SEC_TEXT STATIC INLINE VOID OsAdd2TimerList(LosTaskCB *taskCB， UINT32 timeOut)
{
   SET_SORTLIST_VALUE(&taskCB->sortList， timeOut);//设置idxRollNum的值为timeOut
   OsAdd2SortLink(&OsPercpuGet()->taskSortLink， &taskCB->sortList);//将任务挂到定时器排序链表上
#if (LOSCFG_KERNEL_SMP == YES)//注意:这里的排序不是传统意义上12345的排序，而是根据timeOut的值来决定放到CPU core哪个taskSortLink[0
   taskCB->timerCpu = ArchCurrCpuid();
#endif
}
```

OsAdd2SortLink ，将任务挂到排序链表上，因等待时间不一样，所以内核会对这些任务按时间长短排序。

- 定时器相关三个变量，在系列篇定时器机制篇中已有对定时器的详细描述，可前往 v31.xx (定时器篇) 查看，就不难理解以下三个的作用了。

```
SortLinkAttribute swtmrSortLink;//CPU要处理的定时器链表
UINT32 swtmrHandlerQueue; //队列中放各个定时器的响应函数
UINT32 swtmrTaskID; // 其实就是 OsSwtmrTaskCreate
```

搞明白定时器的机制只需搞明白: 定时器(SWTMR_CTRL_S)，定时任务(swtmrTaskID)，定时器响应函数(SwtmrHandlerItem)，定时器处理队列 swtmrHandlerQueue 四者的关系就可以了。 一句话概括:定时任务 swtmrTaskID 是个系统任务，优先级最高，它循环读取队列 swtmrHandlerQueue 中的已到时间的定时器(SWTMR_CTRL_S)，并执行定时器对应的响应函数 SwtmrHandlerItem 。

- idleTaskID 空闲任务，注意这又是个任务，每个cpu核都有属于自己的空闲任务，cpu没事干的时候就待在里面。空闲任务长什么样？ Look!

```
//创建一个空闲任务
LITE_OS_SEC_TEXT_INIT UINT32 OsIdleTaskCreate(VOID)
{
    UINT32 ret;
    TSK_INIT_PARAM_S taskInitParam;
    Percpu *perCpu = OsPercpuGet();//获取CPU信息
    UINT32 *idleTaskID = &perCpu->idleTaskID;//每个CPU都有一个空闲任务
    (VOID)memset_s((VOID *)(&taskInitParam)， sizeof(TSK_INIT_PARAM_S)， 0， sizeof(TSK_INIT_PARAM_S));//任务初始参数清0
    taskInitParam.pfnTaskEntry = (TSK_ENTRY_FUNC)OsIdleTask;//入口函数
    taskInitParam.uwStackSize = LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE;//任务栈大小 2K
    taskInitParam.pcName = "Idle";//任务名称 叫pcName有点怪怪的，不能换个撒
    taskInitParam.usTaskPrio = OS_TASK_PRIORITY_LOWEST;//默认最低优先级 31
    taskInitParam.uwResved = OS_TASK_FLAG_IDLEFLAG;//默认idle flag
#if (LOSCFG_KERNEL_SMP == YES)//CPU多核情况
    taskInitParam.usCpuAffiMask = CPUID_TO_AFFI_MASK(ArchCurrCpuid());//每个idle任务只在单独的cpu上运行
#endif
    ret = LOS_TaskCreate(idleTaskID， &taskInitParam);//创建task并申请调度，
    OS_TCB_FROM_TID(*idleTaskID)->taskStatus |= OS_TASK_FLAG_SYSTEM_TASK;//设置task状态为系统任务，系统任务运行在内核态。
    //这里说下系统任务有哪些？比如: idle，swtmr(软时钟)，资源回收等等
    return ret;
}
LITE_OS_SEC_TEXT WEAK VOID OsIdleTask(VOID)
{
    while (1) {//只有一个死循环
    #ifdef LOSCFG_KERNEL_TICKLESS //低功耗模式开关， idle task 中关闭tick
    if (OsTickIrqFlagGet()) {
        OsTickIrqFlagSet(0);
        OsTicklessStart();
    }
    #endif
        Wfi();//WFI指令:arm core 立即进入low-power standby state，等待中断，进入休眠模式。
    }
}
```

OsIdleTask 是一个死循环，只有一条汇编指令 Wfi 。 啥意思？ WFI (Wait for interrupt):等待中断到来指令。 WFI 一般用于 cpuidle ， WFI 指令是在处理器发生中断或类似异常之前不需要做任何事情。具体在 自旋锁篇 中有详细描述，可前往查看。说到死循环，这里多说一句，从宏观尺度上来理解，整个内核就是一个死循环。因为有 软硬中断/异常 使得内核能活跃起来，能跳到不同的地方去执行，执行完了又会沉寂下去，等待新的触发到来。

- taskLockCnt 这个简单，记录等锁的任务数量。任务在运行过程中优先级是会不断地变化的， 例如 高优先级的A任务在等某锁，但持有锁的一方B任务优先级低，这时就会调高B的优先级至少到A的等级，提高B被调度算法命中的概率，如此就能快速的释放锁交给A运行。
  taskLockCnt 记录被CPU运行过的正在等锁的任务数量。

- schedFlag 调度的标签。

```
typedef enum {
    INT_NO_RESCH = 0，    /* no needs to schedule *///不需要调度
    INT_PEND_RESCH，      /* pending schedule flag *///阻止调度
} SchedFlag;
```

调度并不是每次都能成功的，在某些情况下内核会阻止调度进行。例如: OS_INT_ACTIVE 硬中断发生的时候。

```
STATIC INLINE VOID LOS_Schedule(VOID)
{
    if (OS_INT_ACTIVE) {//发生硬件中断，调度被阻塞
        OsPercpuGet()->schedFlag = INT_PEND_RESCH;//
        return;
    }
    OsSchedPreempt();//抢占式调度
}
```

- excFlag 标识CPU的运行状态，只在多核CPU下可见。

```
#if (LOSCFG_KERNEL_SMP == YES)
```

```
typedef enum {
    CPU_RUNNING = 0,   ///< cpu is running | CPU正在运行状态
    CPU_HALT,          ///< cpu in the halt | CPU处于暂停状态
    CPU_EXC            ///< cpu in the exc | CPU处于异常状态
} ExcFlag;
#endif
```

以上为内核对CPU描述的全貌，不是很复杂。多CPU的协同工作部分在后续篇中介绍。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

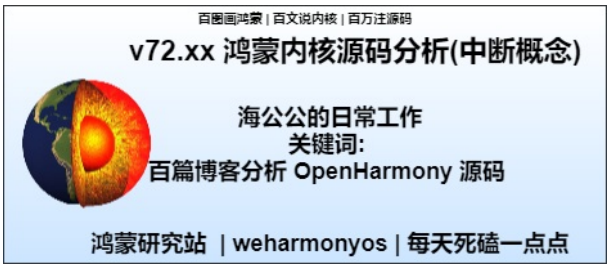
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

72_中断概念篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

关于中断部分系列篇将用三篇详细说明整个过程。

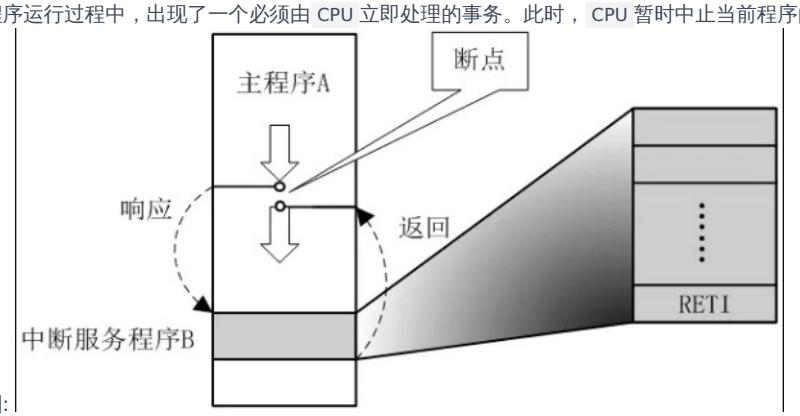
- **中断概念篇(本篇)** 中断概念很多，比如中断控制器，中断源，中断向量，中断共享，中断处理程序等等。本篇做一次整理。先了解透概念才好理解中断过程。本篇的主角是**海公公**，用海公公打比方说明白中断各个概念。
- **中断管理篇** 从中断初始化 `HalIrqInit` 开始，到注册中断的 `LOS_HwiCreate` 函数，到消费中断函数的 `HalIrqHandler` ，剖析鸿蒙内核实现中断的过程，很像设计模式中的观察者模式。 可前往v08。xx 鸿蒙内核源码分析(总目录) 查看。
- **中断切换篇** 用自下而上的方式，从中断源头纯汇编代码往上跟踪代码细节。说清楚保存和恢复中断现场 `TaskIrqContext` 过程。

中断概念

中断模块的核心是中断控制器，这可是 **皇上(CPU)** 身边的大红人海公公，外部人员找皇上办点事都必须经过它。

什么是中断？

- 中断是指程序运行过程中，出现了一个必须由 `CPU` 立即处理的事务。此时， `CPU` 暂时中止当前程序的执行转而处理这个事务，这个过程就叫做中断。如图:

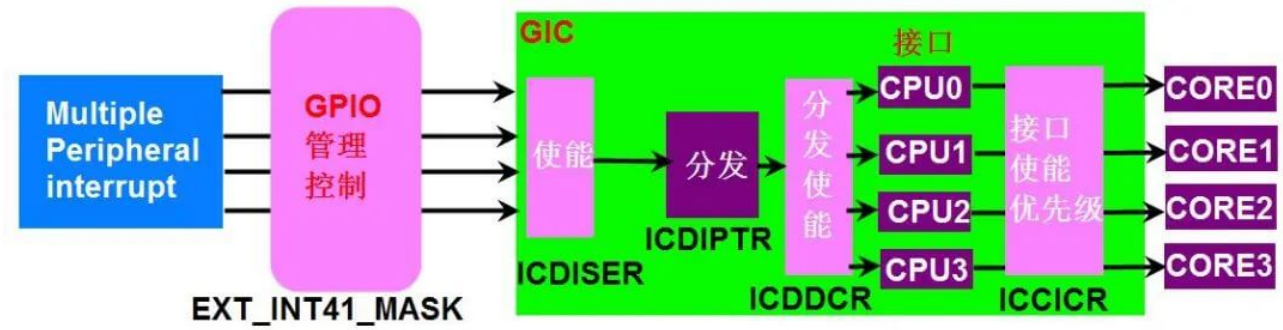


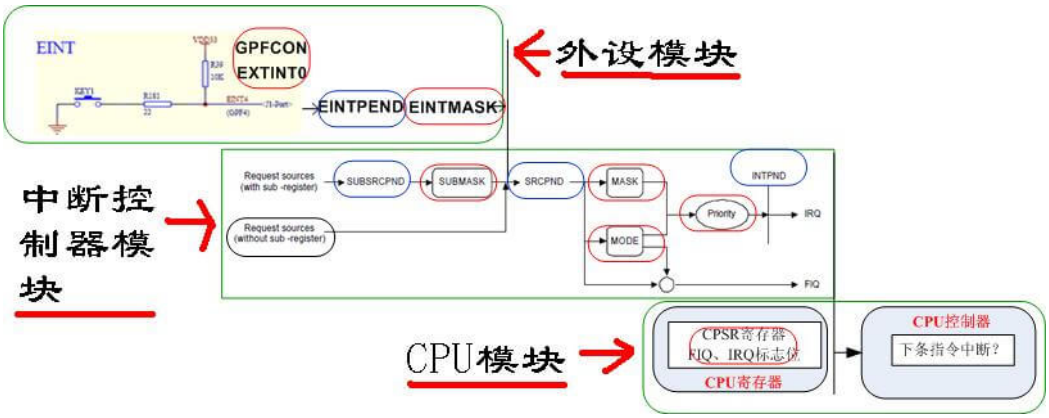
- 外设可以在没有CPU介入的情况下完成一定的工作，但某些情况下也需要CPU为其执行一定的工作。通过中断机制，在外设不需要CPU介入时，CPU可以执行其它任务，而当外设需要CPU时，将通过产生中断信号使CPU立即中断当前任务来响应中断请求。这样可以使CPU避免把大量时间耗费在等待、查询外设状态的操作上，大大提高系统实时性以及执行效率。

中断相关的硬件介绍

与中断相关的硬件可以划分为三类：设备(找皇上办事的事多了去)、中断控制器(海公公)、CPU(皇上威武，执天下耳)。

- **设备** 发起中断的源，当设备需要请求CPU时，产生一个中断信号，该信号连接至中断控制器。
- **中断控制器** 中断控制器是CPU众多外设中的一个，管理外设的外设，外设要使用CPU得先经过它仲裁， 它一方面接收其它外设中断引脚的输入，另一方面它会发出中断信号给CPU。所以可以通过对中断控制器编程来打开和关闭中断源、设置中断源的优先级和触发方式。说的是海公公有权屏蔽大臣们的折子，降低娘娘们被临幸的等级，让你们见不到咱皇上。 常用的中断控制器有VIC（Vector Interrupt Controller）和GIC（General Interrupt Controller）.在ARM Cortex-M系列中使用的中断控制器是NVIC（Nested Vector Interrupt Controller）.在ARM Cortex-A7中使用的中断控制器是GIC。
- **CPU** 中断控制器分发的中断源的请求给各个CPU，CPU收到请求便中断当前正在执行的任务，转而执行中断处理程序。 用二张图说明下三者的关系，能看出咱**海公公**的权利有多大。





中断控制器文档可前往 ARM中断控制器 gic_v2.pdf 查看每个寄存器的作用。以下为鸿蒙内核一小部分GIC寄存器的配置。

```
#ifdef LOSCFG_PLATFORM_BSP_GIC_V2
#define GICC_CTLR               (GICC_OFFSET + 0x00)    /* CPU Interface Control Register */ //CPU接口控制寄存器
#define GICC_PMR                (GICC_OFFSET + 0x04)    /* Interrupt Priority Mask Register */ //中断优先级屏蔽寄存器
#define GICC_BPR                (GICC_OFFSET + 0x08)    /* Binary Point Register */    //二进制点寄存器
#define GICC_IAR                (GICC_OFFSET + 0x0c)    /* Interrupt Acknowledge Register */ //中断确认寄存器
#define GICC_EOIR               (GICC_OFFSET + 0x10)    /* End of Interrupt Register */   //中断结尾寄存器
#define GICC_RPR                (GICC_OFFSET + 0x14)    /* Running Priority Register */   //运行优先寄存器
#define GICC_HPPIR              (GICC_OFFSET + 0x18)    /* Highest Priority Pending Interrupt Register */ //最高优先级挂起中断寄存器
#endif
```

中断源

所谓中断源，即引起中断的事件或原因，或发出中断申请的来源。 可分为外部中断源和内部中断源两大类。

- 外部中断源是指由CPU的外部事件引发的中断。主要包括：
  - 一般中、慢速外设，如键盘、打印机、鼠标等；
  - 数据通道，如磁盘、数据采集装置、网络等；
  - 实时时钟，如定时器定时已到，发中断申请；
  - 故障源，如电源掉电、外设故障、存储器读出出错以及越限报警等事件。

- 内部中断源是指由CPU的内部事件（异常）引发的中断，主要包括：
  - 由CPU执行中断指令INT n引起的中断；
  - 由CPU的某些运算错误引起的中断，如除数为0或商数超过了寄存器所能表达的范围、溢出等；
  - 为调试程序设置的中断，如单步中断、断点中断；
  - 由特殊操作引起的异常，如存储器越限、缺页等。
  - 核间中断，比如cpu a 让 cpu b 停止工作，产生调度等等。

这些都是想找咱皇上办事的人。

中断类型

把中断源划分为三种中断类型

- PPI：私有外设中断(Private Peripheral Interrupt)，是每个CPU私有的中断。最多支持16个PPI中断，硬件中断号从ID16~ID31。PPI通常会送达到指定的CPU上，应用场景有CPU本地时钟。 类似于皇上自己的一些私事，不方便说的，比如大明湖畔的夏雨荷来了。
- SGI：软件触发中断(Software Generated Interrupt)通常用于多核间通讯，最多支持16个SGI中断，硬件中断号从ID0~ID15。SGI通常在内核中被用作核间中断(inter-processor interrupts)，信号会送达到系统指定的CPU上。 主要用于多个皇上(CPU)并存的情况，皇上们直接约一起玩。
- SPI：公用外设中断(Shared Peripheral Interrupt)，最多可以支持988个外设中断，硬件中断号从ID32~ID1019。 属于外部公事，这种事比较多，比如无法预测的吴三桂同志突然造反了，黄河决堤了等等，所以排号也多，除了前面两种其他的都属于这类的。

中断请求

“紧急事件”需向CPU提出申请（发一个电脉冲信号），要求CPU暂停当前执行的任务，转而处理该“紧急事件”，这一申请过程称为中断请求，这个申请必须经过中断控制器仲裁。

找皇上办事的人先写报告走流程，要求都要经过咱海公公处过滤。

中断触发

中断源向中断控制器发送中断信号(电平触发或边沿触发)，中断控制器对中断进行仲裁，确定优先级，将中断信号送给CPU。中断源产生中断信号的时候，会将中断触发器置“1”，表明该中断源产生了中断，要求CPU去响应该中断。

相当于办事的折子，折子统一到了海公公这处理，编号。

中断优先级

为使系统能够及时响应并处理所有中断，系统根据中断时间的重要性和紧迫程度，将中断源分为若干个级别，称作中断优先级。

海公公给折子分好优先级。如花娘娘优先级最高，西施娘娘给的银子少优先级最低。

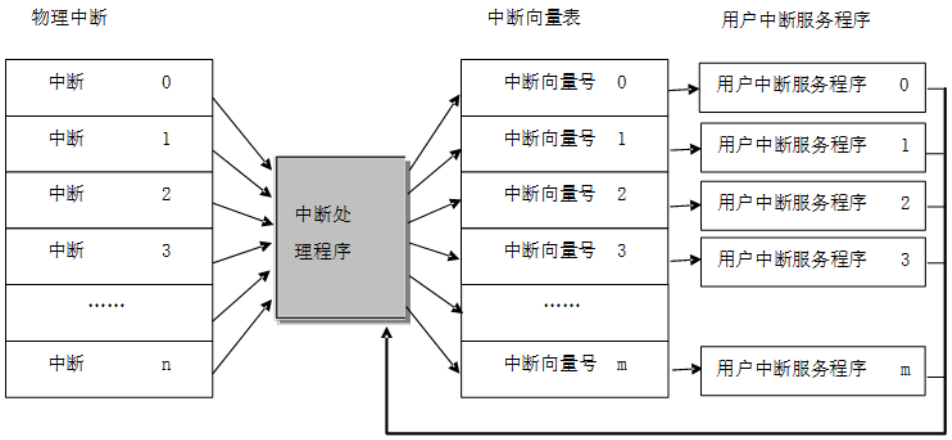
中断处理程序

当外设产生中断请求后，CPU暂停当前的任务，转而响应中断申请，即执行中断处理程序。产生中断的每个设备都有相应的中断处理程序。

海公公把折子交给了咱皇上，皇上一一处理所有折子。

中断向量表

- 中断号 :每个中断请求信号都会有特定的标志，使得计算机能够判断是哪个设备提出的中断请求，这个标志就是中断号。
- 中断向量 :中断服务程序的入口地址。
- 中断向量表是存储中断向量的存储区，中断向量与中断号对应，中断向量在中断向量表中按照中断号顺序存储。 中断向量表是所有中断处理程序的入口，如下图所示中断处理过程：把一个函数（用户中断服务程序）同一个虚拟中断向量表中的中断向量联系在一起。当中断向量对应中断发生的时候，被挂接的用户中断服务程序就会被调用执行。



所有中断都采用中断向量表的方式进行处理，即当一个中断触发时，处理器将直接判定是哪个中断源，然后直接跳转到相应的固定位置进行处理，每个中断服务程序必须排列在一起放在统一的地址上。中断向量表一般由一个数组定义或在起始代码中给出。

皇上把折子一对一的仔细处理，找到给对应折子办事的人。

用户中断服务程序(ISR)

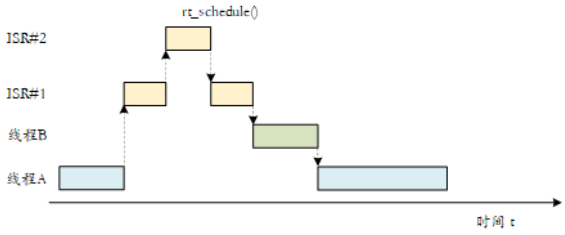
在用户中断服务程序（ISR）中，分为两种情况:

- 第一种情况是不进行线程切换，这种情况下会进行任务中断上下文 `TaskIrqContext` 切换，用户中断服务程序和中断后续程序运行完毕后退出中断模式，返回被中断的线程。
- 另一种情况是，在中断处理过程中需要进行线程切换，这种切换还会进行任务上下文 `TaskContext` 的切换。

具体下面办事的人把事办完。

中断嵌套

在允许中断嵌套的情况下，在执行中断服务程序的过程中，如果出现高优先级的中断，当前中断服务程序的执行将被打断，以执行高优先级中断的中断服务程序，当高优先级中断的处理完成后，被打断的中断服务程序才又得到继续执行，如果需要进行线程调度，线程的上下文切换将在所有中断处理程序都运行结束时才发生，如下图所示。



先把西施娘娘的事停了，现如花娘娘杀到，优先级高，老奴安排皇上先办如花娘娘，再接着办西施娘娘。奴才担心皇上这身子骨吃不吃得消。

中断共享

当外设较少时，可以实现一个外设对应一个中断号，但为了支持更多的硬件设备，可以让多个设备共享一个中断号，共享同一个中断号的中断处理程序形成一个链表。当外部设备产生中断申请时，系统会遍历执行中断号对应的中断处理程序链表直到找到对应设备的中断处理程序。在遍历执行过程中，各中断处理程序可以通过检测设备ID，判断是否是这个中断处理程序对应的设备产生的中断。

简单一句话就是:共用一个折子，分别办多件事。

核间中断

属于SGI中断类型，对于多核系统，中断控制器允许一个CPU的硬件线程去中断其他CPU的硬件线程，这种方式被称为核间中断。核间中断的实现基础是多CPU内存共享，采用核间中断可以减少某个CPU负荷过大，有效提升系统效率。

```
typedef enum {//鸿蒙核间中断
    LOS_MP_IPI_WAKEUP， //唤醒CPU
    LOS_MP_IPI_SCHEDULE，//调度CPU
    LOS_MP_IPI_HALT， //停止CPU
```

```
} MP_IPI_TYPE;
```

可以看出CPU之间可以相互唤醒，调度，停止。

核间中断有点特殊，出现于多个皇上(CPU)的情况。 皇上之间可以相互使唤，停止工作。比如:A皇上通过海公公让B皇上休息。

功能API

|功能分类 |接口名 |描述 | |-|-|-| |创建和删除中断|LOS_HwiCreate |中断创建，注册中断号、中断触发模式、中断优先级、中断处理程序。中断被触发时，handleIrq会调用该中断处理程序| | |LOS_HwiDelete |删除中断| |打开和关闭中断|LOS_IntUnLock |打开当前处理器所有中断响应| | |LOS_IntLock |关闭当前处理器所有中断响应| | |LOS_IntRestore |恢复到使用LOS_IntLock关闭所有中断之前的状态| |使能和屏蔽中断|LOS_HwiDisable |中断屏蔽（通过设置寄存器，禁止CPU响应该中断）| | |LOS_HwiEnable |中断使能（通过设置寄存器，允许CPU响应该中断）| |设置中断优先级|LOS_HwiSetPriority|设置中断优先级| |触发中断 |LOS_HwiTrigger |触发中断（通过写中断控制器的相关寄存器模拟外部中断）| |清除寄存器状态|LOS_HwiClear |清除中断号对应的中断寄存器的状态位，此接口依赖中断控制器版本，非必需| |核间中断 |LOS_HwiSendIpi |向指定核发送核间中断，此接口依赖中断控制器版本和cpu架构，该函数仅在SMP模式下支持| |设置中断亲和性|LOS_HwiSetAffinity|设置中断的亲和性，即设置中断在固定核响应（该函数仅在SMP模式下支持）|

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

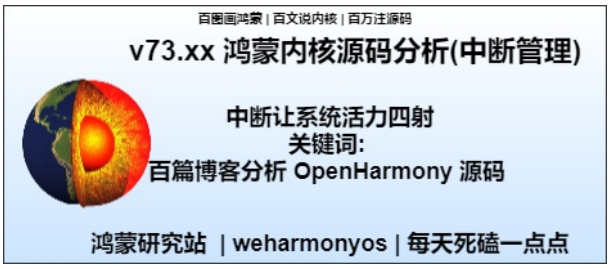
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

73_中断管理篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

硬件架构相关篇为:

- v65.01 鸿蒙内核源码分析(芯片模式) | 回顾芯片行业各位大佬
- v66.03 鸿蒙内核源码分析(ARM架构) | ARMv7 & Cortex(A|R|M)
- v67.01 鸿蒙内核源码分析(指令集) | CICS PK RICS
- v68.01 鸿蒙内核源码分析(协处理器) | CPU的好帮手
- v69.05 鸿蒙内核源码分析(工作模式) | 角色不同 责任不同
- v70.06 鸿蒙内核源码分析(寄存器) | 世界被它们玩出了花
- v71.03 鸿蒙内核源码分析(多核管理) | 并发真正的基础
- v72.05 鸿蒙内核源码分析(中断概念) | 海公公的日常工作
- v73.04 鸿蒙内核源码分析(中断管理) | 没中断太可怕

关于中断部分系列篇将用三篇详细说明整个过程。

- **中断概念篇** 中断概念很多，比如中断控制器，中断源，中断向量，中断共享，中断处理程序等等。本篇做一次整理。先了解透概念才好理解中断过程。用海公公打比方说明白中断各个概念。可前往查看。
- **中断管理篇(本篇)** 从中断初始化 `HalIrqInit` 开始，到注册中断的 `LOS_HwiCreate` 函数，到消费中断函数的 `HalIrqHandler` ，剖析鸿蒙内核实现中断的过程，很像设计模式中的观察者模式。
- **中断切换篇** 用自下而上的方式，从中断源头纯汇编代码往上跟踪代码细节。说清楚保存和恢复中断现场 `TaskIrqContext` 过程。

编译开关

系列篇编译平台为 **hi3516dv300**，整个工程可前往查看。 预编译处理过程会自动生成编译开关 **menuconfig.h** ，供编译阶段选择编译，可前往查看。

```
//....
#define LOSCFG_ARCH_ARM_VER "armv7-a"
#define LOSCFG_ARCH_CPU "cortex-a7"
#define LOSCFG_PLATFORM "hi3516dv300"
#define LOSCFG_PLATFORM_BSP_GIC_V2 1
#define LOSCFG_PLATFORM_ROOTFS 1
#define LOSCFG_KERNEL_CPPSUPPORT 1
#define LOSCFG_HW_RANDOM_ENABLE 1
#define LOSCFG_ARCH_CORTEX_A7 1
#define LOSCFG_DRIVERS_HDF_PLATFORM_RTC 1
#define LOSCFG_DRIVERS_HDF_PLATFORM_UART 1
```

中断初始化

hi3516dv300 中断控制器选择了 `LOSCFG_PLATFORM_BSP_GIC_V2` ，对应代码为 **gic_v2.c** GIC（Generic Interrupt Controller）是ARM公司提供的一个通用的中断控制器。 看这种代码因为涉及硬件部分，需要对照ARM中断控制器 gic_v2.pdf文档看。可前往地址下载查看。

```
//硬件中断初始化
VOID HalIrqInit(VOID)
{
    UINT32 i;
```

```
    /* set externel interrupts to be level triggered， active low. */ //将外部中断设置为电平触发，低电平激活
    for (i = 32; i < OS_HWI_MAX_NUM; i += 16) {
        GIC_REG_32(GICD_ICFGR(i / 16)) = 0;
    }

    /* set externel interrupts to CPU 0 */ //将外部中断设置为CPU 0
    for (i = 32; i < OS_HWI_MAX_NUM; i += 4) {
        GIC_REG_32(GICD_ITARGETSR(i / 4)) = 0x01010101;
    }

    /* set priority on all interrupts */ //设置所有中断的优先级
    for (i = 0; i < OS_HWI_MAX_NUM; i += 4) {
        GIC_REG_32(GICD_IPRIORITYR(i / 4)) = GICD_INT_DEF_PRI_X4;
    }

    /* disable all interrupts. */   //禁用所有中断。
    for (i = 0; i < OS_HWI_MAX_NUM; i += 32) {
        GIC_REG_32(GICD_ICENABLER(i / 32)) = ~0;
    }

    HalIrqInitPercpu();//初始化当前CPU中断信息

    /* enable gic distributor control */
    GIC_REG_32(GICD_CTLR) = 1; //使能分发中断寄存器，该寄存器作用是允许给CPU发送中断信号

#if (LOSCFG_KERNEL_SMP == YES)
    /* register inter-processor interrupt *///注册核间中断，啥意思？就是CPU各核直接可以发送中断信号
    //处理器间中断允许一个CPU向系统其他的CPU发送中断信号，处理器间中断（IPI）不是通过IRQ线传输的，而是作为信号直接放在连接所有CPU本地APIC的总线上
    LOS_HwiCreate(LOS_MP_IPI_WAKEUP， 0xa0， 0， OsMpWakeHandler， 0);//注册唤醒CPU的中断处理函数
    LOS_HwiCreate(LOS_MP_IPI_SCHEDULE， 0xa0， 0， OsMpScheduleHandler， 0);//注册调度CPU的中断处理函数
    LOS_HwiCreate(LOS_MP_IPI_HALT， 0xa0， 0， OsMpScheduleHandler， 0);//注册停止CPU的中断处理函数
#endif
}
//给每个CPU core初始化硬件中断
VOID HalIrqInitPercpu(VOID)
{
    /* unmask interrupts */ //取消中断屏蔽
    GIC_REG_32(GICC_PMR) = 0xFF;

    /* enable gic cpu interface */ //启用gic cpu接口
    GIC_REG_32(GICC_CTLR) = 1;
}
```

解读

- 上来四个循环，是对中断控制器寄存器组的初始化，也就是驱动程序，驱动程序是配置硬件寄存器的过程。寄存器分通用和专用寄存器。下图为gic_v2 的寄存器功能 ，这里对照代码和datasheet重点说下中断配置寄存器(`GICD_ICFGRn`)

GICC_AHPPIR	*Aliased Highest Priority Pending Interrupt Register, GICC_AHPPIR* on page 4-148
GICC_BPR	*Binary Point Register, GICC_BPR* on page 4-133
GICC_CTLR	*CPU Interface Control Register, GICC_CTLR* on page 4-125
GICC_DIR	*Deactivate Interrupt Register, GICC_DIR* on page 4-153
GICC_EOIR	*End of Interrupt Register, GICC_EOIR* on page 4-138
GICC_HPPIR	*Highest Priority Pending Interrupt Register, GICC_HPPIR* on page 4-143
GICC_IAR	*Interrupt Acknowledge Register, GICC_IAR* on page 4-135
GICC_IIDR	*CPU Interface Identification Register, GICC_IIDR* on page 4-152
GICC_NSAPRn	*Non-secure Active Priorities Registers, GICC_NSAPRn* on page 4-151
GICC_PMR	*Interrupt Priority Mask Register, GICC_PMR* on page 4-131
GICC_RPR	*Running Priority Register, GICC_RPR* on page 4-142
GICD_CPENDSGIRn	*SGI Clear-Pending Registers, GICD_CPENDSGIRn* on page 4-115
GICD_CTLR	*Distributor Control Register, GICD_CTLR* on page 4-85
GICD_ICACTIVERn	*Interrupt Clear-Active Registers, GICD_ICACTIVERn* on page 4-103
GICD_ICENABLERn	*Interrupt Clear-Enable Registers, GICD_ICENABLERn* on page 4-95
GICD_ICFGRn	*Interrupt Configuration Registers, GICD_ICFGRn* on page 4-109
GICD_ICPENDRn	*Interrupt Clear-Pending Registers, GICD_ICPENDRn* on page 4-99
GICD_IGROUPRn	*Interrupt Group Registers, GICD_IGROUPRn* on page 4-91
GICD_IIDR	*Distributor Implementer Identification Register, GICD_IIDR* on page 4-90
GICD_IPRIORITYRn	*Interrupt Priority Registers, GICD_IPRIORITYRn* on page 4-104
GICD_ISACTIVERn	*Interrupt Set-Active Registers, GICD_ISACTIVERn* on page 4-102
GICD_ISENABLERn	*Interrupt Set-Enable Registers, GICD_ISENABLERn* on page 4-93
GICD_ISPENDRn	*Interrupt Set-Pending Registers, GICD_ISPENDRn* on page 4-97
GICD_ITARGETSRn	*Interrupt Processor Targets Registers, GICD_ITARGETSRn* on page 4-106

- 以下是GICD_ICFGRn的介绍

> The GICD_ICFGRs provide a 2-bit Int_config field for each interrupt supported by the GIC. This field identifies whether the corresponding interrupt is edge-triggered or level-sensitive

> GICD_ICFGRs为GIC支持的每个中断提供一个2位配置字段。此字段标识相应的中断是边缘触发的还是电平触发的

```
0xC00 - 0xCFC GICD_ICFGRn RW IMPLEMENTATION DEFINED Interrupt Configuration Registers
#define GICD_ICFGR(n)               (GICD_OFFSET + 0xc00 + (n) * 4) /* Interrupt Configuration Registers */  //中断配置寄存器
```

如此一个32位寄存器可以记录16个中断的信息，这也是代码中出现 `GIC_REG_32(GICD_ICFGR(i / 16))` 的原因。

- GIC-v2支持三种类型的中断
  - PPI：私有外设中断(Private Peripheral Interrupt)，是每个CPU私有的中断。最多支持16个PPI中断，硬件中断号从ID16~ID31。PPI通常会送达到指定的CPU上，应用场景有CPU本地时钟。
  - SPI：公用外设中断(Shared Peripheral Interrupt)，最多可以支持988个外设中断，硬件中断号从ID32~ID1019。
  - SGI：软件触发中断(Software Generated Interrupt)通常用于多核间通讯，最多支持16个SGI中断，硬件中断号从ID0~ID15。SGI通常在内核中被用作 IPI 中断(inter-processor interrupts)，并会送达到系统指定的CPU上，函数的最后就注册了三个核间中断的函数。

```
typedef enum {//核间中断
```

```
    LOS_MP_IPI_WAKEUP， //唤醒CPU
    LOS_MP_IPI_SCHEDULE，//调度CPU
    LOS_MP_IPI_HALT， //停止CPU
} MP_IPI_TYPE;
```

中断相关的结构体

```
size_t g_intCount[LOSCFG_KERNEL_CORE_NUM] = {0};//记录每个CPUcore的中断数量
HwiHandleForm g_hwiForm[OS_HWI_MAX_NUM];//中断注册表 @note_why 用 form 来表示？有种写 HTML的感觉 :P
STATIC CHAR *g_hwiFormName[OS_HWI_MAX_NUM] = {0};//记录每个硬中断的名称
STATIC UINT32 g_hwiFormCnt[OS_HWI_MAX_NUM] = {0};//记录每个硬中断的总数量
STATIC UINT32 g_curIrqNum = 0; //记录当前中断号
typedef VOID (*HWI_PROC_FUNC)(VOID); //中断函数指针
typedef struct tagHwiHandleForm {
    HWI_PROC_FUNC pfnHook; //中断处理函数
    HWI_ARG_T uwParam;  //中断处理函数参数
    struct tagHwiHandleForm *pstNext; //节点，指向下一个中断，用于共享中断的情况
} HwiHandleForm;

typedef struct tagIrqParam { //中断参数
    int swIrq;  // 软件中断
    VOID *pDevId; // 设备ID
    const CHAR *pName; //名称
} HwiIrqParam;
```

注册硬中断

```
/*****************************************************************************
 创建一个硬中断
 中断创建，注册中断号、中断触发模式、中断优先级、中断处理程序.中断被触发时，
 handleIrq会调用该中断处理程序
*****************************************************************************/
LITE_OS_SEC_TEXT_INIT UINT32 LOS_HwiCreate(HWI_HANDLE_T hwiNum， //硬中断句柄编号 默认范围[0-127]
                                           HWI_PRIOR_T hwiPrio， //硬中断优先级
                                           HWI_MODE_T hwiMode， //硬中断模式 共享和非共享
                                           HWI_PROC_FUNC hwiHandler，//硬中断处理函数
                                           HwiIrqParam *irqParam) //硬中断处理函数参数
{
    UINT32 ret;

    (VOID)hwiPrio;
    if (hwiHandler == NULL) {//中断处理函数不能为NULL
        return OS_ERRNO_HWI_PROC_FUNC_NULL;
    }
    if ((hwiNum > OS_USER_HWI_MAX) || ((INT32)hwiNum < OS_USER_HWI_MIN)) {//中断数区间限制 [32，96]
        return OS_ERRNO_HWI_NUM_INVALID;
    }

#ifdef LOSCFG_NO_SHARED_IRQ //不支持共享中断
    ret = OsHwiCreateNoShared(hwiNum， hwiMode， hwiHandler， irqParam);
#else
    ret = OsHwiCreateShared(hwiNum， hwiMode， hwiHandler， irqParam);
#endif
    return ret;
}
//创建一个共享硬件中断，共享中断就是一个中断能触发多个响应函数
STATIC UINT32 OsHwiCreateShared(HWI_HANDLE_T hwiNum， HWI_MODE_T hwiMode，
                                HWI_PROC_FUNC hwiHandler， const HwiIrqParam *irqParam)
{
    UINT32 intSave;
    HwiHandleForm *hwiFormNode = NULL;
    HwiHandleForm *hwiForm = NULL;
    HwiIrqParam *hwiParam = NULL;
    HWI_MODE_T modeResult = hwiMode & IRQF_SHARED;

    if (modeResult && ((irqParam == NULL) || (irqParam->pDevId == NULL))) {
```

```
        return OS_ERRNO_HWI_SHARED_ERROR;
    }

    HWI_LOCK(intSave);//中断自旋锁

    hwiForm = &g_hwiForm[hwiNum];//获取中断处理项
    if ((hwiForm->pstNext != NULL) && ((modeResult == 0) || (!(hwiForm->uwParam & IRQF_SHARED)))) {
        HWI_UNLOCK(intSave);
        return OS_ERRNO_HWI_SHARED_ERROR;
    }

    while (hwiForm->pstNext != NULL) {//pstNext指向 共享中断的各处理函数节点，此处一直撸到最后一个
        hwiForm = hwiForm->pstNext;//找下一个中断
        hwiParam = (HwiIrqParam *)(hwiForm->uwParam);//获取中断参数，用于检测该设备ID是否已经有中断处理函数
        if (hwiParam->pDevId == irqParam->pDevId) {//设备ID一致时，说明设备对应的中断处理函数已经存在了。
            HWI_UNLOCK(intSave);
            return OS_ERRNO_HWI_ALREADY_CREATED;
        }
    }

    hwiFormNode = (HwiHandleForm *)LOS_MemAlloc(m_aucSysMem0， sizeof(HwiHandleForm));//创建一个中断处理节点
    if (hwiFormNode == NULL) {
        HWI_UNLOCK(intSave);
        return OS_ERRNO_HWI_NO_MEMORY;
    }

    hwiFormNode->uwParam = OsHwiCpIrqParam(irqParam);//获取中断处理函数的参数
    if (hwiFormNode->uwParam == LOS_NOK) {
        HWI_UNLOCK(intSave);
        (VOID)LOS_MemFree(m_aucSysMem0， hwiFormNode);
        return OS_ERRNO_HWI_NO_MEMORY;
    }

    hwiFormNode->pfnHook = hwiHandler;//绑定中断处理函数
    hwiFormNode->pstNext = (struct tagHwiHandleForm *)NULL;//指定下一个中断为NULL，用于后续遍历找到最后一个中断项(见于以上 while (hwiForm->p
    hwiForm->pstNext = hwiFormNode;//共享中断

    if ((irqParam != NULL) && (irqParam->pName != NULL)) {
        g_hwiFormName[hwiNum] = (CHAR *)irqParam->pName;
    }

    g_hwiForm[hwiNum].uwParam = modeResult;

    HWI_UNLOCK(intSave);
    return LOS_OK;
}
```

解读

- 内核将硬中断进行编号，如:

```
    #define NUM_HAL_INTERRUPT_TIMER0      33
    #define NUM_HAL_INTERRUPT_TIMER1      33
    #define NUM_HAL_INTERRUPT_TIMER2      34
    #define NUM_HAL_INTERRUPT_TIMER3      34
    #define NUM_HAL_INTERRUPT_TIMER4      35
    #define NUM_HAL_INTERRUPT_TIMER5      35
    #define NUM_HAL_INTERRUPT_TIMER6      36
    #define NUM_HAL_INTERRUPT_TIMER7      36
    #define NUM_HAL_INTERRUPT_DMAC        60
    #define NUM_HAL_INTERRUPT_UART0       38
    #define NUM_HAL_INTERRUPT_UART1       39
    #define NUM_HAL_INTERRUPT_UART2       40
    #define NUM_HAL_INTERRUPT_UART3       41
    #define NUM_HAL_INTERRUPT_UART4       42
    #define NUM_HAL_INTERRUPT_TIMER       NUM_HAL_INTERRUPT_TIMER4
```

  例如:时钟节拍处理函数 `OsTickHandler` 就是在 `HalClockInit` 中注册的

```
//硬时钟初始化
VOID HalClockInit(VOID)
{
  // ...
  (void)LOS_HwiCreate(NUM_HAL_INTERRUPT_TIMER， 0xa0， 0， OsTickHandler， 0);//注册OsTickHandler到中断向量表
}
//节拍中断处理函数 ，鸿蒙默认10ms触发一次
  LITE_OS_SEC_TEXT VOID OsTickHandler(VOID)
  {
     UINT32 intSave;
     TICK_LOCK(intSave);//tick自旋锁
     g_tickCount[ArchCurrCpuid()]++;// 累加当前CPU核tick数
     TICK_UNLOCK(intSave);
     OsTimesliceCheck();//时间片检查
     OsTaskScan(); /* task timeout scan *///扫描超时任务 例如:delay(300)
     #if (LOSCFG_BASE_CORE_SWTMR == YES)
        OsSwtmrScan();//扫描定时器，查看是否有超时定时器，加入队列
     #endif
  }
```

- 鸿蒙是支持中断共享的，在`OsHwiCreateShared`中，将函数注册到`g_hwiForm`中。中断向量完成注册后，就是如何触发和回调的问题。触发在v08.xx 鸿蒙内核源码分析(总目录) 中断切换篇中已经讲清楚，触发是从底层汇编向上调用，调用的C函数就是`HalIrqHandler`

中断怎么触发的？

分两种情况:

- 通过硬件触发，比如按键，USB的插拔这些中断源向中断控制器发送电信号(高低电平触发或是上升/下降沿触发)，中断控制器经过过滤后将信号发给对应的CPU处理，通过硬件改变PC和CPSR寄存值，直接跳转到中断向量(固定地址)执行。

```
  b   reset_vector           @开机代码
  b   _osExceptUndefInstrHdl  @异常处理之CPU碰到不认识的指令
  b   _osExceptSwiHdl   @异常处理之:软中断
  b   _osExceptPrefetchAbortHdl @异常处理之:取指异常
  b   _osExceptDataAbortHdl  @异常处理之:数据异常
  b   _osExceptAddrAbortHdl  @异常处理之:地址异常
  b   OsIrqHandler    @异常处理之:硬中断
  b   _osExceptFiqHdl    @异常处理之:快中断
```

- 通过软件触发，常见于核间中断的情况， 核间中断指的是几个CPU之间相互通讯的过程。以下为某一个CPU向其他CPU(可以是多个)发起让这些CPU重新调度`LOS_MpSchedule`的中断请求信号。最终是写了中断控制器的`GICD_SGIR`寄存器，这是一个由软件触发中断的寄存器。中断控制器会将请求分发给对应的CPU处理中断，即触发了`OsIrqHandler`。

```
//给参数CPU发送调度信号
  VOID LOS_MpSchedule(UINT32 target)//target每位对应CPU core
  {
     UINT32 cpuid = ArchCurrCpuid();
     target &= ~(1U << cpuid);//获取除了自身之外的其他CPU
     HalIrqSendIpi(target， LOS_MP_IPI_SCHEDULE);//向目标CPU发送调度信号，核间中断(Inter-Processor Interrupts)，IPI
  }
  //SGI软件触发中断(Software Generated Interrupt)通常用于多核间通讯
  STATIC VOID GicWriteSgi(UINT32 vector， UINT32 cpuMask， UINT32 filter)
  {
     UINT32 val = ((filter & 0x3) << 24) | ((cpuMask & 0xFF) << 16) |
                (vector & 0xF);

     GIC_REG_32(GICD_SGIR) = val;//写SGI寄存器
  }
  //向指定核发送核间中断
  VOID HalIrqSendIpi(UINT32 target， UINT32 ipi)
  {
     GicWriteSgi(ipi， target， 0);
  }
```

中断统一处理入口函数 HalIrqHandler

```
//硬中断统一处理函数，这里由硬件触发，调用见于 ..\arch\arm\arm\src\los_dispatch.S
VOID HalIrqHandler(VOID)
{
    UINT32 iar = GIC_REG_32(GICC_IAR);//从中断确认寄存器获取中断ID号
    UINT32 vector = iar & 0x3FFU;//计算中断向量号
    /*
     * invalid irq number， mainly the spurious interrupts 0x3ff，
     * gicv2 valid irq ranges from 0~1019， we use OS_HWI_MAX_NUM
     * to do the checking.
     */
    if (vector >= OS_HWI_MAX_NUM) {
        return;
    }
    g_curIrqNum = vector;//记录当前中断ID号
    OsInterrupt(vector);//调用上层中断处理函数
    /* use orignal iar to do the EOI */
    GIC_REG_32(GICC_EOIR) = iar;//更新中断结束寄存器
}
VOID OsInterrupt(UINT32 intNum)//中断实际处理函数
{
    HwiHandleForm *hwiForm = NULL;
    UINT32 *intCnt = NULL;

    intCnt = &g_intCount[ArchCurrCpuid()];//当前CPU的中断总数量 ++
    *intCnt = *intCnt + 1;//@note_why 这里没看明白为什么要 +1

#ifdef LOSCFG_CPUP_INCLUDE_IRQ //开启查询系统CPU的占用率的中断
    OsCpupIrqStart();//记录本次中断处理开始时间
#endif

#ifdef LOSCFG_KERNEL_TICKLESS
    OsTicklessUpdate(intNum);
#endif
    hwiForm = (&g_hwiForm[intNum]);//获取对应中断的实体
#ifndef LOSCFG_NO_SHARED_IRQ //如果没有定义不共享中断 ，意思就是如果是共享中断
    while (hwiForm->pstNext != NULL) { //一直撸到最后
        hwiForm = hwiForm->pstNext;//下一个继续撸
#endif
        if (hwiForm->uwParam) {//有参数的情况
            HWI_PROC_FUNC2 func = (HWI_PROC_FUNC2)hwiForm->pfnHook;//获取回调函数
            if (func != NULL) {
                UINTPTR *param = (UINTPTR *)(hwiForm->uwParam);
                func((INT32)(*param)， (VOID *)(*(param + 1)));//运行带参数的回调函数
            }
        } else {//木有参数的情况
            HWI_PROC_FUNC0 func = (HWI_PROC_FUNC0)hwiForm->pfnHook;//获取回调函数
            if (func != NULL) {
                func();//运行回调函数
            }
        }
#ifndef LOSCFG_NO_SHARED_IRQ
    }
#endif
    ++g_hwiFormCnt[intNum];//中断号计数器总数累加

    *intCnt = *intCnt - 1; //@note_why 这里没看明白为什么要 -1
#ifdef LOSCFG_CPUP_INCLUDE_IRQ //开启查询系统CPU的占用率的中断
    OsCpupIrqEnd(intNum);//记录中断处理时间完成时间
#endif
}
```

解读

统一中断处理函数是一个通过一个中断号去找到注册函数的过程，分四步走:

- 第一步:取号，这号是由中断控制器的 `GICC_IAR` 寄存器提供的，这是一个专门保存当前中断号的寄存器。
- 第二步:从注册表 `g_hwiForm` 中查询注册函数，同时取出参数。
- 第三步:执行函数，也就是回调注册函数，分有参和无参两种情况 `func(...)` ，在中断共享的情况，注册函数会指向 next 注册函数 `pstNext` ，依

次执行回调函数，这是中断共享的实现细节。

```
typedef struct tagHwiHandleForm {
    HWI_PROC_FUNC pfnHook; //中断处理函数
    HWI_ARG_T uwParam;  //中断处理函数参数
    struct tagHwiHandleForm *pstNext; //节点，指向next中断，用于共享中断的情况
} HwiHandleForm;
```

- 第四步:销号，本次中断完成了就需要消除记录，中断控制器也有专门的销号寄存器`GICC_EOIR`
- 另外的是一些统一数据，每次中断号处理内核都会记录次数，和耗时，以便定位/跟踪/诊断问题。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

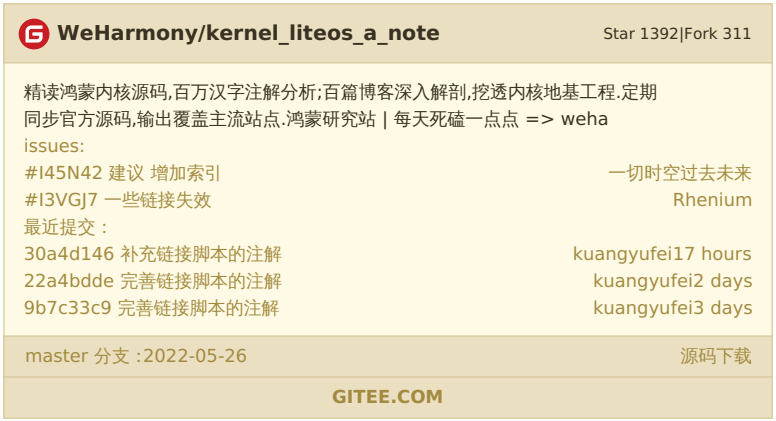
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

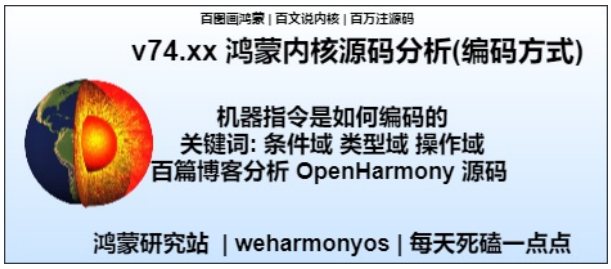


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

74_编码方式篇

本篇关键词：指令格式、条件域、类型域、操作域、数据指令、访存指令、跳转指令、SVC(软件中断)



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

本篇说清楚 `ARM` 指令是如何被编码的，机器指令由哪些部分构成，指令有哪些类型，每种类型的语法又是怎样的？

代码案例 | C -> 汇编 -> 机器指令

看一段C语言编译(clang)成的最后的机器指令(armv7)

```
int main(){
    int a = 0;
    if( a != 1)
        a = 2*a + 1;
    return a;
}

 生成汇编代码如下:

    main:
60c: sub sp, sp, #8
610: mov r0, #0
614: str r0, [sp, #4]
618: str r0, [sp]
61c: ldr r0, [sp]
620: cmp r0, #1
624: beq 640 <main+0x34>
628: b 62c <main+0x20>
62c: ldr r1, [sp]
630: mov r0, #1
634: orr r0, r0, r1, lsl #1
638: str r0, [sp]
63c: b 640 <main+0x34>
640: ldr r0, [sp]
644: add sp, sp, #8
648: bx lr
```

汇编代码对应的机器指令如下图所示:

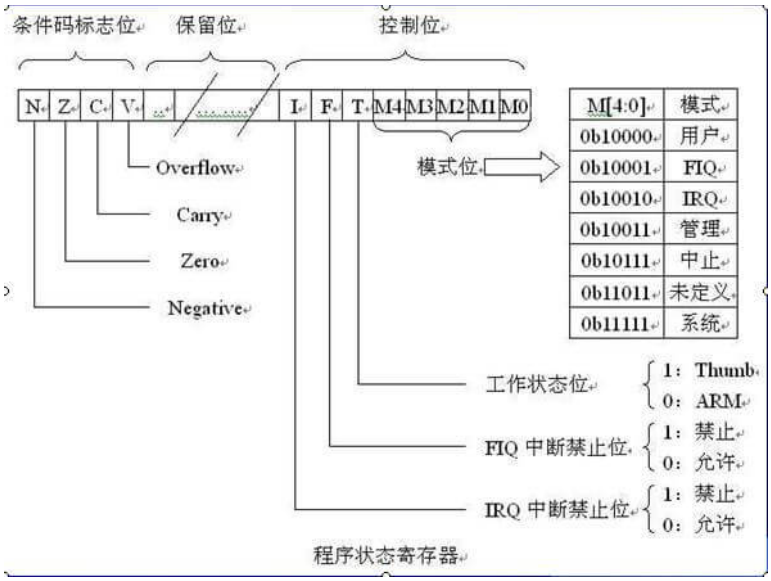
```
main:
 e24dd008
sub     sp, sp, #8
 e3a00000
mov     r0, #0
 e58d0004
str     r0, [sp, #4]
 e58d0000
str     r0, [sp]
 e59d0000
ldr     r0, [sp]
 e3500001                                来源： 鸿蒙内核源码分析（编码方式篇）| weharmonyos.com
cmp     r0, #1                          指令集-编译器： armv7-a clang
 0a000005                               源码：
beq     640 <main+0x34>                 int main(){
 eaffffff                                  int a = 0;
b   62c <main+0x20>                        if( a != 1)
 e59d1000                                     a = 2*a + 1;
ldr     r1, [sp]                           return a;
 e3a00001                               }
mov     r0, #1
 e1800081
orr     r0, r0, r1, lsl #1
 e58d0000
str     r0, [sp]
 eaffffff
b   640 <main+0x34>
 e59d0000
ldr     r0, [sp]
 e28dd008
add     sp, sp, #8
 e12fff1e
bx lr
```

便于后续分析，将以上代码整理成如下表格

汇编代码	机器指令(十六进制表示)	机器指令(二进制表示)
sub sp, sp, #8	e24dd008	1110 0010 0100 1101 1101 0000 0000 1000
mov r0, #0	e3a00000	1110 0011 1010 0000 0000 0000 0000 0000
str r0, [sp, #4]	e58d0004	1110 0101 1000 1101 0000 0000 0000 0100
str r0, [sp]	e58d0000	1110 0101 1000 1101 0000 0000 0000 0000
ldr r0, [sp]	e59d0000	1110 0101 1001 1101 0000 0000 0000 0000
cmp r0, #1	e3500001	1110 0011 0101 0000 0000 0000 0000 0001
beq 640 <main+0x34>	0a000005	0000 1010 0000 0000 0000 0000 0000 0101
b 62c <main+0x20>	eaffffff	1110 1010 1111 1111 1111 1111 1111 1111
ldr r1, [sp]	e59d1000	1110 0101 1001 1101 0001 0000 0000 0010
mov r0, #1	e3a00002	1110 0011 1010 0000 0000 0000 0000 0001
orr r0, r0, r1, lsl #1	e1800081	1110 0001 1000 0000 0000 0000 1000 0001
str r0, [sp]	e58d0000	1110 0101 1000 1101 0000 0000 0000 0000
b 640 <main+0x34>	eaffffff	1110 1010 1111 1111 1111 1111 1111 1111
ldr r0, [sp]	e59d1000	1110 0101 1001 1101 0001 0000 0000 0000
add sp, sp, #8	e28dd008	1110 0010 1000 1101 1101 0000 0000 1000
bx lr	e12fff1e	1110 0001 0010 1111 1111 1111 0001 1110

CPSR寄存器

在理解本篇之前需了解下 CPSR 寄存器的高 4 位 [31,28] 表达的含义。关于寄存器的详细介绍可翻看 系列篇的 **(寄存器篇)**

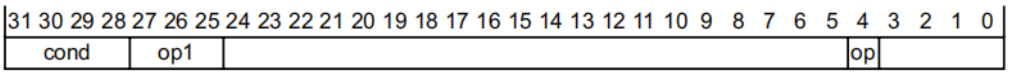


N、Z、C、V 均为条件码标志位。它们的内容可被算术或逻辑运算的结果所改变，并且可以决定某条指令是否被执行!意义重大!

- CPSR 的第 31 位是 N ，符号标志位。它记录相关指令执行后，其结果是否为负。 如果为负 N = 1 ，如果是非负数 N = 0 。
- CPSR 的第 30 位是 Z ， 0 标志位。它记录相关指令执行后，其结果是否为 0 。 如果结果为 0 。那么 Z = 1 。如果结果不为 0 ，那么 Z = 0 。
- CPSR 的第 29位 是 C ，进位标志位 (Carry) 。一般情况下，进行无符号数的运算。 加法运算：当运算结果产生了进位时（无符号数溢出），C=1 ，否则 C=0 。 减法运算（包括 CMP ）：当运算时产生了借位时（无符号数溢出），C=0 ，否则 C=1 。
- CPSR 的第 28 位是 V ，溢出标志位(Overflow)。在进行有符号数运算的时候， 如果超过了机器所能标识的范围，称为溢出。

指令格式

ARM 指令流是一连串的字对齐的四字节指令流。每个 ARM 指令是一个单一的 32 位字(4 字节)，如**图(3)**：



解读 图为 ARM 指令的编码一级格式，所有的指令都必须符合一级格式，分成三部分：

- 条件域: cond[31:28] 表示，条件域会影响 CPSR 的条件码 N、Z、C、V 标志位。
- 类型域: op1[27:25] ， op[4] ， arm 将指令分成了六大类型 。
- 操作域: 剩下的 [24:5] ， [4:0] 即图中的空白位/保留位，这是留给下级自由发挥的，不同的类型对这些保留位有不同的定义。可以理解为因类型变化而变化的二级格式。
- 那有了二级格式会不会有三级格式？答案是必须有， 二级格式只会对保留位定义部分位，会留一部分给具体的指令格式自由发挥。
- 一定要理解这种层次结构才能理解 ARM 指令集的设计总思路，因为**RISC（精简指令集）** 的指令长度是固定的 16/32/64 位，以 32 位为例，所有的指令设计必须全用 32 位来表示，如果只有一层结构是难以满足众多的指令设计需求的，要灵活有包容就得给适当的空间发挥。

条件域

cond 为条件域，每一条可条件执行的条件指令都有 4 位的条件位域， 2^4 能表示 16 种条件

cond	助记符	含义（整型）	含义（浮点型）	条件标志
0000	EQ	相等	相等	Z == 1
0001	NE	不等	不等或无序	Z == 0
0010	CS	进位	大于等于或无序	C == 1
0011	CC	进位清除	小于	C == 0
0100	MI	减、负数	小于	N == 1
0101	PL	加、正数或 0	大于等于或无序	N == 0
0110	VS	溢出	无序	V == 1

0111	VC	未溢出	有序	V == 0
1000	HI	无符号大于	大于或无序	C == 1 and Z == 0
1001	LS	无符号小于或等于	小于或等于	C == 0 or Z == 1
1010	GE	有符号大于或等于	大于或等于	N == V
1011	LT	有符号小于	小于或无序	N != V
1100	GT	有符号大于	大于	Z == 0 and N ==V
1101	LE	有符号大于或等于	小于等于或无序	Z == 1 or N != V
1110	无	无条件	无条件	任何

- 大部分的指令都是 `1110 = e`，无条件执行指令，只要看到 `e` 开头的机器指令都属于这类

```
beq 640 <main+0x34> // 机器码 0a000005 <=> 0000 1010 0000 0000 0000 0000 0000 0101
                                           0000 EQ Equal(相等) Z == 1
```

类型域

图(3) 的 `op1` 域位于 `bits[27:25]`，占三位；`op` 域位于 `bit[4]`，占一位。它们的取值组合在一起，决定指令所属的分类（Instruction Class），其值对应的关系如下

```
op1   op   指令类型
00x   -    数据处理以及杂项指令
010   -    load/store word类型 或者 unsigned byte
011   0    同上
011   1    媒体接口指令
10x   -    跳转指令和块数据操作指令，块数据操作指令指 STMDA 这类，连续内存操作。
11x   -    协处理器指令和 svc 指令，包括高级的 SIMD 和浮点指令。
```

操作域

操作域是因类型变化而变化的二级格式 ，作用于保留位。包含

00x | 数据处理类指令

A5.2.1 Data-processing (register)

The encoding of ARM data-processing (register) instructions is:

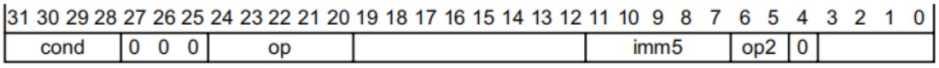


Table A5-3 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-3 Data-processing (register) instructions

op	op2	imm5	Instruction	See
0000x	-	-	Bitwise AND	*AND (register)* on page A8-324
0001x	-	-	Bitwise Exclusive OR	*EOR (register)* on page A8-385
0010x	-	-	Subtract	*SUB (register)* on page A8-713
0011x	-	-	Reverse Subtract	*RSB (register)* on page A8-577
0100x	-	-	Add	*ADD (register, ARM)* on page A8-310
0101x	-	-	Add with Carry	*ADC (register)* on page A8-300
0110x	-	-	Subtract with Carry	*SBC (register)* on page A8-595
0111x	-	-	Reverse Subtract with Carry	*RSC (register)* on page A8-583
10xx0	-	-	See *Data-processing and miscellaneous instructions* on page A5-194	
10001	-	-	Test	*TST (register)* on page A8-747
10011	-	-	Test Equivalence	*TEQ (register)* on page A8-741
10101	-	-	Compare	*CMP (register)* on page A8-370
10111	-	-	Compare Negative	*CMN (register)* on page A8-364
1100x	-	-	Bitwise OR	*ORR (register)* on page A8-519
1101x	00	00000	Move	*MOV (register, ARM)* on page A8-489
		not 00000	Logical Shift Left	*LSL (immediate)* on page A8-469
	01	-	Logical Shift Right	*LSR (immediate)* on page A8-473
	10	-	Arithmetic Shift Right	*ASR (immediate)* on page A8-328
	11	00000	Rotate Right with Extend	*RRX* on page A8-573
		not 00000	Rotate Right	*ROR (immediate)* on page A8-569
1110x	-	-	Bitwise Bit Clear	*BIC (register)* on page A8-340
1111x	-	-	Bitwise NOT	*MVN (register)* on page A8-507

- 上图为涉及数据处理指令的对应编码，由 `op[占5位]` 和 `op2[占2位]` 两项来确定指令的唯一性
- 一般情况下只需 `op` 指定唯一性，图中 `SUB` 指令对应为 `0010x` ，而代码案例中的第一句

```
sub sp, sp, #8  // 机器码 e24dd008 <=> 1110 001`0 0100` 1101 1101 0000 0000 1000
```

对应 `[24:20]` 位就是 `0 0100` ，从而 `CPU` 在译码阶段将其解析为 `SUB` 指令执行

- 需要用到 `op2` 的是 `MOV` 系列指令，包括逻辑/算术左移右移，例如:

```
mov r0, #0 //e3a00000 <=> 1110 0011 1010 0000 0000 0000 0000 0000
```

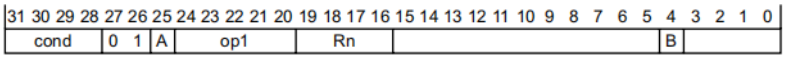
中的 `op = 1 1010` ， `op2 = 00` 对应 **MOV(register,ARM) on page A8-489** `00x` 中的 `x` 表示数据处理分两种情况

- `000` 无立即数参与(寄存器之间），图A5.2.1 表示了这种情况 `[27:25]= 000`
- `001` 有立即参与的运算，例如 `mov r0, #0` 中的 `[27:25]= 001`，此处未展示图，可前往 **ARM体系架构参考手册.pdf** 翻看

010 | 加载存储指令

A5.3 Load/store word and unsigned byte

The encoding of ARM load/store word and unsigned byte instructions is:



These instructions have either A == 0 or B == 0. For instructions with A == 1 and B == 1, see *Media instructions* on page A5-207.

Otherwise, Table A5-15 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-15 Single data transfer instructions

A	op1	B	Rn	Instruction	See
0	xx0x0 not 0x010	-	-	Store Register	*STR (immediate, ARM)* on page A8-675
1	xx0x0 not 0x010	0	-	Store Register	*STR (register)* on page A8-677
0	0x010	-	-	Store Register Unprivileged	*STRT* on page A8-707
1	0x010	0	-		
0	xx0x1 not 0x011	-	not 1111	Load Register (immediate)	*LDR (immediate, ARM)* on page A8-409
			1111	Load Register (literal)	*LDR (literal)* on page A8-411
1	xx0x1 not 0x011	0	-	Load Register	*LDR (register, ARM)* on page A8-415
0	0x011	-	-	Load Register Unprivileged	*LDRT* on page A8-467
1	0x011	0	-		
0	xx1x0 not 0x110	-	-	Store Register Byte (immediate)	*STRB (immediate, ARM)* on page A8-681
1	xx1x0 not 0x110	0	-	Store Register Byte (register)	*STRB (register)* on page A8-683
0	0x110	-	-	Store Register Byte Unprivileged	*STRBT* on page A8-685
1	0x110	0	-		
0	xx1x1 not 0x111	-	not 1111	Load Register Byte (immediate)	*LDRB (immediate, ARM)* on page A8-419
			1111	Load Register Byte (literal)	*LDRB (literal)* on page A8-421
1	xx1x1 not 0x111	0	-	Load Register Byte (register)	*LDRB (register)* on page A8-423
0	0x111	-	-	Load Register Byte Unprivileged	*LDRBT* on page A8-425
1	0x111	0	-		

- `Load/store` 是一组内存访问指令，用来在 `ARM` 寄存器和内存之间进行数据传送， `ARM` 指令中有 `3` 种基本的数据传送指令
  - 单寄存器 `Load/Store` 内存访问指令（ `single register` ）：这些指令为ARM寄存器和存储器提供了更灵活的单数据项传送方式。数据可以使字节，16位半字或32位字
  - 多寄存器 `Load/Store` 内存访问指令：可以实现大量数据的同时传送，主要用于进程的进入和退出、保存和恢复工作寄存器以及复制寄存器中的一片（一块）数据
  - 寄存器交换指令（ `single register swap` ）：实现寄存器数据和内存数据进行交换，而且是在一条指令中完成，执行过程中不会受到中断干扰
- 出现在代码案例中的

```
str r0, [sp, #4] // 机器码 e58d0004 <=> 1110 0101 1000 1101 0000 0000 0000 0100
str r0, [sp]  // 机器码 e58d0000 <=> 1110 0101 1000 1101 0000 0000 0000 0000
                 将r0中的字数据写入以SP为地址的存储器中
ldr r0, [sp]  // 机器码 e59d0000 <=> 1110 0101 1001 1101 0000 0000 0000 0000
                 存储器地址地址为SP的数据读入r0 寄存器
```

`[27:25] = 010` 说明都属于这类指令，完成对内存的读写，包括 `LDR` 、 `LDRB` 、 `LDRH` 、 `STR` 、 `STRB` 、 `STRH` 六条指令。 `ldr` 为加载指令，但是加载到内存还是寄存器，这该怎么记？因为主角是 `CPU` ，加载有进来的意思，将内容加载至寄存器中。 `STR` 有出去的意思，将内容保

存到内存里。 `[sp]` 相当于 C 语言的 `*sp` ， `sp` 指向程序运行栈当前位置

- 具体可看 >> ARM的六条访存指令集---LDR、LDRB、LDRH、STR、STRB、STRH

010 | 多媒体指令

A5.4 Media instructions

The encoding of ARM media instructions is:

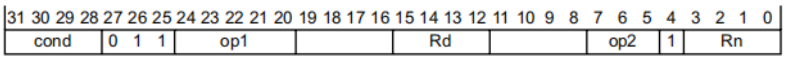


Table A5-16 shows the allocation of encodings in this space.

Other encodings in this space are UNDEFINED.

Table A5-16 Media instructions

op1	op2	Rd	Rn	cond	Instructions	See	Variant
000xx	-	-	-	-	-	*Parallel addition and subtraction, signed* on page A5-208	
001xx	-	-	-	-	-	*Parallel addition and subtraction, unsigned* on page A5-208	
01xxx	-	-	-	-	-	*Packing, unpacking, saturation, and reversal* on page A5-209	
10xxx	-	-	-	-	-	*Signed multiply, signed and unsigned divide* on page A5-211	
11000	000	1111	-	-	Unsigned Sum of Absolute Differences	*USAD8* on page A8-793	v6
	000	not 1111	-	-	Unsigned Sum of Absolute Differences and Accumulate	*USADA8* on page A8-795	v6
1101x	x10	-	-	-	Signed Bit Field Extract	*SBFX* on page A8-599	v6T2
1110x	x00	-	1111	-	Bit Field Clear	*BFC* on page A8-334	v6T2
			not 1111	-	Bit Field Insert	*BFI* on page A8-336	v6T2
1111x	x10	-	-	-	Unsigned Bit Field Extract	*UBFX* on page A8-757	v6T2
11111	111	-	-	1110	Permanently UNDEFINED	*UDF* on page A8-759	All[a]
				not 1110		-[a]	All

多媒体指令使用较少，但是它涉及指令却很多

10x | 跳转/分支/块数据处理 指令

A5.5 Branch, branch with link, and block data transfer

The encoding of ARM branch, branch with link, and block data transfer instructions is:

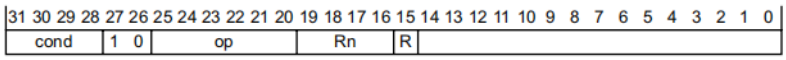


Table A5-21 shows the allocation of encodings in this space. These encodings are in all architecture variants.

Table A5-21 Branch, branch with link, and block data transfer instructions

op	R	Rn	Instructions	See
0000x0	-	-	Store Multiple Decrement After	*STMDA (STMED)* on page A8-667
0000x1	-	-	Load Multiple Decrement After	*LDMDA/LDMFA* on page A8-401
0010x0	-	-	Store Multiple Increment After	*STM (STMIA, STMEA)* on page A8-665
001001	-	-	Load Multiple Increment After	*LDM/LDMIA/LDMFD (ARM)* on page A8-399
001011	-	not 1101	Load Multiple Increment After	*LDM/LDMIA/LDMFD (ARM)* on page A8-399
		1101	Pop multiple registers	*POP (ARM)* on page A8-537
010000	-	-	Store Multiple Decrement Before	*STMDB (STMFD)* on page A8-669
010010	-	not 1101	Store Multiple Decrement Before	*STMDB (STMFD)* on page A8-669
	-	1101	Push multiple registers	*PUSH* on page A8-539
0100x1	-	-	Load Multiple Decrement Before	*LDMDB/LDMEA* on page A8-403
0110x0	-	-	Store Multiple Increment Before	*STMIB (STMFA)* on page A8-671
0110x1	-	-	Load Multiple Increment Before	*LDMIB/LDMED* on page A8-405
0xx1x0	-	-	Store Multiple (user registers)	*STM (User registers)* on page B9-1994
0xx1x1	0	-	Load Multiple (user registers)	*LDM (User registers)* on page B9-1974
	1	-	Load Multiple (exception return)	*LDM (exception return)* on page B9-1972
10xxxx	-	-	Branch	*B* on page A8-332
11xxxx	-	-	Branch with Link	*BL, BLX (immediate)* on page A8-346

- 出现在代码案例中的

```
beq 640 <main+0x34> // 机器码 0a000005 <=> 0000 1010 0000 0000 0000 0000 0000 0101
b 62c <main+0x20> // 机器码 eafffffff <=> 1110 1010 1111 1111 1111 1111 1111 1111
```

`[27:25] = 101` 说明都属于这类指令

- 听得很多的 `pop` ， `push` 也属于这类，成块的数据操作，例如 `push` 常用于将函数的所有参数一次性入栈。
- 内存 <> 寄存器 批量数据搬运指令 `STMDA (STMED)` `LDMDA/LDMF` 。

11x | 软中断/协处理器 指令

A5.6 Coprocessor instructions, and Supervisor Call

The encoding of ARM coprocessor instructions and the Supervisor Call instruction is:

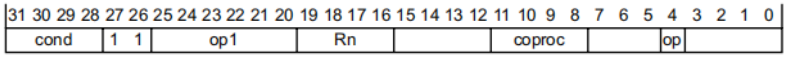


Table A5-22 shows the allocation of encodings in this space:

Table A5-22 Coprocessor instructions, and Supervisor Call

coproc	op1	op	Rn	Instructions	See	Variant
-	00000x	-	-	UNDEFINED	-	-
	11xxxx	-	-	Supervisor Call	*SVC (previously SWI)* on page A8-721	All
not 101x	0xxxx0 not 000x00	-	-	Store Coprocessor	*STC, STC2* on page A8-663	All
	0xxxx1 not 000x01	-	not 1111	Load Coprocessor (immediate)	*LDC, LDC2 (immediate)* on page A8-393	All
			1111	Load Coprocessor (literal)	*LDC, LDC2 (literal)* on page A8-395	All
	000100	-	-	Move to Coprocessor from two ARM core registers	*MCRR, MCRR2* on page A8-479	v5TE
	000101	-	-	Move to two ARM core registers from Coprocessor	*MRRC, MRRC2* on page A8-495	v5TE
	10xxxx	0	-	Coprocessor data operations	*CDP, CDP2* on page A8-356	All
	10xxx0	1	-	Move to Coprocessor from ARM core register	*MCR, MCR2* on page A8-477	All
	10xxx1	1	-	Move to ARM core register from Coprocessor	*MRC, MRC2* on page A8-493	All
101x	0xxxxx not 000x0x	-	-	Advanced SIMD, Floating-point	*Extension register load/store instructions* on page A7-272	
	00010x	-	-	Advanced SIMD, Floating-point	*64-bit transfers between ARM core and extension registers* on page A7-277	
	10xxxx	0	-	Floating-point data processing	*Floating-point data-processing instructions* on page A7-270	
	10xxxx	1	-	Advanced SIMD, Floating-point	*8, 16, and 32-bit transfer between ARM core and extension registers* on page A7-276	

- 其中最有名的就是 `svc 0` ，在系列篇中曾多次提及它，此处详细说下 `svc` ， `svc` 全称是 `Supervisor Call` ， `Supervisor` 是 CPU 的管理模式， `svc` 导致处理器进入管理模式，很多人问的系统调用底层是怎么实现的？ `svc` 就是答案。
- 例如 `printf` 是个标准库函数，在标准库的底层代码中会调用 `svc 0` ，导致用户态的 `ARM` 程序通常将系统调用号传入 `R7` 寄存器(也被鸿蒙内核使用)，然后用 `SVC` 指令调用 `0` 号中断来直接执行系统调用，
- 在以前的ARM架构版本中， `SVC` 指令被称为 `SWI` ，软件中断。
- 描述 `svc` 功能的详细伪代码如下，请尝试读懂它

```
The TakeSVCException() pseudocode procedure describes how the processor takes the exception:
// TakeSVCException()
// ==================
TakeSVCException()
// Determine return information. SPSR is to be the current CPSR, after changing the IT[]
// bits to give them the correct values for the following instruction, and LR is to be
// the current PC minus 2 for Thumb or 4 for ARM, to change the PC offsets of 4 or 8
// respectively from the address of the current instruction into the required address of
// the next instruction, the SVC instruction having size 2bytes for Thumb or 4 bytes for ARM.
ITAdvance();
new_lr_value = if CPSR.T == '1' then PC-2 else PC-4;
new_spsr_value = CPSR;
vect_offset = 8;
// Check whether to take exception to Hyp mode
// if in Hyp mode then stay in Hyp mode
take_to_hyp = (HaveVirtExt() && HaveSecurityExt() && SCR.NS == '1' && CPSR.M == '11010');
// if HCR.TGE is set to 1, take to Hyp mode through Hyp Trap vector
route_to_hyp = (HaveVirtExt() && HaveSecurityExt() && !IsSecure() && HCR.TGE == '1'
&& CPSR.M == '10000'); // User mode
// if HCR.TGE == '1' and in a Non-secure PL1 mode, the effect is UNPREDICTABLE
```

```
preferred_exceptn_return = new_lr_value;
if take_to_hyp then
EnterHypMode(new_spsr_value, preferred_exceptn_return, vect_offset);
elsif route_to_hyp then
EnterHypMode(new_spsr_value, preferred_exceptn_return, 20);
else
// Enter Supervisor ('10011') mode, and ensure Secure state if initially in Monitor
// ('10110') mode. This affects the Banked versions of various registers accessed later
// in the code.
if CPSR.M == '10110' then SCR.NS = '0';
CPSR.M = '10011';
// Write return information to registers, and make further CPSR changes: IRQs disabled,
// IT state reset, instruction set and endianness set to SCTLR-configured values.
SPSR[] = new_spsr_value;
R[14] = new_lr_value;
CPSR.I = '1';
CPSR.IT = '00000000';
CPSR.J = '0'; CPSR.T = SCTLR.TE; // TE=0: ARM, TE=1: Thumb
CPSR.E = SCTLR.EE; // EE=0: little-endian, EE=1: big-endian
// Branch to SVC vector.
BranchTo(ExcVectorBase() + vect_offset);
```

- 这部分内容在系列篇 **(寄存器篇)** ，**(系统调用篇)** ，**(标准库篇)** 中都有提及。

具体指令

细看几条代码案例出现的常用指令

sub sp, sp, #8

```
sub sp, sp, #8  // 机器码 e24dd008 < = > 1110 0010 0100 1101 1101 0000 0000 1000
```

是减法操作指令，减法编码格式为

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
SUB{S}<c> <Rd>, <Rn>, #<const>
```

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
cond	0 0	1	0 0 1 0	S	Rn	Rd	imm12

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-214.

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE SUB (SP minus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

图中除了给出格式语法还有一段伪代码用于描述指令的使用条件

- `sp` 为 `13` 号寄存器， `lr` 为 `14` 号寄存器 ， `pc` 为 `15` 号寄存器。
- 如果是 `PC` 寄存器 (Rn = 15) 且 `S` 等于 `0` 查看 `ADR` 指令。。
- 如果是 `SP` 寄存器 (Rn = 13) 看 `SUB` (申请栈空间)。
- 如果是 `PC` 寄存器 (Rd = 15) 且 `S` 等于 `1` 。查看 `subs pc lr` 相关指令
- 套用格式结合源码

cond	op1	操作码	S	Rn	Rd	imm12(立即数)
1110	001	0010	0	1101	1101	0000 0000 1000
无条件执行	表示数据处理	SUB		sp	sp	8

mov r0, #0

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
MOV{S}<c> <Rd>, <Rm>
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4	3 2 1 0
cond	0 0 0	1 1 0 1	S	(0) (0) (0) (0)	Rd	0 0 0 0 0 0 0 0	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-214.

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
```

```
mov r0, #0 //e3a00000 1110 0011 1010 0000 0000 0000 0000 0000
```

bx lr

A8.8.27 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register.

Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
BX<c> <Rm>
```

Outside or last in IT block

15 14 13 12 11 10	9 8 7	6 5 4 3	2 1 0
0 1 0 0 0 1	1 1 0	Rm	(0) (0) (0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Encoding A1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
BX<c> <Rm>
```

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
cond	0 0 0 1	0 0 1 0	(1) (1) (1) (1)	(1) (1) (1) (1)	(1) (1) (1) (1)	0 0 0 1	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-214.

```
m = UInt(Rm);
```

```
bx lr e12fff1e 1110 0001 0010 1111 1111 1111 0001 1110
```

- `Rm = 1110` 对应 `lr` 寄存器 ，其相当于高级语言的 `return` ，函数执行完了需切回到调用它的函数位置继续执行， `lr` 保存的就是那个位置，从哪里来就回到哪里去。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

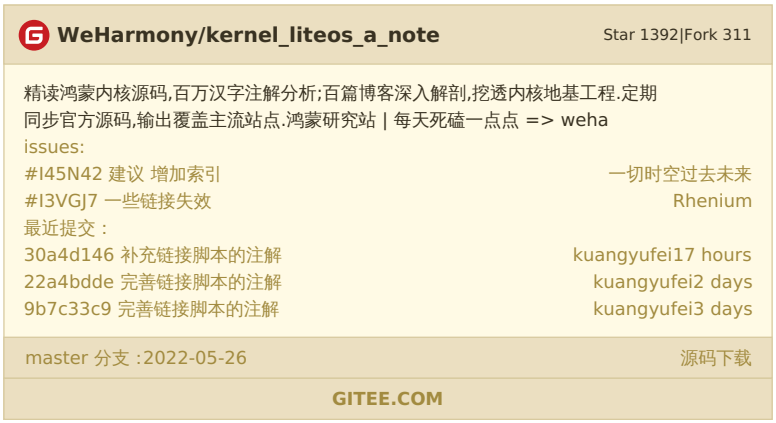
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

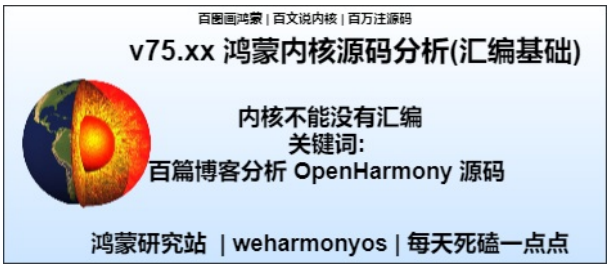


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

75_汇编基础篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

本篇通过拆解一段很简单的汇编代码来快速认识汇编，为读懂鸿蒙汇编打基础。系列篇后续将逐个剖析鸿蒙的汇编文件。

汇编其实很可爱

- 绝大部分IT从业人员终生不用触碰到的汇编，它听着像上古时代遥远的呼唤，总觉得远却又能听到声，汇编再往下就真的是01110011了，汇编指令基本是一一对应了机器指令。
- 所谓内核是对硬件的驱动，对驱动之后资源的良序管理，这里说的资源是CPU(单核/多核)，内存，磁盘，i/o设备。层层封装，步步遮蔽，到了应用层，不知有汉，无论魏晋才好。好是好，但有句话，其实哪有什么岁月静好，只是有人替你负重前行。难道就不想知道别人是怎么负重前行的？
- 越高级的语言是越接近人思维模式的，越低级的语言就是越贴近逻辑与非门的高低电平的起伏。汇编是贴着硬件飞行的，要研究内核就绕不过汇编，觉得神秘是来源于不了解，恐惧是来自于没接近。
- 其实深入分析内核源码之后就会发现，汇编其实很可爱，很容易，比c/c++/java容易太多了，真的是很傻很单纯。

汇编很简单

- 第一： 要认定汇编语言一定是简单的，没有高深的东西，无非就是数据的搬来搬去，运行时数据主要待在两个地方：内存和寄存器。寄存器是 CPU 内部存储器，离运算器最近，所以最快。
- 第二： 运行空间(栈空间)就是 CPU 打卡上班的地方，内核设计者规定谁请 CPU 上班由谁提供场地，用户程序提供的场地叫用户栈，敏感工作CPU要带回公司做，公司提供的场地叫内核栈，敏感工作叫系统调用，系统调用的本质理解是 CPU 要切换工作模式即切换办公场地。
- 第三： CPU 的工作顺序是流水线的，它只认指令，而且只去一个地方（指向代码段的PC寄存器）拿指令运算消化。指令集是告诉外界我 CPU 能干什么活并提供对话指令，汇编语言是人和 CPU 能愉快沟通不拧巴的共识语言。一一对应了 CPU 指令，又能确保记性不好的人类能模块化的设计 idea ， 先看一段 C 编译成汇编代码再来说模块化。

square(c -> 汇编)

```
//编译器: armv7-a clang (trunk)
//++++++++++++ square(c -> 汇编)++++++++++++++++++++++++++
int square(int a , int b){
   return a*b;
}
square(int ， int):
      sub    sp ， sp ， #8    //sp减去8，意思为给square分配栈空间，只用2个栈空间完成计算
```

```
    str    r0，[sp， #4]  //第一个参数入栈
    str    r1，[sp]       //第二个参数入栈
    ldr    r1，[sp， #4]  //取出第一个参数给r1
    ldr    r2，[sp]       //取出第二个参数给r2
    mul    r0， r1， r2     //执行a*b给R0，返回值的工作一直是交给R0的
    add    sp， sp， #8     //函数执行完了，要释放申请的栈空间
    bx     lr             //子程序返回，等同于mov pc，lr，即跳到调用处
```

fp(c -> 汇编)

```
//++++++++++++ fp(c -> 汇编)++++++++++++++++++++++++++
int fp(int b)
{
   int a = 1;
   return square(a+b，a+b);
}
fp(int):
    push   {r11， lr}     //r11(fp)/lr入栈，保存调用者main的位置
    mov    r11， sp       //r11用于保存sp值，函数栈开始位置
    sub    sp， sp， #8    //sp减去8，意思为给fp分配栈空间，只用2个栈空间完成计算
    str    r0，[sp， #4]  //先保存参数值，放在SP+4，此时r0中存放的是参数
    mov    r0， #1        //r0=1
    str    r0，[sp]      //再把1也保存在SP的位置
    ldr    r0，[sp]      //把SP的值给R0
    ldr    r1，[sp， #4]  //把SP+4的值给R1
    add    r1， r0， r1    //执行r1=a+b
    mov    r0， r1        //r0=r1，用r0，r1传参
    bl     square(int， int)//先mov lr， pc 再mov pc square(int， int)
    mov    sp， r11       //函数执行完了，要释放申请的栈空间
    pop    {r11， lr}     //弹出r11和lr，lr是专用标签，弹出就自动复制给lr寄存器
    bx     lr             //子程序返回，等同于mov pc，lr，即跳到调用处
```

main(c -> 汇编)

```
//++++++++++++ main(c -> 汇编)++++++++++++++++++++++++++
int main()
{
   int sum = 0;
   for(int a = 0;a < 100; a++){
      sum = sum + fp(a);
   }
   return sum;
}
main:
    push   {r11， lr}      //r11(fp)/lr入栈，保存调用者的位置
    mov    r11， sp        //r11用于保存sp值，函数栈开始位置
    sub    sp， sp， #16    //sp减去16，意思为给main分配栈空间，只用4个栈空间完成计算
    mov    r0， #0         //初始化r0
    str    r0，[r11， #-4] //执行sum = 0
    str    r0，[sp， #8]   //sum将始终占用SP+8的位置
    str    r0，[sp， #4]   //a将始终占用SP+4的位置
    b      .LBB1_1         //跳到循环开始位置
.LBB1_1:                   //循环开始位置入口
    ldr    r0，[sp， #4]   //取出a的值给r0
    cmp    r0， #99        //跟99比较
    bgt    .LBB1_4         //大于99，跳出循环 mov pc .LBB1_4
    b      .LBB1_2         //继续循环，直接 mov pc .LBB1_2
.LBB1_2:                   //符合循环条件入口
    ldr    r0，[sp， #8]   //取出sum的值给r0，sp+8用于写SUM的值
    str    r0，[sp]        //先保存SUM的值，SP的位置用于读SUM值
    ldr    r0，[sp， #4]   //r0用于传参，取出A的值给r0作为fp的参数
    bl     fp(int)         //先mov lr， pc再mov pc fp(int)
    mov    r1， r0         //fp的返回值为r0，保存到r1
    ldr    r0，[sp]        //取出SUM的值
    add    r0， r0， r1     //计算新sum的值，由R0保存
    str    r0，[sp， #8]   //将新sum保存到SP+8的位置
    b      .LBB1_3         //无条件跳转，直接 mov pc .LBB1_3
```

```
.LBB1_3:                                //完成a++操作入口
        ldr     r0，[sp， #4]  //SP+4中记录是a的值，赋给r0
        add     r0， r0， #1    //r0增加1
        str     r0，[sp， #4]  //把新的a值放回SP+4里去
        b       .LBB1_1         //跳转到比较 a < 100 处
.LBB1_4:                                //循环结束入口
        ldr     r0，[sp， #8]  //最后SUM的结果给R0，返回值的工作一直是交给R0的
        mov     sp， r11        //函数执行完了，要释放申请的栈空间
        pop     {r11， lr}      //弹出r11和lr，lr是专用标签，弹出就自动复制给lr寄存器
        bx      lr              //子程序返回，跳转到lr处等同于 MOV PC， LR
```

代码有点长，都加了注释，如果能直接看懂那么恭喜你，鸿蒙内核的`6`个汇编文件基于也就懂了.这是以下`C`文件全貌

文件全貌

```
#include <stdio.h>
#include <math.h>

int square(int a , int b){
   return a*b;
}

int fp(int b)
{
   int a = 1;
   return square(a+b , a+b);
}

int main()
{
   int sum = 0;
   for(int a = 0;a < 100; a++){
      sum = sum + fp(a);
   }
   return sum;
}
```

代码很简单谁都能看懂，代码很典型，具有代表性，有循环，有判断，有运算，有多级函数调用。编译后的汇编代码基本和`C`语言的结构差不太多， 区别是对循环的实现用了四个模块，四个模块也好理解: 一个是开始块(`LBB1_1`)， 一个符合条件的处理块(`LBB1_2`)，一个条件发生变化块(`LBB1_3`)，最后收尾块(`LBB1_4`)。

按块逐一剖析。

先看最短的那个

```
int square(int a , int b){
   return a*b;
}
//编译成
square(int， int):
        sub     sp， sp， #8    //sp减去8，意思为给square分配栈空间，只用2个栈空间完成计算
        str     r0，[sp， #4]  //第一个参数入栈
        str     r1，[sp]       //第二个参数入栈
        ldr     r1，[sp， #4]  //取出第一个参数给r1
        ldr     r2，[sp]       //取出第二个参数给r2
        mul     r0， r1， r2    //执行a*b给R0，返回值的工作一直是交给R0的
        add     sp， sp， #8    //函数执行完了，要释放申请的栈空间
        bx      lr              //子程序返回，等同于mov pc，lr，即跳到调用处
```

首先上来一句`sub sp， sp， #8`等同于`sp = sp - 8`， CPU运行需要场地，这个场地就是栈 ， SP是指向栈的指针，表示此时用栈的刻度。 代码和鸿蒙内核用栈方式一样，都采用了递减满栈的方式(FD)。 什么是递减满栈？ 递减指的是栈底地址高于栈顶地址，栈的生长方向是递减的， 满栈指的是SP指针永远指向栈顶。 每个函数都有自己独立的栈底和栈顶，之间的空间统称栈帧。可以理解为分配了一块 区域给函数运行， `sub sp， sp， #8`代表申请 2 个栈空间，一个栈空间按四个字节算。 用完要不要释放？当然要， `add sp， sp， #8`就是释放栈空间。 是一对的，减了又加回去，空间就归还了。 `ldr r1，[sp， #4]`的意思是取出`SP+4`这个虚拟地址的值给`r1`寄存器，而`SP`的指向并没有改变的，还是在栈顶， 为什么要`+`呢， `+`就是往回数， 定位到分配的栈空间上。
一定要理解递减满栈，这是关键! 否则读不懂内核汇编代码。

入参方式

一般都是通过寄存器 (r0..r10) 传参， fp 调用 square 之前会先将参数给 (r0..r10)

```
add    r1， r0， r1    //执行r1=a+b
mov    r0， r1        //r0=r1，用r0，r1传参
bl     square(int， int)//先mov lr， pc 再mov pc square(int， int)
```

到了 square 中后，先让 r0 ， r1 入栈，目的是保存参数值， 因为 square 中要用 r0 ， r1 ，

```
str    r0，[sp， #4]  //先入栈保存第一个参数
str    r1，[sp]      //再入栈保存第二个参数
ldr    r1，[sp， #4]  //再取出第一个参数给r1，(a*b)中a值
ldr    r2，[sp]      //再取出第二个参数给r2，用于计算 (a*b)中b值
```

是不是感觉这段汇编很傻，直接不保存计算不就完了吗，这个是流程问题，编译器统一先保存参数，至于你想怎么用它不管，也管不了。 另外返回值都是默认统一给 r0 保存。 square 中将 (a*b) 的结果给了 r0 ，回到 fp 中取出 R0 对 fp 来说这就是 square 的返回值，这是规定。

函数调用 main 和 fp 中都需要调用其他函数，所以都出现了

```
push   {r11， lr}
//....
pop    {r11， lr}
```

这哥俩也是成对出现的，这是函数调用的必备装备，作用是保存和恢复调用者的现场，例如 main -> fp ， fp 要保存 main 的栈帧范围和指令位置， lr 保存的是 main 函数执行到哪个指令的位置， r11 的作用是指向 main 的栈顶位置，如此 fp 执行完后 return 回 main 的时候，先 mov pc，lr ， PC 寄存器的值一变， 表示执行的代码就变了，又回到了 main 的指令和栈帧继续未完成的事业。

内存和寄存器数据怎么搬？

数据主要待在两个地方：内存和寄存器。 搬运方向不同指令也都不一样。 对于于 内存<->寄存器之间的搬运，可理解成将寄存器看成甲方， store 表示将寄存器(甲方)数据放到内存中， load 将内存(乙方)数据加载到寄存器中。

```
str    r1，[sp]     // 寄存器->内存
ldr    r1，[sp， #4] // 内存->寄存器
```

而熟知的 mov r0， r1 用于 寄存器<->寄存器

追问三个问题

第一：如果是可变参数怎么办？ 100 个参数怎么整， 通过寄存器总共就 12 个，不够传参啊

第二：返回值可以有多个吗？

第三：数据搬运可以不经过 CPU 吗？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

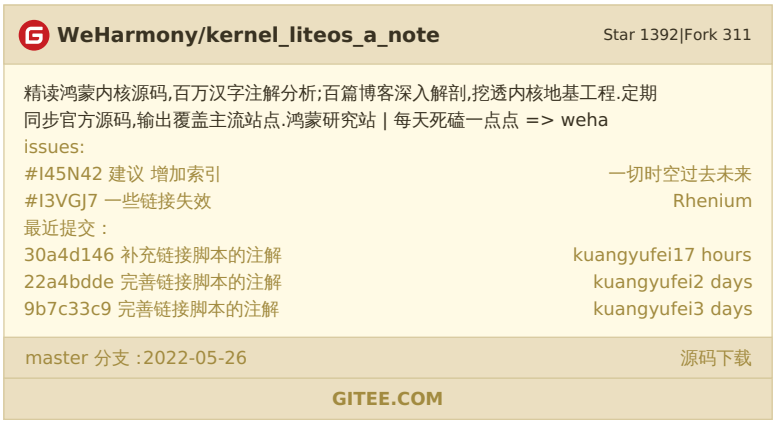
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

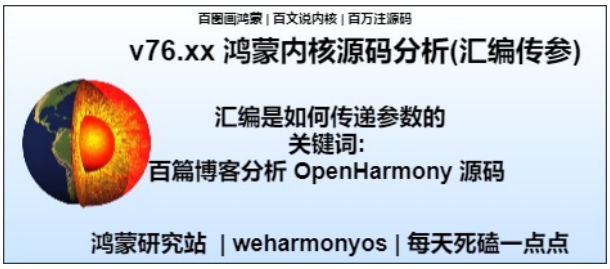


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

76_汇编传参篇

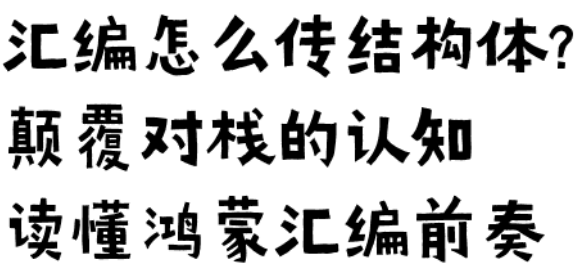
本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...



汇编如何传复杂的参数？

汇编基础篇 中很详细的介绍了一段具有代表性很经典的汇编代码，有循环，有判断，有运算，有多级函数调用。但有一个问题没有涉及，就是很复杂的参数如何处理？ 在实际开发过程中函数参数往往是很复杂的参数，(比如结构体)汇编怎么传递呢？ 先看一段C语言及汇编代码，传递一个稍微复杂的参数来说明汇编传参的过程

```
#include <stdio.h>
#include <math.h>
struct reg{//参数远超寄存器数量
    int Rn[100];
    int pc;
};

int framePoint(reg cpu)
{
    return cpu.Rn[0] * cpu.pc;
}

int main()
{
    reg cpu;
    cpu.Rn[0] = 1;
```

```
    cpu.pc = 2;
    return framePoint(cpu);
}
```

```
//编译器: armv7-a gcc (9.2.1)
framePoint(reg):
        sub     sp， sp， #16    @申请栈空间
        str     fp， [sp， #-4]!  @保护main函数栈帧，等同于push {fp}
        add     fp， sp， #0     @fp变成framePoint栈帧，同时也指向了栈顶
        add     ip， fp， #4     @定位到入栈口，让4个参数依次入栈
        stm     ip， {r0， r1， r2， r3}@r0-r3入栈保存
        ldr     r3， [fp， #4]   @取值cpu.Rn[0] = 1
        ldr     r2， [fp， #404]  @取值cpu.pc = 2，高地址位
        mul     r3， r2， r3     @cpu.Rn[0] * cpu.pc
        mov     r0， r3         @返回值由r0保存
        add     sp， fp， #0     @重置sp，和add     fp， sp， #0配套出现
        ldr     fp， [sp]， #4   @恢复main函数栈帧
        add     sp， sp， #16    @归还栈空间，sp回落到main函数栈顶位置
        bx      lr              @跳回main函数
main:
        push    {fp， lr}       @入栈保存调用函数现场
        add     fp， sp， #4     @fp指向sp+4，即main栈帧的底部
        sub     sp， sp， #800   @分配800个线性地址，即main栈帧的顶部
        mov     r3， #1         @r3 = 1
        str     r3， [fp， #-408] @将1放置 fp-408处，即:cpu.Rn[0]处
        mov     r3， #2         @r3 = 2
        str     r3， [fp， #-8]  @将2放置 fp-8处，即:cpu.pc
        mov     r0， sp         @r0 = sp
        sub     r3， fp， #392   @r3 = fp - 392
        mov     r2， #388       @只拷贝388，剩下4个由寄存器传参
        mov     r1， r3         @保存由r1保存r3，用于memcpy
        bl      memcpy          @拷贝结构体部分内容，将r1的内容拷贝r2的数量到r0
        sub     r3， fp， #408   @定位到结构体剩余未拷贝处
        ldm     r3， {r0， r1， r2， r3} @将剩余结构体内容通过寄存器传参
        bl      framePoint(reg)         @执行framePoint
        mov     r3， r0         @返回值给r3
        nop @用于程序指令的对齐
        mov     r0， r3         @再将返回值给r0
        sub     sp， fp， #4     @恢复SP值
        pop     {fp， lr}       @出栈恢复调用函数现场
        bx      lr              @跳回调用函数
```

两个函数对应两段汇编，干净利落，去除中间各项干扰，只有一个结构体reg，以下详细讲解如何传递它，以及它在栈中的数据变化是怎样的？

入参方式

结构体总共101个栈空间(一个栈空间单位四个字节)，对应就是404个线性地址. main上来就申请了 sub sp， sp， #800 @申请800个线性地址给main，即 200个栈空间

```
int main()
{
    reg cpu;
    cpu.Rn[0] = 1;
    cpu.pc = 2;
    return framePoint(cpu);
}
```

但main函数只有一个变量，只需101个栈空间，其他都算上也用不了200个。为什么要这么做呢？ 而且注意下里面的数字 388， 408， 392 这些都是什么意思？ 看完main汇编能得到一个结论是 200个栈空间中除了存放了main函数本身的变量外 ，还存放了要传递给framePoint函数的部分参数值，存放了多少个？答案是 388/4 = 97个。 注意变量没有共用，而是拷贝了一部份出来。如何拷贝的？继续看

memcpy汇编调用

```
        mov     r0， sp         @r0 = sp
        sub     r3， fp， #392   @r3 = fp - 392
        mov     r2， #388       @只拷贝388，剩下4个由寄存器传参
```

```
    mov     r1， r3         @保存由r1保存r3，用于memcpy
    bl      memcpy          @拷贝结构体部分内容，将r1的内容拷贝r2的数量到r0
    sub     r3， fp， #408    @定位到结构体剩余未拷贝处
    ldm     r3， {r0， r1， r2， r3} @将剩余结构体内容通过寄存器传参
```

看这段汇编拷贝，意思是从r1开始位置拷贝r2数量的数据到r0的位置，注意只拷贝了 388个，也就是 388/4 = 97个栈空间。剩余的4个通过寄存器传的参数。ldm代表从fp-408的位置将内存地址的值连续的给r0 - r3寄存器，即位置(fp-396，fp-400，fp-404，fp-408)的值。 执行下来的结果就是

```
r3 = fp-408， r2 = fp-404 ，r1 = fp-400 ，r0 = fp-396 得到虚拟地址的值，这些值整好是memcpy没有拷贝到变量剩余的值
```

逐句分析 framePoint

```
framePoint(reg):
    sub     sp， sp， #16     @申请栈空间
    str     fp， [sp， #-4]!  @保护main函数栈帧，等同于push {fp}
    add     fp， sp， #0      @fp变成framePoint栈帧，同时也指向了栈顶
    add     ip， fp， #4      @定位到入栈口，让4个参数依次入栈
    stm     ip， {r0， r1， r2， r3}@r0-r3入栈保存
    ldr     r3， [fp， #4]    @取值cpu.Rn[0] = 1
    ldr     r2， [fp， #404]  @取值cpu.pc = 2
    mul     r3， r2， r3      @cpu.Rn[0] * cpu.pc
    mov     r0， r3         @返回值由r0保存
    add     sp， fp， #0      @重置sp，和add     fp， sp， #0配套出现
    ldr     fp， [sp]， #4    @恢复main函数栈帧
    add     sp， sp， #16     @归还栈空间，sp回落到main函数栈顶位置
    bx      lr              @跳回main函数
```

```
framePoint申请了4个栈空间目的是用来存放四个寄存器值的，以上汇编代码逐句分析。

第一句: sub     sp， sp， #16     @申请栈空间，用来存放r0-r3四个参数

第二句: str     fp， [sp， #-4]!  @保护main的fp，等同于push {fp}，为什么这里要把main函数的fp放到 [sp， #-4]! 位置，注意 !号，表示SP的位置要变动，因

第三句: add     fp， sp， #0      @指定framePoint的栈帧位置，同时指向了栈顶 SP

第四句: add     ip， fp， #4      @很关键，用了ip寄存器，因为此时 fp sp 都已经确定了，但别忘了 r0 - r3 还没有入栈呢。从哪个位置入栈呢， fp+4位置，因为 r

第五句: stm     ip， {r0， r1， r2， r3}@r0-r3入栈，填满了剩下的四个空位。

第六句: ldr     r3， [fp， #4]    @取的就是cpu.Rn[0] = 1的值，因为上一句就是从这里依次入栈的，最后一个当然就是cpu.pc了。

第七句: ldr     r2， [fp， #404]  @取值cpu.pc = 2，其实这一句已经是跳到了main函数的栈帧取值了，所以看明白了没有，并不是在传统意义上理解的在framePo

第八句: mul     r3， r2， r3      @cpu.Rn[0] * cpu.pc 做乘法运算

第九句: mov     r0， r3         @返回值r0保存运算结构， 目的是return

第十句: add sp， fp， #0          @重置sp，其实这一句可以优化掉，因为此时sp = fp

第十一句: ldr     fp， [sp]， #4  @恢复fp，等同于pop {fp}，因为函数运行完了，需要回到main函数了，所以要拿到main的栈帧

第十二句: add     sp， sp， #16   @归还栈空间，等于把四个入参抹掉了。

最后一句: bx      lr            @跳回main函数，如此 fp 和 lr 寄存器中保存的都是 main函数的信息，就可以安全着陆了。
```

总结

因为寄存器数量有限，所以只能通过这种方式来传递大的参数，想想也只能在main函数栈中保存大部分参数，同时又必须确保数据的连续性，好像也只能用这种办法了，一部分通过寄存器传，一部分通过拷贝的方式倒是挺有意思的。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很

重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。

- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

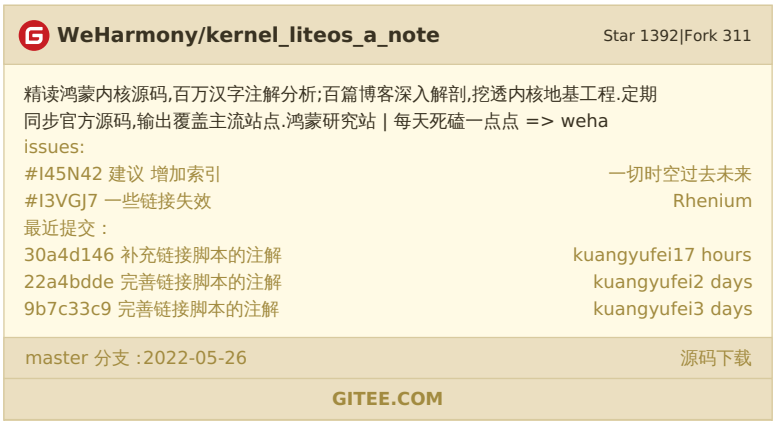
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

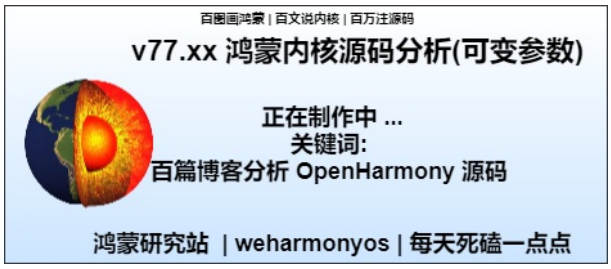


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

77_链接脚本篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

链接器

查看 >> LD官方资料

board.ld

```
#include "los_vm_zone.h"
#define TEXT_BASE  KERNEL_VADDR_BASE //代码区为内核起始地址

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)

MEMORY //链接器的默认配置允许分配所有可用内存,描述链接器可以使用哪些内存区域
{ //ram,sram为存储区域的名字,可以随意取
   ram : ORIGIN = KERNEL_VADDR_BASE, LENGTH = KERNEL_VADDR_SIZE //内核运行空间范围
   sram : ORIGIN = 0x40000000, LENGTH = 0x1000
   user_ram : ORIGIN = 0x1000000, LENGTH = 0x100000 //用户运行空间范围 USER_ASPACE_BASE ,此大小不是真正最后映射到用户空间的大小
}
SECTIONS
{
   /DISCARD/ : { *(.comment .note) } //过滤掉所有输入文件的 .comment .note 段

   .ram_vectors TEXT_BASE : { //内核中断向量区开始位置,此处用 TEXT_BASE 宏有误导嫌疑, 因为真正的(.text)并不在此 @notethinking
      __ram_vectors_vma = .; //定位到当前位置,即TEXT_BASE处
      KEEP (*(.vectors)) //告诉链接器 强制保留所有输入文件中的 .vectors 节
   } > ram //中断向量是开机代码的位置，可翻看 鸿蒙内核源码分析(开机启动篇) 和 (中断管理篇)
   __ram_vectors_lma = LOADADDR(.ram_vectors);//启动时对向量的初始化,加载地址和链接地址一致,说明内核设计者希望从加载地址处运行指令
}
//LMA：加载存储地址，指加载到存储器的地址，即加载或烧写到哪里
//VMA：虚拟存储地址，也就是链接地址，即代码和数据运行的时候应在哪里
USER_INIT_VM_START = 0x1000000; //用户空间初始地址
```

liteos.ld

```
ENTRY(reset_vector) /*指定程序入口地址*/
INCLUDE board.ld // > ram 指放入ram这个地址范围中, ram在board.ld中定义
/* SECTIONS 是脚本中最重要的命令，所有的LD脚本都会有这个命令，用来指定如何将输入文件映射到输出文件等等 */
SECTIONS
{
 //节地址是指该节的VMA地址。如果改地未明确指定，连接器会在考虑严格对齐情况下，首先按照region参数分配内存，其次根据当前位置计数器向下分地址。
    _start = .; //特殊符号 .表示当前位置计数器,即紧挨着中断向量结束的位置
   .set_sysinit_set : {
      __start_set_sysinit_set = ABSOLUTE(.); //
      KEEP (*(.set_sysinit_set))//所有输入文件中的 .set_sysinit_set 链接到此
      __stop_set_sysinit_set = ABSOLUTE(.);//定位结束 .set_sysinit_set 区结束位置
   } > ram

   .got ALIGN(0x4) : { *(.got.plt) *(.got) } > ram

   .gcc_except_table ALIGN (0x8) : { . = .; } > ram  .gcc_except_table : { KEEP(*(.gcc_except_table*)) }
   .exception_ranges ALIGN (0x8) : ONLY_IF_RW { *(.exception_ranges .exception_ranges*) } > ram

   .ARM.extab ALIGN(0x4) : { *(.ARM.extab* .gnu.linkonce.armextab.*) } > ram

   /* .ARM.exidx is sorted, so has to go in its own output section.  */
   .ARM.exidx ALIGN(0x8) : { __exidx_start = .; *(.ARM.exidx* .gnu.linkonce.armexidx.*) ;__exidx_end = .;} > ram

   /* text/read-only data */
   .text ALIGN(0x1000) : { //代码区 按4K对齐
      __text_start = .; //当前位置为 __text_start 开始位置
      *(.text* .sram.text.glue_7* .gnu.linkonce.t.*) //*(.text)
   } > ram
//重定向代码 Relocation ,包括 代码区和数据区的重定位
   .rel.text : { *(.rel.text) *(.rel.text.*) *(.rel.gnu.linkonce.t*) } > ram
   .rela.text : { *(.rela.text) *(.rela.text.*) *(.rela.gnu.linkonce.t*) } > ram
   .rel.data : { *(.rel.data) *(.rel.data.*) *(.rel.gnu.linkonce.d*) } > ram
   .rela.data : { *(.rela.data) *(.rela.data.*) *(.rela.gnu.linkonce.d*) } > ram
   .rel.rodata : { *(.rel.rodata) *(.rel.rodata.*) *(.rel.gnu.linkonce.r*) } > ram
   .rela.rodata : { *(.rela.rodata) *(.rela.rodata.*) *(.rela.gnu.linkonce.r*) } > ram
   .rel.got : { *(.rel.got) } > ram
   .rela.got : { *(.rela.got) } > ram
   .rel.ctors : { *(.rel.ctors) } > ram
   .rela.ctors : { *(.rela.ctors) } > ram
   .rel.dtors : { *(.rel.dtors) } > ram
   .rela.dtors : { *(.rela.dtors) } > ram
   .rel.init : { *(.rel.init) } > ram
   .rela.init : { *(.rela.init) } > ram
   .rel.fini : { *(.rel.fini) } > ram
   .rela.fini : { *(.rela.fini) } > ram
   .rel.bss : { *(.rel.bss) } > ram
   .rela.bss : { *(.rela.bss) } > ram
   .rel.plt : { *(.rel.plt) } > ram
   .rela.plt : { *(.rela.plt) } > ram
   .rel.dyn : { *(.rel.dyn) } > ram

   .dummy_post_text : {
      __text_end = .; //代码区结束位置
   } > ram

   .rodata ALIGN(0x1000) : {
      __rodata_start = .; // 只读数据区开始位置
      __kernel_init_level_0 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.0.*)));
      __kernel_init_level_1 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.1.*)));
      __kernel_init_level_2 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.2.*)));
      __kernel_init_level_3 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.3.*)));
      __kernel_init_level_4 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.4.*)));
      __kernel_init_level_5 = ABSOLUTE(.);
      KEEP(*( SORT (.rodata.init.kernel.5.*)));
      __kernel_init_level_6 = ABSOLUTE(.);
```

```
        KEEP(*( SORT (.rodata.init.kernel.6.*)));
        __kernel_init_level_7 = ABSOLUTE(.);
        KEEP(*( SORT (.rodata.init.kernel.7.*)));
        __kernel_init_level_8 = ABSOLUTE(.);
        KEEP(*( SORT (.rodata.init.kernel.8.*)));
        __kernel_init_level_9 = ABSOLUTE(.);
        KEEP(*( SORT (.rodata.init.kernel.9.*)));
        __kernel_init_level_10 = ABSOLUTE(.);
        *(.rodata .rodata.* .gnu.linkonce.r.*)
        __exc_table_start = .;  //异常表开始位置
        KEEP(*(__exc_table))
        __exc_table_end = .; //异常表结束位置
    } > ram

    /*
     * extra linker scripts tend to insert sections just after .rodata,
     * so we want to make sure this symbol comes after anything inserted above,
     * but not aligned to the next section necessarily.
     */
    .dummy_post_rodata : {
        _hdf_drivers_start = .;
        KEEP(*(.hdf.driver)) // 统一驱动框架也放在了只读区
        _hdf_drivers_end = .;
        __rodata_end = .; //数据只读区结束
    } > ram

    .data ALIGN(0x1000) : {
        /* writable data  */
        __ram_data_start = .; //可写入数据开始位置
        __vdso_data_start = LOADADDR(.data); // vdso区 (virtual dynamic shared object)开始位置
        KEEP(*(.data.vdso.datapage)) //vdso由数据区+代码区两部分组成, 可翻看 鸿蒙内核源码分析(vdso篇)
        . = ALIGN(0x1000);//vdso 的特性是 代码在内核区, 但运行却在用户区
        KEEP(*(.data.vdso.text))//vdso 的代码区
        . = ALIGN(0x1000);//按4K对齐,因啥要按4K对齐,因为需要页表映射,而一页为4K
        __vdso_text_end = .;//vdso区结束位置
        *(.data .data.* .gnu.linkonce.d.*)
        . = ALIGN(0x4);
        KEEP(*( SORT (.liteos.table.*)));
    } > ram

    .ctors : ALIGN(0x4) {
        __ctor_list__ = .;
        KEEP (*(.ctors .init_array))
        __ctor_end__ = .;
    } > ram
    .dtors : ALIGN(0x4) {
        __dtor_list__ = .;
        KEEP (*(.dtors .fini_array))
        __dtor_end__ = .;
    } > ram
    /*
     * extra linker scripts tend to insert sections just after .data,
     * so we want to make sure this symbol comes after anything inserted above,
     * but not aligned to the next section necessarily.
     */
    .dummy_post_data : {
        __ram_data_end = .; //可写入数据区结束
    } > ram
//这里指的是 init 应用程度的位置
    .user_init USER_INIT_VM_START : ALIGN(0x1000) {//开始地址设为 USER_INIT_VM_START = 0x1000000;
        . = ALIGN(0x4);
        __user_init_load_addr = LOADADDR(.user_init);//应用程序的加载地址
        __user_init_entry = .;//应用程序的入口地址
        KEEP(libuserinit.O (.user.entry))
        KEEP(libuserinit.O (.user.text))
        KEEP(libuserinit.O (.user.rodata))
        . = ALIGN(0X4);
        __user_init_data = .;//设置数据段开始位置 __user_init_data
        KEEP(libuserinit.O (.user.data))
        . = ALIGN(0X4);
        __user_init_bss = .;//init 进程的 bss开始位置
```

```
        KEEP(libuserinit.O (.user.bss))
        . = ALIGN(0x1000);
        __user_init_end = .; //init 进程结束位置
    } > user_ram AT > ram

    __user_init_size = __user_init_end - __user_init_entry; //计算init进程占用大小

    /* uninitialized data (in same segment as writable data) | 未初始化数据 */
    .bss : {
        . = ALIGN(0x800); //当前位置按 0x800对齐
        __int_stack_start = .; //内核栈开始位置
        *(.int_stack);
        . = ALIGN(0x4);//4字节对齐
        KEEP(*(.bss.prebss.*))
        . = ALIGN(0x8);
        __bss_start = .; //将当前位置给__bss_start,将所有的目标*(.bss .bss.*) .. 链接到 .bss中
        *(.bss .bss.*)
        *(.gnu.linkonce.b.*)
        *(COMMON)
        . = ALIGN(0x8);
        __bss_end = .;
    } > ram

    . = ALIGN(0x1000);
    _end = .;
    /* mmu temp page table(sys aviliable mem is start with __bss_end) */
    . = ALIGN(0x4000);
    __mmu_ttlb_begin = .;//临时页表开始位置

    /* Strip unnecessary stuff */
    /DISCARD/ 0 : { *(.comment .note) } > ram //过滤不需要的块
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

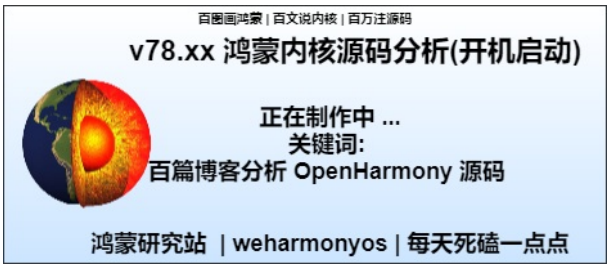
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

78_内核启动篇

本篇关键词：内核重定位、MMU、SVC栈、热启动、内核映射表



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

这应该是系列篇最难写的一篇，全是汇编代码，需大量的底层知识，涉及协处理器，内核镜像重定位，创建内核映射表，初始化 CPU 模式栈，热启动，到最后熟悉的 main() 。

Uboot

在 PC寄存器 指向内核第一行指令之前，需要先将内核从 flash 加载到内存指定位置，这部分工作由引导程序 uboot (与硬件强相关) 完成，而引导代码并不在 kernel_liteos_a (与硬件弱相关) 工程中，而在工程 u-boot-2020.01中 ，前往 >> ，所以系列篇不对引导程序详细说明，后续有机会再分析。引导结束后舞台就交给了内核 reset_vector 处 ，一切从此开始。

内核入口

在链接文件 liteos.ld 中可知内核的入口地址为 `ENTRY(reset_vector)` ， 分别出现在reset_vector_mp.S (多核启动) 和 reset_vector_up.S(单核启动)，系列篇研究多核启动的情况。代码可结合 (协处理器篇) 看更容易懂。

```
reset_vector: //内核启动入口
    /* clear register TPIDRPRW */
    mov     r0, #0     //r0 = 0
    mcr     p15, 0, r0, c13, c0, 4 //复位线程标识符寄存器TPIDRPRW，不复位将导致系统不能启动
    /* do some early cpu setup: i/d cache disable, mmu disabled */
    mrc     p15, 0, r0, c1, c0, 0 //System Control Register-SCTLR | 读取系统控制寄存器内容
    bic     r0, #(1<<12)  //禁用指令缓存功能
    bic     r0, #(1<<2 | 1<<0)  //禁用数据和TLB的缓存功能(bit2) | mmu功能(bit0)
    mcr     p15, 0, r0, c1, c0, 0 //写系统控制寄存器

    /* enable fpu+neon 一些系统寄存器的操作
    | 使能浮点运算(floating point unit)和 NEON就是一种基于SIMD思想的ARM技术，相比于ARMv6或之前的架构，
    NEON结合了64-bit和128-bit的SIMD指令集，提供128-bit宽的向量运算(vector operations)*/
#ifndef LOSCFG_TEE_ENABLE       //Trusted Execution Environment   可信执行环境
    MRC     p15, 0, r0, c1, c1, 2 //非安全模式访问寄存器 (Non-Secure Access Control Register - NSACR)
    ORR     r0, r0, #0xC00        //使能安全和非安全访问协处理器10和11(Coprocessor 10和11)
    BIC     r0, r0, #0xC000       //设置bit15为0，不会影响修改CPACR.ASEDIS寄存器位（控制Advanced SIMD功能）| bit14 reserved
    MCR     p15, 0, r0, c1, c1, 2

    LDR     r0, =(0xF << 20)      //允许在EL0和EL1下，访问协处理器10和11(控制Floating-point和Advanced SIMD特性)
    MCR     p15, 0, r0, c1, c0, 2
    ISB
```

```
#endif
    MOV    r3, #0x40000000     //EN, bit[30] 设置FPEXC的EN位来使能FPU
    VMSR   FPEXC, r3   //浮点异常控制寄存器 (Floating-Point Exception Control register | B4.1.57)

    /* r11: delta of physical address and virtual address | 计算虚拟地址和物理地址之间的差值,目的是为了建立映射关系表 */
    adr     r11, pa_va_offset //获取pa_va_offset变量物理地址，由于这时候mmu已经被关闭，所以这个值就表示pa_va_offset变量的物理地址。
                          /*adr 是一条小范围的地址读取伪指令,它将基于PC的相对偏移的地址值读到目标寄存器中。
                           *编译源程序时，汇编器首先计算当前PC值（当前指令位置）到exper的距离,然后用一条ADD或者SUB指令替换这条伪指令，
                           *例如:ADD register,PC,#offset_to_exper 注意，标号exper与指令必须在同一代码段
                           */
    ldr     r0, [r11]    //r0 = *r11 获取pa_va_offset变量虚拟地址
    sub     r11, r11, r0   //物理地址-虚拟地址 = 映射偏移量 放入r11

    mrc     p15, 0, r12, c0, c0, 5      /* Multiprocessor Affinity Register-MPIDR */
    and     r12, r12, #MPIDR_CPUID_MASK //掩码过滤
    cmp     r12, #0                     //主控核0判断
    bne     secondary_cpu_init          //初始化CPU次核
/*
 * adr是小范围的地址读取伪指令,它将基于PC寄存器相对偏移的地址值读取到寄存器中，
 * 例如: 0x00000004   : adr     r4, __exception_handlers
 * 则此时PC寄存器的值为: 0x00000004 + 8(在三级流水线时,PC和执行地址相差8)，
   * adr指令和标识__exception_handlers的地址相对固定，二者偏移量若为offset,
 * 最后r4 = (0x00000004 + 8) + offset
 */

    /* if we need to relocate to proper location or not | 如果需要重新安装到合适的位置*/
    adr     r4, __exception_handlers            /* r4: base of load address | 加载基址*/
    ldr     r5, =SYS_MEM_BASE                   /* r5: base of physical address | 物理基址*/
    subs    r12, r4, r5                       /* r12: delta of load address and physical address | 二者偏移量*/
    beq     reloc_img_to_bottom_done            /* if we load image at the bottom of physical address | 不相等就需要重定位 */

    /* we need to relocate image at the bottom of physical address | 需要知道拷贝的大小*/
    ldr     r7, =__exception_handlers           /* r7: base of linked address (or vm address) | 链接地址基地址*/
    ldr     r6, =__bss_start                    /* r6: end of linked address (or vm address)，由于目前阶段有用的数据是中断向量表+代码段+只读数据段+数据段，
                所以只需复制[__exception_handlers,__bss_start]这段数据到内存基址处 */
    sub     r6, r7                            /* r6: delta of linked address (or vm address) | 内核镜像大小 */
    add     r6, r4                            /* r6: end of load address | 说明需拷贝[ r4,r4+r6 ] 区间内容到 [ r5,r5+r6 ]*/

reloc_img_to_bottom_loop://重定位镜像到内核物理内存基地址,将内核从加载地址拷贝到内存基址处
    ldr     r7, [r4], #4 // 类似C语言 *r5 = *r4 , r4++ , r5++
    str     r7, [r5], #4 // #4 代表32位的指令长度,此时在拷贝内核代码区内容
    cmp     r4, r6          /* 拷贝完成条件. r4++ 直到等于r6 (加载结束地址) 完成拷贝动作 */
    bne     reloc_img_to_bottom_loop
    sub     pc, r12                           /* 重新校准pc寄存器, 无缝跳到了拷贝后的指令地址处执行 r12是重定位镜像前内核加载基地址和内核物理内存基地址的差值 *
    nop  // 注意执行完成sub       pc, r12后,新的PC寄存器也指向了  nop ,nop是伪汇编指令,等同于 mov r0 r0 通常用于控制时序的目的，强制内存对齐，防止流水线
    sub     r11, r11, r12                     /* r11: eventual address offset | 最终地址映射偏移量, 用于构建MMU页表 */
//内核总大小 __bss_start - __exception_handlers
reloc_img_to_bottom_done:
#ifdef LOSCFG_KERNEL_MMU
    ldr     r4, =g_firstPageTable               /* r4: physical address of translation table and clear it
              内核页表是用数组g_firstPageTable存储 见于los_arch_mmu.c */
    add     r4, r4, r11                       //计算g_firstPageTable页表物理地址
    mov     r0, r4        //因为默认r0 将作为memset_optimized的第一个参数
    mov     r1, #0        //第二个参数,清0
    mov     r2, #MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS //第三个参数是L1表的长度
    bl      memset_optimized                  /* optimized memset since r0 is 64-byte aligned | 将内核页表空间清零*/

    ldr     r5, =g_archMmuInitMapping         //记录映射关系表
    add     r5, r5, r11                       //获取g_archMmuInitMapping的物理地址
init_mmu_loop:                                //初始化内核页表
    ldmia   r5!, {r6-r10}                     /* r6 = phys, r7 = virt, r8 = size, r9 = mmu_flags, r10 = name | 传参: 物理地址、虚拟地址、映射大小、映射属性、
    cmp     r8, 0                           /* if size = 0, the mmu init done | 完成条件 */
    beq     init_mmu_done                   //标志寄存器中Z标志位等于零时跳转到  init_mmu_done处执行
    bl      page_table_build                //创建页表
    b       init_mmu_loop      //循环继续
init_mmu_done:
    orr     r8, r4, #MMU_TTBRx_FLAGS          /* r8 = r4 and set cacheable attributes on translation walk | 设置缓存*/
    ldr     r4, =g_mmuJumpPageTable           /* r4: jump pagetable vaddr | 页表虚拟地址*/
    add     r4, r4, r11
    ldr     r4, [r4]
    add     r4, r4, r11                       /* r4: jump pagetable paddr | 页表物理地址*/
```

```
    /* build 1M section mapping, in order to jump va during turing on mmu:pa == pa, va == pa */
    /* 从当前PC开始建立1MB空间的段映射，分别建立物理地址和虚拟地址方式的段映射页表项
     * 内核临时页表在系统 使能mmu -> 切换到虚拟地址运行 这段时间使用
     */
    mov     r6, pc
    mov     r7, r6                          /* r7: pa (MB aligned)*/
    lsr     r6, r6, #20                     /* r6: pa l1 index */
    ldr     r10, =MMU_DESCRIPTOR_KERNEL_L1_PTE_FLAGS
    add     r12, r10, r6, lsl #20               /* r12: pa |flags */
    str     r12, [r4, r7, lsr #(20 - 2)]        /* jumpTable[paIndex] = pt entry */
    rsb     r7, r11, r6, lsl #20               /* r7: va */
    str     r12, [r4, r7, lsr #(20 - 2)]        /* jumpTable[vaIndex] = pt entry */

    bl      mmu_setup                       /* set up the mmu | 内核映射表已经创建好了,此时可以启动MMU工作了*/
#endif
    /* clear out the interrupt and exception stack and set magic num to check the overflow
    |exc_stack|地址高位
    |svc_stack|地址低位
 清除中断和异常堆栈并设置magic num检查溢出 */
    ldr     r0, =__svc_stack     //stack_init的第一个参数 __svc_stack表示栈顶
    ldr     r1, =__exc_stack_top //stack_init的第二个参数 __exc_stack_top表示栈底, 这里会有点绕, top表高地址位
    bl      stack_init              //初始化各个cpu不同模式下的栈空间
 //设置各个栈顶魔法数字
    STACK_MAGIC_SET __svc_stack, #OS_EXC_SVC_STACK_SIZE, OS_STACK_MAGIC_WORD //中断栈底设成"烫烫烫烫烫烫"
    STACK_MAGIC_SET __exc_stack, #OS_EXC_STACK_SIZE, OS_STACK_MAGIC_WORD     //异常栈底设成"烫烫烫烫烫烫"

warm_reset: //热启动 Warm Reset, warm reboot, soft reboot, 在不关闭电源的情况，由软件控制重启计算机
    /* initialize CPSR (machine state register) */
    mov     r0, #(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|CPSR_SVC_MODE) /* 禁止IRQ中断 | 禁止FIQ中断 | 管理模式-操作系统使用的保护模式 */
    msr     cpsr, r0 //设置CPSR寄存器

    /* Note: some functions in LIBGCC1 will cause a "restore from SPSR"!! */
    msr     spsr, r0 //设置SPSR寄存器

    /* get cpuid and keep it in r12 */
    mrc     p15, 0, r12, c0, c0, 5  //R12保存CPUID
    and     r12, r12, #MPIDR_CPUID_MASK //掩码操作获取当前cpu id

    /* set svc stack, every cpu has OS_EXC_SVC_STACK_SIZE stack | 设置 SVC栈 */
    ldr     r0, =__svc_stack_top //注意这是栈底,高地址位
    mov     r2, #OS_EXC_SVC_STACK_SIZE //栈大小
    mul     r2, r2, r12
    sub     r0, r0, r2                  /* 算出当前core的中断栈栈顶位置，写入所属core的sp */
    mov     sp, r0

    LDR     r0, =__exception_handlers
    MCR     p15, 0, r0, c12, c0, 0      /* Vector Base Address Register - VBAR */

    cmp     r12, #0      //CPU是否为主核
    bne     cpu_start                   //不相等就跳到从核处理分支

clear_bss:                          //主核处理.bss段清零
    ldr     r0, =__bss_start
    ldr     r2, =__bss_end
    mov     r1, #0
    sub     r2, r2, r0
    bl      memset
#if defined(LOSCFG_CC_STACKPROTECTOR_ALL) || \
    defined(LOSCFG_CC_STACKPROTECTOR_STRONG) || \
    defined(LOSCFG_CC_STACKPROTECTOR)
    bl      __stack_chk_guard_setup
#endif

#ifdef LOSCFG_GDB_DEBUG
    /* GDB_START - generate a compiled_breadk,This function will get GDB stubs started, with a proper environment */
    bl      GDB_START
    .word  0xe7ffdeff
#endif

    bl      main                //带LR的子程序跳转, LR = pc - 4, 执行C层main函数
```

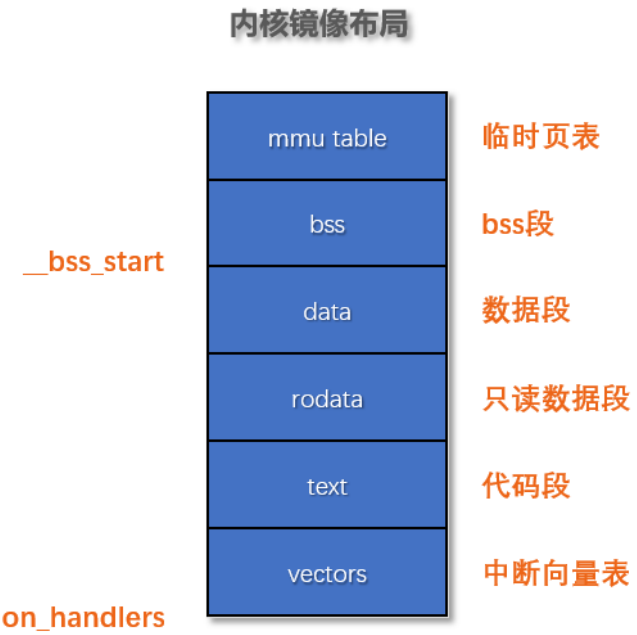
解读

- **第一步：** 操作 **CP15** 协处理器 **TPIDRPRW** 寄存器，它被 **ARM** 设计保存当前运行线程的 **ID**值，在**ARMv7** 架构中才新出现，需**PL1**权限以上才能访问，而硬件不会从内部去改变它的值，也就是说这是一个直接暴露给工程师操作维护的一个寄存器，在鸿蒙内核中被用于记录线程结构体的开始地址，可以搜索 **OsCurrTaskSet** 来跟踪哪些地方会切换当前任务以便更好的理解内核。
- **第二步：** 系统控制寄存器（**SCTLR**），**B4.1.130 SCTLR, System Control Register** 它提供了系统的最高级别控制，高到了玉皇大帝级别，代码中将 `0`、`2`、`12` 位写 `0`。对应关闭 **MMU** 、**数据缓存** 、**指令缓存** 功能。
- **第三步：** 对浮点运算 `FPU` 的设置，在安全模式下使用 `FPU` ，须定义 `NSACR`、`CPACR`、`FPEXC` 三个寄存器
- **第四步：** 计算虚拟地址和物理地址的偏移量，为何要计算它呢？主要目的是为了建立虚拟地址和物理地址的映射关系，因为在 MMU启动之后，运行地址(PC寄存器指向的地址)将变成虚拟地址，使用虚拟地址就离不开映射表，所以两个地址的映射关系需要在MMU启动前就创建好，而有了偏移量就可以创建映射表。但需先搞清楚 **链接地址** 和 **运行地址** 两个概念。
  - **链接地址** 由链接器确定，链接器会将所有输入的 **.o** 文件链接成一个 **.bin** 文件，它们都是**ELF**格式, 链接器给每条指令/数据都赋与一个地址，这个地址叫**链接地址**，它可以是相对的也可以是绝对的。但它们之间的内部距离是固定的，链接具体过程可翻看 **(重定位篇)** 和 **(链接脚本篇)**。
  - **运行地址** 由加载器确定，内核镜像首先通过烧录工具将内核烧录到 **flash** 指定的位置，开机后由**boot loader**工具，例如**uboot**，将内核镜像加载到指定地址后开始执行真正的内核代码，这个地址叫**运行地址**。

  两个地址值往往不一样，而内核设计者希望它们是一样的，那有没有办法检测二者是否一样呢?答案是： **有，那必须的** 。通过一个变量在链接时将其链接地址变成变量的内容 ，无论中间怎么加载变量的内容是不会变的，而获取运行地址是很容易获取的，其实就是PC寄存器的地址，二者相减，加载后偏了多少可不就出来了。别看说了这么一大串，实现却只需4行代码，但如果没有上面的废话每一行都会把您整懵逼。

```
pa_va_offset:
  .word   . //定义一个4字节的pa_va_offset 变量， 链接器生成一个链接地址， . 表示 pa_va_offset = 链接地址 举例: 在地址 0x17321796 中保存了 0x1732

adr     r11, pa_va_offset //代码已执行至此，指令将获取 pa_va_offset 的运行地址(可能不是`0x17321796`) 给r11
ldr     r0, [r11] // [r11]中存的是链接地址 `0x17321796`, 它不会随加载器变化的
sub     r11, r11, r0 // 二者相减得到了偏移地址
```

- **第五步：** 将内核代码从 **__exception_handlers** 处移到 **SYS_MEM_BASE**处,长度是 **__bss_start - __exception_handlers**，**__exception_handlers**是加载后的开始地址, 由加载器决定, 而**SYS_MEM_BASE** 是系统定义的内存地址, 可由系统集成商指定配置, 他们希望内核从这里运行。 下图为内



核镜像布局

具体代码如下:

```
    /* if we need to relocate to proper location or not | 如果需要重新安装到合适的位置*/
adr     r4, __exception_handlers        /* r4: base of load address | 加载基址*/
```

```
  ldr     r5, =SYS_MEM_BASE                 /* r5: base of physical address | 物理基址*/
  subs    r12, r4, r5                     /* r12: delta of load address and physical address | 二者偏移量*/
  beq     reloc_img_to_bottom_done          /* if we load image at the bottom of physical address | 不相等就需要重定位 */

  /* we need to relocate image at the bottom of physical address | 需要知道拷贝的大小*/
  ldr     r7, =__exception_handlers         /* r7: base of linked address (or vm address) | 链接地址基地址*/
  ldr     r6, =__bss_start                /* r6: end of linked address (or vm address)，由于目前阶段有用的数据是中断向量表+代码段+只读数据段+数据段
                所以只需复制[__exception_handlers,__bss_start]这段数据到内存基址处 */
  sub     r6, r7                          /* r6: delta of linked address (or vm address) | 内核镜像大小 */
  add     r6, r4                          /* r6: end of load address | 说明需拷贝[ r4,r4+r6 ] 区间内容到 [ r5,r5+r6 ]*/

  reloc_img_to_bottom_loop://重定位镜像到内核物理内存基地址,将内核从加载地址拷贝到内存基址处
     ldr     r7, [r4], #4 // 类似C语言 *r5 = *r4 , r4++ , r5++
     str     r7, [r5], #4 // #4 代表32位的指令长度,此时在拷贝内核代码区内容
     cmp     r4, r6          /* 拷贝完成条件. r4++ 直到等于r6 (加载结束地址) 完成拷贝动作 */
     bne     reloc_img_to_bottom_loop
     sub     pc, r12                         /* 重新校准pc寄存器, 无缝跳到了拷贝后的指令地址处执行 r12是重定位镜像前内核加载基地址和内核物理内存基地址的
     nop  // 注意执行完成sub       pc, r12后,新的PC寄存器也指向了  nop ,nop是伪汇编指令,等同于 mov r0 r0 通常用于控制时序的目的，强制内存对齐，防止
     sub     r11, r11, r12                   /* r11: eventual address offset | 最终地址偏移量 */
```

- **第六步：** 在打开MMU必须要做好虚拟地址和物理地址的映射关系，需构建页表，关于页表可翻看 **虚实映射篇**, 具体代码如下

```
#ifdef LOSCFG_KERNEL_MMU
ldr     r4, =g_firstPageTable             /* r4: physical address of translation table and clear it
            内核页表是用数组g_firstPageTable存储 见于los_arch_mmu.c */
add     r4, r4, r11                     //计算g_firstPageTable页表物理地址
mov     r0, r4        //因为默认r0 将作为memset_optimized的第一个参数
mov     r1, #0        //第二个参数,清0
mov     r2, #MMU_DESCRIPTOR_L1_SMALL_ENTRY_NUMBERS //第三个参数是L1表的长度
bl      memset_optimized                  /* optimized memset since r0 is 64-byte aligned | 将内核页表空间清零*/

ldr     r5, =g_archMmuInitMapping         //记录映射关系表
add     r5, r5, r11                     //获取g_archMmuInitMapping的物理地址
init_mmu_loop:                            //初始化内核页表
    ldmia   r5!, {r6-r10}                   /* r6 = phys, r7 = virt, r8 = size, r9 = mmu_flags, r10 = name | 物理地址、虚拟地址、映射大小、映射属性、
    cmp     r8, 0                         /* if size = 0, the mmu init done */
    beq     init_mmu_done                 //标志寄存器中Z标志位等于零时跳转到  init_mmu_done处执行
    bl      page_table_build              //创建页表
    b       init_mmu_loop      //循环继续
init_mmu_done:
    orr     r8, r4, #MMU_TTBRx_FLAGS          /* r8 = r4 and set cacheable attributes on translation walk | 设置缓存*/
    ldr     r4, =g_mmuJumpPageTable           /* r4: jump pagetable vaddr | 页表虚拟地址*/
    add     r4, r4, r11
    ldr     r4, [r4]
    add     r4, r4, r11                     /* r4: jump pagetable paddr | 页表物理地址*/

    /* build 1M section mapping, in order to jump va during turing on mmu:pa == pa, va == pa */
    /* 从当前PC开始建立1MB空间的段映射，分别建立物理地址和虚拟地址方式的段映射页表项
    * 内核临时页表在系统 使能mmu -> 切换到虚拟地址运行 这段时间使用
    */
    mov     r6, pc
    mov     r7, r6                        /* r7: pa (MB aligned)*/
    lsr     r6, r6, #20                     /* r6: pa l1 index */
    ldr     r10, =MMU_DESCRIPTOR_KERNEL_L1_PTE_FLAGS
    add     r12, r10, r6, lsl #20             /* r12: pa |flags */
    str     r12, [r4, r7, lsr #(20 - 2)]      /* jumpTable[paIndex] = pt entry */
    rsb     r7, r11, r6, lsl #20            /* r7: va */
    str     r12, [r4, r7, lsr #(20 - 2)]      /* jumpTable[vaIndex] = pt entry */

    bl      mmu_setup                       /* set up the mmu | 内核映射表已经创建好了,此时可以启动MMU工作了*/
#endif
```

- **第七步：** 使能MMU, 有了页表就可以使用虚拟地址了

```
mmu_setup: //启动MMU工作
    mov     r12, #0                       /* TLB Invalidate All entries - TLBIALL */
    mcr     p15, 0, r12, c8, c7, 0            /* Set c8 to control the TLB and set the mapping to invalid */
    isb
```

```
    mcr     p15, 0, r12, c2, c0, 2            /* Translation Table Base Control Register(TTBCR) = 0x0
                                    [31] :0 - Use the 32-bit translation system(虚拟地址是32位)
                                    [5:4]:0 - use TTBR0和TTBR1
                                    [2:0]:0 - TTBCR.N为0；
                                    例如：TTBCR.N为0，TTBR0[31:14-0] | VA[31-0:20] | descriptor-type[1:0]组成32位页表描述符的地址,
                                          VA[31:20]可以覆盖4GB的地址空间，所以TTBR0页表是16KB，不使用TTBR1;
                                    例如：TTBCR.N为1，TTBR0[31:14-1] | VA[31-1:20] | descriptor-type[1:0]组成32位页表描述符的地址,
                                          VA[30:20]可以覆盖2GB的地址空间，所以TTBR0页表是8KB，TTBR1页表是8KB(页表地址必须16KB对齐);
                                    */
    isb
    orr     r12, r4, #MMU_TTBRx_FLAGS   //将临时页表属性[6:0]和基地址[31:14]放到r12
    mcr     p15, 0, r12, c2, c0, 0            /* Set attributes and set temp page table */
    isb
    mov     r12, #0x7                    /* 0b0111 */
    mcr     p15, 0, r12, c3, c0, 0            /* Set DACR with 0b0111, client and manager domian */
    isb
    mrc     p15, 0, r12, c1, c0, 1            /* ACTLR, Auxlliary Control Register */
    orr     r12, r12, #(1 << 6)               /* SMP, Enables coherent requests to the processor. */
    orr     r12, r12, #(1 << 2)               /* Enable D-side prefetch */
    orr     r12, r12, #(1 << 11)               /* Global BP Enable bit */
    mcr     p15, 0, r12, c1, c0, 1            /* ACTLR, Auxlliary Control Register */
    dsb
    /*
    * 开始使能MMU，使用的是内核临时页表，这时cpu访问内存不管是取指令还是访问数据都是需要经过mmu来翻译，
    * 但是在mmu使能之前cpu使用的都是内核的物理地址，即使现在使能了mmu，cpu访问的地址值还是内核的物理地址值(这里仅仅从数值上来看)，
    * 而又由于mmu使能了，所以cpu会把这个值当做虚拟地址的值到页表中去找其对应的物理地址来访问。
    * 所以现在明白了为什么要在内核临时页表里建立一个内核物理地址和虚拟地址一一映射的页表项了吧，因为建立了一一映射，
    * cpu访问的地址经过mmu翻译得到的还是和原来一样的值，这样在cpu真正使用虚拟地址之前也能正常运行。
    */
    mrc     p15, 0, r12, c1, c0, 0
    bic     r12, #(1 << 29 | 1 << 28)         /* disable access flag[bit29]，ap[0]是访问权限位，支持全部的访问权限类型
                                    disable TEX remap[bit28]，使用TEX[2:0]与C Bbit控制memory region属性 */
    orr     r12, #(1 << 0)                    /* mmu enable */
    bic     r12, #(1 << 1)
    orr     r12, #(1 << 2)                   /* D cache enable */
    orr     r12, #(1 << 12)                   /* I cache enable */
    mcr     p15, 0, r12, c1, c0, 0            /* Set SCTLR with r12: Turn on the MMU, I/D cache Disable TRE/AFE */
    isb
    ldr     pc,  =1f                        /* Convert to VA | 1表示标号，f表示forward(往下) - pc值取往下标识符“1”的虚拟地址(跳转到标识符“1”处)
                                    因为之前已经在内核临时页表中建立了内核虚拟地址和物理地址的映射关系，所以接下来cpu切换到虚拟地址空间 */
    1:
        mcr     p15, 0, r8, c2, c0, 0             /* Go to the base address saved in C2: Jump to the page table */
        isb                                 //r8中保存的是内核L1页表基地址和flags，r8写入到TTBR0实现临时页表和内核页表的切换
        mov     r12, #0
        mcr     p15, 0, r12, c8, c7, 0            /* TLB Invalidate All entries - TLBIALL(Invalidate all EL1&0 regime stage 1 and 2 TLB entries) */
        isb
        sub     lr,  r11                        /* adjust lr with delta of physical address and virtual address |
                                               lr中保存的是mmu使能之前返回地址的物理地址值，这时需要转换为虚拟地址，转换算法也很简单，虚拟地址 = 物理地址 -
        bx      lr                            //返回
```

- **第八步：** 设置异常和中断栈 ，初始化栈内值和栈顶值

```
//初始化栈内值
    ldr     r0, =__svc_stack     //stack_init的第一个参数 __svc_stack表示栈顶
    ldr     r1, =__exc_stack_top //stack_init的第二个参数 __exc_stack_top表示栈底, 这里会有点绕, top表高地址位
    bl      stack_init              //初始化各个cpu不同模式下的栈空间
    //设置各个栈顶魔法数字
    STACK_MAGIC_SET __svc_stack, #OS_EXC_SVC_STACK_SIZE, OS_STACK_MAGIC_WORD //中断栈底设成"烫烫烫烫烫烫"
    STACK_MAGIC_SET __exc_stack, #OS_EXC_STACK_SIZE, OS_STACK_MAGIC_WORD     //异常栈底设成"烫烫烫烫烫烫"
stack_init:
    ldr     r2, =OS_STACK_INIT //0xCACACACA
    ldr     r3, =OS_STACK_INIT
    /* Main loop sets 32 bytes at a time. | 主循环一次设置 32 个字节*/
stack_init_loop:
    .irp    offset, #0, #8, #16, #24
    strd    r2, r3, [r0, \offset]    /* 等价于strd r2, r3, [r0, 0], strd r2, r3, [r0, 8], ... , strd r2, r3, [r0, 24] */
    .endr
    add     r0, #32   //加跳32个字节,说明在地址范围上 r1 > r0 ==> __exc_stack_top > __svc_stack
    cmp     r0, r1   //是否到栈底
```

```
    blt     stack_init_loop
    bx      lr
```

```
//初始化栈顶值
excstack_magic:
    mov     r3, #0 //r3 = 0
excstack_magic_loop:
    str     r2, [r0]   //栈顶设置魔法数字
    add     r0, r0, r1 //定位到栈底
    add     r3, r3, #1 //r3++
    cmp     r3, #CORE_NUM //栈空间等分成core_num个空间，所以每个core的栈顶需要magic num
    blt     excstack_magic_loop
    bx      lr
/* param0 is stack top, param1 is stack size, param2 is magic num */
.macro STACK_MAGIC_SET param0, param1, param2
    ldr     r0, =\param0
    mov     r1, \param1
    ldr     r2, =\param2
    bl      excstack_magic
.endm
STACK_MAGIC_SET __svc_stack, #OS_EXC_SVC_STACK_SIZE, OS_STACK_MAGIC_WORD //中断栈底设成"烫烫烫烫烫烫"
STACK_MAGIC_SET __exc_stack, #OS_EXC_STACK_SIZE, OS_STACK_MAGIC_WORD     //异常栈底设成"烫烫烫烫烫烫"
```

- **第九步：** 热启动

```
warm_reset: //热启动 Warm Reset, warm reboot, soft reboot, 在不关闭电源的情况，由软件控制重启计算机
  /* initialize CPSR (machine state register) */
  mov    r0, #(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|CPSR_SVC_MODE) /* 禁止IRQ中断 | 禁止FIQ中断 | 管理模式-操作系统使用的保护模式 */
  msr    cpsr, r0

  /* Note: some functions in LIBGCC1 will cause a "restore from SPSR"!! */
  msr    spsr, r0

  /* get cpuid and keep it in r12 */
  mrc     p15, 0, r12, c0, c0, 5  //R12保存CPUID
  and     r12, r12, #MPIDR_CPUID_MASK //掩码操作获取当前cpu id

  /* set svc stack, every cpu has OS_EXC_SVC_STACK_SIZE stack */
  ldr    r0, =__svc_stack_top
  mov    r2, #OS_EXC_SVC_STACK_SIZE
  mul    r2, r2, r12
  sub    r0, r0, r2                  /* 算出当前core的中断栈栈顶位置，写入所属core的sp */
  mov    sp, r0

  LDR    r0, =__exception_handlers
  MCR    p15, 0, r0, c12, c0, 0       /* Vector Base Address Register - VBAR */

  cmp    r12, #0
  bne    cpu_start                    //从核处理分支
```

- **第十步：** 进入 C 语言的 main()

```
bl     main                //带LR的子程序跳转, LR = pc - 4, 执行C层main函数

LITE_OS_SEC_TEXT_INIT INT32 main(VOID)//由主CPU执行,默认0号CPU 为主CPU
 {
    UINT32 ret = OsMain();
    if (ret != LOS_OK) {
        return (INT32)LOS_NOK;
    }

    CPU_MAP_SET(0, OsHwIDGet());//设置CPU映射,参数0 代表0号CPU

    OsSchedStart();//调度开始

    while (1) {
```

```
        __asm volatile("wfi");//WFI: wait for Interrupt 等待中断，即下一次中断发生前都在此hold住不干活
    }
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

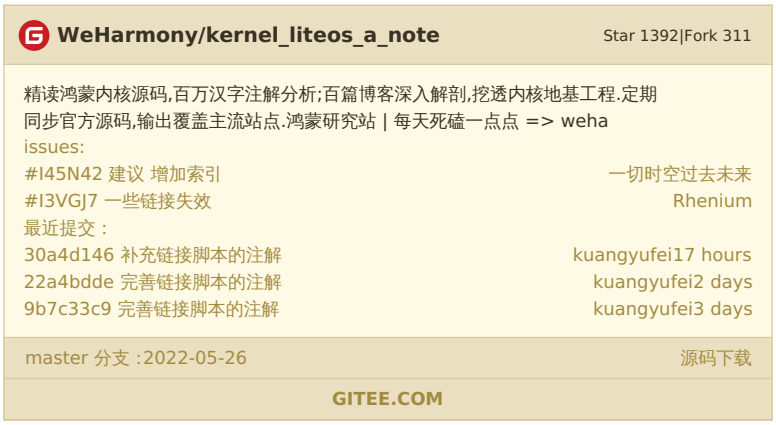
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

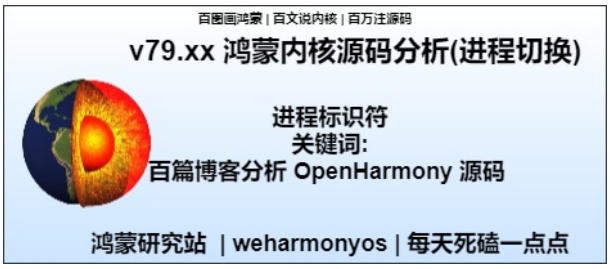


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

79_进程切换篇

本篇关键词：进程上下文、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

再看下那些地方会调用 `LOS_ArchMmuContextSwitch`，下图一目了然。

哪些地方会切换`mmu`上下文

- 第一：通过调度算法，被选中的进程的空间改变了，自然映射页表就跟着变了，需要切换mmu上下文，还是直接看代码。代码不是很多，就都贴出来了，都加了注释，不记得调度算法的可去系列篇中看 鸿蒙内核源码分析(调度机制篇)，里面有详细的阐述。

```
//调度算法-进程切换
STATIC VOID OsSchedSwitchProcess(LosProcessCB *runProcess， LosProcessCB *newProcess)
{
    if (runProcess == newProcess) {
        return;
    }
#if (LOSCFG_KERNEL_SMP == YES)
    runProcess->processStatus = OS_PROCESS_RUNTASK_COUNT_DEC(runProcess->processStatus);
    newProcess->processStatus = OS_PROCESS_RUNTASK_COUNT_ADD(newProcess->processStatus);

    LOS_ASSERT(!(OS_PROCESS_GET_RUNTASK_COUNT(newProcess->processStatus) > LOSCFG_KERNEL_CORE_NUM));
    if (OS_PROCESS_GET_RUNTASK_COUNT(runProcess->processStatus) == 0) {//获取当前进程的任务数量
#endif
        runProcess->processStatus &= ~OS_PROCESS_STATUS_RUNNING;
        if ((runProcess->threadNumber > 1) && !(runProcess->processStatus & OS_PROCESS_STATUS_READY)) {
            runProcess->processStatus |= OS_PROCESS_STATUS_PEND;
        }
#if (LOSCFG_KERNEL_SMP == YES)
    }
#endif
    LOS_ASSERT(!(newProcess->processStatus & OS_PROCESS_STATUS_PEND));//断言进程不是阻塞状态
    newProcess->processStatus |= OS_PROCESS_STATUS_RUNNING;//设置进程状态为运行状态
    if (OsProcessIsUserMode(newProcess)) {//用户模式下切换进程mmu上下文
        LOS_ArchMmuContextSwitch(&newProcess->vmSpace->archMmu);//新进程->虚拟空间中的->Mmu部分入参
    }
#ifdef LOSCFG_KERNEL_CPUP
    OsProcessCycleEndStart(newProcess->processID， OS_PROCESS_GET_RUNTASK_COUNT(runProcess->processStatus) + 1);
#endif /* LOSCFG_KERNEL_CPUP */
```

```
    OsCurrProcessSet(newProcess);//将进程置为 g_runProcess
    if ((newProcess->timeSlice == 0) && (newProcess->policy == LOS_SCHED_RR)) {//为用完时间片或初始进程分配时间片
        newProcess->timeSlice = OS_PROCESS_SCHED_RR_INTERVAL;//重新分配时间片，默认 20ms
    }
}
```

这里再啰嗦一句，系列篇中已经说了两个上下文切换了，一个是这里的因进程切换引起的mmu上下文切换，还一个是因task切换引起的CPU的上下文切换，还能想起来吗？

- 第二：是加载 `ELF` 文件的时候会切换 `mmu` ，一个崭新的进程诞生了，具体翻看(进程映像篇)。其余是虚拟空间回收和刷新空间的时候，这个就自己看代码去吧。mmu是如何快速的通过虚拟地址找到物理地址的呢？答案是： `TLB` ，注意上面还有个 `TTB` ，一个是寄存器， 一个是 `cache` ，别搞混了。

asid寄存器

asid(Adress Space ID) 进程标识符，属于CP15协处理器的C13号寄存器，ASID可用来唯一标识进程，并为进程提供地址空间保护。当TLB试图解析虚拟页号时，它确保当前运行进程的ASID与虚拟页相关的ASID相匹配。如果不匹配，那么就作为TLB失效。除了提供地址空间保护外，ASID允许TLB同时包含多个进程的条目。如果TLB不支持独立的ASID，每次选择一个页表时（例如，上下文切换时），TLB就必须被冲刷（flushed）或删除，以确保下一个进程不会使用错误的地址转换。

TLB页表中有一个bit来指明当前的entry是global(nG=0，所有process都可以访问)还是non-global(nG=1，only本process允许访问)。如果是global类型，则TLB中不会tag ASID；如果是non-global类型，则TLB会tag上ASID，且MMU在TLB中查询时需要判断这个ASID和当前进程的ASID是否一致，只有一致才证明这条entry当前process有权限访问。

看到了吗？如果每次mmu上下文切换时，把TLB全部刷新已保证TLB中全是新进程的映射表，固然是可以，但效率太低了！！！进程的切换其实是秒级亚秒级的，地址的虚实转换是何等的频繁啊，怎么会这么现实呢，真实的情况是TLB中有很多很多其他进程占用的物理内存的记录还在，当然他们对物理内存的使用权也还在。所以当应用程序 new了10M内存以为是属于自己的时候，其实在内核层面根本就不属于你，还是别人在用，只有你用了1M的那一瞬间真正1M物理内存才属于你，而且当你的进程被其他进程切换后，很大可能你用的那1M也已经不在物理内存中了，已经被置换到硬盘上了。明白了吗？只关注应用开发的同学当然可以说这关我鸟事，给我的感觉有就行了，但想熟悉内核的同学就必须要明白，这是每分每秒都在发生的事情。

最后一个函数留给大家，asid是如何分配的？

```
/* allocate and free asid */
status_t OsAllocAsid(UINT32 *asid)
{
    UINT32 flags;
    LOS_SpinLockSave(&g_cpuAsidLock， &flags);
    UINT32 firstZeroBit = LOS_BitmapFfz(g_asidPool， 1UL << MMU_ARM_ASID_BITS);
    if (firstZeroBit >= 0 && firstZeroBit < (1UL << MMU_ARM_ASID_BITS)) {
        LOS_BitmapSetNBits(g_asidPool， firstZeroBit， 1);
        *asid = firstZeroBit;
        LOS_SpinUnlockRestore(&g_cpuAsidLock， flags);
        return LOS_OK;
    }

    LOS_SpinUnlockRestore(&g_cpuAsidLock， flags);
    return firstZeroBit;
}
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

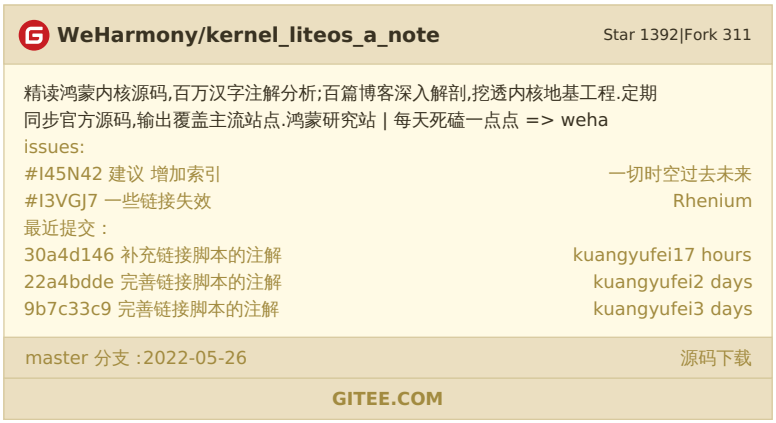
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

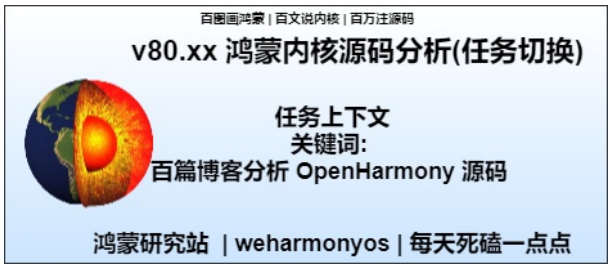


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

80_任务切换篇

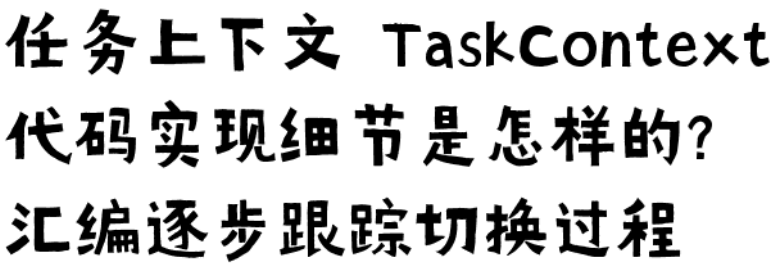
本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...



本篇说清楚线程环境下的任务切换

在鸿蒙的内核线程就是任务，系列篇中说的任务和线程当一个东西去理解。

一般二种场景下需要切换任务上下文:

- 在线程环境下，从当前线程切换到目标线程，这种方式也称为软切换，能由软件控制的自主式切换。哪些情况下会出现软切换呢？
  - 运行的线程申请某种资源(比如各种锁，读/写消息队列)失败时，需要主动释放CPU的控制权，将自己挂入等待队列，调度算法重新调度新任务运行。
  - 每隔10ms就执行一次的 `OsTickHandler` 节拍处理函数，检测到任务的时间片用完了，就发起任务的重新调度，切换到新任务运行。
  - 不管是内核态的任务还是用户态的任务，于切换而言是统一处理，一视同仁的，因为切换是需要换栈运行，寄存器有限，需要频繁的复用，这就需要将当前寄存器值先保存到任务自己的栈中，以便别人用完了轮到自己再用时恢复寄存器当时的值，确保老任务还能继续跑下去。而保存寄存器顺序的结构体叫:任务上下文(`TaskContext`)。
- 在中断环境下，从当前线程切换到目标线程，这种方式也称为硬切换。不由软件控制的被动式切换。哪些情况下会出现硬切换呢？
  - 由硬件产生的中断，比如 鼠标，键盘外部设备每次点击和敲打，屏幕的触摸，USB的插拔等等这些都是硬中断。同样的需要切换栈运行，需要复用寄存器，但与软切换不一样的是，硬切换会切换工作模式(中断模式)。所以会更复杂点，但道理还是一样要保存和恢复切换现场寄存器的值， 而保存寄存器顺序的结构体叫:任务中断上下文(`TaskIrqContext`)。

本篇说清楚在线程环境下切换(软切换)的实现过程。中断切换(硬切换)实现过程将在v08。xx 鸿蒙内核源码分析(总目录) 中断切换篇中详细说明。

本篇具体说清楚以下几个问题:

- 任务上下文(TaskContext)怎么保存的？
- 代码的实现细节是怎样的？
- 如何保证切换不会发生错误，指令不会丢失？

在 v08.xx 鸿蒙内核源码分析(总目录) 系列篇中已经说清楚了调度机制，线程概念，寄存器，CPU，工作模式，这些是读懂本篇的基础，建议先前往翻看，不然理解本篇会费劲。本篇代码量较多，涉及C和汇编代码，代码都添加了注释，试图把任务的整个切换过程逐行逐行说清楚。

前置条件

一个任务要跑起来，需要两个必不可少的硬性条件:

- 1.从代码段哪个位置取指令？ 也就是入口地址，main函数是应用程序的入口地址， 注意main函数也是一个线程，只是不需要你来new而已，加载程序阶段会默认创建好。 run()是new一个线程执行的入口地址。高级语言是这么叫，但到了汇编层的叫法就是PC寄存器。给PC寄存器赋什么值，指令就从哪里开始执行。
- 2.运行的场地(栈空间)在哪里？ ARM有7种工作模式，到了进程层面只需要考虑内核模式和用户模式两种，对应到任务会有内核态栈空间和用户态栈空间。内核模式的任务只有内核态的栈空间，用户模式任务二者都有。栈空间是在初始化一个任务时就分配指定好的。以下是两种栈空间的初始化过程。为了精练省去了部分代码，留下了核心部分。

```
//任务控制块中对两个栈空间的描述
typedef struct {
    VOID            *stackPointer;      /**< Task stack pointer */  //内核态栈指针，SP位置，切换任务时先保存上下文并指向TaskContext位置。
    UINT32          stackSize;          /**< Task stack size */     //内核态栈大小
    UINTPTR         topOfStack;         /**< Task stack top */      //内核态栈顶 bottom = top + size
    // ....
    UINTPTR         userArea;       //使用区域，由运行时划定，根据运行态不同而不同
    UINTPTR         userMapBase;    //用户态下的栈底位置
    UINT32          userMapSize;    /**< user thread stack size ，real size : userMapSize + USER_STACK_MIN_SIZE */
} LosTaskCB;
```

```
//内核态运行栈初始化
LITE_OS_SEC_TEXT_INIT VOID *OsTaskStackInit(UINT32 taskID， UINT32 stackSize， VOID *topStack， BOOL initFlag)
{
    UINT32 index = 1;
    TaskContext *taskContext = NULL;
    taskContext = (TaskContext *)(((UINTPTR)topStack + stackSize) - sizeof(TaskContext));//上下文存放在栈的底部
    /* initialize the task context */ //初始化任务上下文
    taskContext->PC = (UINTPTR)OsTaskEntry;//程序计数器，CPU首次执行task时跑的第一条指令位置
    taskContext->LR = (UINTPTR)OsTaskExit;  /* LR should be kept， to distinguish it's THUMB or ARM instruction */
    taskContext->resved = 0x0;
    taskContext->R[0] = taskID;             /* R0 */
    taskContext->R[index++] = 0x01010101;   /* R1， 0x01010101 : reg initialed magic word */ //0x55
    for (; index < GEN_REGS_NUM; index++) {//R2 - R12的初始化很有意思
        taskContext->R[index] = taskContext->R[index - 1] + taskContext->R[1]; /* R2 - R12 */
    }
    taskContext->regPSR = PSR_MODE_SVC_ARM;   /* CPSR (Enable IRQ and FIQ interrupts， ARM-mode) */
    return (VOID *)taskContext;
}
```

```
//用户态运行栈初始化
LITE_OS_SEC_TEXT_INIT VOID OsUserTaskStackInit(TaskContext *context， TSK_ENTRY_FUNC taskEntry， UINTPTR stack)
{
    context->regPSR = PSR_MODE_USR_ARM;//工作模式:用户模式 + 工作状态:arm
    context->R[0] = stack;//栈指针给r0寄存器
    context->SP = TRUNCATE(stack， LOSCFG_STACK_POINT_ALIGN_SIZE);//给SP寄存器值使用
    context->LR = 0;//保存子程序返回地址 例如 a call b ，在b中保存 a地址
    context->PC = (UINTPTR)taskEntry;//入口函数
}
```

您一定注意到了 TaskContext ，说的全是它，这就是任务上下文结构体，理解它是理解任务切换的钥匙。它不仅在C语言层面出现，而且还在汇编层出现， TaskContext 是连接或者说打通 C->汇编->C 实现任务切换的最关键概念。本篇全是围绕着它来展开。先看看它张啥样，LOOK!

TaskContext 任务上下文

```
typedef struct {
#if !defined(LOSCFG_ARCH_FPU_DISABLE)
    UINT64 D[FP_REGS_NUM]; /* D0-D31 */
    UINT32 regFPSCR;       /* FPSCR */
    UINT32 regFPEXC;       /* FPEXC */
#endif
    UINT32 resved;         /* It's stack 8 aligned */
    UINT32 regPSR;
    UINT32 R[GEN_REGS_NUM]; /* R0-R12 */
    UINT32 SP;             /* R13 */
    UINT32 LR;             /* R14 */
    UINT32 PC;             /* R15 */
} TaskContext;
```

- 结构很简单，目的更简单，就是用来保存寄存器现场的值的。 v08.xx 鸿蒙内核源码分析(总目录) 系列寄存器篇中已经说过了，到了汇编层就是寄存器在玩，当CPU工作在用户和系统模式下时寄存器是复用的，玩的是17个寄存器和内存地址，访问内存地址也是通过寄存器来玩。
- 哪17个？ R0~R15和CPSR。 当调度(主动式)或者中断(被动式)发生时。将这17个寄存器压入任务的内核栈的过程叫保护案发现场。从任务栈中弹出依次填入寄存器的过程叫恢复案发现场。
- 从栈空间的具体哪个位置开始恢复呢？ 答案是: `stackPointer`，任务控制块(LosTaskCB)的首个变量。对应到汇编层的就是SP寄存器。
- 而 `TaskContext` (任务上下文)就是一定的顺序来保存和恢复这17个寄存器的。任务上下文在任务还没有开始执行的时候就已经保存在内核栈中了，只不过是一些默认的值， `OsTaskStackInit` 干的就是这个默认的事。 而 `OsUserTaskStackInit` 是对用户栈的初始化，改变的是(CPSR)工作模式和SP寄存器。
- 新任务的运行栈指针(`stackPointer`)给SP寄存器意味着切换了运行栈，这是本篇最重要的一句话。

以下通过汇编代码逐行分析如何保存和恢复 `TaskContext` (任务上下文)

OsSchedResched 调度算法

```
//调度算法的实现
VOID OsSchedResched(VOID)
{
    // ...此处省去 ...
    /* do the task context switch */
    OsTaskSchedule(newTask， runTask);//切换任务上下文，注意OsTaskSchedule是一个汇编函数 见于 los_dispatch.S
}
```

- 在v08.xx 鸿蒙内核源码分析(总目录) 之调度机制篇中，留了一个问题， `OsTaskSchedule` 不是一个C函数，而是个汇编函数，就没有往下分析了，本篇要完成整个分析过程。 `OsTaskSchedule` 实现了任务的上下文切换，汇编代码见于los_dispatch.S中
- `OsTaskSchedule` 的参数指向的是新老两个任务，这两个参数分别保存在R0，R1寄存器中。

OsTaskSchedule 汇编实现

读这段汇编代码一定要对照上面的 `TaskContext` ，不然很难看懂，容易懵圈，但对照着看就秒懂。

```
/*
 * R0: new task
 * R1: run task
 */
OsTaskSchedule: /*任务调度，OsTaskSchedule的目的是将寄存器值按TaskContext的格式保存起来*/
    MRS     R2， CPSR  /*MRS 指令用于将特殊寄存器(如 CPSR 和 SPSR)中的数据传递给通用寄存器，要读取特殊寄存器的数据只能使用 MRS 指令*/
    STMFD   SP!， {LR} /*返回地址入栈，LR = PC-4 ，对应TaskContext->PC(R15寄存器)*/
    STMFD   SP!， {LR} /*再次入栈对应，对应TaskContext->LR(R14寄存器)*/
    /* jump sp */
    SUB     SP， SP， #4 /* 跳的目的是为了，对应TaskContext->SP(R13寄存器)*/
    /* push r0-r12*/
    STMFD   SP!， {R0-R12}  @对应TaskContext->R[GEN_REGS_NUM](R0~R12寄存器)。
    STMFD   SP!， {R2}  /*R2 入栈 对应TaskContext->regPSR*/
    /* 8 bytes stack align */
    SUB     SP， SP， #4  @栈对齐，对应TaskContext->resved
    /* save fpu registers */
    PUSH_FPU_REGS   R2 /*保存fpu寄存器*/
    /* store sp on running task */
```

```
    STR     SP， [R1] @在运行的任务栈中保存SP，即runTask->stackPointer = sp

OsTaskContextLoad: @加载上下文
    /* clear the flag of ldrex */ @LDREX 可从内存加载数据，如果物理地址有共享TLB属性，则LDREX会将该物理地址标记为由当前处理器独占访问，并且会清除该
    CLREX @清除ldrex指令的标记
    /* switch to new task's sp */
    LDR     SP， [R0] @ 即:sp =  task->stackPointer
    /* restore fpu registers */
    POP_FPU_REGS    R2 @恢复fpu寄存器，这里用了汇编宏R2是宏的参数
    /* 8 bytes stack align */
    ADD     SP， SP， #4 @栈对齐
    LDMFD   SP!， {R0}  @此时SP!位置保存的是CPSR的内容，弹出到R0
    MOV     R4， R0 @R4=R0，将CPSR保存在r4， 将在OsKernelTaskLoad中保存到SPSR
    AND     R0， R0， #CPSR_MASK_MODE @R0 =R0&CPSR_MASK_MODE ，目的是清除高16位
    CMP     R0， #CPSR_USER_MODE @R0 和 用户模式比较
    BNE     OsKernelTaskLoad @非用户模式则跳转到OsKernelTaskLoad执行，跳出
    /*此处省去 LOSCFG_KERNEL_SMP 部分*/
    MVN     R3， #CPSR_INT_DISABLE @按位取反 R3 = 0x3F
    AND     R4， R4， R3  @使能中断
    MSR     SPSR_cxsf， R4 @修改spsr值
    /* restore r0-r12， lr */
    LDMFD   SP!， {R0-R12} @恢复寄存器值
    LDMFD   SP， {R13， R14}^ @恢复SP和LR的值，注意此时SP值已经变了，CPU换地方上班了。
    ADD     SP， SP， #(2 * 4)@sp = sp + 8
    LDMFD   SP!， {PC}^  @恢复PC寄存器值，如此一来 SP和PC都有了新值，完成了上下文切换。完美!
OsKernelTaskLoad:    @内核任务的加载
    MSR     SPSR_cxsf， R4  @将R4整个写入到程序状态保存寄存器
    /* restore r0-r12， lr */
    LDMFD   SP!， {R0-R12}  @出栈，依次保存到 R0-R12，其实就是恢复现场
    ADD     SP， SP， #4   @sp=SP+4
    LDMFD   SP!， {LR， PC}^  @返回地址赋给pc指针，直接跳出。
```

解读

- 汇编分成了三段 `OsTaskSchedule` ， `OsTaskContextLoad` ， `OsKernelTaskLoad` 。
- 第一段 `OsTaskSchedule` 其实就是在保存现场。代码都有注释，对照着 `TaskContext` 来的，它就干了一件事把17个寄存器的值按 `TaskContext` 的格式入栈，因为鸿蒙用栈方式采用的是满栈递减的方式，所以存放顺序是从最后一个往前依次入栈。
- 连着来两句 `STMFD SP!， {LR}` 之前让笔者懵圈了很久， 看了 `TaskContext` 才恍然大悟，因为三级流水线的原因，LR和PC寄存器之间是差了一条指令的，LR指向了处于译码阶段指令，而PC指向了取指阶段的指令，所以此处做了两次LR入栈，其实是保存了未执行的译码指令地址，确保执行不会丢失一条指令。
- R1是正在运行的任务栈， `OsTaskSchedule` 总的理解是在任务R1的运行栈中插入一个 `TaskContext` 结构块。而 `STR SP，[R1]` ，是改变了LosTaskCB->stackPointer的值，这个值只能在汇编层进行精准的改变，而在整个鸿蒙内核C代码层面都没有看到对它有任何修改的地方。这个改变意义极为重要。因为新的任务被调度后的第一件事情就是恢复现场!!!
- 在 `OsTaskSchedule` 执行完成后，因为PC寄存器并没有发生跳转，所以紧接着往下执行 `OsTaskContextLoad`
- `OsTaskContextLoad` 的任务就是恢复现场，谁的现场？当然是R0: new task的，所以第一条指令就是 `CLREX` ，清除干净后立马执行 `LDR SP，[R0]` ，所指向的就是 `LosTaskCB->stackPointer` ，这个位置存的是新任务的 `TaskContext` 结构块，是上一次R0任务被打断时保存下来当时这17个寄存器的值啊，依次出栈就是恢复这17个寄存器的值。
- `OsTaskContextLoad` 在开始之前会判断下工作模式，即判断下是内核栈还是用户栈，两种处理方式稍有不同。但都是在恢复现场。
- `BNE OsKernelTaskLoad` 是查询CPSR后判断此时为内核栈的现场恢复过程，代码很简单就是恢复17个寄存器。 如此一来，任务执行的两个条件，第一个SP的在 `LDR SP，[R0]` 时就有了。第二个条件:PC寄存器的值也在最后一条汇编 `LDMFD SP!，{LR，PC}^` 也已经有了。改变了PC和LR有了新值，下一条指令位置一样是上次任务被中断时还没被执行的处于译码阶段的指令地址。
- 如果是用户态区别是需要恢复中断。因为用户模式的优先级是最低的，必须允许响应中断，也是依次恢复各寄存器的值，最后一句 `LDMFD SP!，{PC}^` 结束本次旅行，下一条指令位置一样是上次任务被中断时还没被执行的处于译码阶段的指令地址。
- 如此，说清楚了任务上下文切换的整个过程，初看可能不太容易理解，建议多看几篇，用笔画下栈的运行过程，脑海中会很清晰的浮现出整个切换过程的运行图。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很

重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。

- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

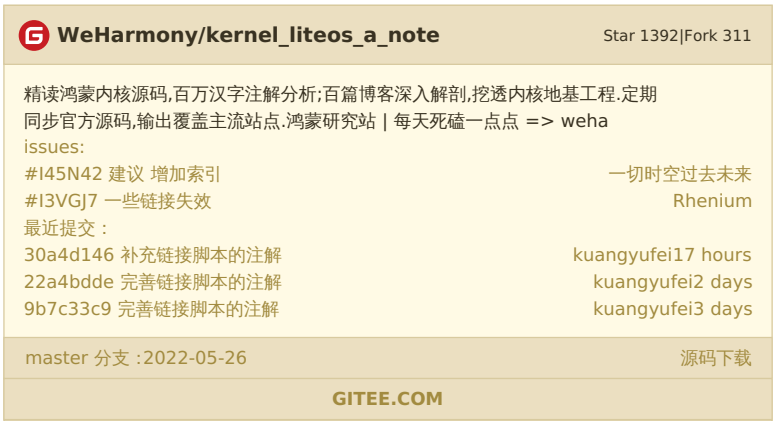
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		

编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

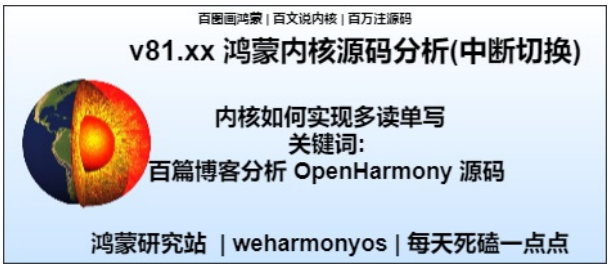


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

81_中断切换篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

关于中断部分系列篇将用三篇详细说明整个过程。

- **中断概念篇** 中断概念很多，比如中断控制器，中断源，中断向量，中断共享，中断处理程序等等。本篇做一次整理。先了解透概念才好理解中断过程。用海公公打比方说明白中断各个概念。可前往 v08.xx 鸿蒙内核源码分析(总目录) 查看。
- **中断管理篇** 从中断初始化 `HalIrqInit` 开始，到注册中断的 `LOS_HwiCreate` 函数，到消费中断函数的 `HalIrqHandler` ，剖析鸿蒙内核实现中断的过程，很像设计模式中的观察者模式。
- **中断切换篇(本篇)** 用自下而上的方式，从中断源头纯汇编代码往上跟踪代码细节。说清楚保存和恢复中断现场 `TaskIrqContext` 过程。

中断环境下的任务切换

在鸿蒙的内核线程就是任务，系列篇中说的任务和线程当一个东西去理解。

一般二种场景下需要切换任务上下文:

- 在中断环境下，从当前线程切换到目标线程，这种方式也称为硬切换。它们通常由硬件产生或是软件发生异常时的被动式切换。哪些情况下会出现硬切换呢？
    - 中断源可分外部和内部中断源两大类，比如 鼠标，键盘外部设备每次点击和敲打，屏幕的触摸，USB的插拔等等这些都是外部中断源。存储器越限、缺页，核间中断，断点中断等等属于内部中断源。由此产生的硬切换都需要换栈运行，硬切换硬在需切换工作模式(中断模式)。所以会比线程环境下的切换更复杂点，但道理还是一样要保存和恢复切换现场寄存器的值， 而保存寄存器顺序格式结构体叫:任务中断上下文(`TaskIrqContext`)。
- 在线程环境下，从当前线程切换到目标线程，这种方式也称为软切换，能由软件控制的自主式切换。哪些情况下会出现软切换呢？
    - 运行的线程申请某种资源(比如各种锁，读/写消息队列)失败时，需要主动释放CPU的控制权，将自己挂入等待队列，调度算法重新调度新任务运行。
    - 每隔10ms就执行一次的 `OsTickHandler` 节拍处理函数，检测到任务的时间片用完了，就发起任务的重新调度，切换到新任务运行。
    - 不管是内核态的任务还是用户态的任务，于切换而言是统一处理，一视同仁的，因为切换是需要换栈运行，寄存器有限，需要频繁的复用，这就需要将当前寄存器值先保存到任务自己的栈中，以便别人用完了轮到自己再用时恢复寄存器当时的值，确保老任务还能继续跑下去。而保存寄存器顺序格式结构体叫:任务上下文(`TaskContext`)。

本篇说清楚在中断环境下切换(硬切换)的实现过程。线程切换(软切换)实现过程已在v08.xx 鸿蒙内核源码分析(总目录) 任务切换篇中详细说明。

ARM的七种工作模式中，有两个是和中断相关。

- **普通中断模式（irq）**：一般中断模式也叫普通中断模式，用于处理一般的中断请求，通常在硬件产生中断信号之后自动进入该模式，该模式可以自由访问系统硬件资源。
- **快速中断模式（fiq）**：快速中断模式是相对一般中断模式而言的，用来处理高优先级中断的模式，处理对时间要求比较紧急的中断请求，主要用于高速数据传输及通道处理中。

此处分析普通中断模式下的任务切换过程。

普通中断模式相关寄存器

这张图一定要刻在脑海里，系列篇会多次拿出来，目的是为了能牢记它。

- 普通中断模式(图中IRQ列)是一种异常模式，有自己独立运行的栈空间。一个(IRQ)中断发生后，硬件会将CPSR寄存器工作模式置为IRQ模式。并跳转到入口地址`OsIrqHandler`执行。

```
#define OS_EXC_IRQ_STACK_SIZE    64 //中断模式栈大小 64个字节
__irq_stack:
    .space OS_EXC_IRQ_STACK_SIZE * CORE_NUM
__irq_stack_top:
```

- `OsIrqHandler`汇编代码实现过程，就干了三件事:
  - 1.保存任务中断上下文`TaskIrqContext`
  - 2.执行中断处理程序`HalIrqHandler`，这是个C函数，由汇编调用
  - 3.恢复任务中断上下文`TaskIrqContext`，返回被中断的任务继续执行

TaskIrqContext 和 TaskContext

先看本篇结构体 TaskIrqContext

```
#define TASK_IRQ_CONTEXT \
        unsigned int R0;     \
        unsigned int R1;     \
```

```
    unsigned int R2;    \
    unsigned int R3;    \
    unsigned int R12;   \
    unsigned int USP;   \
    unsigned int ULR;   \
    unsigned int CPSR;  \
    unsigned int PC;

typedef struct {//任务中断上下文
#if !defined(LOSCFG_ARCH_FPU_DISABLE)
  UINT64 D[FP_REGS_NUM]; /* D0-D31 */
  UINT32 regFPSCR;      /* FPSCR */
  UINT32 regFPEXC;      /* FPEXC */
#endif
  UINT32 resved;
  TASK_IRQ_CONTEXT
} TaskIrqContext;
```

```
typedef struct {//任务上下文，已在任务切换篇中详细说明，放在此处是为了对比
#if !defined(LOSCFG_ARCH_FPU_DISABLE)
  UINT64 D[FP_REGS_NUM]; /* D0-D31 */
  UINT32 regFPSCR;      /* FPSCR */
  UINT32 regFPEXC;      /* FPEXC */
#endif
  UINT32 resved;        /* It's stack 8 aligned */
  UINT32 regPSR;        //保存CPSR寄存器
  UINT32 R[GEN_REGS_NUM]; /* R0-R12 */
  UINT32 SP;            /* R13 */
  UINT32 LR;            /* R14 */
  UINT32 PC;            /* R15 */
} TaskContext;
```

- 两个结构体很简单，目的更简单，就是用来保存寄存器现场的值的。 `TaskContext` 把17个寄存器全部保存了， `TaskIrqContext` 保存的少些，在栈中并没有保存R4-R11寄存器的值，这说明在整个中断处理过程中，都不会用到R4-R11寄存器。不会用到就不会改变，当然就没必要保存了。这也说明内核开发者的严谨程度，不造成时间和空间上的一丁点浪费。效率的提升是从细节处入手的，每个小地方优化那么一丢丢，整体性能就上来了。
- `TaskIrqContext` 中有两个变量有点奇怪 `unsigned int USP;` `unsigned int ULR;` 指的是用户模式下的SP和LR值， 这个要怎么理解？ 因为对一个正运行的任务而言，中断的到来是颗不定时炸弹，无法预知，也无法提前准备，中断一来它立即被打断，压根没有时间去保存现场到自己的栈中，那保存工作只能是放在IRQ栈或者SVC栈中。而IRQ栈非常的小，只有64个字节，16个栈空间，指望不上了，就保存在SVC栈中，SVC模式栈可是有 8K空间的。
- 从接下来的 `OsIrqHandler` 代码中可以看出，鸿蒙内核整个中断的工作其实都是在SVC模式下完成的，而irq的栈只是个过渡栈。具体看汇编代码逐行注解分析。

普通中断处理程序

```
OsIrqHandler: @硬中断处理，此时已切换到硬中断栈
  SUB    LR， LR， #4 @记录译码指令地址，以防切换过程丢失指令

  /* push r0-r3 to irq stack */ @irq栈只是个过渡栈
  STMFD  SP， {R0-R3}  @r0-r3寄存器入 irq 栈
  SUB    R0， SP， #(4 * 4)@r0 = sp - 16，目的是记录{R0-R3}4个寄存器保存的开始位置，届时从R3开始出栈
  MRS    R1， SPSR  @获取程序状态控制寄存器
  MOV    R2， LR   @r2=lr

  /* disable irq， switch to svc mode */@超级用户模式(SVC 模式)，主要用于 SWI(软件中断)和 OS(操作系统)。
  CPSID  i， #0x13    @切换到SVC模式，此处一切换，后续指令将在SVC栈运行
     @CPSID i为关中断指令，对应的是CPSIE
  @TaskIrqContext 开始保存中断现场 ......
  /* push spsr and pc in svc stack */
  STMFD  SP!， {R1， R2} @实际是将 SPSR，和PC入栈对应TaskIrqContext.PC，TaskIrqContext.CPSR，
  STMFD  SP， {LR}   @LR再入栈，SP不自增，如果是用户模式，LR值将被 282行:STMFD  SP， {R13， R14}^覆盖
    @如果非用户模式，将被 286行:SUB    SP， SP， #(2 * 4) 跳过。
  AND    R3， R1， #CPSR_MASK_MODE @获取CPU的运行模式
  CMP    R3， #CPSR_USER_MODE  @中断是否发生在用户模式
  BNE    OsIrqFromKernel   @非用户模式不用将USP，ULR保存在TaskIrqContext
```

```
    /* push user sp， lr in svc stack */
    STMFD   SP， {R13， R14}^   @将用户模式的sp和LR入svc栈

OsIrqFromKernel: @从内核发起中断
    /* from svc not need save sp and lr */@svc模式下发生的中断不需要保存sp和lr寄存器值
    SUB     SP， SP， #(2 * 4) @目的是为了留白给 TaskIrqContext.USP，TaskIrqContext.ULR
        @TaskIrqContext.ULR已经在 276行保存了，276行用的是SP而不是SP!，所以此处要跳2个空间
    /* pop r0-r3 from irq stack*/
    LDMFD   R0， {R0-R3}      @从R0位置依次出栈

    /* push caller saved regs as trashed regs in svc stack */
    STMFD   SP!， {R0-R3， R12} @寄存器入栈，对应 TaskIrqContext.R0~R3，R12

    /* 8 bytes stack align */
    SUB     SP， SP， #4   @栈对齐 对应TaskIrqContext.resved

    /*
     * save fpu regs in case in case those been
     * altered in interrupt handlers.
     */
    PUSH_FPU_REGS   R0 @保存fpu regs，以防中断处理程序中的fpu regs被修改。
    @TaskIrqContext 结束保存中断现场.....
    @开始执行真正的中断处理函数了。
#ifdef LOSCFG_IRQ_USE_STANDALONE_STACK @是否使用了独立的IRQ栈
    PUSH    {R4} @R4先入栈保存，接下来要切换栈，需保存现场
    MOV     R4， SP @R4=SP
    EXC_SP_SET __svc_stack_top， OS_EXC_SVC_STACK_SIZE， R1， R2 @切换到svc栈
#endif
 /*BLX 带链接和状态切换的跳转*/
    BLX     HalIrqHandler /* 调用硬中断处理程序，无参 ，说明HalIrqHandler在svc栈中执行 */

#ifdef LOSCFG_IRQ_USE_STANDALONE_STACK @是否使用了独立的IRQ栈
    MOV     SP， R4 @恢复现场，sp = R4
    POP     {R4} @弹出R4
#endif

    /* process pending signals */  @处理挂起信号
    BL      OsTaskProcSignal   @跳转至C代码

    /* check if needs to schedule */@检查是否需要调度
    CMP     R0， #0 @是否需要调度，R0为参数保存值
    BLNE    OsSchedPreempt @不相等，即R0非0，一般是 1

    MOV     R0，SP @参数
    MOV     R1，R7 @参数
    BL      OsSaveSignalContextIrq @跳转至C代码

    /* restore fpu regs */
    POP_FPU_REGS    R0 @恢复fpu寄存器值

    ADD     SP， SP， #4 @sp = sp + 4

OsIrqContextRestore: @恢复硬中断环境
    LDR     R0， [SP， #(4 * 7)] @R0 = sp + 7，目的是跳到恢复中断现场TaskIrqContext.CPSR位置，刚好是TaskIrqContext倒数第7的位置。
    MSR     SPSR_cxsf， R0  @恢复spsr 即:spsr = TaskIrqContext.CPSR
    AND     R0， R0， #CPSR_MASK_MODE @掩码找出当前工作模式
    CMP     R0， #CPSR_USER_MODE @是否为用户模式？
    @TaskIrqContext 开始恢复中断现场 .....
    LDMFD   SP!， {R0-R3， R12} @从SP位置依次出栈 对应 TaskIrqContext.R0~R3，R12
        @此时已经恢复了5个寄存器，接来下是TaskIrqContext.USP，TaskIrqContext.ULR
    BNE     OsIrqContextRestoreToKernel @看非用户模式，怎么恢复中断现场。

    /* load user sp and lr， and jump cpsr */
    LDMFD   SP， {R13， R14}^ @出栈，恢复用户模式sp和lr值 即:TaskIrqContext.USP，TaskIrqContext.ULR
    ADD     SP， SP， #(3 * 4) @跳3个位置，跳过 CPSR ，因为上一句不是 SP!，所以跳3个位置，刚好到了保存TaskIrqContext.PC的位置

    /* return to user mode */
    LDMFD   SP!， {PC}^ @回到用户模式，整个中断过程完成
    @TaskIrqContext 结束恢复中断现场(用户模式下) .....

OsIrqContextRestoreToKernel:@从内核恢复中断
```

```
    /* svc mode not load sp */
    ADD     SP， SP， #4 @其实是跳过TaskIrqContext.USP，因为在内核模式下并没有保存这个值，翻看 287行
    LDMFD   SP!， {LR} @弹出LR
    /* jump cpsr and return to svc mode */
    ADD     SP， SP， #4 @跳过cpsr
    LDMFD   SP!， {PC}^ @回到svc模式，整个中断过程完成
    @TaskIrqContext 结束恢复中断现场(内核模式下) ......
```

逐句解读

- 跳转到 `OsIrqFromKernel` 硬件会自动切换到 `__irq_stack` 执行
- 1句: `SUB LR， LR， #4` 在arm执行过程中一般分为取指，译码，执行阶段，而PC是指向取指，正在执行的指令为 PC-8 ，译码指令为PC-4。当中断发生时硬件自动执行 mov lr pc， 中间的PC-4译码指令因为没有寄存器去记录它，就会被丢失掉。所以 `SUB LR， LR， #4` 的结果是lr = PC -4 ，定位到了被中断时译码指令，将在栈中保存这个位置，确保回来后能继续执行。
- 2句: `STMFD SP， {R0-R3}` 当前4个寄存器入 `__irq_stack` 保存
- 3句: `SUB R0， SP， #(4 * 4)` 因为SP没有自增，R0跳到保存R0内容地址
- 4，5句:读取SPSR，LR寄存器内容，目的是为了后面在SVC栈中保存 `TaskIrqContext`
- 6句: `CPSID i， #0x13` 禁止中断和切换SVC模式，执行完这条指令后工作模式将切到 SVC模式
- @TaskIrqContext 开始保存中断现场 ......
- 中间代码需配合 `TaskIrqContext` 来看，不然100%懵逼。结合看就秒懂，代码都已经注释，不再做解释，注解中提到的 翻看276行 是指源码的第276行，请对照注解源码看理解会更透彻。 **进入源码注解地址查看**
- @TaskIrqContext 结束保存中断现场 ......
- `TaskIrqContext` 保存完现场后就真正的开始处理中断了。

```
 /*BLX 带链接和状态切换的跳转*/
    BLX     HalIrqHandler /* 调用硬中断处理程序，无参 ，说明HalIrqHandler在svc栈中执行 */
#ifdef LOSCFG_IRQ_USE_STANDALONE_STACK @是否使用了独立的IRQ栈
    MOV     SP， R4 @恢复现场，sp = R4
    POP     {R4} @弹出R4
#endif
    /* process pending signals */  @处理挂起信号
    BL      OsTaskProcSignal   @跳转至C代码
    /* check if needs to schedule */@检查是否需要调度
    CMP     R0， #0 @是否需要调度，R0为参数保存值
    BLNE    OsSchedPreempt @不相等，即R0非0，一般是 1
    MOV     R0，SP @参数
    MOV     R1，R7 @参数
    BL      OsSaveSignalContextIrq @跳转至C代码
    /* restore fpu regs */
    POP_FPU_REGS    R0 @恢复fpu寄存器值
    ADD     SP， SP， #4 @sp = sp + 4
```

- 这段代码都是跳转到C语言去执行，分别是 `HalIrqHandler` `OsTaskProcSignal` `OsSchedPreempt` `OsSaveSignalContextIrq` C语言部分内容很多，将在中断管理篇中说明。
- @TaskIrqContext 开始恢复中断现场 ......
- 同样的中间代码需配合 `TaskIrqContext` 来看，不然100%懵逼。结合看就秒懂，代码都已经注释，不再做解释，注解中提到的 翻看287行 是指源码的第287行，请对照注解源码看理解会更透彻。**进入源码注解地址查看**
- @TaskIrqContext 结束恢复中断现场 ......

百文说内核 | 抓住主脉络

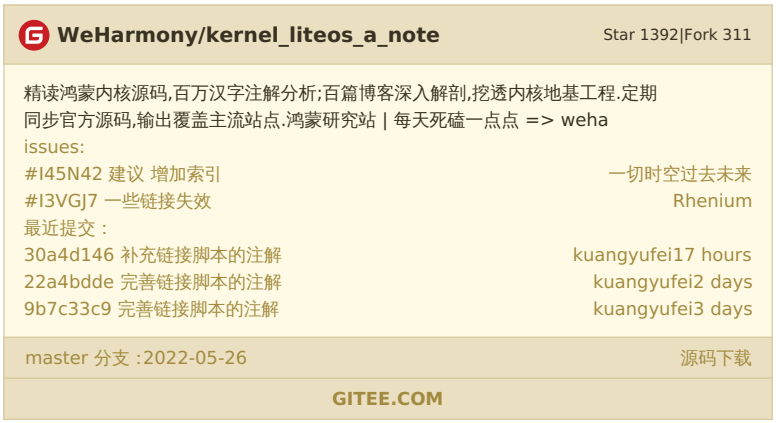
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

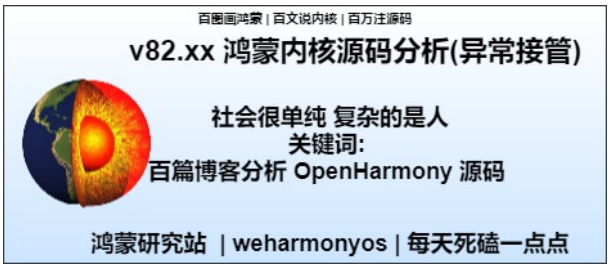


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

82_异常接管篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

本篇需结合 << ARM体系架构参考手册(ARMv7-A/R).pdf >> 阅读。

为何要有异常接管？

拿小孩成长打比方，大人总希望孩子能健康成长，但在成长过程中总会遇到各种各样的问题，树欲静而风不止，成长路上有危险，有时是自己的问题有时是外在环境问题。就像抖音最近的流行口水歌一样，社会很单纯，复杂的是人啊，每次听到都想站起来扭几下。哎! 老衲到底做错什么了？

比如:老被其他小朋友欺负怎么弄？ 发现乱花钱怎么搞？ 青春期发育怎么应对？ 失恋要跳楼又怎么办？ 意思是超过他的认知范围，靠它自己解决不了了，就需要有更高权限，更高智慧的人介入进来，帮着解决，干擦屁股的事。

那么应用程序就是那个小孩，内核就是监护人，有更高的权限，更高的智慧。而且监护人还不止一个，而是六个，每个监护人对应解决一种情况，情况发生了就由它来接管这件事的处理，小朋友你就别管了哈，先把你关家里，处理好了外面安全了再把应用程序放出来玩去。

这六个人处理问题都自带工具，有标准的解决方案，有自己独立的办公场所，办公场所就是栈空间(独立的)，标准解决方案就是私有代码段，放在固定的位置。而自带的工具就是 `SPSR_***` ， `SP_***` ， `LR_***` 寄存器组。详见 **系列篇之工作模式篇** ，这里再简单回顾下有哪些工作模式，包括小孩自己(用户模式)一共是七种模式。

七种工作模式

本篇需结合 << ARM体系架构参考手册(ARMv7-A/R).pdf >> 阅读。

The ARM720T supports seven modes of operation as listed in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution state
FIQ	fiq	Designed to support a data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

- **用户模式（usr）**：该模式是用户程序的工作模式，它运行在操作系统的用户态，它没有权限去操作其它硬件资源，只能执行处理自己的数据，也不能切换到其它模式下，要想访问硬件资源或切换到其它模式只能通过软中断或产生异常。
- **快速中断模式（fiq）**：快速中断模式是相对一般中断模式而言的，用来处理高优先级中断的模式，处理对时间要求比较紧急的中断请求，主要用于高速数据传输及通道处理中。
- **普通中断模式（irq）**：一般中断模式也叫普通中断模式，用于处理一般的中断请求，通常在硬件产生中断信号之后自动进入该模式，该模式可以自由访问系统硬件资源。
- **管理模式（svc）**：操作系统保护模式，CPU上电复位和当应用程序执行 SVC 指令调用系统服务时也会进入此模式，操作系统内核的普通代码通常工作在这个模式下。
- **终止模式（abt）**：当数据或指令预取终止时进入该模式，中止模式用于支持虚拟内存或存储器保护，当用户程序访问非法地址，没有权限读取的内存地址时，会进入该模式，
- **系统模式（sys）**：供操作系统使用的高特权用户模式，与用户模式类似，但具有可以直接切换到其他模式等特权，用户模式与系统模式两者使用相同的寄存器，都没有SPSR（Saved Program Statement Register，已保存程序状态寄存器），但系统模式比用户模式有更高的权限，可以访问所有系统资源。
- **未定义模式（und）**：未定义模式用于支持硬件协处理器的软件仿真，CPU在指令的译码阶段不能识别该指令操作时，会进入未定义模式。

除用户模式外，其余6种工作模式都属于特权模式

- 特权模式中除了系统模式以外的其余5种模式称为异常模式
- 大多数程序运行于用户模式
- 进入特权模式是为了处理中断、异常、或者访问被保护的系统资源
- 硬件权限级别：系统模式 > 异常模式 > 用户模式
- 快中断(fiq)与慢中断(irq)区别：快中断处理时禁止中断

每种模式都有自己独立的入口和独立的运行栈空间。 **系列篇之CPU篇** 已介绍过只要提供了入口函数和运行空间，CPU就可以干活了。入口函数解决了指令来源问题，运行空间解决了指令的运行场地问题。 而且在多核情况下，每个CPU核的每种特权模式都有自己独立的栈空间。注意是特权模式下的栈空间，用户模式的栈空间是由用户(应用)程序提供的。

官方概念

异常接管是操作系统对运行期间发生的异常情况（芯片硬件异常）进行处理的一系列动作，例如打印异常发生时当前函数的调用栈信息、CPU现场信息、任务的堆栈情况等。 异常接管作为一种调测手段，可以在系统发生异常时给用户提供有用的异常信息，譬如异常类型、发生异常时的系统状态等，方便用户定位分析问题。

鸿蒙的异常接管，在系统发生异常时的处理动作为：显示异常发生时正在运行的任务信息（包括任务名、任务号、堆栈大小等），以及CPU现场等信息。

进入和退出异常方式

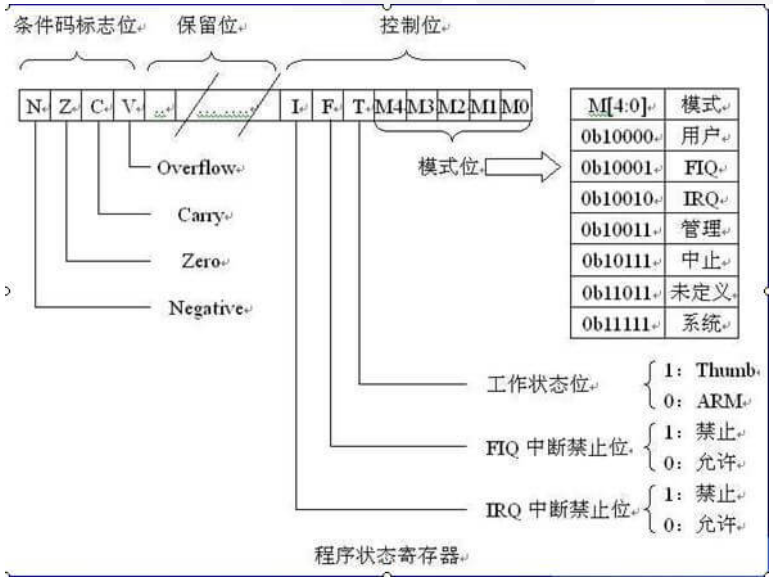
异常接管切换需要处理好两件事:

- 一个是代码要切到哪个位置，也就是要重置PC寄存器，每种异常模式下的切换方式如图:

Table 2-3 Exception entry and exit

Exception	Return Instruction	Previous State	
进入和退出异常模式指令		ARM R14_x	Thumb R14_x
BL[a]	MOV PC, R14	PC + 4	PC + 2
SWI[a]	MOVS PC, R14_svc	PC + 4	PC + 2
UDEF[a]	MOVS PC, R14_und	PC + 4	PC + 2
FIQ[b]	SUBS PC, R14_fiq, #4	PC + 4	PC + 4
IRQ [b]	SUBS PC, R14_irq, #4	PC + 4	PC + 4
PABT[a]	SUBS PC, R14_abt, #4	PC + 4	PC + 4
DABT[c]	SUBS PC, R14_abt, #8	PC + 8	PC + 8
RESET[d]	NA	-	-

- 另一个是要恢复每种模式的状态，即 CPSR(1个) 和 SPSR(共5个) 的关系，对 M[4:0] 的修改，如图:



以下是 M[4:0] 在每种模式下具体操作方式:

weharmony.gitee.io 异常模式切换CPSR寄存器值管理

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10000	User	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR
10001	FIQ	R7 to R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7 to R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7 to R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12 to R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7 to R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12 to R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc

weharmony.gitee.io 来源:鸿蒙内核源码分析

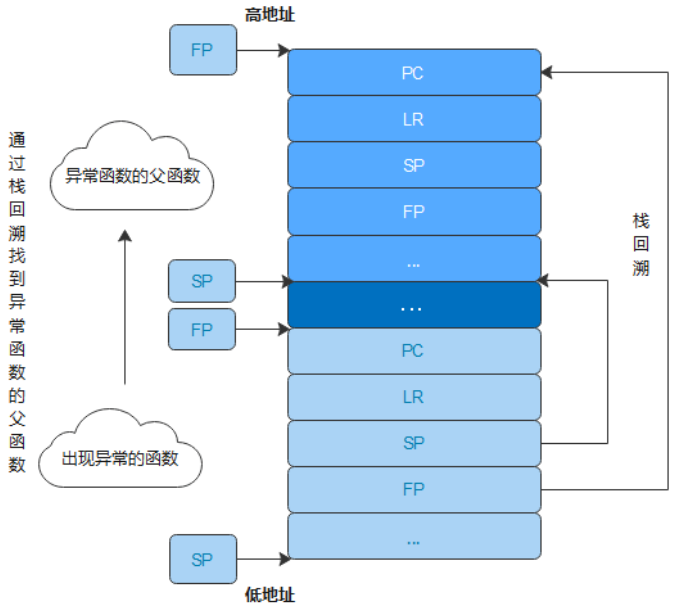
Table 2-2 PSR mode bit values (continued)

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10111	Abort	R7 to R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12 to R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7 to R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12 to R0, R14_und, R13_und, PC, CPSR, SPSR_und
11111	System	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR

栈帧

每个函数都有自己的栈空间，称为栈帧。调用函数时，会创建子函数的栈帧，同时将函数入参、局部变量、寄存器入栈。栈帧从高地址向低地址生长，也就是说栈底是高地址，栈顶是底地址。 详见 **系列篇之用栈方式篇**

以 ARM32 CPU 架构为例，每个栈帧中都会保存 PC 、 LR 、 SP 和 FP 寄存器的历史值。 堆栈分析原理如下图所示，实际堆栈信息根据不同CPU架构有所差异，此处仅做示意。 图中不同颜色的寄存器表示不同的函数。可以看到函数调用过程中，寄存器的保存。通过FP寄存器，栈回溯到异常函数的父函数，继续按照规律对栈进行解析，推出函数调用关系，方便用户定位问题。



解读

- LR寄存器（Link Register），链接寄存器，指向函数的返回地址。
- R11：可以用作通用寄存器，在开启特定编译选项时可以用作帧指针寄存器FP，用来实现栈回溯功能。 GNU编译器（gcc）默认将R11作为存储变量的通用寄存器，因而默认情况下无法使用FP的栈回溯功能。为支持调用栈解析功能，需要在编译参数中添加-fno-omit-frame-pointer选项，提示编译器将R11作为FP使用。
- FP寄存器（Frame Point），帧指针寄存器，指向当前函数的父函数的栈帧起始地址。利用该寄存器可以得到父函数的栈帧，从栈帧中获取父函数的FP，就可以得到祖父函数的栈帧，以此类推，可以追溯程序调用栈，得到函数间的调用关系。 当系统发生异常时，系统打印异常函数的栈帧中保存的寄存器内容，以及父函数、祖父函数的栈帧中的LR、FP寄存器内容，用户就可以据此追溯函数间的调用关系，定位异常原因。

六种异常模式实现代码

```
/* Define exception type ID */  //ARM处理器一共有7种工作模式，除了用户和系统模式其余都叫异常工作模式
#define OS_EXCEPT_RESET         0x00 //重置功能，例如：开机就进入CPSR_SVC_MODE模式
#define OS_EXCEPT_UNDEF_INSTR   0x01 //未定义的异常，就是others
#define OS_EXCEPT_SWI           0x02 //软中断
#define OS_EXCEPT_PREFETCH_ABORT 0x03 //预取异常(取指异常)， 指令三步骤: 取指，译码，执行，
#define OS_EXCEPT_DATA_ABORT    0x04 //数据异常
#define OS_EXCEPT_FIQ           0x05 //快中断异常
#define OS_EXCEPT_ADDR_ABORT    0x06 //地址异常
#define OS_EXCEPT_IRQ           0x07 //普通中断异常
```

地址异常处理(Address abort)

```
@ Description: Address abort exception handler
_osExceptAddrAbortHdl: @地址异常处理
    SUB     LR， LR， #8                            @ LR offset to return from this exception: -8.
    STMFD   SP， {R0-R7}                             @ Push working registers， but don`t change SP.

    MOV     R0， #OS_EXCEPT_ADDR_ABORT                  @ Set exception ID to OS_EXCEPT_ADDR_ABORT.

    B       _osExceptDispatch                       @跳到异常分发统一处理
```

快中断处理(fiq)

```
@ Description: Fast interrupt request exception handler
_osExceptFiqHdl: @快中断异常处理
    SUB     LR， LR， #4                            @ LR offset to return from this exception: -4.
    STMFD   SP， {R0-R7}                             @ Push working registers.
```

```
    MOV     R0 ， #OS_EXCEPT_FIQ                        @ Set exception ID to OS_EXCEPT_FIQ.

    B       _osExceptDispatch                       @ Branch to global exception handler.
```

解读

- 快中断处理时需禁用普通中断

取指异常(Prefectch abort)

```
@ Description: Prefectch abort exception handler
_osExceptPrefetchAbortHdl:
#ifdef LOSCFG_GDB
#if __LINUX_ARM_ARCH__ >= 7
    GDB_HANDLE OsPrefetchAbortExcHandleEntry
#endif
#else
    SUB     LR ， LR ， #4                          @ LR offset to return from this exception: -4.
    STMFD   SP ， {R0-R7}                           @ Push working registers ， but don`t change SP.
    MOV     R5 ， LR
    MRS     R1 ， SPSR

    MOV     R0 ， #OS_EXCEPT_PREFETCH_ABORT             @ Set exception ID to OS_EXCEPT_PREFETCH_ABORT.

    AND     R4 ， R1 ， #CPSR_MASK_MODE                 @ Interrupted mode
    CMP     R4 ， #CPSR_USER_MODE                   @ User mode
    BEQ     _osExcPageFault                         @ Branch if user mode

_osKernelExceptPrefetchAbortHdl:
    MOV     LR ， R5
    B       _osExceptDispatch                       @ Branch to global exception handler.
#endif
```

数据访问异常(Data abort)

```
@ Description: Data abort exception handler
_osExceptDataAbortHdl: @数据异常处理，缺页就属于数据异常
#ifdef LOSCFG_GDB
#if __LINUX_ARM_ARCH__ >= 7
    GDB_HANDLE OsDataAbortExcHandleEntry
#endif
#else
    SUB     LR ， LR ， #8                          @ LR offset to return from this exception: -8.
    STMFD   SP ， {R0-R7}                           @ Push working registers ， but don`t change SP.
    MOV     R5 ， LR
    MRS     R1 ， SPSR

    MOV     R0 ， #OS_EXCEPT_DATA_ABORT                 @ Set exception ID to OS_EXCEPT_DATA_ABORT.

    B     _osExcPageFault   @跳到缺页异常处理
#endif
```

软中断处理(swi)

```
@ Description: Software interrupt exception handler
_osExceptSwiHdl: @软中断异常处理
    SUB     SP ， SP ， #(4 * 16) @先申请16个栈空间用于处理本次软中断
    STMIA   SP ， {R0-R12} @保存R0-R12寄存器值
    MRS     R3 ， SPSR   @读取本模式下的SPSR值
    MOV     R4 ， LR    @保存回跳寄存器LR

    AND     R1 ， R3 ， #CPSR_MASK_MODE                 @ Interrupted mode 获取中断模式
    CMP     R1 ， #CPSR_USER_MODE                   @ User mode 是否为用户模式
    BNE     OsKernelSVCHandler                      @ Branch if not user mode 非用户模式下跳转
```

```
@ 当为用户模式时，获取SP和LR寄出去值
  @ we enter from user mode， we need get the values of  USER mode r13(sp) and r14(lr).
  @ stmia with ^ will return the user mode registers (provided that r15 is not in the register list).
  MOV    R0， SP          @获取SP值，R0将作为OsArmA32SyscallHandle的参数
  STMFD  SP!， {R3}                              @ Save the CPSR 入栈保存CPSR值
  ADD    R3， SP， #(4 * 17)                       @ Offset to pc/cpsr storage 跳到PC/CPSR存储位置
  STMFD  R3!， {R4}                              @ Save the CPSR and r15(pc) 保存LR寄存器
  STMFD  R3， {R13， R14}^                           @ Save user mode r13(sp) and r14(lr) 保存用户模式下的SP和LR寄存器
  SUB    SP， SP， #4
  PUSH_FPU_REGS R1 @保存中断模式(用户模式模式)

  MOV    FP， #0                                 @ Init frame pointer
  CPSIE  I @开中断，表明在系统调用期间可响应中断
  BLX    OsArmA32SyscallHandle /*交给C语言处理系统调用*/
  CPSID  I @执行后续指令前必须先关中断

  POP_FPU_REGS R1          @弹出FP值给R1
  ADD    SP， SP，#4         @ 定位到保存旧SPSR值的位置
  LDMFD  SP!， {R3}                              @ Fetch the return SPSR 弹出旧SPSR值
  MSR    SPSR_cxsf， R3                          @ Set the return mode SPSR 恢复该模式下的SPSR值

  @ we are leaving to user mode， we need to restore the values of USER mode r13(sp) and r14(lr).
  @ ldmia with ^ will return the user mode registers (provided that r15 is not in the register list)

  LDMFD  SP!， {R0-R12}        @恢复R0-R12寄存器
  LDMFD  SP， {R13， R14}^                           @ Restore user mode R13/R14 恢复用户模式的R13/R14寄存器
  ADD    SP， SP， #(2 * 4)       @定位到保存旧PC值的位置
  LDMFD  SP!， {PC}^                             @ Return to user 切回用户模式运行
```

普通中断处理(irq)

```
OsIrqHandler: @硬中断处理，此时已切换到硬中断栈
  SUB    LR， LR， #4
  /* push r0-r3 to irq stack */
  STMFD  SP， {R0-R3}  @r0-r3寄存器入 irq 栈
  SUB    R0， SP， #(4 * 4)@r0 = sp - 16
  MRS    R1， SPSR  @获取程序状态控制寄存器
  MOV    R2， LR   @r2=lr

  /* disable irq， switch to svc mode */@超级用户模式(SVC 模式)，主要用于 SWI(软件中断)和 OS(操作系统)。
  CPSID  i， #0x13    @切换到SVC模式，此处一切换，后续指令将入SVC的栈
     @CPSID i为关中断指令，对应的是CPSIE
  /* push spsr and pc in svc stack */
  STMFD  SP!， {R1， R2} @实际是将 SPSR，和LR入栈，入栈顺序为 R1，R2，SP自增
  STMFD  SP， {LR}   @LR再入栈，SP不自增

  AND    R3， R1， #CPSR_MASK_MODE @获取CPU的运行模式
  CMP    R3， #CPSR_USER_MODE  @中断是否发生在用户模式
  BNE    OsIrqFromKernel   @中断不发生在用户模式下则跳转到OsIrqFromKernel

  /* push user sp， lr in svc stack */
  STMFD  SP， {R13， R14}^   @sp和LR入svc栈
```

解读

- 普通中断处理时可以响应快中断

未定义异常处理(undef)

```
@ Description: Undefined instruction exception handler
_osExceptUndefInstrHdl:@出现未定义的指令处理
#ifdef LOSCFG_GDB
  GDB_HANDLE OsUndefIncExcHandleEntry
#else
                                       @ LR offset to return from this exception:  0.
  STMFD  SP， {R0-R7}                            @ Push working registers， but don`t change SP.
```

```
    MOV     R0 ， #OS_EXCEPT_UNDEF_INSTR                    @ Set exception ID to OS_EXCEPT_UNDEF_INSTR.

    B       _osExceptDispatch                           @ Branch to global exception handler.

#endif
```

异常分发统一处理

```
_osExceptDispatch: @异常模式统一分发处理
    MRS     R2 ， SPSR                                  @ Save CPSR before exception.
    MOV     R1 ， LR                                    @ Save PC before exception.
    SUB     R3 ， SP ， #(8 * 4)                            @ Save the start address of working registers.

    MSR     CPSR_c ， #(CPSR_INT_DISABLE | CPSR_SVC_MODE)     @ Switch to SVC mode ， and disable all interrupts
    MOV     R5 ， SP
    EXC_SP_SET __exc_stack_top ， OS_EXC_STACK_SIZE ， R6 ， R7

    STMFD   SP! ， {R1}                                 @ Push Exception PC
    STMFD   SP! ， {LR}                                 @ Push SVC LR
    STMFD   SP! ， {R5}                                 @ Push SVC SP
    STMFD   SP! ， {R8-R12}                               @ Push original R12-R8 ，
    LDMFD   R3! ， {R4-R11}                               @ Move original R7-R0 from exception stack to original stack.
    STMFD   SP! ， {R4-R11}
    STMFD   SP! ， {R2}                                 @ Push task`s CPSR (i.e. exception SPSR).

    CMP     R0 ， #OS_EXCEPT_DATA_ABORT   @是数据异常吗？
    BNE     1f        @不是跳到 锚点1处
    MRC     P15 ， 0 ， R8 ， C6 ， C0 ， 0    @R8=C6(内存失效的地址) 0(访问数据失效)
    MRC     P15 ， 0 ， R9 ， C5 ， C0 ， 0    @R9=C5(内存失效的状态) 0(无效整个指令cache)
    B       3f        @跳到锚点3处执行
1:  CMP     R0 ， #OS_EXCEPT_PREFETCH_ABORT  @是预取异常吗？
    BNE     2f        @不是跳到 锚点2处
    MRC     P15 ， 0 ， R8 ， C6 ， C0 ， 2    @R8=C6(内存失效的地址) 2(访问指令失效)
    MRC     P15 ， 0 ， R9 ， C5 ， C0 ， 1    @R9=C5(内存失效的状态) 1(虚拟地址)
    B       3f        @跳到锚点3处执行
2:  MOV     R8 ， #0
    MOV     R9 ， #0

3:  AND     R2 ， R2 ， #CPSR_MASK_MODE
    CMP     R2 ， #CPSR_USER_MODE                         @ User mode
    BNE     4f @不是用户模式
    STMFD   SP ， {R13 ， R14}^                           @ save user mode sp and lr
4:
    SUB     SP ， SP ， #(4 * 2) @sp=sp-(4*2)
```

非常重要的ARM37个寄存器

ARM state general registers and program counter

31个通用寄存器,r13_*这些是单独算的,r13,r14各六个　来源: 鸿蒙内核源码分析

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

注意: r15(pc寄存器)七种工作模式通用, 因为代码是共用的, 所以可以通用.

6个状态寄存器,系统和用户模式寄存器共用　详见: weharmony.gitee.io

ARM state program status registers weharmony.github.io

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

详见 v08.xx 鸿蒙内核源码分析(总目录)之寄存器篇

结尾

以上为异常接管对应的代码处理，具体每种异常发生的场景和代码细节处理，因内容太多，太复杂，系列篇后续将分篇一一分析。敬请关注!

百文说内核 | 抓住主脉络

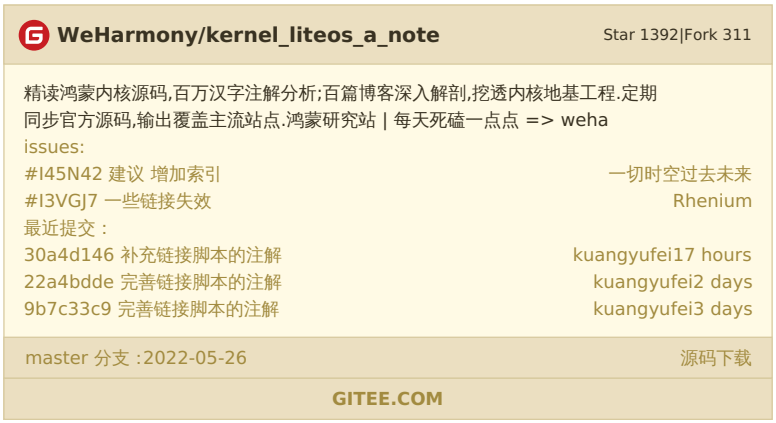
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

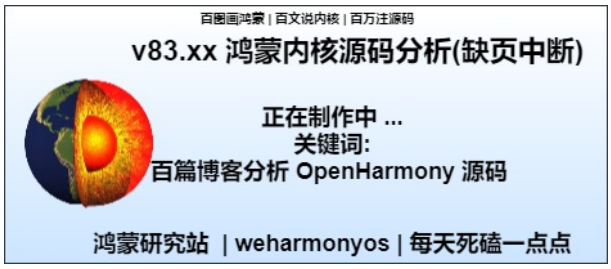


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

83_缺页中断篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

内核汇编相关篇为:

- v74.01 鸿蒙内核源码分析(编码方式) | 机器指令是如何编码的
- v75.03 鸿蒙内核源码分析(汇编基础) | CPU上班也要打卡
- v76.04 鸿蒙内核源码分析(汇编传参) | 如何传递复杂的参数
- v77.01 鸿蒙内核源码分析(链接脚本) | 正在制作中 ...
- v78.01 鸿蒙内核源码分析(内核启动) | 从汇编到main()
- v79.01 鸿蒙内核源码分析(进程切换) | 正在制作中 ...
- v80.03 鸿蒙内核源码分析(任务切换) | 看汇编如何切换任务
- v81.05 鸿蒙内核源码分析(中断切换) | 系统因中断活力四射
- v82.06 鸿蒙内核源码分析(异常接管) | 社会很单纯 复杂的是人
- v83.01 鸿蒙内核源码分析(缺页中断) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

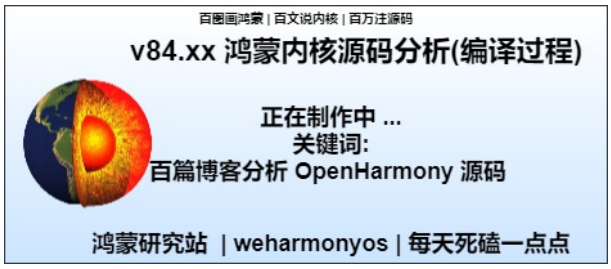
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

84_编译过程篇

本篇关键词：、、、

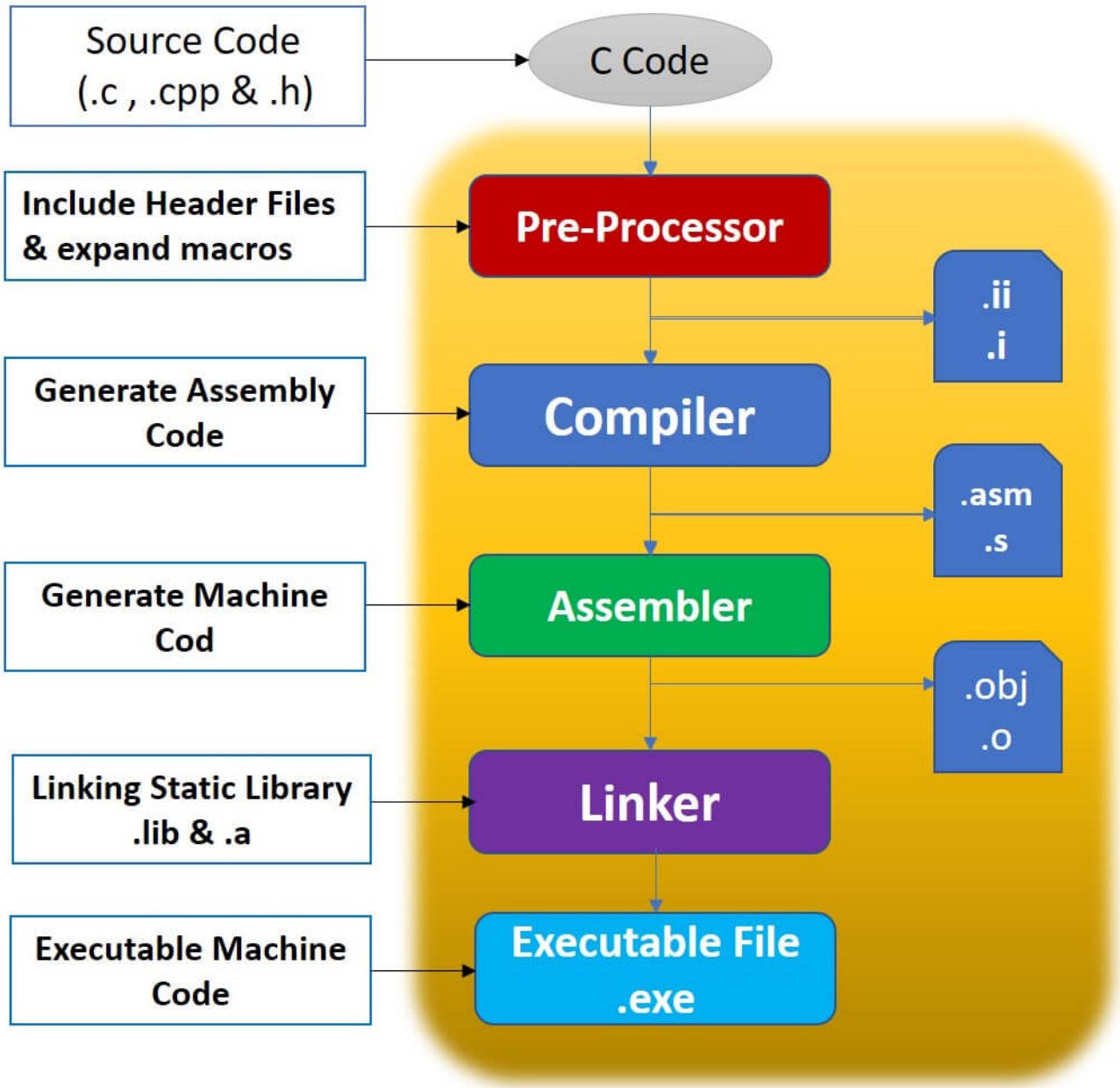


下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

一个.c源文件编译的整个过程如图。



编译过程要经过：源文件 --> 预处理 --> 编译(cc1) --> 汇编器(as) --> 链接器(ld) --> 可执行文件(PE/ELF)

GCC

GCC（GNU Compiler Collection，GNU编译器套件），官网:https://gcc.gnu.org/ ，是由 GNU 开发的编程语言编译器

- GCC源码仓库:https://github.com/gcc-mirror/gcc 有兴趣的可以去阅读源码。
- GCC用法

```
gcc [options] infile
-o 重定位输入文件位置；
-E 只对源文件进行预处理，输出.i文件；
-S 对源文件进行预处理，编译，输出.s文件；
-c 对源文件进行预处理，编译，汇编，输出.o文件；
-I 包含头文件路径，如 g++ -Iopencv/include/；
-L 包含库文件路径，如 g++ -Lopencv/lib/ ；
-l 链接库文件，如链接lib.so：g++ -llib；
-shared 编译.so库；
-fPIC 生成位置无法码；
```

```
-Wall 对代码有问题的地方发出警告；
-g 在目标文件中嵌入调试信息，便于gdb调试；
```

本篇以 main.c 文件举例，说明白两个问题

- main.c是怎么编译的？ 详细的整个过程是怎样的，是如何一步步走到了可执行文件的。
- main.c中的代码，数据，栈，堆是如何形成和构建的？ 通过变量地址窥视其内存布局。
- 这两部分内容将掺杂在一起尽量同时说明白，main.c文件它将经历以下变化过程 main.c --> main.i --> main.S --> main.o --> main

插播 case_code_100

case_code_100 是百篇博客分析过程中用到的案例汇总，其中可能是代码，也可能是一些文章，序号与博客序号一一对应。仓库地址: https://gitee.com/weharmony/case_code_100，本篇为 `57`

源文件 | main.c

```
#include <stdio.h>
#include <stdlib.h>
#define HARMONY_OS "hello harmonyos \n"
const int g_const = 10; //全局常量区
int g_init = 57; //全局变量
int g_no_init;  //全局变量无初值
static int s_exter_init = 58;//静态外部变量有初值
static int s_exter_no_init;  //静态外部变量无初值
/*****************************************************
* main:通过简单案例窥视编译过程和内存布局
*****************************************************/
int main()
{
  static int s_inter_init = 59;//静态内部变量有初值
  static int s_inter_no_init;  //静态内部变量无初值
  int l_init = 60;    //局部变量有初值
  int l_no_init;     //局部变量无初值
  const int l_const = 11;   //局部常量
  char *heap = (char *)malloc(100);//堆区
  printf(HARMONY_OS);
  //----------------------
  printf("全局常量 g_const : %p\n"， &g_const);
  printf("全局外部有初值 g_init : %p\n"， &g_init);
  printf("全局外部无初值 g_no_init : %p\n"， &g_no_init);
  printf("静态外部有初值 s_exter_init : %p\n"， &s_exter_init);
  printf("静态外静无初值 s_exter_no_init : %p\n"， &s_exter_no_init);
  printf("静态内部有初值 s_inter_init : %p\n"， &s_inter_init);
  printf("静态内部无初值 s_inter_no_init : %p\n"， &s_inter_no_init);
  //----------------------
  printf("局部栈区有初值 l_init : %p\n"， &l_init);
  printf("局部栈区无初值  l_no_init : %p\n"， &l_no_init);
  printf("局部栈区常量  l_const : %p\n"， &l_const);
  //----------------------
  printf("堆区地址 heap : %p\n"， heap);
  //----------------------
  printf("代码区地址 : %p\n"， &main);
  return 0;
}
```

解读

- 函数具有代表性，有宏，注释，有全局，局部，静态外部，静态内部变量，堆申请。
- 通过这些值的变化看其中间过程和最后内存布局。

预处理 | main.i

```
root@5e3abe332c5a:/home/docker/case_code_100/57# gcc -E main.c -o main.i
root@5e3abe332c5a:/home/docker/case_code_100/57# cat main.i
# 1 "main.c"
# 1 "<built-in>"
```

```
# 1 "<command-line>"
......
typedef __u_char u_char;
typedef __u_short u_short;
extern int printf (const char *__restrict __format , ...);
......
const int g_const = 10;
int g_init = 57;
int g_no_init;
static int s_exter_init = 58;
static int s_exter_no_init;
int main()
{
  static int s_inter_init = 59;
  static int s_inter_no_init;

  int l_init = 60;
  int l_no_init;
  const int l_const = 11;

  char *heap = (char *)malloc(100);

  printf("hello harmonyos \n");

  printf("全局常量 g_const : %p\n" , &g_const);
  printf("全局外部有初值 g_init : %p\n" , &g_init);
  printf("全局外部无初值 g_no_init : %p\n" , &g_no_init);
  printf("静态外部有初值 s_exter_init : %p\n" , &s_exter_init);
  printf("静态外静无初值 s_exter_no_init : %p\n" , &s_exter_no_init);
  printf("静态内部有初值 s_inter_init : %p\n" , &s_inter_init);
  printf("静态内部无初值 s_inter_no_init : %p\n" , &s_inter_no_init);

  printf("局部栈区有初值 l_init : %p\n" , &l_init);
  printf("局部栈区无初值  l_no_init : %p\n" , &l_no_init);
  printf("局部栈区常量  l_const : %p\n" , &l_const);

  printf("堆区地址 heap : %p\n" , heap);

  printf("代码区地址 : %p\n" , &main);
  return 0;
}
```

解读

main.i文件很大，1000多行，此处只列出重要部分，全部代码前往case_code_100查看 预处理过程主要处理那些源代码中以#开始的预处理指令，主要处理规则如下：

- 将所有的#define删除，并且展开所有的宏定义；
- 处理所有条件编译指令，如#if，#ifdef等；
- 处理#include预处理指令，将被包含的文件插入到该预处理指令的位置。该过程递归进行，及被包含的文件可能还包含其他文件。
- 删除所有的注释//和 /**/；
- 添加行号和文件标识，如# 1 "main.c"，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号信息；
- 保留所有的#pragma编译器指令，因为编译器须要使用它们；
- 经过预编译后的.i文件不包含任何宏定义，因为所有的宏都已经被展开，并且包含的文件也已经被插入到.i文件中。所以当无法判断宏定义是否正确或头文件包含是否正确使，可以查看预编译后的文件来确定问题。

编译 | main.S

编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。这个过程是整个程序构建的核心部分，也是最复杂的部分之一。

```
//编译器: armv7-a clang (trunk)
root@5e3abe332c5a:/home/docker/case_code_100/57# gcc -S main.i -o main.S
root@5e3abe332c5a:/home/docker/case_code_100/57# cat main.S
main:
        push    {r11 , lr}     @保存r11和lr寄存器，因为内部有函数调用，lr表示函数的返回地址，所以需要保存
        mov     r11 , sp       @保存sp
```

```
    sub    sp， sp， #24   @申请栈空间
    mov    r0， #0       @r0 = 0，这个代表 return 0;
    str    r0， [sp]     @栈顶保存 main函数返回值
    str    r0， [r11， #-4]@r0 = 0，即变量l_no_init入栈，优化了指令的顺序
    mov    r0， #60      @r0 = 60，即变量l_init
    str    r0， [r11， #-8]@l_init入栈
    mov    r0， #11      @r0 = 11，即变量l_const
    str    r0， [sp， #8]  @l_const入栈
    mov    r0， #100     @r0=100，为malloc参数
    bl     malloc       @调用malloc
    str    r0， [sp， #4]  @malloc函数返回值入栈
    ldr    r0， .LCPI0_0  @为printf准备参数 "hello harmonyos \n"
    bl     printf       @调用printf("hello harmonyos \n");
    ldr    r0， .LCPI0_1  @准备参数
    ldr    r1， .LCPI0_2  @准备参数
    bl     printf       @调用printf("全局常量 g_const : %p\n"， &g_const);
    ldr    r0， .LCPI0_3  @准备参数
    ldr    r1， .LCPI0_4  @准备参数
    bl     printf       @调用printf("全局外部有初值 g_init : %p\n"， &g_init);
    ldr    r0， .LCPI0_5  @准备参数
    ldr    r1， .LCPI0_6  @准备参数
    bl     printf       @调用printf("全局外部无初值 g_no_init : %p\n"， &g_no_init);
    ldr    r0， .LCPI0_7  @准备参数
    ldr    r1， .LCPI0_8  @准备参数
    bl     printf       @调用printf("静态外部有初值 s_exter_init : %p\n"， &s_exter_init);
    ldr    r0， .LCPI0_9  @准备参数
    ldr    r1， .LCPI0_10 @准备参数
    bl     printf       @调用printf("静态外静无初值 s_exter_no_init : %p\n"， &s_exter_no_init);
    ldr    r0， .LCPI0_11 @准备参数
    ldr    r1， .LCPI0_12 @准备参数
    bl     printf       @调用printf("静态内部有初值 s_inter_init : %p\n"， &s_inter_init);
    ldr    r0， .LCPI0_13 @准备参数
    ldr    r1， .LCPI0_14 @准备参数
    bl     printf       @调用printf("静态内部无初值 s_inter_no_init : %p\n"， &s_inter_no_init);
    ldr    r0， .LCPI0_15 @准备参数
    sub    r1， r11， #8   @r1=&l_init
    bl     printf       @调用printf("局部栈区有初值 l_init : %p\n"， &l_init);
    ldr    r0， .LCPI0_16 @准备参数
    add    r1， sp， #12   @r1=&l_no_init
    bl     printf       @调用printf("局部栈区无初值  l_no_init : %p\n"， &l_no_init);
    ldr    r0， .LCPI0_17 @准备参数
    add    r1， sp， #8    @r1=&l_const
    bl     printf       @调用printf("局部栈区常量  l_const : %p\n"， &l_const);
    ldr    r1， [sp， #4]  @r1=heap，即malloc返回值出栈
    ldr    r0， .LCPI0_18 @准备参数
    bl     printf       @调用printf("堆区地址 heap : %p\n"， heap);
    ldr    r0， .LCPI0_19 @准备参数
    ldr    r1， .LCPI0_20 @准备参数
    bl     printf       @调用printf("代码区地址 : %p\n"， &main);
    ldr    r0， [sp]     @即 r0=0，代表main函数的返回值 对应开头的 str    r0， [sp]
    mov    sp， r11      @恢复值，对应开头的  mov    r11， sp
    pop    {r11， lr}    @出栈，对应开头的 push   {r11， lr}
    bx     lr           @退出main，跳到调用main()函数处，返回值保存在r0 | =0
.LCPI0_0:   @以下全部是申请和定义代码中的一个个变量
    .long   .L.str
.LCPI0_1:
    .long   .L.str.1
.LCPI0_2:
    .long   g_const
.LCPI0_3:
    .long   .L.str.2
.LCPI0_4:
    .long   g_init
.LCPI0_5:
    .long   .L.str.3
.LCPI0_6:
    .long   g_no_init
.LCPI0_7:
    .long   .L.str.4
.LCPI0_8:
    .long   s_exter_init
```

```
.LCPI0_9:
        .long   .L.str.5
.LCPI0_10:
        .long   _ZL15s_exter_no_init
.LCPI0_11:
        .long   .L.str.6
.LCPI0_12:
        .long   main::s_inter_init
.LCPI0_13:
        .long   .L.str.7
.LCPI0_14:
        .long   _ZZ4mainE15s_inter_no_init
.LCPI0_15:
        .long   .L.str.8
.LCPI0_16:
        .long   .L.str.9
.LCPI0_17:
        .long   .L.str.10
.LCPI0_18:
        .long   .L.str.11
.LCPI0_19:
        .long   .L.str.12
.LCPI0_20:
        .long   main
g_init:
        .long   57                      @ 0x39

g_no_init:
        .long   0                       @ 0x0

main::s_inter_init:
        .long   59                      @ 0x3b

.L.str:
        .asciz  "hello harmonyos \n"

.L.str.1:
        .asciz  "\345\205\250\345\261\200\345\270\270\351\207\217 g_const\357\274\232%p\n"

g_const:
        .long   10                      @ 0xa

.L.str.2:
        .asciz  "\345\205\250\345\261\200\345\244\226\351\203\250\346\234\211\345\210\235\345\200\274 g_init\357\274\232%p\n"

.L.str.3:
        .asciz  "\345\205\250\345\261\200\345\244\226\351\203\250\346\227\240\345\210\235\345\200\274 g_no_init\357\274\232%p\n"

.L.str.4:
        .asciz  "\351\235\231\346\200\201\345\244\226\351\203\250\346\234\211\345\210\235\345\200\274 s_exter_init\357\274\232%p\n"

s_exter_init:
        .long   58                      @ 0x3a

.L.str.5:
        .asciz  "\351\235\231\346\200\201\345\244\226\351\235\231\346\227\240\345\210\235\345\200\274 s_exter_no_init\357\274\232%p\n"

.L.str.6:
        .asciz  "\351\235\231\346\200\201\345\206\205\351\203\250\346\234\211\345\210\235\345\200\274 s_inter_init\357\274\232%p\n"

.L.str.7:
        .asciz  "\351\235\231\346\200\201\345\206\205\351\203\250\346\227\240\345\210\235\345\200\274 s_inter_no_init\357\274\232%p\n"

.L.str.8:
        .asciz  "\345\261\200\351\203\250\346\240\210\345\214\272\346\234\211\345\210\235\345\200\274 l_init\357\274\232%p\n"

.L.str.9:
        .asciz  "\345\261\200\351\203\250\346\240\210\345\214\272\346\227\240\345\210\235\345\200\274  l_no_init\357\274\232%p\n"

.L.str.10:
        .asciz  "\345\261\200\351\203\250\346\240\210\345\214\272\345\270\270\351\207\217  l_const\357\274\232%p\n"
```

```
.L.str.11:
        .asciz  "\345\240\206\345\214\272\345\234\260\345\235\200 heap\357\274\232%p\n"

.L.str.12:
        .asciz  "\344\273\243\347\240\201\345\214\272\345\234\260\345\235\200\357\274\232%p\n"
```

解读

- 汇编代码全部贴出，都已经加上了注释，不要嫌多，忍忍吧。
- 系列篇到了这里，读上面的汇编应该没什么难度了，不是很清楚的读以下两篇
  - v23.xx 鸿蒙内核源码分析(汇编传参篇)
  - v22.xx 鸿蒙内核源码分析(汇编基础篇)

汇编 | main.o

汇编器是将汇编代码转变成机器可以执行的命令，每一个汇编语句几乎都对应一条机器指令。汇编相对于编译过程比较简单，根据汇编指令和机器指令的对照表一一翻译即可。 main.o的内容为机器码，不能以文本形式方便的呈现，不过可以利用 objdump -S file 查看源码反汇编

```
root@5e3abe332c5a:/home/docker/case_code_100/57# gcc -c main.S -o main.o
root@5e3abe332c5a:/home/docker/case_code_100/57#objdump -S main.o
main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  13:   00 00
  15:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  19:   31 c0                   xor    %eax,%eax
  1b:   c7 45 e4 3c 00 00 00    movl   $0x3c,-0x1c(%rbp)
  22:   c7 45 ec 0b 00 00 00    movl   $0xb,-0x14(%rbp)
  29:   bf 64 00 00 00          mov    $0x64,%edi
  2e:   e8 00 00 00 00          callq  33 <main+0x33>
  33:   48 89 45 f0             mov    %rax,-0x10(%rbp)
  37:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 3e <main+0x3e>
  3e:   e8 00 00 00 00          callq  43 <main+0x43>
  43:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 4a <main+0x4a>
  4a:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 51 <main+0x51>
  51:   b8 00 00 00 00          mov    $0x0,%eax
  56:   e8 00 00 00 00          callq  5b <main+0x5b>
  5b:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 62 <main+0x62>
  62:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 69 <main+0x69>
  69:   b8 00 00 00 00          mov    $0x0,%eax
  6e:   e8 00 00 00 00          callq  73 <main+0x73>
  73:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 7a <main+0x7a>
  7a:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 81 <main+0x81>
  81:   b8 00 00 00 00          mov    $0x0,%eax
  86:   e8 00 00 00 00          callq  8b <main+0x8b>
  8b:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 92 <main+0x92>
  92:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 99 <main+0x99>
  99:   b8 00 00 00 00          mov    $0x0,%eax
  9e:   e8 00 00 00 00          callq  a3 <main+0xa3>
  a3:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # aa <main+0xaa>
  aa:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # b1 <main+0xb1>
  b1:   b8 00 00 00 00          mov    $0x0,%eax
  b6:   e8 00 00 00 00          callq  bb <main+0xbb>
  bb:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # c2 <main+0xc2>
  c2:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # c9 <main+0xc9>
  c9:   b8 00 00 00 00          mov    $0x0,%eax
  ce:   e8 00 00 00 00          callq  d3 <main+0xd3>
  d3:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # da <main+0xda>
  da:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # e1 <main+0xe1>
  e1:   b8 00 00 00 00          mov    $0x0,%eax
  e6:   e8 00 00 00 00          callq  eb <main+0xeb>
  eb:   48 8d 45 e4             lea    -0x1c(%rbp),%rax
```

```
 ef:   48 89 c6              mov    %rax , %rsi
 f2:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi       # f9 <main+0xf9>
 f9:   b8 00 00 00 00         mov    $0x0 , %eax
 fe:   e8 00 00 00 00         callq  103 <main+0x103>
103:   48 8d 45 e8           lea    -0x18(%rbp) , %rax
107:   48 89 c6              mov    %rax , %rsi
10a:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi       # 111 <main+0x111>
111:   b8 00 00 00 00         mov    $0x0 , %eax
116:   e8 00 00 00 00         callq  11b <main+0x11b>
11b:   48 8d 45 ec           lea    -0x14(%rbp) , %rax
11f:   48 89 c6              mov    %rax , %rsi
122:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi       # 129 <main+0x129>
129:   b8 00 00 00 00         mov    $0x0 , %eax
12e:   e8 00 00 00 00         callq  133 <main+0x133>
133:   48 8b 45 f0           mov    -0x10(%rbp) , %rax
137:   48 89 c6              mov    %rax , %rsi
13a:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi       # 141 <main+0x141>
141:   b8 00 00 00 00         mov    $0x0 , %eax
146:   e8 00 00 00 00         callq  14b <main+0x14b>
14b:   48 8d 35 00 00 00 00    lea    0x0(%rip) , %rsi       # 152 <main+0x152>
152:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi       # 159 <main+0x159>
159:   b8 00 00 00 00         mov    $0x0 , %eax
15e:   e8 00 00 00 00         callq  163 <main+0x163>
163:   b8 00 00 00 00         mov    $0x0 , %eax
168:   48 8b 55 f8           mov    -0x8(%rbp) , %rdx
16c:   64 48 33 14 25 28 00    xor    %fs:0x28 , %rdx
173:   00 00
175:   74 05                 je     17c <main+0x17c>
177:   e8 00 00 00 00         callq  17c <main+0x17c>
17c:   c9                    leaveq
17d:   c3                    retq
```

解读

- 此时的main.o是个 `REL (Relocatable file)` 重定位文件，关于重定位可前往翻看 v55.xx 鸿蒙内核源码分析(重定位篇)

链接 | main

链接器ld将各个目标文件组装在一起，解决符号依赖，库依赖关系，并生成可执行文件。

```
root@5e3abe332c5a:/home/docker/case_code_100/57# gcc main.o -o main
root@5e3abe332c5a:/home/docker/case_code_100/57# readelf -lW  main
Elf file type is DYN (Shared object file)
Entry point 0x10c0
There are 13 program headers ,  starting at offset 64
Program Headers:
  Type           Offset   VirtAddr           PhysAddr           FileSiz  MemSiz   Flg Align
  PHDR           0x000040 0x0000000000000040 0x0000000000000040 0x0002d8 0x0002d8 R   0x8
  INTERP         0x000318 0x0000000000000318 0x0000000000000318 0x00001c 0x00001c R   0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000 0x0000000000000000 0x0000000000000000 0x0006c8 0x0006c8 R   0x1000
  LOAD           0x001000 0x0000000000001000 0x0000000000001000 0x0003b5 0x0003b5 R E 0x1000
  LOAD           0x002000 0x0000000000002000 0x0000000000002000 0x000338 0x000338 R   0x1000
  LOAD           0x002da0 0x0000000000003da0 0x0000000000003da0 0x00027c 0x000290 RW  0x1000
  DYNAMIC        0x002db0 0x0000000000003db0 0x0000000000003db0 0x0001f0 0x0001f0 RW  0x8
  NOTE           0x000338 0x0000000000000338 0x0000000000000338 0x000020 0x000020 R   0x8
  NOTE           0x000358 0x0000000000000358 0x0000000000000358 0x000044 0x000044 R   0x4
  GNU_PROPERTY   0x000338 0x0000000000000338 0x0000000000000338 0x000020 0x000020 R   0x8
  GNU_EH_FRAME   0x0021e8 0x00000000000021e8 0x00000000000021e8 0x000044 0x000044 R   0x4
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
  GNU_RELRO      0x002da0 0x0000000000003da0 0x0000000000003da0 0x000260 0x000260 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
```

```
  05     .init_array .fini_array .dynamic .got .data .bss
  06     .dynamic
  07     .note.gnu.property
  08     .note.gnu.build-id .note.ABI-tag
  09     .note.gnu.property
  10     .eh_frame_hdr
  11
  12     .init_array .fini_array .dynamic .got
root@5e3abe332c5a:/home/docker/case_code_100/57# readelf -SW  main
There are 31 section headers ,  starting at offset 0x3b10:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        0000000000000318 000318 00001c 00   A  0   0  1
  [ 2] .note.gnu.property NOTE           0000000000000338 000338 000020 00   A  0   0  8
  [ 3] .note.gnu.build-id NOTE           0000000000000358 000358 000024 00   A  0   0  4
  [ 4] .note.ABI-tag     NOTE            000000000000037c 00037c 000020 00   A  0   0  4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0 0003a0 000024 00   A  6   0  8
  [ 6] .dynsym           DYNSYM          00000000000003c8 0003c8 0000f0 18   A  7   1  8
  [ 7] .dynstr           STRTAB          00000000000004b8 0004b8 0000ab 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          0000000000000564 000564 000014 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         0000000000000578 000578 000030 00   A  7   1  8
  [10] .rela.dyn         RELA            00000000000005a8 0005a8 0000c0 18   A  6   0  8
  [11] .rela.plt         RELA            0000000000000668 000668 000060 18  AI  6  24  8
  [12] .init             PROGBITS        0000000000001000 001000 00001b 00  AX  0   0  4
  [13] .plt              PROGBITS        0000000000001020 001020 000050 10  AX  0   0 16
  [14] .plt.got          PROGBITS        0000000000001070 001070 000010 10  AX  0   0 16
  [15] .plt.sec          PROGBITS        0000000000001080 001080 000040 10  AX  0   0 16
  [16] .text             PROGBITS        00000000000010c0 0010c0 0002e5 00  AX  0   0 16
  [17] .fini             PROGBITS        00000000000013a8 0013a8 00000d 00  AX  0   0  4
  [18] .rodata           PROGBITS        0000000000002000 002000 0001e8 00   A  0   0  8
  [19] .eh_frame_hdr     PROGBITS        00000000000021e8 0021e8 000044 00   A  0   0  4
  [20] .eh_frame         PROGBITS        0000000000002230 002230 000108 00   A  0   0  8
  [21] .init_array       INIT_ARRAY      0000000000003da0 002da0 000008 08  WA  0   0  8
  [22] .fini_array       FINI_ARRAY      0000000000003da8 002da8 000008 08  WA  0   0  8
  [23] .dynamic          DYNAMIC         0000000000003db0 002db0 0001f0 10  WA  7   0  8
  [24] .got              PROGBITS        0000000000003fa0 002fa0 000060 08  WA  0   0  8
  [25] .data             PROGBITS        0000000000004000 003000 00001c 00  WA  0   0  8
  [26] .bss              NOBITS          000000000000401c 00301c 000014 00  WA  0   0  4
  [27] .comment          PROGBITS        0000000000000000 00301c 00002a 01  MS  0   0  1
  [28] .symtab           SYMTAB          0000000000000000 003048 000708 18     29  50  8
  [29] .strtab           STRTAB          0000000000000000 003750 0002a3 00      0   0  1
  [30] .shstrtab         STRTAB          0000000000000000 0039f3 00011a 00      0   0  1
Key to Flags:
  W (write) ,  A (alloc) ,  X (execute) ,  M (merge) ,  S (strings) ,  I (info) ,
  L (link order) ,  O (extra OS processing required) ,  G (group) ,  T (TLS) ,
  C (compressed) ,  x (unknown) ,  o (OS specific) ,  E (exclude) ,
  l (large) ,  p (processor specific)
```

解读

- 区头表位置的顺序将是加载到内存中映像的顺序，即虚拟地址的大小顺序，可以看出 `.text` < `.rodata` < `.data` < `.bss`
  - v54.xx 鸿蒙内核源码分析(静态链接篇)

运行 | ./main

```
root@5e3abe332c5a:/home/docker/case_code_100/57# ./main
hello harmonyos
全局常量 g_const : 0x5599b1a00008
全局外部有初值 g_init : 0x5599b1a02010
全局外部无初值 g_no_init : 0x5599b1a02028
静态外部有初值 s_exter_init : 0x5599b1a02014
静态外静无初值 s_exter_no_init : 0x5599b1a02020
静态内部有初值 s_inter_init : 0x5599b1a02018
静态内部无初值 s_inter_no_init : 0x5599b1a02024
局部栈区有初值 l_init : 0x7ffda7f4dc94
局部栈区无初值  l_no_init : 0x7ffda7f4dc98
局部栈区常量  l_const : 0x7ffda7f4dc9c
```

```
堆区地址 heap：0x5599b2f522a0
代码区地址：0x5599b19ff1a9
```

解读

- 栈区地址最高 `0x7ffda7******`，局部变量是放在栈中的，这些局部变量地址大小为 `l_init` < `l_no_init` < `l_const`， 这和变量定义的方向是一致的，从而佐证了其用栈方式为递减满栈。越是在前面的变量内存的虚拟地址越小。这个在
  - v01.xx 鸿蒙内核源码分析(双向链表篇) | 谁是内核最重要结构体
  - v20.xx 鸿蒙内核源码分析(用栈方式篇) | 程序运行场地由谁提供 都已说过，请自行翻看。
- 代码区 `.text` 地址最低 `0x5599b1******`，代码区第二个 `LOAD` 加载段，其 `flag` 为(R/E)
- 全局地址顺序是 `g_no_init` (.bss) > `g_init` (.data) > `g_const` (rodata)，刚好和三个区的地址范围吻合。
- 关于静态变量，看地址的顺序 `s_exter_init` < `s_inter_init` < `s_exter_no_init` < `s_inter_no_init` ，说明前两个因为有初始值都放在了 `.data` 区，后两个都放到了 `.bss` 。
- 对于同样在 `.bss` 区的三个变量地址顺序是 `s_exter_no_init(2020)` < `s_inter_no_init(2024)` < `g_no_init(2028)`
- 对于同样在 `.data` 区的三个变量地址顺序是 `g_init(2010)` < `s_exter_init(2014)` < `s_inter_init(2018)`
- 从地址上看 `.bss` ， `.data` 挨在一起的，因为实际的ELF区分布上它们也确实是挨在一起的

```
[25] .data             PROGBITS        0000000000004000 003000 00001c 00  WA  0   0  8
[26] .bss              NOBITS          000000000000401c 00301c 000014 00  WA  0   0  4
```

- 堆区在中间位置 `0x5599b2******`，并且可以发现在 `.bss(0x5599b1a0****)` 和 `.heap(0x5599b2f5****)` 区之间还有大量的虚拟地址没有被使用
- ELF如何被加载运行可翻看 v56.xx 鸿蒙内核源码分析(进程映像篇)

问题

细心的可能会发现了一个问题，`s_inter_init(2018)` ， `s_exter_no_init(2020)` 这两个地址之间只相差两个字节，但是int变量是4个字节，这是为什么呢？

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

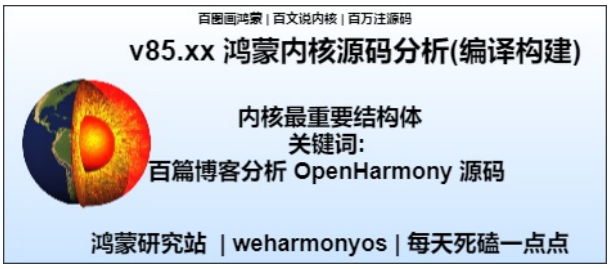
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

85_编译构建篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

构建的必要性

- 前端开发有构建工具：`Grunt`、`Gulp`、`Webpack`
- 后台开发有构建工具: `Maven`、`Ant`、`Gradle`

构建工具重要性不言而喻，它描述了整个工程的如何编译、连接，打包等规则，其中包括：工程中的哪些源文件需要编译以及如何编译、需要创建那些库文件以及如何创建这些库文件、如何最后输出我们想要的文件。

鸿蒙轻内核(L1/liteos)的编译构建工具是`hb`，`hb`是`ohos-build`的简称， 而`ohos`又是`openharmony os`的简称。

hb | ohos-build

hb通过以下命令安装，是用 python写的一个构建工具。

```
python3 -m pip install --user ohos-build
```

其源码在 ./build/lite 目录下，含义如下:

```
build/lite
├── components              # 组件描述文件
├── figures                 # readme中的图片
├── hb                      # hb pip安装包源码
│   ├── build               # hb build 命令实现
│   ├── clean               # hb clean 命令实现
│   ├── common              # 通用类， 提供 Device，Config，Product，utils 类
│   ├── cts                 # hb cts 命令实现
│   ├── deps                # hb deps 命令实现
│   ├── env                 # hb env 命令实现
│   ├── set                 # hb set 命令实现
├── make_rootfs             # 文件系统镜像制作脚本
├── config                  # 编译配置项
```

```
│   ├── component          # 组件相关的模板定义
│   ├── kernel           # 内核相关的编译配置
│   └── subsystem          # 子系统编译配置
├── platform             # ld脚本
├── testfwk              # 测试编译框架
└── toolchain            # 编译工具链配置，包括：编译器路径、编译选项、链接选项等
```

构建组成

鸿蒙构建系统由 `python` ， `gn` ， `ninja` ， `makefile` 几个部分组成，每个部分都有自己的使命，干自己最擅长的活。

- python：胶水语言，最擅长的是对参数，环境变量，文件操作，它任务是做好编译前的准备工作和为gn收集命令参数。不用python直接用gn行不行？ 也行，但很麻烦。比如:直接使用下面的命令行也可以生成 .ninja文件，最后的效果是一样的。但相比只使用 `hb build` 哪个更香， `hb build` 也会生成下面这一坨坨， 但怎么来是python的强项。

  ```
  /home/tools/gn gen /home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries \
  --root=/home/openharmony/code-v1.1.1-LTS \
  --dotfile=/home/openharmony/code-v1.1.1-LTS/build/lite/.gn \
  --script-executable=python3 \
  '--args=ohos_build_type="debug" ohos_build_compiler_specified="clang" ohos_build_compiler_dir="/home/tools/llvm" product_path="/home/
  ```

  图为绕过hb python部分直接执行gn gen 的结果:

- gn：类似构建界的高级语言，gn和ninja的关系有点像C和汇编语言的关系，与它对标的是cmake，它的作用是生成.ninja文件，不用gn直接用ninja行不行？ 也行，但更麻烦。就跟全用汇编写鸿蒙系统一样，理论上可行，可谁会这么去干呢。
- ninja:类似构建界的汇编语言，与它对标的是make，由它完成对编译器clang，链接器ld的使用。
- makefile:鸿蒙有些模块用的还是make编译， 听说后面会统一使用ninja，是不是以后就看不到make文件了，目前是还有大量的make存在。

如何调试 hb

推荐使用vscode来调试，在调试面板点击 `create a launch.json file` 创建调试文件，复制以下内容就可以调试hb了。

```
{
  "version": "0.2.0",
  "configurations": [
    {// hb set
      "name": "hb set",
      "type": "python",
      "request": "launch",
      "program": "./build/lite/hb/__main__.py",
      "console": "integratedTerminal",
      "args": ["set"],
      "stopOnEntry": true
    },
    {//hb build
      "name": "hb build debug",
      "type": "python",
      "request": "launch",
      "program": "./build/lite/hb/__main__.py",
      "console": "integratedTerminal",
      "args": ["build"],
      "stopOnEntry": true
    },
    {//hb clean
      "name": "hb clean",
      "type": "python",
      "request": "launch",
      "program": "./build/lite/hb/__main__.py",
      "console": "integratedTerminal",
      "args": ["clean"],
```

```
        "stopOnEntry": true
      },
    ]
}
```

构建流程

编译构建流程图所示，主要分设置和编译两步：

本篇调试图中的 `hb set` | `hb build` 两个命令

hb set | 选择项目

源码见于: `./build/lite/hb/set/set.py` `hb set` 执行的大致流程是这样的:

- 你可以在任何目录下执行 `hb set` ， 它尝试读取当前目录下的 `ohos_config.json` 配置文件，如果没有会让你输入代码的路径

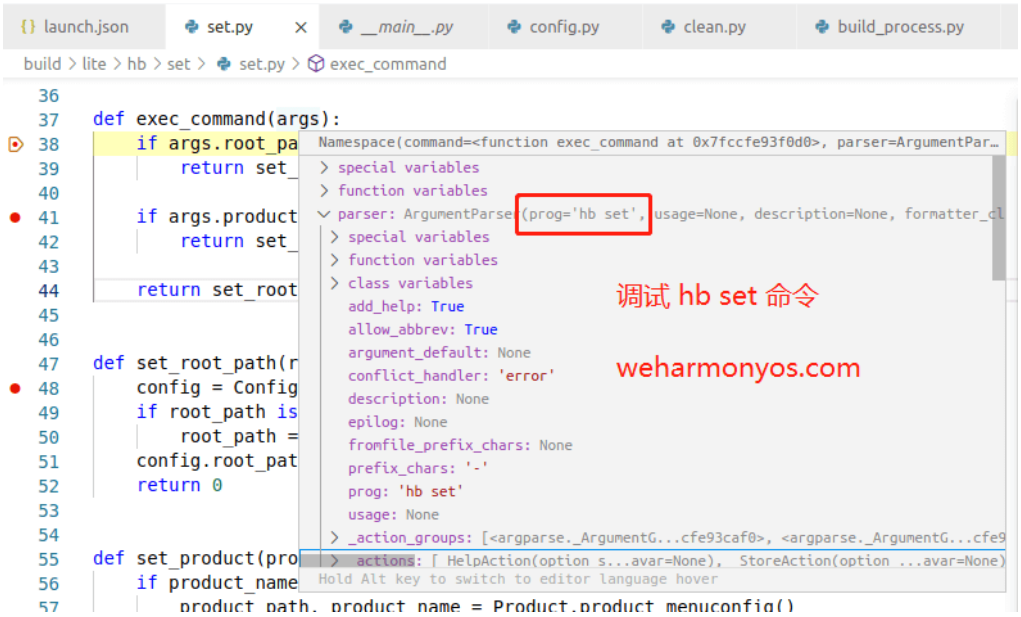
  ```
  [OHOS INFO] Input code path:
  ```

  也就是源码根目录， 生成 `ohos_config.json` 配置文件，配置内容项是固定的，由 `Config` 类管理。

- 可以在以下位置打上断点调试 `set` 命令，跟踪整个过程。

  ```
  def exec_command(args):
      if args.root_path is not None:
          return set_root_path(root_path=args.root_path)

      if args.product:
          return set_product()

      return set_root_path() == 0 and set_product() == 0
  ```

  图为断点调试现场



- 最后生成的配置文件如下:

  ```
  {
  "root_path": "/home/openharmony/code-v1.1.1-LTS" ,
  "board": "hispark_aries" ,
  "kernel": "liteos_a" ,
  "product": "ipcamera_hispark_aries" ,
  "product_path": "/home/openharmony/code-v1.1.1-LTS/vendor/hisilicon/hispark_aries" ,
  ```

```
  "device_path": "/home/openharmony/code-v1.1.1-LTS/device/hisilicon/hispark_aries/sdk_liteos" ,
  "patch_cache": null
  }
```

有了这些路径就为后续 `hb build` 铺好了路。

hb build | 编译项目

源码见于: `./build/lite/hb/build/*.py` 建议大家去调试下源码，非常有意思，能看清楚所有的细节。本篇将编译工具中重要代码都加上了注解。 也可以前往 weharmony | 注解鸿蒙编译工具 查看对其的代码注释工程。

总体步骤是分两步:

- 调用 gn_build 使用 `gn gen` 生成 *.ninja 文件
- 调用 ninja_build 使用 `ninja -w dupbuild=warn -C` 生成 *.o *.so *.bin 等最后的文件

gn_build

关于gn的资料可以前往 GN参考手册查看。 具体gn是如何生成.ninja文件的，后续有篇幅详细介绍其语法及在鸿蒙中的使用。

```
#执行gn编译
def gn_build(self， cmd_args):
  # Clean out path
  remove_path(self.config.out_path) #先删除out目录
  makedirs(self.config.out_path)   #创建out目录

  # Gn cmd init and execute ，生成 build.ninja， args.gn
  gn_path = self.config.gn_path
  gn_args = cmd_args.get('gn'， [])
  gn_cmd = [gn_path，#gn的安装路径 ~/gn
        'gen'，
        self.config.out_path， #/home/openharmony/out/hispark_aries/ipcamera_hispark_aries
        '--root={}'.format(self.config.root_path)， #项目的根例如:/home/openharmony
        '--dotfile={}/.gn'.format(self.config.build_path)，#/home/openharmony/build/lite/.gn -> root = "//build/lite"
        f'--script-executable={sys.executable}'，#python3
        '--args={}'.format(" ".join(self._args_list))] + gn_args
  #   ohos_build_type="debug"
  #   ohos_build_compiler_specified="clang"
  #   ohos_build_compiler_dir="/root/llvm"
  #   product_path="/home/openharmony/vendor/hisilicon/hispark_aries"
  #   device_path="/home/openharmony/device/hisilicon/hispark_aries/sdk_liteos"
  #   ohos_kernel_type="liteos_a"
  #   enable_ohos_appexecfwk_feature_ability = false
  #   ohos_full_compile=true'
  #   这些参数也将在exec_command后保存在 args.gn文件中
  exec_command(gn_cmd， log_path=self.config.log_path)#执行 gn gen .. 命令，在./out/hispark_aries/ipcamera_hispark_aries目录下生成如下文件
  #   obj 子编译项生成的 ninja文件目录，例如:obj/base/global/resmgr_lite/frameworks/resmgr_lite/global_resmgr.ninja
  #   args.gn 各种参数， ohos_build_type="debug" ...
  #   build.ninja 子编译项 例如: build aa: phony dev_tools/bin/aa
  #   build.ninja.d 由那些模块产生的子编译项 例如:../../../base/global/resmgr_lite/frameworks/resmgr_lite/BUILD.gn
  #   toolchain.ninja 工具链 放置了各种编译/链接规则 rule cxx rule alink
```

ninja_build

关于ninja 的资料可以前往 ninja 参考手册查看。 具体ninja是如何运行的，后续有篇幅详细介绍其语法及在鸿蒙中的使用。

```
# ninja 编译过程
def ninja_build(self， cmd_args):
  ninja_path = self.config.ninja_path

  ninja_args = cmd_args.get('ninja'， [])
  ninja_cmd = [ninja_path，
          '-w'，
          'dupbuild=warn'，
          '-C'，
          self.config.out_path] + ninja_args
  # ninja -w dupbuild=warn -C /home/openharmony/out/hispark_aries/ipcamera_hispark_aries
  # 将读取gn生成的文件，完成编译的第二步，最终编译成 .o .bin 文件
```

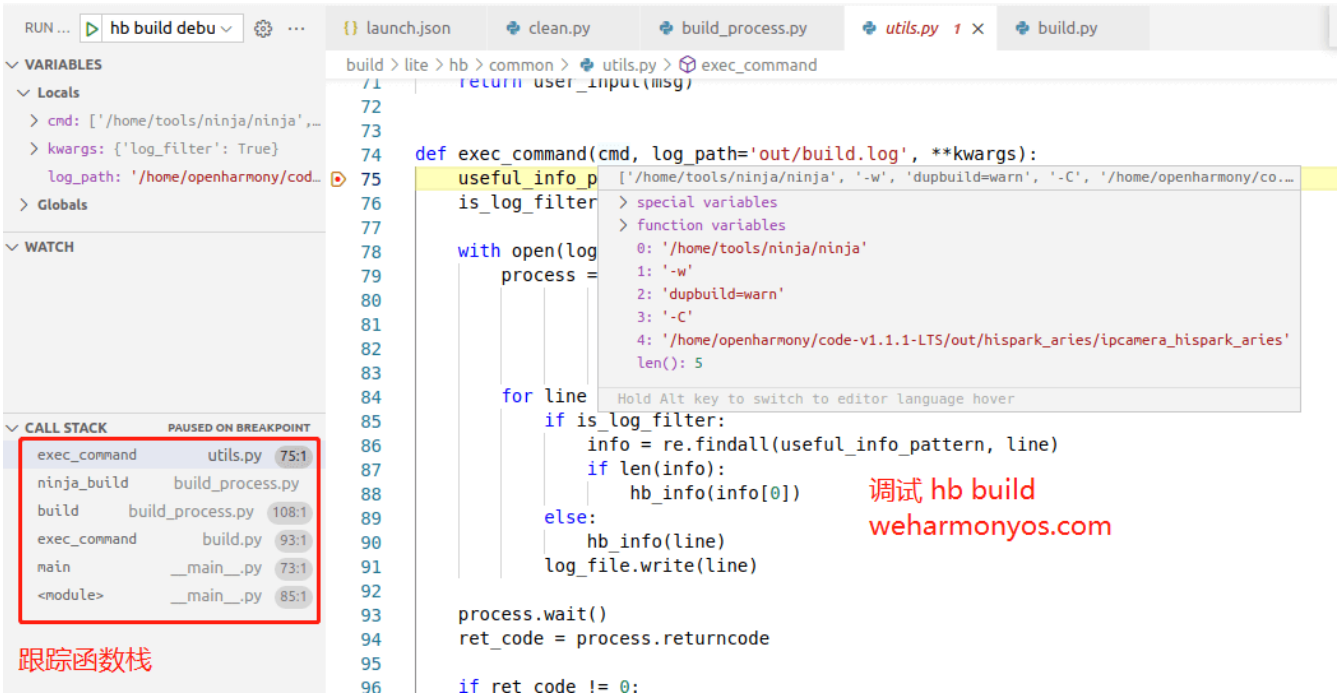
```
        exec_command(ninja_cmd， log_path=self.config.log_path， log_filter=True)
        #生成以下部分文件
        #NOTICE_FILE     OHOS_Image.bin  bin         build.ninja           config     etc           libs           obj             server.map  test_info           us
        #OHOS_Image      OHOS_Image.map  bm_tool.map  build.ninja.d         data       foundation.map  liteos.bin       rootfs.tar       suites      toggl
        #OHOS_Image.asm  args.gn         build.log    bundle_daemon_tool.map  dev_tools  gen           media_server.map  rootfs_jffs2.img  test        t
```

exec_command | utils.py

`gn_build` 和 `ninja_build` 最后都会调用 `exec_command` 来执行命令， `exec_command` 是个通用方法，见于 `build/lite/hb/common/utils.py` ，调试时建议在这里打断点，顺瓜摸藤，跟踪相关函数的实现细节。

```
def exec_command(cmd， log_path='out/build.log'， **kwargs):
    useful_info_pattern = re.compile(r'\[\d+/\d+\].+')
    is_log_filter = kwargs.pop('log_filter'， False)

    with open(log_path， 'at'， encoding='utf-8') as log_file:
        process = subprocess.Popen(cmd，
                                   stdout=subprocess.PIPE，
                                   stderr=subprocess.PIPE，
                                   encoding='utf-8'，
                                   **kwargs)
        for line in iter(process.stdout.readline， ''):
            if is_log_filter:
                info = re.findall(useful_info_pattern， line)
                if len(info):
                    hb_info(info[0])
            else:
                hb_info(line)
            log_file.write(line)

    process.wait()
    ret_code = process.returncode

    if ret_code != 0:
        with open(log_path， 'at'， encoding='utf-8') as log_file:
            for line in iter(process.stderr.readline， ''):
                if 'ninja: warning' in line:
                    log_file.write(line)
                    continue
                hb_error(line)
                log_file.write(line)

        if is_log_filter:
            get_failed_log(log_path)

        hb_error('you can check build log in {}'.format(log_path))
        if isinstance(cmd， list):
            cmd = ' '.join(cmd)
        raise Exception("{} failed， return code is {}".format(cmd， ret_code))
```

图为断点调试现场



百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

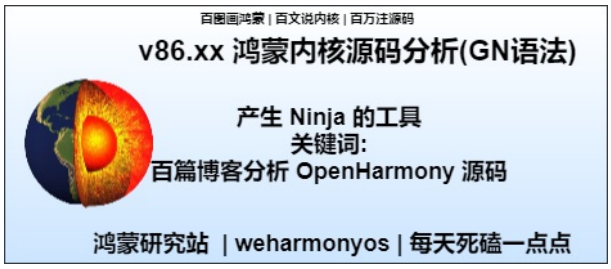
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

86_GN语法篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

gn是什么？

`gn` 存在的意义是为了生成 `ninja`，如果熟悉前端开发，二者关系很像 `Sass` 和 `CSS` 的关系。 为什么会有 `gn`，说是有个叫 `even` 的谷歌负责构建系统的工程师在使用传统的 `makefile` 构建 `chrome` 时觉得太麻烦，不高效，所以设计了一套更简单，更高效新的构建工具 `gn` + `ninja`，然后就被广泛的使用了。

gn语法和配置

`gn` 有大量的内置变量和库函数，熟悉这些库函数基本就知道gn能干什么，gn的官方文档很齐全，前往 gn参考手册翻看查找 。 `gn` 的语法简单，了解以下几点基本就能看懂 `gn` 代码。重点了解下函数的调用方式，和其他高级语言不太一样。

字符串

```
a = "mypath"
b = "$a/foo.cc"  # b -> "mypath/foo.cc"
c = "foo${a}bar.cc"  # c -> "foomypathbar.cc"
```

列表

```
a = [ "first" ]
a += [ "second" ]  # [ "first" , "second" ]
a += [ "third" , "fourth" ]  # [ "first" , "second" , "third" , "fourth" ]
b = a + [ "fifth" ]  # [ "first" , "second" , "third" , "fourth" , "fifth" ]
```

条件语句

```
if (is_linux || (is_win && target_cpu == "x86")) {
    sources -= [ "something.cc" ]
  }else {
    ...
```

```
  }
```

循环

```
foreach(i， mylist) {
  print(i)  # Note: i is a copy of each element， not a reference to it.
}
```

函数调用

```
print("hello， world")
assert(is_win， "This should only be executed on Windows") # 如果is_win为真，就打印后面的内容
static_library("mylibrary") {
  sources = [ "a.cc" ]
}
```

解读:

- `print` ， `assert` ， `static_library` 都是库函数，或者叫内置函数。
- `static_library` 的调用有点奇怪，它是个函数，函数内部会使用 `sources` 这个内置变量， `sources = [ "a.cc" ]` 相当于先传参给这个函数的内置变量，并调用这个函数。学习ng得习惯这种语法方式，被大量的使用。

模板 | Templates

`gn` 提供了很多内置函数，使用偏傻瓜式，若构建不复杂的系统，熟悉这些内置函数，选择填空就可以交作业了，如果想高阶点，想自己定义函数怎么办？ 答案是模板，模板就是自定义函数。

```
#定义模板， 文件路径: //tools/idl_compiler.gni， 后缀.gni 代表这是一个 gn import file
template("idl") { #自定义一个名称为 "idl"的函数
  source_set(target_name) { #调用内置函数 source_set
    sources = invoker.sources #invoker为内置变量，含义为调用者内容 即:[ "a"， "b" ]的内容
  }
}
```

```
#如何使用模板， 用import，类似 C语言的 #include
import("//tools/idl_compiler.gni")
idl("my_interfaces") { #等同于调用 idl
  sources = [ "a"， "b" ] #给idl传参， 参数的接收方是 invoker.sources
}
```

明白了模板的使用，阅读鸿蒙gn代码就不会有太大的障碍。

目标项 | Targets

目标是构建图中的一个节点。它通常表示将生成某种可执行文件或库文件。整个构建是由一个个的目标组成。 目标包含:

```
action: 运行脚本以生成文件
executable: 生成可执行文件
group: 生成依赖关系组
shared_library: 生成.dll或.so动态链接库
static_library: 生成.lib或.a 静态链接库
...
```

配置项 | Configs

记录完成目标项所需的配置信息，例如:

```
config("myconfig") {#创建一个标签为`myconfig`的配置项
    include_dirs = [ "include/common" ]
    defines = [ "ENABLE_DOOM_MELON" ]
  }

executable("mything") {#生成可执行文件
```

```
  configs = [ ":myconfig" ]#使用标签为`myconfig`的配置项来生成目标文件
}
```

gn在鸿蒙中的使用

有了以上基础铺垫，正式开始 gn 在 openharomny 中的使用。

从哪开始

在构建工具篇中已经说清楚了 hb 的 python 部分有个工作任务是生成 gn 命令所需参数. gn 生成 ninja 的命令是 gn gen ...

```
/home/tools/gn gen /home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries \
--root=/home/openharmony/code-v1.1.1-LTS \
--dotfile=/home/openharmony/code-v1.1.1-LTS/build/lite/.gn \
--script-executable=python3 \
'--args=ohos_build_type="debug" \
      ohos_build_compiler_specified="clang" \
      ohos_build_compiler_dir="/home/tools/llvm" \
      product_path="/home/openharmony/code-v1.1.1-LTS/vendor/hisilicon/hispark_aries" \
      device_path="/home/openharmony/code-v1.1.1-LTS/device/hisilicon/hispark_aries/sdk_liteos" \
      ohos_kernel_type="liteos_a" \
      enable_ohos_appexecfwk_feature_ability = false \
      ohos_full_compile=true'
```

解读

- root ， dotfile ， script-executable 是gn内置的固定参数，一切从 dotfile 指向的文件开始。即 build/lite/.gn 相当于 main() 函数的作用，
- 打开 build/lite/.gn看下内容，只有两句，先配置好内部参数再工作。

  ```
  # The location of the build configuration file. #1.完成gn的配置工作
  buildconfig = "//build/lite/config/BUILDCONFIG.gn"

  # The source root location. #2.完成gn的编译工作
  root = "//build/lite"
  ```

- args 为用户自定义的参数，它们将会在解析 BUILD.gn ， BUILDCONFIG.gn 过程中被使用。

BUILDCONFIG.gn | 构建配置项

BUILDCONFIG.gn 为 BUILD.gn 做准备，填充好编译所需的配置信息。即生成配置项 详细请查看 build/lite/config/BUILDCONFIG.gn 文件全部内容，本篇只贴出部分。

```
import("//build/lite/ohos_var.gni")
import("${device_path}/config.gni")
....
arch = "arm"
if (ohos_kernel_type == "liteos_a") {
  target_triple = "$arch-liteos"
} else if (ohos_kernel_type == "linux") {
  target_triple = "$arch-linux-ohosmusl"
}
...
template("executable") { #生成可执行文件
  executable(target_name) {
    forward_variables_from(invoker， Variables_Executable)
    if (!defined(invoker.deps)) {
      deps = [ "//build/lite:prebuilts" ]
    } else {
      deps += [ "//build/lite:prebuilts" ]
    }
    if (defined(invoker.configs)) {
      configs = []
      configs += invoker.configs
    }
  }
```

```
}
set_defaults("executable") {#设置目标类型的默认值
  configs = default_executable_configs
  configs += [ "//build/lite/config:board_exe_ld_flags" ]
}
...
```

解读

- `${device_path}` 为命令行参数，本篇为 `home/openharmony/code-v1.1.1-LTS/device/hisilicon/hispark_aries/sdk_liteos`
- 查看构建-配置内容， `BUILDCONFIG` 主要任务就是对其中的内置变量赋值。例如:
  - 指定编译器 clang，
  - 如何生成可执行文件的方法等

BUILD.gn | 启动构建

查看 build/lite/BUILD.gn 文件，文件注解较多。

```
#目的是要得到项目各个模块的编译入口
group("ohos") {
  deps = []
  if (ohos_build_target == "") {
    # Step 1: Read product configuration profile。
    # 第一步:读取配置文件product_path的值来源于根目录的ohos_config.json，如下，内容由 hb set 命令生成
    # {
    #  "root_path": "/home/openharmony" ,
    #  "board": "hispark_aries" ,
    #  "kernel": "liteos_a" ,
    #  "product": "ipcamera_hispark_aries" ,
    #  "product_path": "/home/openharmony/vendor/hisilicon/hispark_aries" ,
    #  "device_path": "/home/openharmony/device/hisilicon/hispark_aries/sdk_liteos" ,
    #  "patch_cache": null
    #}
    product_cfg = read_file("${product_path}/config.json" ,  "json")

    # Step 2: Loop subsystems configured by product.
    # 第二步:循环处理各自子系统，config.json中子系统部分格式如下hb
    #"subsystems": [
    #  {
    #    "subsystem": "aafwk" ,
    #    "components": [
    #      { "component": "ability" ,  "features":[ "enable_ohos_appexecfwk_feature_ability = false" ] }
    #    ]
    #  } ,
    #  ...
    #  {
    #    "subsystem": "distributed_schedule" ,
    #    "components": [
    #   { "component": "system_ability_manager" ,  "features":[] } ,
    #   { "component": "foundation" ,  "features":[] } ,
    #   { "component": "distributed_schedule" ,  "features":[] }
    #    ]
    #  } ,
    #  {
    #      "subsystem": "kernel" ,
    #      "components": [
    #        { "component": "liteos_a" ,  "features":[] }
    #      ]
    #   } ,
    #]
    foreach(product_configed_subsystem ,  product_cfg.subsystems) {#对子系统数组遍历操作
      subsystem_name = product_configed_subsystem.subsystem #读取一个子系统 aafwk，hiviewdfx，security ==
      subsystem_info = {
      }

      # Step 3: Read OS subsystems profile.
     # 第三步: 读取各个子系统的配置文件
      subsystem_info =
          read_file("//build/lite/components/${subsystem_name}.json" ,  "json")
```

```
    # Step 4: Loop components configured by product.
    # 第四步: 循环读取子系统内各控件的配置信息
    # 此处以内核为例://build/lite/components/kernel.json"
    # "components": [
    #   {
    #     "component": "liteos_a" ,              # 组件名称
    #     "description": "liteos-a kernel" ,      # 组件一句话功能描述
    #     "optional": "false" ,                  # 组件是否为最小系统必选
    #     "dirs": [                          # 组件源码路径
    #       "kernel/liteos_a"
    #     ] ,
    #     "targets": [                       # 组件编译入口
    #       "//kernel/liteos_a:kernel"
    #     ] ,
    #     "rom": "1.98MB" ,                      # 组件ROM值
    #     "ram": "" ,                          # 组件RAM估值
    #     "output": [                        # 组件编译输出
    #       "liteos.bin"
    #     ] ,
    #     "adapted_board": [                 # 组件已适配的主板
    #       "hispark_aries" ,
    #       "hispark_taurus" ,
    #       "hi3518ev300" ,
    #    "hi3516dv300" ,
    #     ] ,
    #     "adapted_kernel": [ "liteos_a" ] ,      # 组件已适配的内核
    #     "features": [] ,                     # 组件可配置的特性
    #     "deps": {
    #       "components": [] ,                   # 组件依赖的其他组件
    #       "third_party": [                 # 组件依赖的三方开源软件
    #         "FreeBSD" ,
    #         "musl" ,
    #         "zlib" ,
    #         "FatFs" ,
    #         "Linux_Kernel" ,
    #         "lwip" ,
    #         "NuttX" ,
    #         "mtd-utils"
    #       ]
    #     }
    #   } ,
    # ]
    foreach(product_configed_component ,
           product_configed_subsystem.components) { #遍历项目控件数组
      # Step 5: Check whether the component configured by product is exist.
    # 第五步: 检查控件配置信息是否存在
      component_found = false #初始为不存在
      foreach(system_component ,  subsystem_info.components) {#项目控件和子系统中的控件遍历对比
        if (product_configed_component.component ==
            system_component.component) { #找到了liteos_a
          component_found = true
        }
      }
    #如果没找到的信息，则打印项目控件查找失败日志
      assert(
          component_found ,
          "Component \"${product_configed_component.component}\" not found" +
              " ,  please check your product configuration.")

      # Step 6: Loop OS components and check validity of product configuration.
# 第六步: 检查子系统控件的有效性并遍历控件组，处理各个控件
      foreach(component ,  subsystem_info.components) {
        kernel_valid = false #检查内核
        board_valid = false #检查开发板

        # Step 6.1: Skip component which not configured by product.
        if (component.component == product_configed_component.component) {
          # Step 6.1.1: Loop OS components adapted kernel type.
          foreach(component_adapted_kernel ,  component.adapted_kernel) {
            if (component_adapted_kernel == product_cfg.kernel_type &&
```

```
            kernel_valid == false) { #内核检测是否已适配
          kernel_valid = true
        }
       }
  # 如果内核未适配，则打印未适配日志
       assert(
          kernel_valid，
          "Invalid component configed， ${subsystem_name}:${product_configed_component.component} " + "not available for kernel: ${produ

       # Step 6.1.2: Add valid component for compiling.
  # 添加有效组件进行编译
       foreach(component_target， component.targets) {//遍历组件的编译入口
        deps += [ component_target ] #添加到编译列表中
       }
      }
     }
    }
   }

   # Step 7: Add device and product target by default.
   # 第七步: 添加设备和项目的编译单元
   # "product_path": "/home/openharmony/vendor/hisilicon/hispark_aries"，
   # "device_path": "/home/openharmony/device/hisilicon/hispark_aries/sdk_liteos"，
    deps += [
    "${device_path}/../"， #添加 //device/hisilicon/hispark_aries 进入编译项
    "${product_path}"  #添加 //vendor/hisilicon/hispark_aries 进入编译项
   ]
  } else {#编译指定的组件，例如 hb build -T targetA&&targetB
   deps += string_split(ohos_build_target， "&&")
  }
 }
```

解读

- 有三个概念贯彻整个鸿蒙系统，子系统(subsystems)，组件(components)，功能(features)。理解它们的定位和特点是解读鸿蒙的关键所在。
- 先找到 product_path 下的 配置文件 config.json ，里面配置了项目所要使用的子系统和组件。
- 再遍历项目所使用的组件是否能再 //build/lite/components/*。json 组件集中能找到。
- 将找到的组件 targets 加入到编译列表 deps 中. targets 指向了要编译的组件目录。例如内核组件时指向了://kernel/liteos_a:kernel，

```
import("//build/lite/config/component/lite_component.gni") #组件模板函数
import("//build/lite/config/subsystem/lite_subsystem.gni") #子系统模板函数
lite_subsystem("kernel") {#编译内核子系统/组件入口
  subsystem_components = []

  if (enable_ohos_kernel_liteos_a_ext_build == false) {
    subsystem_components += [
      "//kernel/liteos_a/kernel"，
      "//kernel/liteos_a/net"，
      "//kernel/liteos_a/lib"，
      "//kernel/liteos_a/compat"，
      "//kernel/liteos_a/fs"，
      "//kernel/liteos_a/arch:platform_cpu"，
    ]
    if (LOSCFG_SHELL) {
      subsystem_components += [ "//kernel/liteos_a/shell" ]
    }
  } else {
    deps = [ ":make" ]
  }
 }
```

lite_subsystem 是个模板函数(自定义函数)，再查看lite_subsystem.gni函数原型，它的目的只有一个填充 deps， deps 是私有链接依赖关系，最终会形成一颗依赖树.gn会根据这些依赖关系生成最终的.ninja文件。

```
# 定义一个子系统
```

```
# lite_subsystem template模板定义了子系统中包含的所有模块
# 参数
#   subsystem_components (必须))
#     [范围列表] 定义子系统的所有模块。
template("lite_subsystem") {
    assert(defined(invoker.subsystem_components) ， "subsystem_components in required.")

    lite_subsystem_components = invoker.subsystem_components

    group(target_name) {
        deps = []
        if(defined(invoker.deps)) {
            deps += invoker.deps
        }
        # add subsystem packages
        foreach(pkg_label ， lite_subsystem_components) {
            deps += [ pkg_label ]
        }
    }
}
```

生成了哪些文件

执行后 `gn gen` 会生成如下文件和目录

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries$ ls
args.gn  build.ninja  build.ninja.d  NOTICE_FILE  obj  test_info  toolchain.ninja
```

`build.ninja.d` 中记录依赖的 `BUILD.gn` 文件路径

```
build.ninja: ../../../base/global/resmgr_lite/frameworks/resmgr_lite/BUILD.gn \
             ../../../base/hiviewdfx/hilog_lite/frameworks/featured/BUILD.gn \
             ../../../base/hiviewdfx/hilog_lite/services/apphilogcat/BUILD.gn \
             ....
```

`gn` 根据这些组件的 `BUILD.gn` 在 `obj` 目录下对应生成了每个组件的 `.ninja` 文件。此处列出鸿蒙L1所有的 `.ninja` 文件， 具体 `ninja` 是如何编译成最终的库和可执行文件的，将在后续篇中详细介绍其语法和应用。

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries/obj$ tree
├── base
│   ├── global
│   │   └── resmgr_lite
│   │       └── frameworks
│   │           └── resmgr_lite
│   │               └── global_resmgr.ninja
│   ├── hiviewdfx
│   │   └── hilog_lite
│   │       ├── frameworks
│   │       │   └── featured
│   │       │       ├── hilog_shared.ninja
│   │       │       └── hilog_static.ninja
│   │       └── services
│   │           ├── apphilogcat
│   │           │   ├── apphilogcat.ninja
│   │           │   └── apphilogcat_static.ninja
│   │           └── hilogcat
│   │               ├── hilogcat.ninja
│   │               └── hilogcat_static.ninja
│   ├── security
│   │   ├── appverify
│   │   │   └── interfaces
│   │   │       └── innerkits
│   │   │           └── appverify_lite
│   │   │               ├── products
│   │   │               │   └── ipcamera
│   │   │               │       └── verify_base.ninja
```

```
│   │   │           ├── unittest
│   │   │           │   └── app_verify_test.ninja
│   │   │           └── verify.ninja
│   │   ├── deviceauth
│   │   │   └── frameworks
│   │   │       └── deviceauth_lite
│   │   │           └── source
│   │   │               └── hichainsdk.ninja
│   │   ├── huks
│   │   │   └── frameworks
│   │   │       └── huks_lite
│   │   │           └── source
│   │   │               └── huks.ninja
│   │   └── permission
│   │       └── services
│   │           └── permission_lite
│   │               ├── ipc_auth
│   │               │   └── ipc_auth_target.ninja
│   │               ├── pms
│   │               │   └── pms_target.ninja
│   │               ├── pms_base
│   │               │   └── pms_base.ninja
│   │               └── pms_client
│   │                   └── pms_client.ninja
│   └── startup
│       ├── appspawn_lite
│       │   └── services
│       │       ├── appspawn.ninja
│       │       └── test
│       │           └── unittest
│       │               └── common
│       │                   └── appspawn_test.ninja
│       ├── bootstrap_lite
│       │   └── services
│       │       └── source
│       │           └── bootstrap.ninja
│       ├── init_lite
│       │   └── services
│       │       ├── init.ninja
│       │       └── test
│       │           └── unittest
│       │               └── common
│       │                   └── init_test.ninja
│       └── syspara_lite
│           └── frameworks
│               ├── parameter
│               │   └── src
│               │       └── sysparam.ninja
│               ├── token
│               │   └── token_shared.ninja
│               └── unittest
│                   └── parameter
│                       └── ParameterTest.ninja
├── build
│   └── lite
│       └── config
│           └── component
│               ├── cJSON
│               │   ├── cjson_shared.ninja
│               │   └── cjson_static.ninja
│               ├── openssl
│               │   ├── openssl_shared.ninja
│               │   └── openssl_static.ninja
│               └── zlib
│                   ├── zlib_shared.ninja
│                   └── zlib_static.ninja
├── drivers
│   ├── adapter
│   │   └── uhdf
│   │       ├── manager
│   │       │   └── hdf_core.ninja
```

```
│   │   ├── platform
│   │   │   └── hdf_platform.ninja
│   │   ├── posix
│   │   │   └── hdf_posix_osal.ninja
│   │   └── test
│   │       └── unittest
│   │           ├── common
│   │           │   └── hdf_test_common.ninja
│   │           ├── config
│   │           │   └── hdf_adapter_uhdf_test_config.ninja
│   │           ├── manager
│   │           │   ├── hdf_adapter_uhdf_test_door.ninja
│   │           │   ├── hdf_adapter_uhdf_test_ioservice.ninja
│   │           │   ├── hdf_adapter_uhdf_test_manager.ninja
│   │           │   └── hdf_adapter_uhdf_test_sbuf.ninja
│   │           ├── osal
│   │           │   └── hdf_adapter_uhdf_test_osal.ninja
│   │           └── platform
│   │               └── hdf_adapter_uhdf_test_platform.ninja
│   └── peripheral
│       ├── input
│       │   └── hal
│       │       └── hdi_input.ninja
│       └── wlan
│           ├── client
│           │   └── wifi_driver_client.ninja
│           ├── hal
│           │   └── wifi_hal.ninja
│           └── test
│               ├── performance
│               │   └── hdf_peripheral_wlan_test_performance.ninja
│               └── unittest
│                   └── hdf_peripheral_wlan_test.ninja
├── foundation
│   ├── aafwk
│   │   └── aafwk_lite
│   │       ├── frameworks
│   │       │   ├── ability_lite
│   │       │   │   └── ability.ninja
│   │       │   ├── abilitymgr_lite
│   │       │   │   └── abilitymanager.ninja
│   │       │   └── want_lite
│   │       │       └── want.ninja
│   │       └── services
│   │           └── abilitymgr_lite
│   │               ├── abilityms.ninja
│   │               ├── tools
│   │               │   └── aa.ninja
│   │               └── unittest
│   │                   └── test_lv0
│   │                       └── page_ability_test
│   │                           └── ability_test_pageAbilityTest_lv0.ninja
│   ├── ai
│   │   └── engine
│   │       ├── services
│   │       │   ├── client
│   │       │   │   ├── ai_client.ninja
│   │       │   │   ├── client_executor
│   │       │   │   │   └── client_executor.ninja
│   │       │   │   └── communication_adapter
│   │       │   │       └── ai_communication_adapter.ninja
│   │       │   ├── common
│   │       │   │   ├── platform
│   │       │   │   │   ├── dl_operation
│   │       │   │   │   │   └── dlOperation.ninja
│   │       │   │   │   ├── event
│   │       │   │   │   │   └── event.ninja
│   │       │   │   │   ├── lock
│   │       │   │   │   │   └── lock.ninja
│   │       │   │   │   ├── os_wrapper
│   │       │   │   │   │   └── ipc
```

```
│   │   │   │   │   │   └── aie_ipc.ninja
│   │   │   │   │   ├── semaphore
│   │   │   │   │   │   └── semaphore.ninja
│   │   │   │   │   ├── threadpool
│   │   │   │   │   │   └── threadpool.ninja
│   │   │   │   │   └── time
│   │   │   │   │       └── time.ninja
│   │   │   │   ├── protocol
│   │   │   │   │   └── data_channel
│   │   │   │   │       └── data_channel.ninja
│   │   │   │   └── utils
│   │   │   │       └── encdec
│   │   │   │           └── encdec.ninja
│   │   │   └── server
│   │   │       ├── ai_server.ninja
│   │   │       ├── communication_adapter
│   │   │       │   └── ai_communication_adapter.ninja
│   │   │       ├── plugin_manager
│   │   │       │   └── plugin_manager.ninja
│   │   │       └── server_executor
│   │   │           └── server_executor.ninja
│   │   └── test
│   │       ├── common
│   │       │   ├── ai_test_common.ninja
│   │       │   └── dl_operation
│   │       │       └── dl_operation_so
│   │       │           └── dlOperationSo.ninja
│   │       ├── function
│   │       │   ├── ai_test_function.ninja
│   │       │   └── death_callback
│   │       │       ├── testDeathCallbackLibrary.ninja
│   │       │       └── testDeathCallback.ninja
│   │       ├── performance
│   │       │   └── ai_test_performance_unittest.ninja
│   │       └── sample
│   │           ├── asyncDemoPluginCode.ninja
│   │           ├── sample_plugin_1.ninja
│   │           ├── sample_plugin_2.ninja
│   │           └── syncDemoPluginCode.ninja
│   ├── appexecfwk
│   │   └── appexecfwk_lite
│   │       ├── frameworks
│   │       │   └── bundle_lite
│   │       │       └── bundle.ninja
│   │       └── services
│   │           └── bundlemgr_lite
│   │               ├── bundle_daemon
│   │               │   └── bundle_daemon.ninja
│   │               ├── bundlems.ninja
│   │               └── tools
│   │                   └── bm.ninja
│   ├── communication
│   │   ├── ipc_lite
│   │   │   └── liteipc_adapter.ninja
│   │   └── softbus_lite
│   │       └── softbus_lite.ninja
│   ├── distributedschedule
│   │   ├── dmsfwk_lite
│   │   │   ├── dmslite.ninja
│   │   │   └── moduletest
│   │   │       └── dtbschedmgr_lite
│   │   │           └── distributed_schedule_test_dms.ninja
│   │   ├── safwk_lite
│   │   │   └── foundation.ninja
│   │   └── samgr_lite
│   │       ├── communication
│   │       │   └── broadcast
│   │       │       └── broadcast.ninja
│   │       ├── samgr
│   │       │   ├── adapter
│   │       │   │   └── samgr_adapter.ninja
```

```
│   │   │   ├── samgr.ninja
│   │   │   └── source
│   │   │       └── samgr_source.ninja
│   │   ├── samgr_client
│   │   │   └── client.ninja
│   │   ├── samgr_endpoint
│   │   │   ├── endpoint_source.ninja
│   │   │   └── store_source.ninja
│   │   └── samgr_server
│   │       └── server.ninja
│   ├── graphic
│   │   ├── surface
│   │   │   ├── surface.ninja
│   │   │   └── test
│   │   │       └── lite_surface_unittest.ninja
│   │   ├── ui
│   │   │   └── ui.ninja
│   │   ├── utils
│   │   │   ├── graphic_hals.ninja
│   │   │   ├── graphic_utils.ninja
│   │   │   └── test
│   │   │       ├── graphic_test_color.ninja
│   │   │       ├── graphic_test_container.ninja
│   │   │       ├── graphic_test_geometry2d.ninja
│   │   │       ├── graphic_test_math.ninja
│   │   │       └── graphic_test_style.ninja
│   │   └── wms
│   │       ├── wms_client.ninja
│   │       └── wms_server.ninja
│   └── multimedia
│       ├── audio_lite
│       │   └── frameworks
│       │       └── audio_capturer_lite.ninja
│       ├── camera_lite
│       │   └── frameworks
│       │       └── camera_lite.ninja
│       ├── media_lite
│       │   ├── frameworks
│       │   │   ├── player_lite
│       │   │   │   └── player_lite.ninja
│       │   │   └── recorder_lite
│       │   │       └── recorder_lite.ninja
│       │   ├── interfaces
│       │   │   └── kits
│       │   │       └── player_lite
│       │   │           └── js
│       │   │               └── builtin
│       │   │                   └── audio_lite_api.ninja
│       │   └── services
│       │       └── media_server.ninja
│       └── utils
│           └── lite
│               └── media_common.ninja
├── test
│   ├── developertest
│   │   ├── examples
│   │   │   └── lite
│   │   │       └── cxx_demo
│   │   │           └── test
│   │   │               └── unittest
│   │   │                   └── common
│   │   │                       └── CalcSubTest.ninja
│   │   └── third_party
│   │       └── lib
│   │           └── cpp
│   │               ├── gtest_main.ninja
│   │               └── gtest.ninja
│   └── xts
│       ├── acts
│       │   ├── aafwk_lite
│       │   │   └── ability_posix
```

```
│   │   │   └── module_ActsAbilityMgrTest.ninja
│   │   ├── ai_lite
│   │   │   └── ai_engine_posix
│   │   │       └── base
│   │   │           ├── module_ActsAiEngineTest.ninja
│   │   │           └── src
│   │   │               └── sample
│   │   │                   ├── asyncDemoPluginCode.ninja
│   │   │                   ├── sample_plugin_1_sync.ninja
│   │   │                   ├── sample_plugin_2_async.ninja
│   │   │                   └── syncDemoPluginCode.ninja
│   │   ├── appexecfwk_lite
│   │   │   └── bundle_mgr_posix
│   │   │       └── module_ActsBundleMgrTest.ninja
│   │   ├── communication_lite
│   │   │   ├── lwip_posix
│   │   │   │   └── module_ActsLwipTest.ninja
│   │   │   └── softbus_posix
│   │   │       └── module_ActsSoftBusTest.ninja
│   │   ├── distributed_schedule_lite
│   │   │   └── samgr_posix
│   │   │       └── module_ActsSamgrTest.ninja
│   │   ├── graphic_lite
│   │   │   ├── graphic_utils
│   │   │   │   ├── a
│   │   │   │   │   └── module_ActsUiInterfaceTest1.ninja
│   │   │   │   ├── color_posix
│   │   │   │   │   └── module_ActsColorTest.ninja
│   │   │   │   ├── geometry2d_posix
│   │   │   │   │   └── module_ActsGeometyr2dTest.ninja
│   │   │   │   ├── graphic_math_posix
│   │   │   │   │   └── module_ActsGraphicMathTest.ninja
│   │   │   │   ├── heap_base_posix
│   │   │   │   │   └── module_ActsHeapBaseTest.ninja
│   │   │   │   ├── list_posix
│   │   │   │   │   └── module_ActsListTest.ninja
│   │   │   │   ├── mem_api_posix
│   │   │   │   │   └── module_ActsGraphMemApiTest.ninja
│   │   │   │   ├── rect_posix
│   │   │   │   │   └── module_ActsRectTest.ninja
│   │   │   │   ├── transform_posix
│   │   │   │   │   └── module_ActsTransformTest.ninja
│   │   │   │   └── version_posix
│   │   │   │       └── module_ActsGraphVersionTest.ninja
│   │   │   ├── surface
│   │   │   │   └── surface_posix
│   │   │   │       └── module_ActsSurfaceTest.ninja
│   │   │   └── ui
│   │   │       ├── a
│   │   │       │   └── module_ActsUiInterfaceTest.ninja
│   │   │       ├── animator_posix
│   │   │       │   └── module_ActsAnimatorTest.ninja
│   │   │       ├── easing_equation_posix
│   │   │       │   └── module_ActsEasingEquationTest.ninja
│   │   │       ├── events_posix
│   │   │       │   └── module_ActsEventsTest.ninja
│   │   │       ├── flexlayout_posix
│   │   │       │   └── module_ActsFlexlaoutTest.ninja
│   │   │       ├── gridlayout_posix
│   │   │       │   └── module_ActsGridLayoutTest.ninja
│   │   │       ├── image_posix
│   │   │       │   └── module_ActsImageTest.ninja
│   │   │       ├── interpolation_posix
│   │   │       │   └── module_ActsInterpoliationTest.ninja
│   │   │       ├── layout_posix
│   │   │       │   └── module_ActsLayoutTest.ninja
│   │   │       ├── listlayout_posix
│   │   │       │   └── module_ActsListlayoutTest.ninja
│   │   │       ├── screen_posix
│   │   │       │   └── module_ActsScreenTest.ninja
│   │   │       ├── style_posix
```

```
│   │   │   │   └── module_ActsStyleTest.ninja
│   │   │   ├── theme_posix
│   │   │   │   └── module_ActsThemeTest.ninja
│   │   │   ├── ui_abstract_progress_posix
│   │   │   │   └── module_ActsUIAbstractProgressTest.ninja
│   │   │   ├── ui_analog_clock_posix
│   │   │   │   └── module_ActsUIAnalogClockTest.ninja
│   │   │   ├── uianimator_posix
│   │   │   │   └── module_ActsUIAnimatorTest.ninja
│   │   │   ├── ui_arc_lable_posix
│   │   │   │   └── module_ActsUIArcLabelTest.ninja
│   │   │   ├── ui_axis_posix
│   │   │   │   └── module_ActsUIAxisTest.ninja
│   │   │   ├── ui_box_porgress_posix
│   │   │   │   └── module_ActsUIBoxProgressTest.ninja
│   │   │   ├── ui_button_posix
│   │   │   │   └── module_ActsUIButtonTest.ninja
│   │   │   ├── ui_canvas_posix
│   │   │   │   └── module_ActsUICanvasTest.ninja
│   │   │   ├── ui_chart_posix
│   │   │   │   └── module_ActsUIChartTest.ninja
│   │   │   ├── ui_checbox_posix
│   │   │   │   └── module_ActsUICheckboxTest.ninja
│   │   │   ├── ui_circle_progress_posix
│   │   │   │   └── module_ActsUICircleProgressTest.ninja
│   │   │   ├── ui_digital_clock_posix
│   │   │   │   └── module_ActsUIDigitalClockTest.ninja
│   │   │   ├── ui_image_animator_posix
│   │   │   │   └── module_ActsUIImageAnimatorTest.ninja
│   │   │   ├── ui_image_posix
│   │   │   │   └── module_ActsUIImageTest.ninja
│   │   │   ├── ui_label_button_posix
│   │   │   │   └── module_ActsUILabelButtonTest.ninja
│   │   │   ├── ui_label_posix
│   │   │   │   └── module_ActsUILabelTest.ninja
│   │   │   ├── ui_list_posix
│   │   │   │   └── module_ActsUIListTest.ninja
│   │   │   ├── ui_picker_posix
│   │   │   │   └── module_ActsUIPickerTest.ninja
│   │   │   ├── ui_radio_button_posix
│   │   │   │   └── module_ActsUIRadioButtonTest.ninja
│   │   │   ├── ui_repeat_button_posix
│   │   │   │   └── module_ActsUIRepeatButtonTest.ninja
│   │   │   ├── ui_screenshot_posix
│   │   │   │   └── module_ActsUIScreenshotTest.ninja
│   │   │   ├── ui_scroll_view_posix
│   │   │   │   └── module_ActsUIScrollViewTest.ninja
│   │   │   ├── ui_slider_posix
│   │   │   │   └── module_ActsUISliderTest.ninja
│   │   │   ├── ui_surface_view_posix
│   │   │   │   └── module_ActsUISurfaceViewTest.ninja
│   │   │   ├── ui_swipe_view_posix
│   │   │   │   └── module_ActsUISwipeViewTest.ninja
│   │   │   ├── ui_text_posix
│   │   │   │   └── module_ActsUITextTest.ninja
│   │   │   ├── ui_texture_mapper_posix
│   │   │   │   └── module_ActsUITextureMapperTest.ninja
│   │   │   ├── ui_time_picker_posix
│   │   │   │   └── module_ActsUITimePickerTest.ninja
│   │   │   ├── ui_toggle_button_posix
│   │   │   │   └── module_ActsUIToggleButtonTest.ninja
│   │   │   ├── ui_view_group_posix
│   │   │   │   └── module_ActsUIViewGroupTest.ninja
│   │   │   └── ui_view_posix
│   │   │       └── module_ActsUIViewTest.ninja
│   │   ├── hiviewdfx_lite
│   │   │   └── hilog_posix
│   │   │       └── module_ActsHilogTest.ninja
│   │   ├── kernel_lite
│   │   │   ├── dyload_posix
│   │   │   │   └── module_ActsDyloadTest.ninja
```

```
│   │   │   ├── fs_posix
│   │   │   │   ├── jffs
│   │   │   │   │   └── module_ActsJFFS2Test.ninja
│   │   │   │   ├── nfs
│   │   │   │   │   └── module_ActsNFSTest.ninja
│   │   │   │   ├── vfat
│   │   │   │   │   └── module_ActsVFATTest.ninja
│   │   │   │   └── vfat_storage
│   │   │   │       └── module_ActsVFATstorageTest.ninja
│   │   │   ├── futex_posix
│   │   │   │   └── module_ActsFutexApiTest.ninja
│   │   │   ├── io_posix
│   │   │   │   └── module_ActsIoApiTest.ninja
│   │   │   ├── ipc_posix
│   │   │   │   ├── message_queue
│   │   │   │   │   └── module_ActsIpcMqTest.ninja
│   │   │   │   ├── pipe_fifo
│   │   │   │   │   └── module_ActsIpcPipeTest.ninja
│   │   │   │   ├── semaphore
│   │   │   │   │   └── module_ActsIpcSemTest.ninja
│   │   │   │   ├── shared_memory
│   │   │   │   │   └── module_ActsIpcShmTest.ninja
│   │   │   │   └── signal
│   │   │   │       └── module_ActsIpcSignalTest.ninja
│   │   │   ├── math_posix
│   │   │   │   ├── complexTest.ninja
│   │   │   │   └── module_ActsMathApiTest.ninja
│   │   │   ├── mem_posix
│   │   │   │   └── module_ActsMemApiTest.ninja
│   │   │   ├── net_posix
│   │   │   │   └── module_ActsNetTest.ninja
│   │   │   ├── process_posix
│   │   │   │   └── module_ActsProcessApiTest.ninja
│   │   │   ├── sched_posix
│   │   │   │   └── module_ActsSchedApiTest.ninja
│   │   │   ├── sys_posix
│   │   │   │   └── module_ActsSysApiTest.ninja
│   │   │   ├── time_posix
│   │   │   │   └── module_ActsTimeApiTest.ninja
│   │   │   ├── util_posix
│   │   │   │   └── module_ActsUtilApiTest.ninja
│   │   │   └── utils
│   │   │       ├── libfs.ninja
│   │   │       ├── libmt_utils.ninja
│   │   │       └── libutils.ninja
│   │   ├── multimedia_lite
│   │   │   └── media_lite_posix
│   │   │       └── recorder_native
│   │   │           └── module_ActsMediaRecorderTest.ninja
│   │   ├── security_lite
│   │   │   ├── datahuks_posix
│   │   │   │   └── module_ActsSecurityDataTest.ninja
│   │   │   └── permission_posix
│   │   │       ├── capability
│   │   │       │   ├── capability_shared.ninja
│   │   │       │   ├── jffs
│   │   │       │   │   └── module_ActsJFFS2CapabilityTest.ninja
│   │   │       │   └── vfat
│   │   │       │       └── module_ActsVFATCapabilityTest.ninja
│   │   │       ├── dac
│   │   │       │   ├── jffs
│   │   │       │   │   └── module_ActsJFFS2DACTest.ninja
│   │   │       │   └── vfat
│   │   │       │       └── module_ActsVFATDACTest.ninja
│   │   │       └── pms
│   │   │           └── module_ActsPMSTest.ninja
│   │   ├── startup_lite
│   │   │   ├── bootstrap_posix
│   │   │   │   └── module_ActsBootstrapTest.ninja
│   │   │   └── syspara_posix
│   │   │       └── module_ActsParameterTest.ninja
```

```
│       │   └── utils_lite
│       │       └── kv_store_posix
│       │           └── module_ActsKvStoreTest.ninja
│       └── tools
│           └── lite
│               ├── hcpptest
│               │   ├── gmock_main.ninja
│               │   ├── gmock.ninja
│               │   ├── hcpptest_main.ninja
│               │   └── hcpptest.ninja
│               └── others
│                   └── query
│                       └── query.ninja
├── third_party
│   ├── bounds_checking_function
│   │   ├── libsec_shared.ninja
│   │   └── libsec_static.ninja
│   ├── freetype
│   │   └── freetype.ninja
│   ├── giflib
│   │   └── libgif.ninja
│   ├── iniparser
│   │   └── iniparser.ninja
│   ├── libjpeg
│   │   └── libjpeg.ninja
│   ├── libpng
│   │   └── libpng.ninja
│   ├── mbedtls
│   │   ├── mbedtls_gt.ninja
│   │   ├── mbedtls_shared.ninja
│   │   └── mbedtls_static.ninja
│   └── qrcodegen
│       └── qrcodegen.ninja
├── utils
│   └── native
│       └── lite
│           ├── kv_store
│           │   └── src
│           │       └── utils_kv_store.ninja
│           └── os_dump
│               └── os_dump.ninja
└── vendor
    └── hisilicon
        └── hispark_aries
            └── hals
                ├── security
                │   └── permission_lite
                │       └── hal_pms.ninja
                └── utils
                    ├── sys_param
                    │   └── hal_sysparam.ninja
                    └── token
                        └── haltoken_shared.ninja
```

百文说内核 | 抓住主脉络

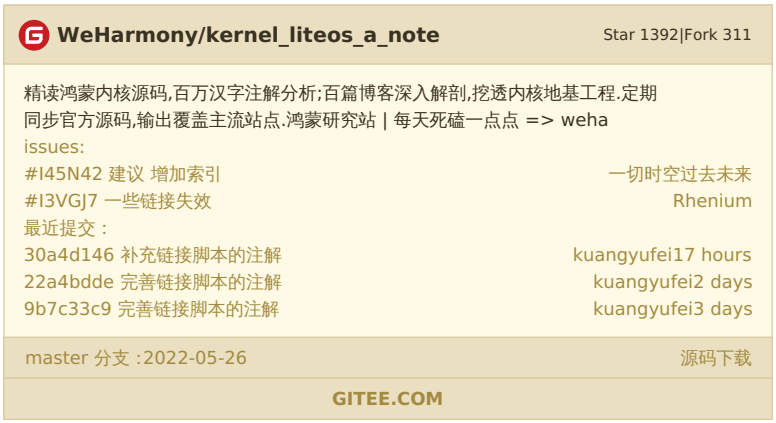
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

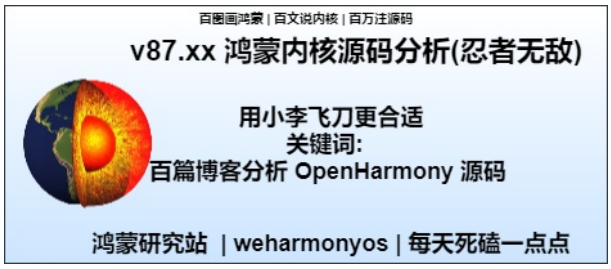


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

87_忍者无敌篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

ninja | 忍者

ninja 是一个叫 Evan Martin 的谷歌工程师开源的一个自定义的构建系统，最早是用于 chrome 的构建， Martin 给它取名 ninja (忍者)的原因是因为它 strikes quickly (快速出击)。这是忍者的特点，看来 Martin 很迷恋岛国文化，动作片估计没少看。 可惜 Martin 不了解中国文化，不然叫小李飞刀更合适些。究竟有多块呢？ 用 Martin 自己的话说是当一个文件被修改后， ninja 从发现到编译速度是 make 的十倍。有没有十倍不是本篇讨论的重点，人家做出来了，就算是牛皮也该人家吹。本篇是要对鸿蒙如何使用ninja做一个比较详细的阐述。

ninja 是一个重视速度的构建系统，与其对标的是 Make ，它们都依赖于文件的时间戳进行检测重编。

- 它的设计目的是让更高级别的构建系统生成其输入端文件，其并不希望你手动去编 .ninja 文件，可以生成 .ninja 的工具有 gn ， cmake ， premake ，甚至你自己都可以写个 ninja 生成工具。
- ninja 非常高效，可理解为构建系统中的汇编语言。
- ninja 文件没有分支、循环的流程控制，是被指定了一堆规则的文件，所以要比 Makefile 简单很多
- 目前已知的 GoogleChrome ， Android 的一部分， LLVM ， V8 ， 方舟编译器， 鸿蒙 等大型系统都使用到了 ninja 构建。

基本概念

```
概念 中译 解释
edge 边     即build语句，指定目标（输出）、规则与输入，是编译过程拓扑图中的一条边（edge）。
target 目标 编译过程需要产生的目标，由build语句指定。
output 输出 build语句的前半段，是target的另一种称呼。
input 输入 build语句的后半段，用来产生output的文件或目标，另一种称呼是依赖。
rule 规则 通过指定command与一些内置变量，决定如何从输入产生输出。
pool 池     一组rule或edge，通过指定其depth，可以控制并行上限。
scope 作用域 变量的作用范围，有rule与build语句的块级，也有文件级别。rule也有scope。
-------------------------------------------------------------------------------------
关键字   作用
build   定义一个edge。
rule   定义一个rule。
pool   定义一个pool。
default   指定默认的一个或多个target。
```

```
include   添加一个ninja文件到当前scope。
subninja  添加一个ninja文件，其scope与当前文件不同。
phony     一个内置的特殊规则，指定非文件的target。
```

简单的ninja

首先 `ninja` 一定是简单的，呆板的。凡是能被工具生成的东西，一定是在不断的重复某种简单，众多的简单按一定的规则有效叠加起来就能解决复杂的问题，请仔细想想是不是这个道理。 `ninja` 简单到没什么语法，只是几个概念和规则。 以至于 ninja参考手册 比 gn参考手册 简单的太多。

看个示例:

```
cflags = -Wall -Werror  #全局变量
rule cc
  command = gcc $cflags -c $in -o $out

build foo.o: cc foo.c

build special.o: cc special.c
  cflags = -Wall #局部变量，范围只在编译special.c上有效
```

解读

- `cflags` :定义一个用户变量，用于给规则传参。
- `rule` :定义一个叫 `cc` 的规则。
  - `command` :将生成bash命令，接收外部三个参数
- 第一个 `build` ，将 `foo.c` 用 `cc` 规则编译成 `foo.o`
  - 最终编译选项: `gcc -Wall -Werror -c foo.c -o foo.o`
- 第二个 `build` ，将 `special.c` 用 `cc` 规则编译成 `special.o`
  - 最终编译选项: `gcc -Wall -c foo.c -o foo.o`
- `in` ， `out` 是 `ninja` 的两个内置变量。

phony规则

跟称呼 `弗拉基米尔·弗拉基米罗维奇·普京` 为 `普总` 一样， 有些文件路径会很长， `ninja` 提供取别名的功能，这仅仅是为了方便。

```
build ability: phony ./libability.so
build ability_notes: phony obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/ability_notes.stamp
build ability_test: phony obj/foundation/aafwk/aafwk_lite/services/abilitymgr_lite/unittest/ability_test.stamp
build ability_test_pageAbilityTest_group_lv0: phony obj/foundation/aafwk/aafwk_lite/services/abilitymgr_lite/unittest/test_lv0/page_ability_test/abil
```

有了上面的铺垫，读懂鸿蒙的 `ninja` 部分应该没多大障碍了。

鸿蒙 | ninja

在[v60.xx 鸿蒙内核源码分析(gn应用篇) | gn语法及在鸿蒙的使用]篇的末尾已说明通过 `gn gen` 生成了以下文件和目录

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries$ ls
args.gn  build.ninja  build.ninja.d  NOTICE_FILE  obj  test_info  toolchain.ninja
```

- `args.gn` :一些参数
- `build.ninja` : `ninja` 的主文件
- `build.ninja.d` :记录生成所有 `.ninja` 所依赖的BUILD.gn文件路劲列表，一个BUILD.gn就生成一个.ninja文件
- obj :各组件模块构建/编译文件输出地。
- toolchain :放置ninja规则，将被 subninja 进 build.ninja

build.ninja

`build.ninja` 内容如下:

```
ninja_required_version = 1.7.2
```

```
rule gn
  command = ../../../../tools/gn --root=../../.. -q --dotfile=../../../build/lite/.gn --script-executable=python3 gen .
  description = Regenerating ninja files

build build.ninja: gn
  generator = 1
  depfile = build.ninja.d

subninja toolchain.ninja


build ability: phony ./libability.so
build ability_notes: phony obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/ability_notes.stamp
build ability_test: phony obj/foundation/aafwk/aafwk_lite/services/abilitymgr_lite/unittest/ability_test.stamp
build ability_test_pageAbilityTest_group_lv0: phony obj/foundation/aafwk/aafwk_lite/services/abilitymgr_lite/unittest/test_lv0/page_ability_test/abil
#此处省略诸多 phony ..

build all: phony $
    ./libcameraApp.so $
    obj/applications/sample/camera/cameraApp/cameraApp_hap.stamp $
    ./libgallery.so $
    ...

default all
```

解读

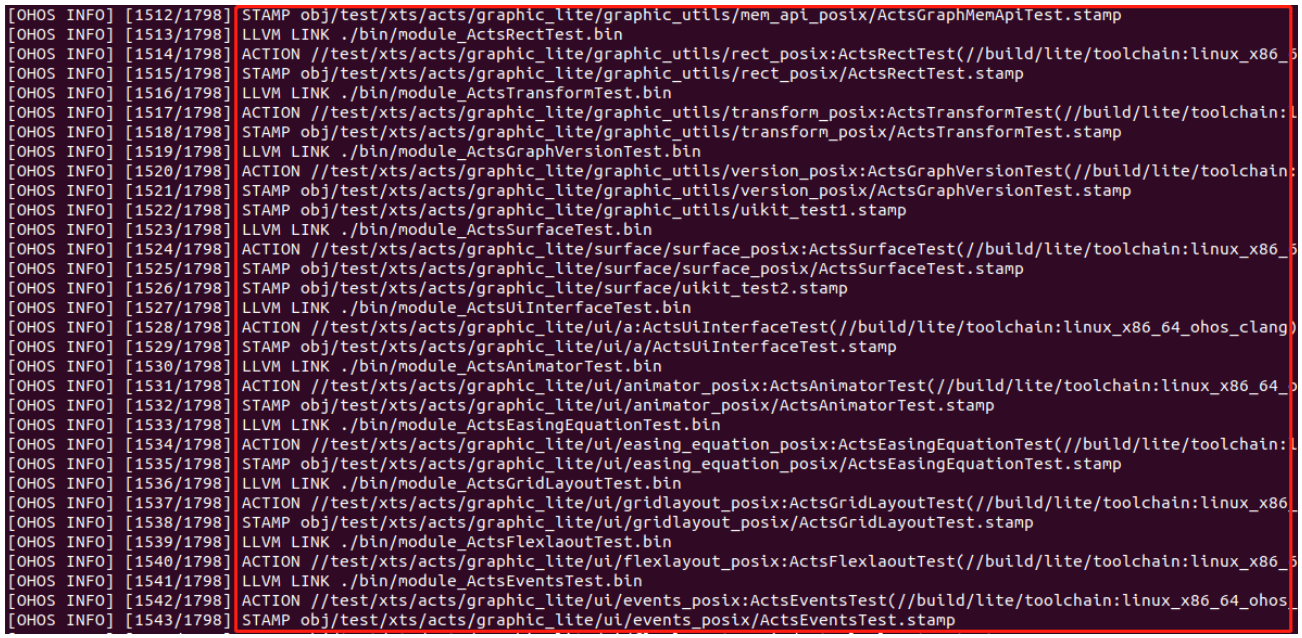
- 前面部分是定义一个 `gn` 规则，用于干嘛呢？ 重新生成一遍 `*ninja` 文件
- `subninja` 相当于 `#include` 文件
- `default all`，指定默认的一个或多个target

toolchain | 定义规则

`toolchain.ninja` 定义了编译c，c++，汇编器，链接，静态/动态链接库，时间戳，拷贝等规则。 内容如下:

```
rule cxx
  command = /root/llvm/bin/clang++ ${defines} ${include_dirs} ${cflags_cc} -c ${in} -o ${out}
  description = clang++ ${out}
  depfile = ${out}.d
  deps = gcc
rule alink
  command = /root/llvm/bin/llvm-ar -cr ${out} @"${out}.rsp"
  description = AR ${out}
  rspfile = ${out}.rsp
  rspfile_content = ${in}
rule link
  command = /root/llvm/bin/clang ${ldflags} ${in} ${libs}   -o ${output_dir}/bin/${target_output_name}${output_extension}
  description = LLVM LINK ${output_dir}/bin/${target_output_name}${output_extension}
  rspfile = ${output_dir}/bin/${target_output_name}${output_extension}.rsp
  rspfile_content = ${in}
rule solink
  command = /root/llvm/bin/clang -shared ${ldflags} ${in} ${libs} -o ${output_dir}/${target_output_name}${output_extension}
  description = SOLINK ${output_dir}/${target_output_name}${output_extension}
  rspfile = ${output_dir}/${target_output_name}${output_extension}.rsp
  rspfile_content = ${in}
rule stamp
  command = /usr/bin/touch ${out}
  description = STAMP ${out}
rule asm
  command = /root/llvm/bin/clang ${include_dirs} ${asmflags} -c ${in} -o ${out}
  description = ASM ${out}
  depfile = ${out}.d
  deps = gcc
rule cc
  command = /root/llvm/bin/clang ${defines} ${include_dirs} ${cflags} ${cflags_c} -c ${in} -o ${out}
  description = clang ${out}
rule copy
  command = cp -afd ${in} ${out}
  description = COPY ${in} ${out}
```

- 注意这些规则中的描述 `description` 字段，其后面的内容会打到控制台上，每一条输出都是一次 `build` ，如图所示，通过这些描述就知道使用了什么规则去构建。



组件编译

本篇以编译 `ability` 组件为例说明 `ninja` 对组件的编译情况。每个组件都有自己的 `.ninja` ，描述组件的编译细节。而整个鸿蒙系统就是由众多的类似 .ninja 构建编译完成的。

```
├── foundation
│   ├── aafwk
│   │   └── aafwk_lite
│   │       ├── frameworks
│   │       │   ├── ability_lite
│   │       │   │   └── ability.ninja
```

`ability.ninja` 内容如下:

```
defines = -DOHOS_APPEXECFWK_BMS_BUNDLEMANAGER \
        -D_XOPEN_SOURCE=700 -DOHOS_DEBUG \
        -D_FORTIFY_SOURCE=2 \
        -D__LITEOS__ -D__LITEOS_A__
include_dirs = -I../../../foundation/aafwk/aafwk_lite/frameworks/abilitymgr_lite/include \
               -I../../../foundation/aafwk/aafwk_lite/frameworks/want_lite/include \
               -I../../../foundation/aafwk/aafwk_lite/interfaces/innerkits/abilitymgr_lite \
               -I../../../foundation/aafwk/aafwk_lite/interfaces/kits/want_lite \
               -I../../../foundation/aafwk/aafwk_lite/interfaces/kits/ability_lite \
               -I../../../foundation/appexecfwk/appexecfwk_lite/utils/bundle_lite \
               -I../../../foundation/appexecfwk/appexecfwk_lite/interfaces/kits/bundle_lite \
               -I../../../foundation/appexecfwk/appexecfwk_lite/frameworks/bundle_lite/include \
               -I../../../foundation/graphic/ui/frameworks -I../../../foundation/graphic/surface/interfaces/kits \
               -I../../../foundation/distributedschedule/samgr_lite/interfaces/kits/registry \
               -I../../../foundation/distributedschedule/samgr_lite/interfaces/kits/samgr \
               -I../../../foundation/communication/ipc_lite/frameworks/liteipc/include \
               -I../../../kernel/liteos_a/kernel/include \
               -I../../../kernel/liteos_a/kernel/common \
               -I../../../third_party/bounds_checking_function/include \
               -I../../../third_party/freetype/include \
               -I../../../utils/native/lite/kv_store/innerkits \
               -I../../../utils/native/lite/include \
               -I../../../foundation/aafwk/aafwk_lite/frameworks/ability_lite/include \
               -I../../../foundation/aafwk/aafwk_lite/frameworks/ability_lite \
```

```
            -I/root/llvm/include/c++/v1 \
            -I../../../prebuilts/lite/sysroot/usr/include/arm-liteos \
            -I../../../base/hiviewdfx/hilog_lite/interfaces/native/innerkits/hilog \
            -I../../../base/hiviewdfx/hilog_lite/interfaces/native/innerkits \
            -I../../../third_party/bounds_checking_function/include \
            -I../../../third_party/bounds_checking_function/include \
            -I../../../foundation/communication/ipc_lite/interfaces/kits \
            -I../../../utils/native/lite/include
cflags = -Wall -Wno-format -Wno-format-extra-args -fPIC \
        --target=arm-liteos \
        --sysroot=/home/openharmony/prebuilts/lite/sysroot \
        -Oz -flto -mfloat-abi=softfp -mcpu=cortex-a7 -nostdlib -fno-common -fno-builtin -fno-strict-aliasing -Wall -fsigned-char -mno-unaligned-acce
cflags_cc = -Wall -Wno-format -Wno-format-extra-args -fPIC \
          --target=arm-liteos \
          --sysroot=/home/openharmony/prebuilts/lite/sysroot \
          -Oz -flto -mfloat-abi=softfp -mcpu=cortex-a7 -nostdlib -fno-common -fno-builtin -fno-strict-aliasing -Wall -mno-unaligned-access -fno-omit-
target_output_name = libability

build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/ability_lite/s
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_context.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/abi
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_env.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/ability_
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_env_impl.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/ab
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_event_handler.o: cxx ../../../foundation/aafwk/aafwk_lite/framewo
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_loader.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/abili
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_main.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/ability
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_scheduler.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/a
build obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_thread.o: cxx ../../../foundation/aafwk/aafwk_lite/frameworks/abili

build ./libability.so: solink \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_context.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_env.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_env_impl.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_event_handler.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_loader.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_main.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_scheduler.o \
  obj/foundation/aafwk/aafwk_lite/frameworks/ability_lite/src/libability.ability_thread.o \
  ./libabilitymanager.so ./libbundle.so ./libhilog_shared.so ./libliteipc_adapter.so \
  ./libsec_shared.so ./libutils_kv_store.so || obj/utils/native/lite/kv_store/kv_store.stamp
  ldflags = -lstdc++ \
  --target=arm-liteos \
  --sysroot=/home/openharmony/prebuilts/lite/sysroot \
  -L/root/llvm/lib/arm-liteos/c++ \
  -L/home/openharmony/prebuilts/lite/sysroot/usr/lib/arm-liteos \
  -L/root/llvm/lib/clang/9.0.0/lib/arm-liteos \
  -lclang_rt.builtins -lc -lc++ -lc++abi \
  --sysroot=/home/openharmony/prebuilts/lite/sysroot \
  -mcpu=cortex-a7 -lc \
  -L/home/openharmony/out/hispark_aries/ipcamera_hispark_aries \
  -Wl,-rpath-link=/home/openharmony/out/hispark_aries/ipcamera_hispark_aries -Wl,-z,now -Wl,-z,relro -Wl,-z,noexecstack
  libs =
  frameworks =
  output_extension = .so
  output_dir = .
```

解读

- `defines`，`include_dirs`，`cflags_cc`都是用户自定义变量，为了给 `rule cxx`准备参数，对`.cpp`的编译使用了这个规则

```
rule cxx
  command = /root/llvm/bin/clang++ ${defines} ${include_dirs} ${cflags_cc} -c ${in} -o ${out}
  description = clang++ ${out}
  depfile = ${out}.d
  deps = gcc
```

- `in`，`out`是两个内置变量，无须定义，值由`build`提供，如此就编译成了一个个的 `.o`文件。
- 在最后在当前目录下使用了`solink`规则，生成一个动态链接库`libability.so`。

```
rule solink
  command = /root/llvm/bin/clang -shared ${ldflags} ${in} ${libs} -o ${output_dir}/${target_output_name}${output_extension}
  description = SOLINK ${output_dir}/${target_output_name}${output_extension}
  rspfile = ${output_dir}/${target_output_name}${output_extension}.rsp
  rspfile_content = ${in}
```

ability | 最终生成文件

```
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries/obj/foundation/aafwk/aafwk_lite/frameworks/abilit
.
├── aafwk_abilitykit_lite.stamp
├── ability.ninja
├── ability_notes.stamp
└── src
    ├── libability.ability_context.o
    ├── libability.ability_env_impl.o
    ├── libability.ability_env.o
    ├── libability.ability_event_handler.o
    ├── libability.ability_loader.o
    ├── libability.ability_main.o
    ├── libability.ability.o
    ├── libability.ability_scheduler.o
    └── libability.ability_thread.o

1 directory， 12 files
turing@ubuntu:/home/openharmony/code-v1.1.1-LTS/out/hispark_aries/ipcamera_hispark_aries/obj/foundation/aafwk/aafwk_lite/frameworks/abilit
  File: ability_notes.stamp
  Size: 0         Blocks: 0          IO Block: 4096   regular empty file
Device: 805h/2053d Inode: 1217028     Links: 1
Access : (0644/-rw-r--r--)  Uid : ( 1000/  turing)   Gid : (    0/    root)
Access: 2021-07-21 00:38:52.237373740 -0700
Modify: 2021-07-21 00:34:30.207312566 -0700
Change: 2021-07-21 00:34:30.207312566 -0700
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

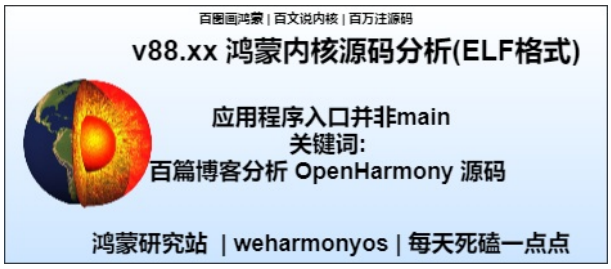
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

88_ELF格式篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

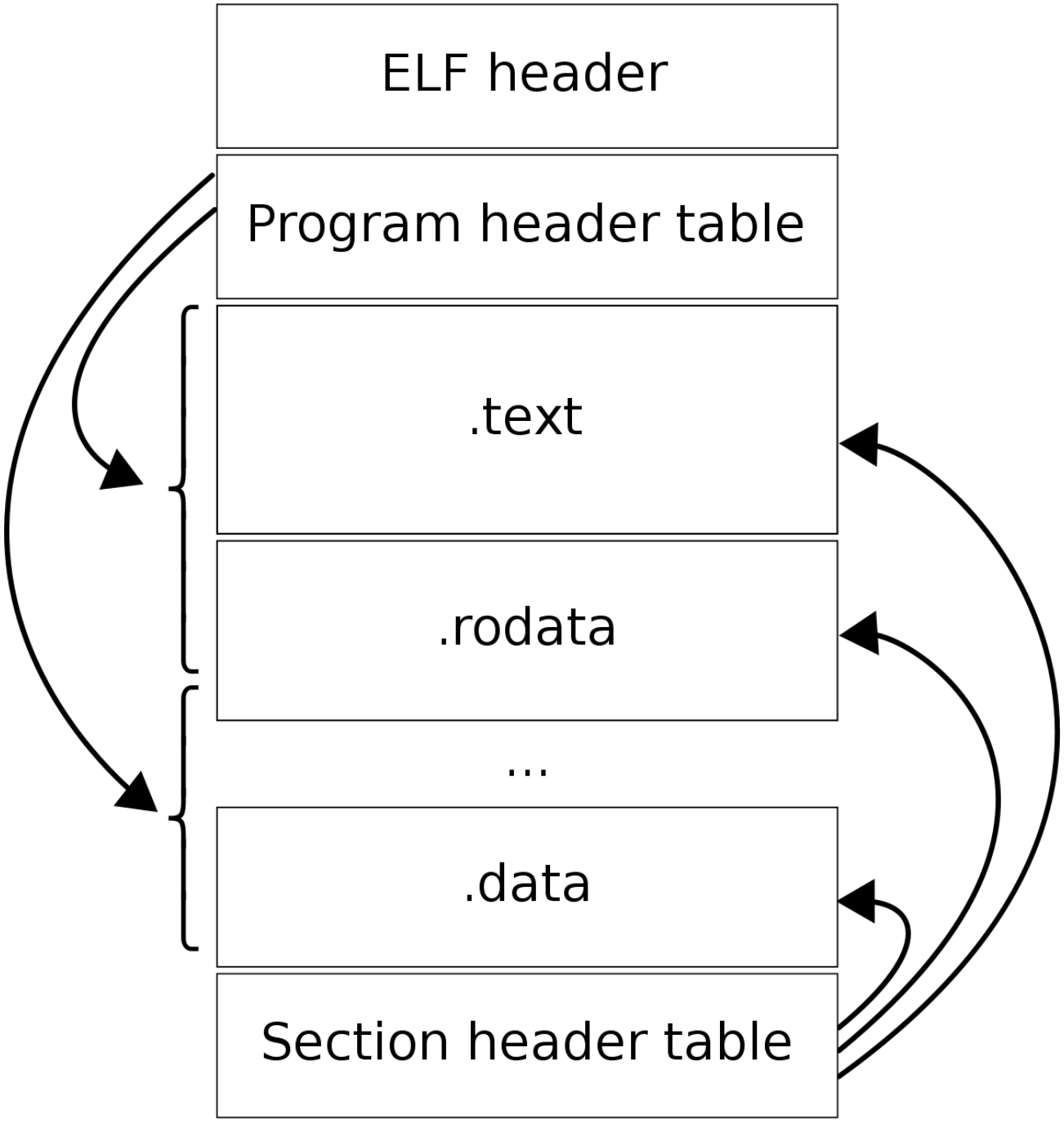
编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

阅读之前的说明

先说明，本篇很长，也很枯燥，若不是绝对的技术偏执狂是看不下去的。将通过一段简单代码去跟踪编译成ELF格式后的内容。看看 `ELF` 究竟长了怎样的一副花花肠子，用 `readelf` 命令去窥视ELF的全貌，最后用 `objdump` 命令反汇编 `ELF` 。找到了大家熟悉 `main` 函数。 开始之前先说结论: ELF 分四块，其中三块是描述信息(也叫头信息)，另一块是内容，放的是所有段/区的内容。

- 1。 ELF头定义全局性信息
- 2。 Segment(段)头，内容描述段的名字，开始位置，类型，偏移，大小及每段由哪些区组成。
- 3。 内容区，ELF有两个重要概念 `Segment` (段) 和 `Section` (区)，段比区大，二者之间关系如下:
  - 每个 `Segment` 可以包含多个 `Section`
  - 每个 `Section` 可以属于多个 `Segment`
  - `Segment` 之间可以有重合的部分
  - 拿大家熟知的 `.text` ， `.data` ， `。bss` 举例，它们都叫区，但它们又属于 `LOAD` 段。
- 4。 Section(区)头，内容描述区的名字，开始位置，类型，偏移，大小等信息
- ELF一体两面，面对不同的场景扮演不同的角色，这是理解ELF的关键，链接器只关注1，3(区)，4 的内容，加载器只关注1，2，3(段)的内容
- 鸿蒙对 `EFL` 的定义在 `kernel\extended\dynload\include\los_ld_elf_pri.h` 文件中



示例代码

在windows目录 `E:\harmony\docker\case_code_100` 下创建 main。c文件，如下:

```
#include <stdio.h>
void say_hello(char *who)
{
   printf("hello ,  %s!\n" ,  who);
}
char *my_name = "harmony os";

int main()
{
```

```
    say_hello(my_name);
    return 0;
}
```

因在

v50。xx (编译环境篇) | docker编译鸿蒙真的很香

篇中已做好了环境映射，所以文件会同时出现在docker中。编译生成`ELF`->运行->`readelf -h`查看`app`头部信息。

```
root@5e3abe332c5a:/home/docker/case_code_100# ls
main.c
root@5e3abe332c5a:/home/docker/case_code_100# gcc -o app main.c
root@5e3abe332c5a:/home/docker/case_code_100# ls
app  main.c
root@5e3abe332c5a:/home/docker/case_code_100# ./app
hello ， harmony os!
```

名正才言顺

一下是关于ELF的所有中英名词对照。建议先仔细看一篇再看系列篇部分。

```
可执行可连接格式 : ELF(Executable and Linking Format)
ELF文件头:ELF header
基地址:base address
动态连接器: dynamic linker
动态连接: dynamic linking
全局偏移量表: got(global offset table)
进程链接表: plt(Procedure Linkage Table)
哈希表: hash table
初始化函数 : initialization function
连接编辑器 : link editor
目标文件 : object file
函数连接表 : procedure linkage table
程序头: program header
程序头表 : program header table
程序解析器 : program interpreter
重定位: relocation
共享目标 : shared object
区(节): section
区(节)头 : section header
区(节)表: section header table
段 : segment
字符串表 : string table
符号表: symbol table
终止函数 : termination function
```

ELF历史

- ELF(Executable and Linking Format)，即"可执行可连接格式"，最初由UNIX系统实验室(UNIX System Laboratories – USL)做为应用程序二进制接口(Application Binary Interface - ABI)的一部分而制定和发布。是鸿蒙的主要可执行文件格式。
- ELF的最大特点在于它有比较广泛的适用性，通用的二进制接口定义使之可以平滑地移植到多种不同的操作环境上。这样，不需要为每一种操作系统都定义一套不同的接口，因此减少了软件的重复编码与编译，加强了软件的可移植性。

ELF整体布局

ELF规范中把ELF文件宽泛地称为"目标文件 (object file)"，这与我们平时的理解不同。一般地，我们把经过编译但没有连接的文件(比如Unix/Linux上的。o文件)称为目标文件，而ELF文件仅指连接好的可执行文件；在ELF规范中，所有符合ELF格式规范的都称为ELF文件，也称为目标文件，这两个名字是相同的，而经过编译但没有连接的文件则称为"可重定位文件 (relocatable file)"或"待重定位文件 (relocatable file)"。本文采用与此规范相同的命名方式，所以当提到可重定位文件时，一般可以理解为惯常所说的目标文件；而提到目标文件时，即指各种类型的ELF文件。

ELF格式可以表达四种类型的二进制对象文件(object files):

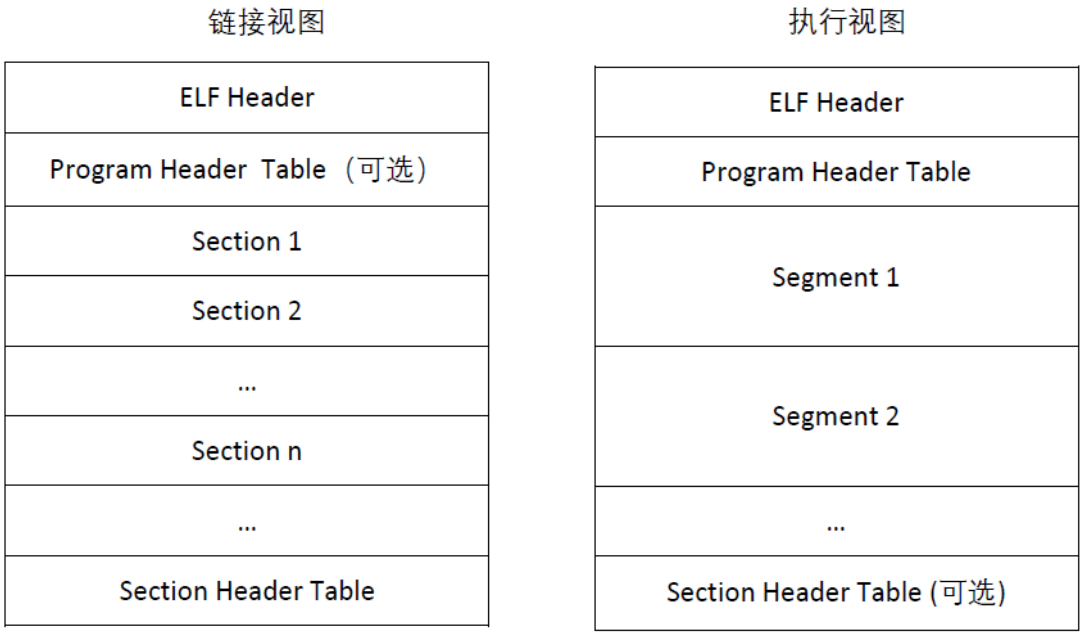
- 可重定位文件(relocatable file)，用于与其它目标文件进行连接以构建可执行文件或动态链接库。可重定位文件就是常说的目标文件，由源文件编译而成，但还没有连接成可执行文件。在UNIX系统下，一般有扩展名".o"。之所以称其为"可重定位"，是因为在这些文件中，如果引用到其它

目标文件或库文件中定义的符号（变量或者函数）的话，只是给出一个名字，这里还并不知道这个符号在哪里，其具体的地址是什么。需要在连接的过程中，把对这些外部符号的引用重新定位到其真正定义的位置上，所以称目标文件为"可重定位"或者"待重定位"的。

- 可执行文件(executable file)包含代码和数据，是可以直接运行的程序。其代码和数据都有固定的地址 （或相对于基地址的偏移 ），系统可根据这些地址信息把程序加载到内存执行。
- 共享目标文件(shared object file)，即动态连接库文件。它在以下两种情况下被使用:第一，在连接过程中与其它动态链接库或可重定位文件一起构建新的目标文件；第二，在可执行文件被加载的过程中，被动态链接到新的进程中，成为运行代码的一部分。包含了代码和数据，这些数据是在链接时被链接器（ld）和运行时动态链接器（ld.so.l、libc.so.l、ld-linux.so.l）使用的。
- 核心转储文件(core dump file，就是core dump文件)

```
可重定位文件用在编译和链接阶段。
可执行文件用在程序运行阶段。
共享库则同时用在编译链接和运行阶段，本篇 app 就是个 DYN，可直接运行。
    Type:                          DYN (Shared object file)
```

在不同阶段，我们可以用不同视角来理解 ELF 文件，整体布局如下图所示:



从上图可见，ELF格式文件整体可分为四大部分:

- ELF Header : 在文件的开始，描述整个文件的组织。即 readelf -h app 看到的内容
- Program Header Table : 告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件可以不需要这个表。表描述所有段(Segment)信息，即 readelf -l app 看到的前半部分内容。
- Segments :段(Segment)由若干区(Section)组成。是从加载器角度来描述 ELF 文件。加载器只关心 ELF header ， Program header table 和 Segment 这三部分内容。 在加载阶段可以忽略 section header table 来处理程序（所以很多加固手段删除了 section header table ）
- Sections : 是从链接器角度来描述 ELF 文件。 链接器只关心 ELF header ， Sections 以及 Section header table 这三部分内容。在链接阶段，可以忽略 program header table 来处理文件。
- Section Header Table :描述区(Section)信息的数组，每个元素对应一个区，通常包含在可重定位文件中，可执行文件中为可选(通常包含)即 readelf -S app 看到的内容
- 从图中可以看出 Segment : Section (M:N)是多对多的包含关系。 Segment 是由多个 Section 组成， Section 也能属于多个段。

ELF头信息

ELF 头部信息对应鸿蒙源码结构体为 LDElf32Ehdr ， 各字段含义已一一注解，很容易理解。

```
//kernel\extended\dynload\include\los_ld_elf_pri.h
/* Elf header */
#define LD_EI_NIDENT        16
typedef struct {
```

```
    UINT8      elfIdent[LD_EI_NIDENT]; /* Magic number and other info *///含前16个字节，又可细分成class、data、version等字段，具体含义不用太关心，
    UINT16     elfType;              /* Object file type *///表示具体ELF类型，可重定位文件/可执行文件/共享库文件
    UINT16     elfMachine;           /* Architecture *///表示cpu架构
    UINT32     elfVersion;           /* Object file version *///表示文件版本号
    UINT32     elfEntry;             /* Entry point virtual address *///对应`Entry point address`，程序入口函数地址，通过进程虚拟地址空间地址表达
    UINT32     elfPhoff;             /* Program header table file offset *///对应`Start of program headers`，表示program header table在文件内的偏移位置
    UINT32     elfShoff;             /* Section header table file offset *///对应`Start of section headers`，表示section header table在文件内的偏移位置
    UINT32     elfFlags;             /* Processor-specific flags *///表示与CPU处理器架构相关的信息
    UINT16     elfHeadSize;          /* ELF header size in bytes *///对应`Size of this header`，表示本ELF header自身的长度
    UINT16     elfPhEntSize;         /* Program header table entry size *///对应`Size of program headers`，表示program header table中每个元素的大小
    UINT16     elfPhNum;             /* Program header table entry count *///对应`Number of program headers`，表示program header table中元素个数
    UINT16     elfShEntSize;         /* Section header table entry size *///对应`Size of section headers`，表示section header table中每个元素的大小
    UINT16     elfShNum;             /* Section header table entry count *///对应`Number of section headers`，表示section header table中元素的个数
    UINT16     elfShStrIndex;        /* Section header string table index *///对应`Section header string table index`，表示描述各section字符名称的strin
} LDElf32Ehdr;
root@5e3abe332c5a:/home/docker/case_code_100# readelf -h app
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement，little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14784 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

解读

显示的信息，就是 ELF header 中描述的所有内容了。这个内容与结构体 `LDElf32Ehdr` 中的成员变量是一一对应的！ `Size of this header: 64 (bytes)` 也就是说：ELF header 部分的内容，一共是 64 个字节。64个字节码长啥样可以用命令 `od -Ax -t x1 -N 64 app`看，并对照结构体 `LDElf32Ehdr` 来理解。

```
root@5e3abe332c5a:/home/docker/case_code_100/51# od -Ax -t x1 -N 64 app
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
000010 03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00
000020 40 00 00 00 00 00 00 00 c0 39 00 00 00 00 00 00
000030 00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00
000040
```

简单解释一下命令的几个选项：

```
-Ax: 显示地址的时候，用十六进制来表示。如果使用 -Ad，意思就是用十进制来显示地址;
-t -x1: 显示字节码内容的时候，使用十六进制(x)，每次显示一个字节(1);
-N 64：只需要读取64个字节;
```

这里留意这几个内容，下面会说明，先记住。

```
Entry point address:               0x1060   //代码区 .text 起始位置，即程序运行开始位置
Size of program headers:           56 (bytes)//每个段头大小
Number of program headers:         13       //段数量
Size of section headers:           64 (bytes)//每个区头大小
Number of section headers:         31       //区数量
Section header string table index: 30       //字符串数组索引，该区记录所有区名称
```

段(Segment)头信息

段(Segment)信息对应鸿蒙源码结构体为 `LDElf32Phdr` ，

```
//kernel\extended\dynload\include\los_ld_elf_pri.h
/* Program Header */
typedef struct {
    UINT32 type;    /* Segment type */ //段类型
    UINT32 offset;  /* Segment file offset */  //此数据成员给出本段内容在文件中的位置，即段内容的开始位置相对于文件开头的偏移量。
    UINT32 vAddr;   /* Segment virtual address */ //此数据成员给出本段内容的开始位置在进程空间中的虚拟地址。
    UINT32 phyAddr;  /* Segment physical address */ //此数据成员给出本段内容的开始位置在进程空间中的物理地址。对于目前大多数现代操作系统而言，应用程
    UINT32 fileSize; /* Segment size in file */  //此数据成员给出本段内容在文件中的大小，单位是字节，可以是0。
    UINT32 memSize;  /* Segment size in memory */ //此数据成员给出本段内容在内容镜像中的大小，单位是字节，可以是0。
    UINT32 flags;   /* Segment flags */   //此数据成员给出了本段内容的属性。
    UINT32 align;   /* Segment alignment */  //对于可装载的段来说，其p_vaddr和p_offset的值至少要向内存页面大小对齐。
} LDElf32Phdr;
```

解读

用 `readelf -l` 查看 `app` 段头部表内容，先看命令返回的前半部分:

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -l app
Elf file type is DYN (Shared object file)
Entry point 0x1060
There are 13 program headers，starting at offset 64
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz             Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so。2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000618 0x0000000000000618  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000225 0x0000000000000225  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000000190 0x0000000000000190  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000260 0x0000000000000268  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8
  GNU_EH_FRAME   0x000000000000201c 0x000000000000201c 0x000000000000201c
                 0x000000000000004c 0x000000000000004c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000248 0x0000000000000248  R      0x1
```

数一下一共13个段，其实在ELF头信息也告诉了我们共13个段

```
Size of program headers:           56 (bytes)//每个段头大小
Number of program headers:         13        //段数量
```

仔细看下这些段的开始地址和大小，发现有些段是重叠的。那是因为一个区可以被多个段所拥有。例如: `0x2db8` 对应的 `.init_array` 区就被第四 `LOAD` 和 `GNU_RELRO` 两段所共有。

`PHDR` ，此类型header元素描述了program header table自身的信息。从这里的内容看出，示例程序的program header table在文件中的偏移(`Offset`)为 `0x40` ，即64号字节处。该段映射到进程空间的虚拟地址(`VirtAddr`)为 `0x40` 。 `PhysAddr` 暂时不用，其保持和 `VirtAddr` 一致。该段占用的文件大

小 FileSiz 为 0x2d8 。运行时占用进程空间内存大小 MemSiz 也为 0x2d8 。 Flags 标记表示该段的读写权限，这里 R 表示只读， Align 对齐为8，表明本段按8字节对齐。

INTERP ，此类型header元素描述了一个特殊内存段，该段内存记录了动态加载解析器的访问路径字符串。示例程序中，该段内存位于文件偏移 0x318 处，即紧跟program header table。映射的进程虚拟地址空间地址为 0x318 。文件长度和内存映射长度均为 0x1c ，即28个字符，具体内容为 /lib64/ld-linux-x86-64.so。2 。段属性为只读，并按字节对齐。

LOAD ，此类型 header 元素描述了可加载到进程空间的代码区或数据区:

- 其第二段包含了代码区，文件内偏移为0x1000，文件大小为0x225，映射到进程地址0x001000处，属性为只读可执行(RE)，段地址按0x1000(4K)边界对齐。
- 其第四段包含了数据区，文件内偏移为0x2db8，文件大小为0x260，映射到进程地址0x003db8处，属性为可读可写(RW)，段地址也按0x1000(4K)边界对齐。

DYNAMIC ，此类型 header 元素描述了动态加载段，其内部通常包含了一个名为 。dynamic 的动态加载区。这也是一个数组，每个元素描述了与动态加载相关的各方面信息，将在系列篇(动态加载篇)中介绍。该段是从文件偏移 0x2dc8 处开始，长度为 0x1f0 ，并映射到进程的 0x3dc8 。可见该段和上一个段 LOAD4 0x2db8 是有重叠的。

GNU_STACK ，可执行栈，即栈区，在加载段的过程中，当发现存在PT_GNU_STACK，也就是GNU_STACK segment 的存在，如果存在这个这个段的话，看这个段的 flags 是否有可执行权限，来设置对应的值。必须为RW方式。

再看命令返回内容的后半部分-段区映射关系

```
Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
  03     .init .plt .plt.got .plt.sec .text .fini
  04     .rodata .eh_frame_hdr .eh_frame
  05     .init_array .fini_array .dynamic .got .data .bss
  06     .dynamic
  07     .note.gnu.property
  08     .note.gnu.build-id .note.ABI-tag
  09     .note.gnu.property
  10     .eh_frame_hdr
  11
  12     .init_array .fini_array .dynamic .got
```

13个段和31个区的映射关系，右边其实不止31个区，是因为一个区可以共属于多个段，例如 .dynamic ， .interp ， .got Segment:Section(M:N)是多对多的包含关系.Segment是由多个Section组成，Section也能属于多个段。这个很重要，说第二遍了。

- INTERP 段只包含了 .interp 区
- LOAD2 段包含 .interp 、 .plt 、 .text 等区， .text 代码区位于这个段。 这个段是 'RE'属性，只读可执行的。
- LOAD4 包含 .dynamic 、 .data 、 .bss 等区， 数据区位于这个段。这个段是 'RW'属性，可读可写。 .data 、 .bss 都是数据区，有何区别呢？
- .data(ZI data) 它用来存放初始化了的(initailized)全局变量(global)和初始化了的静态变量(static)。
- .bss(RW data) 它用来存放未初始化的(uninitailized)全局变量(global)和未初始化的静态变量。
- DYNAMIC 段包含 .dynamic 区。

区表

区(section)头表信息对应鸿蒙源码结构体为 LDElf32Shdr ，

```
//kernel\extended\dynload\include\los_ld_elf_pri.h
/* Section header */
typedef struct {
    UINT32 shName;      /* Section name (string tbl index) *///表示每个区的名字
    UINT32 shType;      /* Section type *///表示每个区的功能
    UINT32 shFlags;     /* Section flags *///表示每个区的属性
    UINT32 shAddr;      /* Section virtual addr at execution *///表示每个区的进程映射地址
    UINT32 shOffset;    /* Section file offset *///表示文件内偏移
    UINT32 shSize;      /* Section size in bytes *///表示区的大小
    UINT32 shLink;      /* Link to another section *///Link和Info记录不同类型区的相关信息
    UINT32 shInfo;      /* Additional section information *///Link和Info记录不同类型区的相关信息
    UINT32 shAddrAlign; /* Section alignment *///表示区的对齐单位
    UINT32 shEntSize;   /* Entry size if section holds table *///表示区中每个元素的大小(如果该区为一个数组的话，否则该值为0)
```

```
} LDElf32Shdr;
```

示例程序共生成31个区。其实在头文件中也已经告诉我们了

```
Size of section headers:           64 (bytes)//每个区头大小
Number of section headers:         31       //区数量
```

通过 `readelf -S` 命令看看示例程序中 section header table的内容，如下所示。

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -S app
There are 31 section headers ,  starting at offset 0x39c0:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000         0     0     0
  [ 1] .interp           PROGBITS        0000000000000318  00000318
       000000000000001c  0000000000000000   A      0     0     1
  [ 2] .note.gnu.propert NOTE            0000000000000338  00000338
       0000000000000020  0000000000000000   A      0     0     8
  [ 3] .note.gnu.build-i NOTE            0000000000000358  00000358
       0000000000000024  0000000000000000   A      0     0     4
  [ 4] .note.ABI-tag     NOTE            000000000000037c  0000037c
       0000000000000020  0000000000000000   A      0     0     4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0  000003a0
       0000000000000024  0000000000000000   A      6     0     8
  [ 6] .dynsym           DYNSYM          00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A      7     1     8
  [ 7] .dynstr           STRTAB          0000000000000470  00000470
       0000000000000084  0000000000000000   A      0     0     1
  [ 8] .gnu.version      VERSYM          00000000000004f4  000004f4
       000000000000000e  0000000000000002   A      6     0     2
  [ 9] .gnu.version_r    VERNEED         0000000000000508  00000508
       0000000000000020  0000000000000000   A      7     1     8
  [10] .rela.dyn         RELA            0000000000000528  00000528
       00000000000000d8  0000000000000018   A      6     0     8
  [11] .rela.plt         RELA            0000000000000600  00000600
       0000000000000018  0000000000000018  AI      6    24     8
  [12] .init             PROGBITS        0000000000001000  00001000
       000000000000001b  0000000000000000  AX      0     0     4
  [13] .plt              PROGBITS        0000000000001020  00001020
       0000000000000020  0000000000000010  AX      0     0     16
  [14] .plt.got          PROGBITS        0000000000001040  00001040
       0000000000000010  0000000000000010  AX      0     0     16
  [15] .plt.sec          PROGBITS        0000000000001050  00001050
       0000000000000010  0000000000000010  AX      0     0     16
  [16] .text             PROGBITS        0000000000001060  00001060
       00000000000001b5  0000000000000000  AX      0     0     16
  [17] .fini             PROGBITS        0000000000001218  00001218
       000000000000000d  0000000000000000  AX      0     0     4
  [18] .rodata           PROGBITS        0000000000002000  00002000
       000000000000001b  0000000000000000   A      0     0     4
  [19] .eh_frame_hdr     PROGBITS        000000000000201c  0000201c
       000000000000004c  0000000000000000   A      0     0     4
  [20] .eh_frame         PROGBITS        0000000000002068  00002068
       0000000000000128  0000000000000000   A      0     0     8
  [21] .init_array       INIT_ARRAY      0000000000003db8  00002db8
       0000000000000008  0000000000000008  WA      0     0     8
  [22] .fini_array       FINI_ARRAY      0000000000003dc0  00002dc0
       0000000000000008  0000000000000008  WA      0     0     8
  [23] .dynamic          DYNAMIC         0000000000003dc8  00002dc8
       00000000000001f0  0000000000000010  WA      7     0     8
  [24] .got              PROGBITS        0000000000003fb8  00002fb8
       0000000000000048  0000000000000008  WA      0     0     8
  [25] .data             PROGBITS        0000000000004000  00003000
       0000000000000018  0000000000000000  WA      0     0     8
  [26] .bss              NOBITS          0000000000004018  00003018
```

```
     0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00003018
     000000000000002a  0000000000000001  MS       0     0     1
  [28] .symtab           SYMTAB           0000000000000000  00003048
     0000000000000648  0000000000000018       29    46     8
  [29] .strtab           STRTAB           0000000000000000  00003690
     0000000000000216  0000000000000000         0     0     1
  [30] .shstrtab         STRTAB           0000000000000000  000038a6
     000000000000011a  0000000000000000         0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

String Table

在 ELF header 的最后 `2` 个字节是 `0x1e` `0x00`，即 `30`。 它对应结构体中的成员 `elfShStrIndex`，意思是这个 `ELF` 文件中，字符串表是一个普通的 `Section`，在这个 `Section` 中，存储了 `ELF` 文件中使用到的所有的字符串。 我们使用`readelf -x`读出下标30区的数据:

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -x 30 app

Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strtab
  0x00000010 002e7368 73747274 6162002e 696e7465 ..shstrtab..inte
  0x00000020 7270002e 6e6f7465 2e676e75 2e70726f rp..note.gnu.pro
  0x00000030 70657274 79002e6e 6f74652e 676e752e perty..note.gnu.
  0x00000040 6275696c 642d6964 002e6e6f 74652e41 build-id..note.A
  0x00000050 42492d74 6167002e 676e752e 68617368 BI-tag..gnu.hash
  0x00000060 002e6479 6e73796d 002e6479 6e737472 ..dynsym..dynstr
  0x00000070 002e676e 752e7665 7273696f 6e002e67 ..gnu.version..g
  0x00000080 6e752e76 65727369 6f6e5f72 002e7265 nu.version_r..re
  0x00000090 6c612e64 796e002e 72656c61 2e706c74 la.dyn..rela.plt
  0x000000a0 002e696e 6974002e 706c742e 676f7400 ..init..plt.got.
  0x000000b0 2e706c74 2e736563 002e7465 7874002e .plt.sec..text..
  0x000000c0 66696e69 002e726f 64617461 002e6568 fini..rodata..eh
  0x000000d0 5f667261 6d655f68 6472002e 65685f66 _frame_hdr..eh_f
  0x000000e0 72616d65 002e696e 69745f61 72726179 rame..init_array
  0x000000f0 002e6669 6e695f61 72726179 002e6479 ..fini_array..dy
  0x00000100 6e616d69 63002e64 61746100 2e627373 namic..data..bss
  0x00000110 002e636f 6d6d656e 7400              ..comment.
```

可以发现，这里其实是一堆字符串，这些字符串对应的就是各个区的名字。因此section header table中每个元素的Name字段其实是这个string table的索引。为节省空间而做的设计，再回头看看ELF header中的 `elfShStrIndex`，

```
Section header string table index: 30 //字符串数组索引，该区记录所有区名称
```

它的值正好就是30，指向了当前的string table。

符号表 Symbol Table

Section Header Table中，还有一类 `SYMTAB` (DYNSYM)区，该区叫符号表。符号表中的每个元素对应一个符号，记录了每个符号对应的实际数值信息，通常用在重定位过程中或问题定位过程中，进程执行阶段并不加载符号表。符号表对应鸿蒙源码结构体为 `LDElf32Sym`。
//kernel\extended\dynload\include\los_ld_elf_pri.h

```
/* Symbol table */
typedef struct {
    UINT32 stName;  /* Symbol table name (string tbl index) *///表示符号对应的源码字符串，为对应String Table中的索引
    UINT32 stValue; /* Symbol table value *///表示符号对应的数值
    UINT32 stSize;  /* Symbol table size *///表示符号对应数值的空间占用大小
    UINT8 stInfo;   /* Symbol table type and binding *///表示符号的相关信息 如符号类型(变量符号、函数符号)
    UINT8 stOther;  /* Symbol table visibility */
    UINT16 stShndx; /* Section table index *///表示与该符号相关的区的索引，例如函数符号与对应的代码区相关
} LDElf32Sym;
```

用 `readelf -s` 读出示例程序中的符号表，如下所示

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -s app

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 67 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000318     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000338     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000358     0 SECTION LOCAL  DEFAULT    3
     4: 000000000000037c     0 SECTION LOCAL  DEFAULT    4
     5: 00000000000003a0     0 SECTION LOCAL  DEFAULT    5
     6: 00000000000003c8     0 SECTION LOCAL  DEFAULT    6
     7: 0000000000000470     0 SECTION LOCAL  DEFAULT    7
     8: 00000000000004f4     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000000508     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000000528     0 SECTION LOCAL  DEFAULT   10
    11: 0000000000000600     0 SECTION LOCAL  DEFAULT   11
    12: 0000000000001000     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000001020     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000001040     0 SECTION LOCAL  DEFAULT   14
    15: 0000000000001050     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000001060     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000001218     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000002000     0 SECTION LOCAL  DEFAULT   18
    19: 000000000000201c     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000002068     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000003db8     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000003dc0     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000003dc8     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000003fb8     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000004000     0 SECTION LOCAL  DEFAULT   25
    26: 0000000000004018     0 SECTION LOCAL  DEFAULT   26
    27: 0000000000000000     0 SECTION LOCAL  DEFAULT   27
    28: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    29: 0000000000001090     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
    30: 00000000000010c0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
    31: 0000000000001100     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
    32: 0000000000004018     1 OBJECT  LOCAL  DEFAULT   26 completed.8060
    33: 0000000000003dc0     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtors_aux_fin
    34: 0000000000001140     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    35: 0000000000003db8     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_init_array_
    36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
    37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    38: 000000000000218c     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
    39: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    40: 0000000000003dc0     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_end
    41: 0000000000003dc8     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
    42: 0000000000003db8     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_start
    43: 000000000000201c     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
    44: 0000000000003fb8     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    45: 0000000000001000     0 FUNC    LOCAL  DEFAULT   12 _init
    46: 0000000000001210     5 FUNC    GLOBAL DEFAULT   16 __libc_csu_fini
    47: 0000000000004010     8 OBJECT  GLOBAL DEFAULT   25 my_name
    48: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
    49: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    50: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    51: 0000000000001218     0 FUNC    GLOBAL HIDDEN    17 _fini
    52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
```

```
    54: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    55: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    56: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
    57: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    58: 00000000000011a0   101 FUNC    GLOBAL DEFAULT   16 __libc_csu_init
    59: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   26 _end
    60: 0000000000001060    47 FUNC    GLOBAL DEFAULT   16 _start
    61: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    62: 0000000000001174    30 FUNC    GLOBAL DEFAULT   16 main
    63: 0000000000001149    43 FUNC    GLOBAL DEFAULT   16 say_hello
    64: 0000000000004018     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    66: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2
```

在最后位置找到了亲切的老朋友 `main` 和 `say_hello`

```
    62: 0000000000001174    30 FUNC    GLOBAL DEFAULT   16 main
    63: 0000000000001149    43 FUNC    GLOBAL DEFAULT   16 say_hello
```

`main` 函数符号对应的数值为 `0x1174` ，其类型为 `FUNC` ，大小为30字节，对应的代码区索引为16。 `say_hello` 函数符号对应数值为 `0x1149` ，其类型为 `FUNC` ，大小为43字节，对应的代码区索引同为16。 Section Header Table:

```
  [16] .text           PROGBITS        0000000000001060  00001060
       00000000000001b5  0000000000000000  AX       0     0     16
```

反汇编代码区

在理解了 `String Table` 和 `Symbol Table` 的作用后，通过 `objdump` 反汇编来理解一下 `.text` 代码区:

```
root@5e3abe332c5a:/home/docker/case_code_100# objdump -j .text -l -C -S app

0000000000001149 <say_hello>:
say_hello():
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp , %rbp
    1151:       48 83 ec 10             sub    $0x10 , %rsp
    1155:       48 89 7d f8             mov    %rdi , -0x8(%rbp)
    1159:       48 8b 45 f8             mov    -0x8(%rbp) , %rax
    115d:       48 89 c6                mov    %rax , %rsi
    1160:       48 8d 3d 9d 0e 00 00    lea    0xe9d(%rip) , %rdi        # 2004 <_IO_stdin_used+0x4>
    1167:       b8 00 00 00 00          mov    $0x0 , %eax
    116c:       e8 df fe ff ff          callq  1050 <printf@plt>
    1171:       90                      nop
    1172:       c9                      leaveq
    1173:       c3                      retq

0000000000001174 <main>:
main():
    1174:       f3 0f 1e fa             endbr64
    1178:       55                      push   %rbp
    1179:       48 89 e5                mov    %rsp , %rbp
    117c:       48 8b 05 8d 2e 00 00    mov    0x2e8d(%rip) , %rax        # 4010 <my_name>
    1183:       48 89 c7                mov    %rax , %rdi
    1186:       e8 be ff ff ff          callq  1149 <say_hello>
    118b:       b8 00 00 00 00          mov    $0x0 , %eax
    1190:       5d                      pop    %rbp
    1191:       c3                      retq
    1192:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax , %rax , 1)
    1199:       00 00 00
    119c:       0f 1f 40 00             nopl   0x0(%rax)
```

`0x1149` `0x1174` 正是 `say_hello` ， `main` 函数的入口地址。并看到了激动人心的指令

```
1186:       e8 be ff ff ff          callq  1149 <say_hello>
```

很佩服你还能看到这里，牛逼，牛逼! 看了这么久还记得开头的C代码的样子吗？ 再看一遍：）

```
#include <stdio.h>
void say_hello(char *who)
{
   printf("hello ， %s!\n" ， who);
}
char *my_name = "harmony os";
int main()
{
   say_hello(my_name);
   return 0;
}
root@5e3abe332c5a:/home/docker/case_code_100# ./app
hello ， harmony os!
```

但是!!! 晕，怎么还有but，西卡西...，上面请大家记住的还有一个地方没说到

```
Entry point address:               0x1060   //代码区 .text 起始位置，即程序运行开始位置
```

它的地址并不是main函数位置`0x1174` ，是`0x1060`!而且代码区的开始位置是`0x1060`没错的。

```
 [16] .text           PROGBITS        0000000000001060  00001060
      00000000000001b5  0000000000000000  AX       0     0     16
```

难度`main`不是入口地址？ 那`0x1060`上放的是何方神圣，再查符号表发现是

```
   60: 0000000000001060    47 FUNC    GLOBAL DEFAULT   16 _start
```

从反汇编堆中找到 `_start`

```
0000000000001060 <_start>:
_start():
   1060:      f3 0f 1e fa           endbr64
   1064:      31 ed                 xor    %ebp , %ebp
   1066:      49 89 d1              mov    %rdx , %r9
   1069:      5e                    pop    %rsi
   106a:      48 89 e2              mov    %rsp , %rdx
   106d:      48 83 e4 f0           and    $0xfffffffffffffff0 , %rsp
   1071:      50                    push   %rax
   1072:      54                    push   %rsp
   1073:      4c 8d 05 96 01 00 00    lea    0x196(%rip) , %r8       # 1210 <__libc_csu_fini>
   107a:      48 8d 0d 1f 01 00 00    lea    0x11f(%rip) , %rcx       # 11a0 <__libc_csu_init>
   1081:      48 8d 3d ec 00 00 00    lea    0xec(%rip) , %rdi       # 1174 <main>
   1088:      ff 15 52 2f 00 00       callq  *0x2f52(%rip)       # 3fe0 <__libc_start_main@GLIBC_2.2.5>
   108e:      f4                    hlt
   108f:      90                    nop
```

这才看到了`0x1174`的`main`函数。所以真正的说法是:

- 从内核动态加载的视角看，程序运行首个函数并不是`main` ，而是`_start` 。
- 但从应用程序开发者视角看， `main`就是启动函数。

百文说内核 | 抓住主脉络

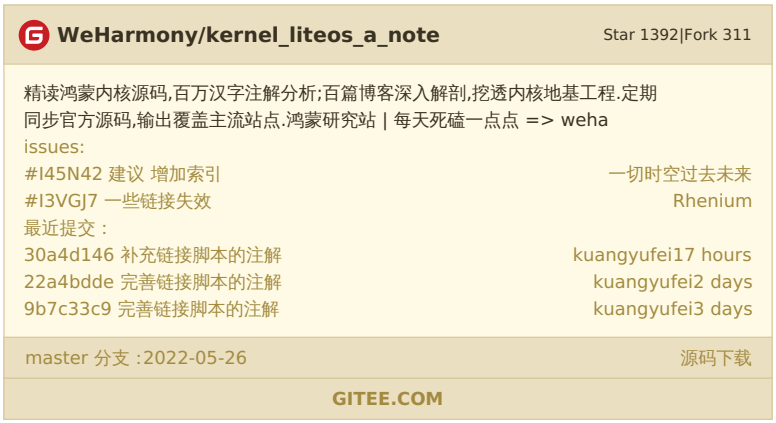
- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

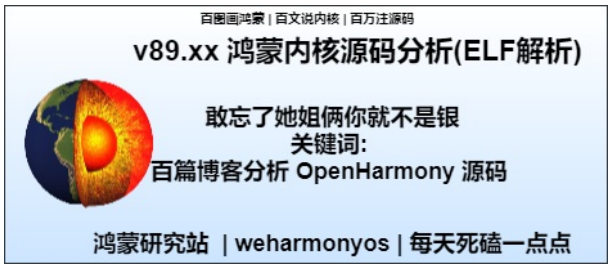


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

89_ELF解析篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

ELF，它实在是太重要了，内核加载的就是它，不说清楚它怎么去说清楚应用程序运行的过程呢。看到下面这一坨一坨的，除了 `.text` ， `.bss` ， `.data` 听过见过外，其他的咱也没啥交情。

```
01    .interp
02    .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
03    .init .plt .plt.got .plt.sec .text .fini
04    .rodata .eh_frame_hdr .eh_frame
05    .init_array .fini_array .dynamic .got .data .bss
```

系列篇要全说清楚也不太可能，可以去看 ELF官方文档(106页) ，本篇试图与它多些交情，混个脸熟，方便后续推进。从两个命令入手。 `readelf -S app` 和 `readelf -s app` 这俩宝贝长的很像，但仔细看中间参数是大S和小s，说到大S小s又有点意思了，这姐妹俩上了点年纪的码农都应该不陌生，据说是性格完全不同。个人喜欢大的，甜美安静，小的太聒噪，受不了，码农最需要安静了。

readelf -S app

先看老大是干啥的，其实她是她们家老二，上面还有个姐姐，没啥存在感，不管她了。

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -h
  ...
 -S --section-headers   Display the sections' header
    --sections          An alias for --section-headers
 -s --syms              Display the symbol table
    --symbols           An alias for --syms
```

显示所有区头信息 | sections' header

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -S app
There are 31 section headers， starting at offset 0x39c0:

Section Headers:
```

```
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000         0    0     0
  [ 1] .interp           PROGBITS        0000000000000318  00000318
       000000000000001c  0000000000000000   A       0    0     1
  [ 2] .note.gnu.propert NOTE            0000000000000338  00000338
       0000000000000020  0000000000000000   A       0    0     8
  [ 3] .note.gnu.build-i NOTE            0000000000000358  00000358
       0000000000000024  0000000000000000   A       0    0     4
  [ 4] .note.ABI-tag     NOTE            000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0    0     4
  [ 5] .gnu.hash         GNU_HASH        00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       6    0     8
  [ 6] .dynsym           DYNSYM          00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A       7    1     8
  [ 7] .dynstr           STRTAB          0000000000000470  00000470
       0000000000000084  0000000000000000   A       0    0     1
  [ 8] .gnu.version      VERSYM          00000000000004f4  000004f4
       000000000000000e  0000000000000002   A       6    0     2
  [ 9] .gnu.version_r    VERNEED         0000000000000508  00000508
       0000000000000020  0000000000000000   A       7    1     8
  [10] .rela.dyn         RELA            0000000000000528  00000528
       00000000000000d8  0000000000000018   A       6    0     8
  [11] .rela.plt         RELA            0000000000000600  00000600
       0000000000000018  0000000000000018  AI       6   24     8
  [12] .init             PROGBITS        0000000000001000  00001000
       000000000000001b  0000000000000000  AX       0    0     4
  [13] .plt              PROGBITS        0000000000001020  00001020
       0000000000000020  0000000000000010  AX       0    0     16
  [14] .plt.got          PROGBITS        0000000000001040  00001040
       0000000000000010  0000000000000010  AX       0    0     16
  [15] .plt.sec          PROGBITS        0000000000001050  00001050
       0000000000000010  0000000000000010  AX       0    0     16
  [16] .text             PROGBITS        0000000000001060  00001060
       00000000000001b5  0000000000000000  AX       0    0     16
  [17] .fini             PROGBITS        0000000000001218  00001218
       000000000000000d  0000000000000000  AX       0    0     4
  [18] .rodata           PROGBITS        0000000000002000  00002000
       000000000000001b  0000000000000000   A       0    0     4
  [19] .eh_frame_hdr     PROGBITS        000000000000201c  0000201c
       000000000000004c  0000000000000000   A       0    0     4
  [20] .eh_frame         PROGBITS        0000000000002068  00002068
       0000000000000128  0000000000000000   A       0    0     8
  [21] .init_array       INIT_ARRAY      0000000000003db8  00002db8
       0000000000000008  0000000000000008  WA       0    0     8
  [22] .fini_array       FINI_ARRAY      0000000000003dc0  00002dc0
       0000000000000008  0000000000000008  WA       0    0     8
  [23] .dynamic          DYNAMIC         0000000000003dc8  00002dc8
       00000000000001f0  0000000000000010  WA       7    0     8
  [24] .got              PROGBITS        0000000000003fb8  00002fb8
       0000000000000048  0000000000000008  WA       0    0     8
  [25] .data             PROGBITS        0000000000004000  00003000
       0000000000000018  0000000000000000  WA       0    0     8
  [26] .bss              NOBITS          0000000000004018  00003018
       0000000000000008  0000000000000000  WA       0    0     1
  [27] .comment          PROGBITS        0000000000000000  00003018
       000000000000002a  0000000000000001  MS       0    0     1
  [28] .symtab           SYMTAB          0000000000000000  00003048
       0000000000000648  0000000000000018          29   46     8
  [29] .strtab           STRTAB          0000000000000000  00003690
       0000000000000216  0000000000000000           0    0     1
  [30] .shstrtab         STRTAB          0000000000000000  000038a6
       000000000000011a  0000000000000000           0    0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

解读

命令结果主要三个部分，区名称(Section Head Name)，区类型 (Section Head Type) 和区标签(Section Head Flag)

- `Name` 部分 出现了一些熟悉的内容 `.bss` ， `.text` ，但更多是看不懂的 `.fini` ， `.plt` ， `.relname`
- `Type` 部分 就有更多看不懂的 `NULL` ， `PROGBITS` ， `INIT_ARRAY` 等等。
- `Flag` 部分 好像也似懂非懂。

一个区只属于一个类型，具有排它性，跟男人，女人一样。 但身上可以贴多个标签。可以是码农，可以是高富帅，可以是脱发男，不对!!! 码农你还想是高富帅，想多了。脱发才是你的标配。例如:

- 代码区(`.text`)属于 `PROGBITS` 类型被贴上了 `AX` (`alloc` + `execute`)标签。原来代码区可以被CPU取指运行是因为在ELF中被贴上了可运行标签。但注意 `.text` 是只读不可写，因为它身上没有 `write` 标签。
- 再看熟悉两个数据区 `.bss` 和 `.data` ，它们都有 `WA` (`write` + `alloc`)标签，可写+运行过程中需要占用内存，但二者区别是类型的不同， `.bss` 是 `NOBITS` 类型 `.data` 是 `PROGBITS` 类型

区名称 | Section Head Name

简称: `SHN`

在ELF文件中有一些特定的区是预定义好的，其内容是指令代码或者控制信息。这些区专门为操作系统使用，对于不同的操作系统，这些区的类型和属性有所不同。

在构建可执行程序时，链接器(linker)可能需要把一些独立的目标文件和库文件链接在一起，在这个过程中，链接器要解析各个文件中的相互引用，调整某些目标文件中的绝对引用，并重定位指令码。

每种操作系统都有自己的一套链接模型，但总的来说，不外乎静态和动态两类：

- 静态链接：所有的目标文件和动态链接库被静态地绑定在一起，所有的符号都被解析出来。所创建的目标文件是完整的，运行时不依赖于任何外部的库。
- 动态链接：所有的目标文件，系统共享资源以及共享库以动态的形式链接在一起，外部库的内容没有完整地拷贝进来。如果创建的是可执行文件的话，程序在运行的时候，在构建时所依赖的那些库必须在系统中能找到，把它们一并装载之后，程序才能运行起来。运行期间如何解析那些动态链接进来的符号引用，不同的系统有各自不同的方式。

根据区功能划分:

- 有些区包含调试信息，比如.debug和.line区。
- 有些区包含程序控制信息，比如.bss，.data，.data1，.rodata和.rodata1这些区。
- 还有一些区含有程序或控制信息，这些区由系统使用，有指定的类型和属性。它们中的大多数都将用于链接过程。动态链接过程所需要的信息由.dynsym，.dynstr，.interp，.hash，.dynamic，.rel，.rela，.got，.plt等区提供。其中有些区(比如.plt和.got)的内容依处理器而不同，但它们都支持同样的链接模型。

以点号"."为前缀的区名字是为系统保留的。应用程序也可以构造自己的区，但最好不要取与上述系统已定义的区相同的名字，也不要取以点号开头的名字，以避免潜在的冲突，注意，目标文件中区的名字并不具有唯一性，可以存在多个相同名字的区。具体如下:

```
区名            描述说明
.bss          本区中包含目标文件中未初始化的全局变量。一般情况下，可执行程序在开始运行的时候，系统会把这一区内容清零。但是，
              在运行期间的bss区是由系统初始化而成的，在目标文件中.bss区并不包含任何内容，其长度为0，所以它的区类型为NOBITS。
.comment        本区包含版本控制信息。
.data         这个区不陌生，用于存放程序中被初始化过的全局变量。在目标文件中，它们是占用实际的存储空间的，与.bss区不同。
.debug          本区中含有调试信息，内容格式没有统一规定。所有以".debug"为前缀的区名字都是保留的。
.dynamic        本区包含动态链接信息，并且可能有SHF_ALLOC和SHF_WRITE等属性。是否具有SHF_WRITE属性取决于操作系统和处理器。
.dynstr         本区含有用于动态链接的字符串，一般是那些与符号表相关的名字。具有SHF_ALLOC属性
.dynsym         本区含有动态链接符号表。具有SHF_ALLOC属性，因为它需要在运行时被加载
.got          本区包含全局偏移量表(global offset table)。
.hash           本区包含一张符号哈希表。
.init         本区包含进程初始化时要执行的程序指令，当程序开始运行时，系统会在进入主函数之前执行这一区中的代码。
.fini         程序终止代码区，当程序结束运行时，系统会在最后执行这一区中的代码。
.interp         本区含有ELF程序解析器的路径名。如果本区被包含在某个可装载的区中，那么本区的属性中应置SHF_ALLOC标志位，否则不置此标志。
.line         本区也是一个用于调试的区，它包含那些调试符号的行号，为程序指令码与源文件的行号建立起联系。其内容格式没有统一规定。
.note           本区所包含的信息在第2章"注释区(note section)"部分描述。
.plt          本区包含函数链接表。动态链接时使用的过程链接表(precedure linkage table)
.relname        同下
.relaname       这两个区含有重定位信息。如果本区被包含在某个可装载的区中，那么本区的属性中应置SHF_ALLOC标志位，否则不置此标志。注意，这两个区的名字
              对哪一区做重定位就把"name"换成哪一区的名字。比如，.text区的重定位区的名字将是.rel.text或.rela.text。
.rodata         同下
.rodata1        本区包含程序中的只读数据，在程序装载时，它们一般会被装入进程空间中那些只读的区中去。
```

```
.shstrtab      本区是"区名字表"，含有所有其它区的名字，如 `.data`，`.bss`，`.text`...
.strtab        本区用于存放字符串，主要是那些符号表项的名字。如果一个目标文件有一个可装载的区，并且其中含有符号表，存储的是变量名，函数名等。
.symtab        本区用于存放符号表。如果一个目标文件有一个可载入的区，并且其中含有符号表，那么本区的属性中应该有SHF_ALLOC。
.text          本区包含程序指令代码。
.rel.text      重定位的地方在.text段内，以offset指定具体要定位位置。在连接时候由连接器完成。注意比较。text段前后变化。
               指的是比较.o文件和最终的执行文件（或者动态库文件）。就是重定位前后比较，以上是说明了具体比较对象而已。
               一般由编译器编译产生，存在于obj文件内。
.rel.dyn       重定位的地方在.got段内。主要是针对外部数据变量符号。例如全局数据。重定位在程序运行时定位，
               一般是在.init段内。定位过程：获得符号对应value后，根据rel.dyn表中对应的offset，修改.got表对应位置的
               value。另外，.rel.dyn 含义是指和dyn有关，一般是指在程序运行时候，动态加载。
               区别于rel.plt，rel.plt是指和plt相关，具体是指在某个函数被调用时候加载。
               一般由连接器产生，存在于可执行文件或者动态库文件。
.rel.plt       重定位的地方在.got.plt段内（注意也是。got内，具体区分而已）。 主要是针对外部函数符号。一般是函数首次
               被调用时候重定位。可看汇编，理解其首次访问是如何重定位的，实际很简单，就是初次重定位函数地址，
               然后把最终函数地址放到.got.plt内，以后读取该.got.plt就直接得到最终函数地址(参考过程说明)。
               所有外部函数调用都是经过一个对应桩函数，这些桩函数都在.plt段内。
               一般由连接器产生，存在于可执行文件或者动态库文件。
               借助这两个辅助段可以动态修改对应.got和.got.plt段，从而实现运行时重定位。
.rel.data      常量区重定位信息
.rel.rodata    数据段重定位信息
```

详细解读

- `.text` 通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码区为可写，即允许修改程序。在代码区中，也有可能包含一些只读的常数变量，例如字符串常量等。
- `.rodata` 和 `.data` 区类型一样但标签有别，`.rodata` 只有 `A` 标，是个只读区，比如字符串常量，全局const变量和#define定义的常量，又称为常量区 但是注意，并不是所有的常量都放在rodata区的，其特殊情况如下：
  - 有些立即数与指令编译在一起直接放在代码区。
  - 对于字符串常量，编译器会去掉重复的常量，让程序的每个字符串常量只有一份
  - 有些系统中rodata区是多个进程共享的，目的是为了提高空间的利用率
- `.bss` 和 `.data` 是标签一样但类型有别，`.bss` 区属于静态内存分配。通常是指用来存放程序中未初始化的全局变量和未初始化的局部静态变量。未初始化的全局变量和未初始化的局部静态变量默认值是0，本来这些变量也可以放到 `data` 区的，但是因为它们都是0，所以它们在 `data` 区分配空间并且存放数据0是没有必要的。在程序运行时，才会给BSS区里面的变量分配内存空间。在目标文件(*.o)和可执行文件中，`.bss` 只是为未初始化的全局变量和未初始化的局部静态变量预留位置而已，它并没有内容，所以它不占据空间。
- `.data` 通常是指用来存放程序中已初始化的全局变量和已初始化的静态变量的一块内存区域，属于静态内存分配。

区类型 | Section Head Type

简称: `SHT`

```
SHT_NULL       本区头是一个无效的（非活动的）区头，它也没有对应的区。本区头中的其它成员的值也都是没有意义的。
SHT_PROGBITS   本区所含有的信息是由程序定义的，本区内容的格式和含义都由程序来决定。
SHT_SYMTAB     同DYNSYM
SHT_DYNSYM     这两类区都含有符号表。目前，目标文件中最多只能各包含一个这两种区，但这种限制以后可能会取消。
               一般来说，SYMTAB提供的符号用于在创建目标文件的时候编辑链接，在运行期间也有可能会用于动态链接。
               SYMTAB包含完整的符号表，它往往会包含很多在运行期间(动态链接)用不到的符号。所以，一个目标文件
               可以再有一个DYNSYM区，它含有一个较小的符号表，专门用于动态链接。
SHT_STRTAB     本区是字符串表。目标文件中可以包含多个字符串表区。
SHT_RELA       本区是一个重定位区，含有带明确加数(addend)的重定位项，对于32位类型的目标文件来说，
               这个加数就是Elf32_Rela。一个目标文件可能含有多个重定位区。
SHT_HASH       本区包含一张哈希表。所有参与动态链接的目标文件都必须要包含一个符号哈希表。目前，一个目标文件中最多只能有一个哈希表，
               但这一限制以后可能会取消。
SHT_DYNAMIC    本区包含的是动态链接信息。目前，一个目标文件中最多只能有一个DYNAMIC区，
               但这一限制以后可能会取消。
SHT_NOTE       本区包含的信息用于以某种方式来标记本文件。
SHT_NOBITS     这一区的内容是空的，区并不占用实际的空间。只代表一个逻辑上的位置概念，并不代表实际的内容。
SHT_REL        本区是一个重定位区，含有带明确加数的重定位项，对于32位类型的目标文件来说，这个加数就是Elf32_Rel。一个目标文件可能含有多个重定位区。
SHT_SHLIB      此值是一个保留值，暂未指定语义。
SHT_LOPROC     为特殊处理器保留的区类型索引值的下边界。
SHT_HIPROC     为特殊处理器保留的区类型索引值的上边界。LOPROC ~ HIPROC区间是为特殊处理器区类型的保留值。
SHT_LOUSER     为应用程序保留区类型索引值的下边界。
SHT_HIUSER     为应用程序保留区类型索引值的下边界。LOUSER ~ HIUSER区间的区类型可由应用程序自行定义，是一区保留值。
```

解读

- `.bss` 类型为 `NOBITS`，这一区的内容是空的，区并不占用实际的空间， 没有初值的全局变量就放在这个区。它是真没有值，由运行过程中映

射到哪个地址就取哪个地址的值。鬼知道跑哪个位置的。

- PROGBITS本区内容的格式和含义都由程序来决定，属于这个区的内容还挺多的 .text ， .data ， .init ， .rodata ，这些区默认自带运行时数据。不需要你额外提供，区别是这些自带数据运行时可不可以被改变。 .data 可以被程序运行时逻辑所修改， .rodata 不可改，即常量数据。

区标签 | Section Head Flag

简称: SHF

```
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

```
名字         值      描述
SHF_WRITE      0x01      如果此标志被设置，表示本区所包含的内容在进程运行过程中是可写的。
SHF_ALLOC      0x02      如果此标志被设置，表示本区内容在进程运行过程中要占用内存单元。并不是所有区。
                    都会占用实际的内存，有一些起控制作用的区，在目标文件映射到进程空间时，并不需要占用内存。
SHF_EXECUTE    0x04       如果此标志被设置，表示本区内容是指令代码。
```

解读

此处看下与数据相关的三个区，仔细对照看参数发现其真正的区别。

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [18] .rodata           PROGBITS        0000000000002000  00002000
       000000000000001b  0000000000000000   A       0     0     4
  [25] .data             PROGBITS        0000000000004000  00003000
       0000000000000018  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS          0000000000004018  00003018
       0000000000000008  0000000000000000  WA       0     0     1
```

readelf -s app

说完大S再来说小S

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -h
   ...
  -S --section-headers   Display the sections' header
     --sections          An alias for --section-headers
  -s --syms              Display the symbol table
     --symbols           An alias for --syms
```

显示所有符号表 | Symbol Table。

```
root@5e3abe332c5a:/home/docker/case_code_100# readelf -s app
Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)

Symbol table '.symtab' contains 67 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000318     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000338     0 SECTION LOCAL  DEFAULT    2
```

```
   3: 0000000000000358     0 SECTION LOCAL  DEFAULT    3
   4: 000000000000037c     0 SECTION LOCAL  DEFAULT    4
   5: 00000000000003a0     0 SECTION LOCAL  DEFAULT    5
   6: 00000000000003c8     0 SECTION LOCAL  DEFAULT    6
   7: 0000000000000470     0 SECTION LOCAL  DEFAULT    7
   8: 00000000000004f4     0 SECTION LOCAL  DEFAULT    8
   9: 0000000000000508     0 SECTION LOCAL  DEFAULT    9
  10: 0000000000000528     0 SECTION LOCAL  DEFAULT   10
  11: 0000000000000600     0 SECTION LOCAL  DEFAULT   11
  12: 0000000000001000     0 SECTION LOCAL  DEFAULT   12
  13: 0000000000001020     0 SECTION LOCAL  DEFAULT   13
  14: 0000000000001040     0 SECTION LOCAL  DEFAULT   14
  15: 0000000000001050     0 SECTION LOCAL  DEFAULT   15
  16: 0000000000001060     0 SECTION LOCAL  DEFAULT   16
  17: 0000000000001218     0 SECTION LOCAL  DEFAULT   17
  18: 0000000000002000     0 SECTION LOCAL  DEFAULT   18
  19: 000000000000201c     0 SECTION LOCAL  DEFAULT   19
  20: 0000000000002068     0 SECTION LOCAL  DEFAULT   20
  21: 0000000000003db8     0 SECTION LOCAL  DEFAULT   21
  22: 0000000000003dc0     0 SECTION LOCAL  DEFAULT   22
  23: 0000000000003dc8     0 SECTION LOCAL  DEFAULT   23
  24: 0000000000003fb8     0 SECTION LOCAL  DEFAULT   24
  25: 0000000000004000     0 SECTION LOCAL  DEFAULT   25
  26: 0000000000004018     0 SECTION LOCAL  DEFAULT   26
  27: 0000000000000000     0 SECTION LOCAL  DEFAULT   27
  28: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
  29: 0000000000001090     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
  30: 00000000000010c0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
  31: 0000000000001100     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
  32: 0000000000004018     1 OBJECT  LOCAL  DEFAULT   26 completed.8060
  33: 0000000000003dc0     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtors_aux_fin
  34: 0000000000001140     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
  35: 0000000000003db8     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_init_array_
  36: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
  37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
  38: 000000000000218c     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
  39: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
  40: 0000000000003dc0     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_end
  41: 0000000000003dc8     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
  42: 0000000000003db8     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_start
  43: 000000000000201c     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
  44: 0000000000003fb8     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
  45: 0000000000001000     0 FUNC    LOCAL  DEFAULT   12 _init
  46: 0000000000001210     5 FUNC    GLOBAL DEFAULT   16 __libc_csu_fini
  47: 0000000000004010     8 OBJECT  GLOBAL DEFAULT   25 my_name
  48: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
  49: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
  50: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   25 _edata
  51: 0000000000001218     0 FUNC    GLOBAL HIDDEN    17 _fini
  52: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
  53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
  54: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
  55: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
  56: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
  57: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
  58: 00000000000011a0   101 FUNC    GLOBAL DEFAULT   16 __libc_csu_init
  59: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   26 _end
  60: 0000000000001060    47 FUNC    GLOBAL DEFAULT   16 _start
  61: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
  62: 0000000000001174    30 FUNC    GLOBAL DEFAULT   16 main
  63: 0000000000001149    43 FUNC    GLOBAL DEFAULT   16 say_hello
  64: 0000000000004018     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
  65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
  66: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2
```

解读

`.dynsym`，.`symtab` 两区的类型如下，是一个含义。

```
SHT_SYMTAB      同DYNSYM
```

```
SHT_DYNSYM      这两类区都含有符号表。目前，目标文件中最多只能各包含一个这两种区，但这种限制以后可能会取消。
                一般来说，SYMTAB提供的符号用于在创建目标文件的时候编辑链接，在运行期间也有可能会用于动态链接。
                SYMTAB包含完整的符号表，它往往会包含很多在运行期间(动态链接)用不到的符号。所以，一个目标文件
                可以再有一个DYNSYM区，它含有一个较小的符号表，专门用于动态链接。
```

正如描述所言，`.dynsym`是`.symtab`的缩小版，在其中能看到亲切的`printf`。具体请参考以下四个维度来理解符号表。

符号表绑定 | Symbol Table Bind

简称 `STB`

```
STB_LOCAL       表明本符号是一个本地符号。它只出现在本文件中，在本文件外该符号无效。
                所以在不同的文件中可以定义相同的符号名，它们之间不会互相影响。
STB_GLOBAL      表明本符号是一个全局符号。当有多个文件被链接在一起时，在所有文件中该符号都是可见的。
                正常情况下，在一个文件中定义的全局符号，一定是在其它文件中需要被引用，否则无须定义为全局。
STB_WEAK        类似于全局符号，但是相对于STB_GLOBAL，它们的优先级更低。
                全局符号(global symbol)和弱符号(weak symbol)在以下两方面有区别：
                • 当链接编辑器把若干个可重定位目标文件链接起来时，同名的STB_GLOBAL符号不允许出现多次。
                而如果在一个目标文件中已经定义了一个全局的符号(global symbol)，当一个同名的弱符号(weak symbol)出现时，并不会发生错误。
                链接编辑器会以全局符号为准，忽略弱符号。与全局符号相似，
                如果已经存在的是一个公用符号，即st_shndx域为SHN_COMMON值的符号，当一个同名的弱符号(weak symbol)出现时，也不会发生错误。
                链接编辑器会以公用符号为准，忽略弱符号。
                • 在查找符号定义时，链接编辑器可能会搜索存档的库文件。如果是查找全局符号，
                链接编辑器会提取包含该未定义的全局符号的存档成员，存档成员可能是一个全局的符号，
                也可能是弱符号。而如果是查找弱符号，链接编辑器不会去提取存档成员。未解析的弱符号值为0。
STB_LOPROC ~ STB_HIPROC 为特殊处理器保留的属性区间。
```

符号表类型 | Symbol Table Type

简称 `STT`

```
STT_NOTYPE      本符号类型未指定。
STT_OBJECT      本符号是一个数据对象，比如变量，数组等。
STT_FUNC        本符号是一个函数，或者其它的可执行代码。函数符号在共享目标文件中有特殊的意义。当另外一个目标文件引用一个共享目标文件中的函数符号时，
STT_SECTION     本符号与一个区相关联，用于重定位，通常具有STB_LOCAL属性。
STT_FILE        本符号是一个文件符号，它具有STB_LOCAL属性，它的区索引值是SHN_ABS。在符号表中如果存在本类符号的话，它会出现在所有STB_LOCAL类符
STT_LOPROC ~ STT_HIPROC 这一区间的符号类型为特殊处理器保留。
```

符号表可见性 | Symbol Table Visibility

简称 `STV`

```
STV_DEFAULT     当符号的可见性是STV_DEFAULT时，那么该符号的可见性由符号的绑定属性决定。
                这类情况下，（可执行文件和共享库中的）全局符号和弱符号默认是外部可访问的，
                本地符号默认外部是无法被访问的。但是，可见性是STV_DEFAULT的全局符号和弱符号是可被覆盖的。
                什么意思？举个最典型的例子，共享库中的可见性值为STV_DEFAULTD的全局符号和弱符号
                是可被可执行文件中的同名符号覆盖的。
STV_HIDDEN      当符号的可见性是STV_HIDDEN时，证明该符号是外部无法访问的。这个属性主要
                用来控制共享库对外接口的数量。需要注意的是，一个可见性为STV_HIDDEN的数据对象，
                如果能获取到该符号的地址，那么依然是可以访问或者修改该数据对象的。在可重定位文件中，
                如果一个符号的可见性是STV_HIDDEN的话，那么在链接生成可执行文件或者共享库的过程中，
                该符号要么被删除，要么绑定属性变成STB_LOCAL。
STV_PROTECTED   当符号的可见性是STV_PROTECTED时，它是外部可见的，这点跟可见性是STV_DEFAULT的一样，
                但不同的是它是不可覆盖的。这样的符号在共享库中比较常见。不可覆盖意味着如果是在该符号
                所在的共享库中访问这个符号，那么就一定是访问的这个符号，尽管可执行文件中也会存在
                同样名字的符号也不会被覆盖掉。规定绑定属性为STB_LOCAL的符号的可见性不可以是STV_PROTECTED。
STV_INTERNAL    该可见性属性的含义可以由处理器补充定义，以进一步约束隐藏的符号。 处理器补充程序的定义
                应使通用工具可以安全地将内部符号视为隐藏符号。当可重定位对象包含在可执行文件或共享对象中时，
                可重定位对象中包含的内部符号必须被链接编辑器删除或转换为STB_LOCAL绑定。
```

符号表索引 | Symbol Table Ndx

简称 `STN` 任何一个符号表项的定义都与某一个"区"相联系，因为符号是为区而定义，在区中被引用。本数据成员即指明了相关联的区。本数据成员是一个索引值，它指向相关联的区在区头表中的索引。在重定位过程中，区的位置会改变，本数据成员的值也随之改变，继续指向区的新位置。当本

数据成员指向下面三种特殊的区索引值时，本符号具有如下特别的意义：

```
SHN_ABS     符号的值是绝对的，具有常量性，在重定位过程中，此值不需要改变。
SHN_COMMON  本符号所关联的是一个还没有分配的公共区，本符号的值规定了其内容的字区对齐规则，
            与sh_addralign相似。也就是说，链接器会为本符号分配存储空间，而且其起始地址是
            向st_value对齐的。本符号的值指明了要分配的字区数。
SHN_UNDEF   当一个符号指向第1区(SHN_UNDEF)时，表明本符号在当前目标文件中未定义，在链接过程中，
            链接器会找到此符号被定义的文件，并把这些文件链接在一起。
            本文件中对该符号的引用会被链接到实际的定义上去。
```

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

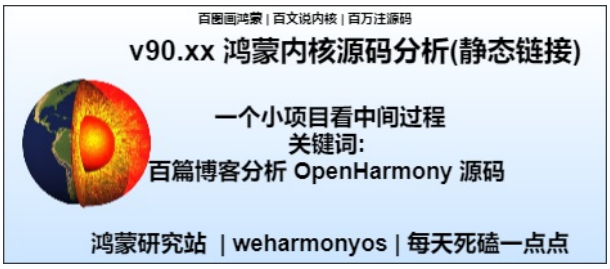
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

90_静态链接篇

本篇关键词：、、、

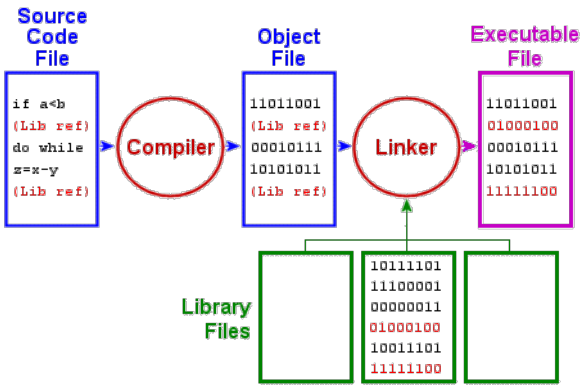


下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

下图是一个可执行文件编译，链接的过程。



本篇将通过一个完整的小工程来阐述ELF编译，链接过程，并分析。o和bin文件中各区，符号表之间的关系。从一个崭新的视角去看中间过程。

准备工作

先得有个小工程，麻雀虽小，但五脏俱全，标准的文件夹和Makefile结构，如下:

目录结构

```
root@5e3abe332c5a:/home/docker/test4harmony/54# tree
.
├── bin
│   └── weharmony
├── include
│   └── part.h
```

```
├── Makefile
├── obj
│   ├── main.o
│   └── part.o
└── src
    ├── main.c
    └── part.c

4 directories， 7 files
```

看到 .c .h .o 就感觉特别的亲切：），项目很简单，但具有代表性，有全局变量/函数，`extern`，多文件链接，和动态链接库的`printf`，用cat命令看看三个文件内容。

cat .c .h

```
root@5e3abe332c5a:/home/docker/test4harmony/54# cat ./src/main.c
#include <stdio.h>
#include "part.h"
extern int g_int;
extern char *g_str;

int main() {
        int loc_int = 53;
        char *loc_str = "harmony os";
        printf("main 开始 - 全局 g_int = %d， 全局 g_str = %s。\n"， g_int， g_str);
        func_int(loc_int);
        func_str(loc_str);
        printf("main 结束 - 全局 g_int = %d， 全局 g_str = %s。\n"， g_int， g_str);
        return 0;
}
```

```
root@5e3abe332c5a:/home/docker/test4harmony/54# cat ./src/part.c
#include <stdio.h>
#include "part.h"

int g_int = 51;
char *g_str = "hello world";

void func_int(int i) {
 int tmp = i;
 g_int = 2 * tmp ;
 printf("func_int g_int = %d , tmp = %d。\n"， g_int , tmp);
}
void func_str(char *str) {
        g_str = str;
        printf("func_str g_str = %s。\n"， g_str);
}
```

```
root@5e3abe332c5a:/home/docker/test4harmony/54# cat ./include/part.h
#ifndef _PART_H_
#define _PART_H_
void func_int(int i);
void func_str(char *str);
#endif
```

cat Makefile

`Makefile`采用标准写法，关于makefile系列篇会在编译过程篇中详细说明，此处先看点简单的。

```
root@5e3abe332c5a:/home/docker/test4harmony/54# cat Makefile
DIR_INC = ./include
DIR_SRC = ./src
DIR_OBJ = ./obj
DIR_BIN = ./bin
```

```
SRC = $(wildcard ${DIR_SRC}/*.c)
OBJ = $(patsubst %.c , ${DIR_OBJ}/%.o , $(notdir ${SRC}))

TARGET = weharmony

BIN_TARGET = ${DIR_BIN}/${TARGET}

CC = gcc
CFLAGS = -g -Wall -I${DIR_INC}

${BIN_TARGET}:${OBJ}
	$(CC) $(OBJ)  -o $@

${DIR_OBJ}/%.o:${DIR_SRC}/%.c
	$(CC) $(CFLAGS) -c  $< -o $@
。PHONY:clean
clean:
	find ${DIR_OBJ} -name *.o -exec rm -rf {}
```

编译.链接.运行.看结果

```
root@5e3abe332c5a:/home/docker/test4harmony/54# make
gcc -g -Wall -I./include -c  src/part.c -o obj/part.o
gcc -g -Wall -I./include -c  src/main.c -o obj/main.o
gcc ./obj/part.o ./obj/main.o  -o bin/weharmony
root@5e3abe332c5a:/home/docker/test4harmony/54# ./bin/weharmony
main 开始 - 全局 g_int = 51， 全局 g_str = hello world。
func_int g_int = 106 , tmp = 53。
func_str g_str = harmony os。
main 结束 - 全局 g_int = 106， 全局 g_str = harmony os。
```

结果很简单，没什么好说的。

开始分析

准备工作完成，开始了真正的分析。 因为命令输出内容太多，本篇做了精简，去除了干扰项。对这些命令还不行清楚的请翻看系列篇其他文章，此处不做介绍，阅读本篇需要一定的基础。

readelf 大S小s ./obj/main.o

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -S ./obj/main.o
There are 22 section headers， starting at offset 0x1498:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] 。text             PROGBITS        0000000000000000  00000040
       000000000000007b  0000000000000000  AX       0     0     1
  [ 2] 。rela。text        RELA            0000000000000000  00000c80
       0000000000000108  0000000000000018   I      19     1     8
  [ 3] .data             PROGBITS        0000000000000000  000000bb
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] 。bss              NOBITS          0000000000000000  000000bb
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] 。rodata           PROGBITS        0000000000000000  000000c0
       000000000000007d  0000000000000000   A       0     0     8
       ......
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -s ./obj/main.o

Symbol table '.symtab' contains 22 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
```

```
    ...
  15: 0000000000000000   123 FUNC    GLOBAL DEFAULT    1 main
  16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND g_str
  17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND g_int
  18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
  19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
  20: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND func_int
  21: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND func_str
```

解读

编译 main.c 后 main.o 告诉了链接器以下信息

- 有一个文件 叫 main.c `(Type=FILE)`
- 文件中有个函数叫 main `(Type=FUNC)`，并且这是一个全局函数，`(Bind = GLOBAL ， Vis = DEFAULT`，全局的意思就是可以被外部文件所引用。
- 剩下的 `g_str` ， `printf` ， `func_int` ，...。，都是需要外部提供，并未在本文件中定义的符号 `(Ndx = UND ， Type = NOTYPE)`，至于怎么顺藤摸瓜找到这些符号那我不管，。o文件是独立存在，它只是告诉你我用了哪些东西，但我也不知道在哪里。
- `printf` 和 `func_int` 对它来说一视同仁，都是外部链接符号，没有特殊对待。

readelf 大S小s ./obj/part.o

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -S ./obj/part.o
  [ 1] 。text           PROGBITS        0000000000000000  00000040
       0000000000000078  0000000000000000  AX       0     0     1
  [ 2] 。rela。text       RELA            0000000000000000  00000cf0
       00000000000000c0  0000000000000018   I     21     1     8
  [ 3] .data            PROGBITS        0000000000000000  000000b8
       0000000000000004  0000000000000000  WA       0     0     4
  [ 4] 。bss            NOBITS          0000000000000000  000000bc
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] 。rodata         PROGBITS        0000000000000000  000000c0
       0000000000000045  0000000000000000   A       0     0     8
  [ 6] .data。rel。local   PROGBITS        0000000000000000  00000108
       0000000000000008  0000000000000000  WA       0     0     8
       ......
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -s ./obj/part.o

Symbol table '.symtab' contains 22 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS part.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
       ...
    16: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 g_int
    17: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    6 g_str
    18: 0000000000000000    52 FUNC    GLOBAL DEFAULT    1 func_int
    19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    20: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    21: 0000000000000034    57 FUNC    GLOBAL DEFAULT    1 func_str
```

解读

编译 part.c 后part。o告诉了链接器以下信息

- 有一个文件 叫 part.c `(Type=FILE)`
- 文件中有两个函数叫 `func_int` ， `func_str` `(Type=FUNC)`，并且都是全局函数，`(Bind = GLOBAL ， Vis = DEFAULT`，全局的意思就是可以被外部文件所引用。
- 文件中有两个对象叫 `g_int` ， `g_str` `(Type=OBJECT)`，并且都是全局对象，同样可以被外部使用。
- 剩下的 `printf` ， `_GLOBAL_OFFSET_TABLE_` ，都是需要外部提供，并未在本文件中定义的符号 `(Ndx = UND ， Type = NOTYPE)`
- 另外 part.c的局部变量 `tmp` 并没有出现在符号表中。因为符号表相当于外交部，只有对外的内容。
- `func_int` ， `func_str` 在1区代码区 。`text` 。

- `g_int` 在3区 `.data` 数据区， 打开3区，发现了 0x33 就是源码中 int g_int = 51;的值

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -x 3 ./obj/part.o
Hex dump of section '.data':
0x00000000 33000000                            3...
```

- `g_str` 在6区， `.data.rel.local` 数据区，打开6区看结果

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -x 6 ./obj/part.o
Hex dump of section '.data.rel.local':
NOTE: This section has relocations against it， but these have NOT been applied to this dump.
0x00000000 00000000 00000000                   ........
```

并未发现 char *g_str = "hello world";的身影，反而抛下一句话 NOTE: This section has relocations against it， but these have NOT been applied to this dump.翻译过来是 注意：此部分已针对它进行重定位，但是尚未将其应用于此转储。 最后在5区 '.rodata'找到了 `hello world`

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -x 5 ./obj/part.o
  Hex dump of section '.rodata':
  0x00000000 68656c6c 6f20776f 726c6400 00000000 hello world.....
  0x00000010 66756e63 5f696e74 20675f69 6e74203d func_int g_int =
  0x00000020 2025642c 746d7020 3d202564 2e0a0066  %d，tmp = %d...f
  0x00000030 756e635f 73747220 675f7374 72203d20 unc_str g_str =
  0x00000040 25732e0a 00                         %s..
```

至于重定向是如何实现的，在系列篇 重定向篇中已有详细说明，不再此展开说。

- 看完两个符号表总结下来就是三句话
  - 我是谁，我在哪
  - 我能提供什么给别人用
  - 我需要别人提供什么给我用。

readelf 大S小s ./bin/weharmony

`weharmony` 是将 `main.o` ， `part.o` 和库文件链接完成后的可执行文件。

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -S ./bin/weharmony
There are 36 section headers， starting at offset 0x4908:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
       ......
  [16] .text             PROGBITS        0000000000001060  00001060
       0000000000000255  0000000000000000  AX       0     0     16
  [17] .fini             PROGBITS        00000000000012b8  000012b8
       000000000000000d  0000000000000000  AX       0     0     4
  [18] .rodata           PROGBITS        0000000000002000  00002000
       00000000000000cd  0000000000000000   A       0     0     8
       ......
  [25] .data             PROGBITS        0000000000004000  00003000
       0000000000000020  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS          0000000000004020  00003020
       0000000000000008  0000000000000000  WA       0     0     1
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -s ./bin/weharmony

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
```

```
Symbol table '.symtab' contains 75 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000318     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000000338     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000358     0 SECTION LOCAL  DEFAULT    3
     ....
    33: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    34: 0000000000001090     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
    35: 00000000000010c0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
    36: 0000000000001100     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
    37: 0000000000004020     1 OBJECT  LOCAL  DEFAULT   26 completed.8060
    38: 0000000000003dc0     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtors_aux_fin
    39: 0000000000001140     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    40: 0000000000003db8     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_init_array_
    41: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS part.c
    42: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
    43: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    44: 000000000000225c     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
    45: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    46: 0000000000003dc0     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_end
    47: 0000000000003dc8     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
    48: 0000000000003db8     0 NOTYPE  LOCAL  DEFAULT   21 __init_array_start
    49: 00000000000020c0     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
    50: 0000000000003fb8     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    51: 0000000000001000     0 FUNC    LOCAL  DEFAULT   12 _init
    52: 00000000000012b0     5 FUNC    GLOBAL DEFAULT   16 __libc_csu_fini
    53: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
    54: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    55: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    56: 00000000000012b8     0 FUNC    GLOBAL HIDDEN    17 _fini
    57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
    58: 0000000000004010     4 OBJECT  GLOBAL DEFAULT   25 g_int
    59: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    60: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    61: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    62: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
    63: 0000000000004018     8 OBJECT  GLOBAL DEFAULT   25 g_str
    64: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    65: 0000000000001240   101 FUNC    GLOBAL DEFAULT   16 __libc_csu_init
    66: 0000000000001149    52 FUNC    GLOBAL DEFAULT   16 func_int
    67: 0000000000004028     0 NOTYPE  GLOBAL DEFAULT   26 _end
    68: 0000000000001060    47 FUNC    GLOBAL DEFAULT   16 _start
    69: 000000000000117d    57 FUNC    GLOBAL DEFAULT   16 func_str
    70: 0000000000004020     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    71: 00000000000011b6   123 FUNC    GLOBAL DEFAULT   16 main
    72: 0000000000004020     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    73: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    74: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@@GLIBC_2.2
```

解读

链接后的可执行文件 `weharmony` 将告诉加载器以下信息

- 涉及文件有哪些 `Type = FILE`
- 涉及函数有哪些 `Type = FUNC` func_str，func_int，_start，main
- 涉及对象有哪些 `Type = OBJECT` g_int，g_str，....它将这些数据统一归到了25区。 前往25区查看下数据，同样只发现了 int g_int = 51; 的数据。

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -x 25 ./bin/weharmony
  Hex dump of section '.data':
  0x00004000 00000000 00000000 08400000 00000000 .........@......
  0x00004010 33000000 00000000 08200000 00000000 3........ ......
```

是不是和part.o一样也被放在了 `.rodata` 区，再反查 18区，果然发了 main.c和part.c的数据都放在了这里。

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -x 18 ./bin/weharmony
  Hex dump of section '.rodata':
  0x00002000 01000200 00000000 68656c6c 6f20776f ........hello wo
  0x00002010 726c6400 00000000 66756e63 5f696e74 rld.....func_int
  0x00002020 20675f69 6e74203d 2025642c 746d7020  g_int = %d , tmp
  0x00002030 3d202564 2e0a0066 756e635f 73747220 = %d...func_str
  0x00002040 675f7374 72203d20 25732e0a 00000000 g_str = %s......
  0x00002050 6861726d 6f6e7920 6f730000 00000000 harmony os......
  0x00002060 6d61696e 20e5bc80 e5a78b20 2d20e585 main ...... - ..
  0x00002070 a8e5b180 20675f69 6e74203d 2025642c .... g_int = %d ,
  0x00002080 20e585a8 e5b18020 675f7374 72203d20  ..... g_str =
  0x00002090 25732e0a 00000000 6d61696e 20e7bb93 %s......main ...
  0x000020a0 e69d9f20 2d20e585 a8e5b180 20675f69 ... - ..... g_i
  0x000020b0 6e74203d 2025642c 20e585a8 e5b18020 nt = %d ,  .....
  0x000020c0 675f7374 72203d20 25732e0a 00       g_str = %s...
```

- 另外还有注意 printf 的变化，从 Type = NOTYPE 变成了 Type = FUNC ，告诉了后续的动态链接这是个函数

```
    57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
```

但是内容依然是 Ndx=UND ，weharmony也提供不了，内容需要运行时环境提供。并在需要动态链接表中也已经注明了内容清单，运行环境必须提供以下内容才能真正跑起来weharmony。

```
Symbol table '.dynsym' contains 7 entries:
 Num:    Value          Size Type    Bind   Vis      Ndx Name
   0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
   1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
   2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.2.5 (2)
   3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
   4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
   5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
   6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (2)
```

本例在windows环境中一般是跑不起来的。除非提供对应的运行时环境。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

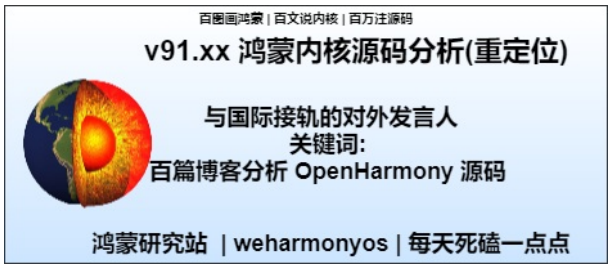
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

91_重定位篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

一个程序从源码到被执行，当中经历了3个过程：

- 编译：将.c文件编译成.o文件，不关心.o文件之间的联系。
- 静态链接：将所有.o文件合并成一个.so或.out文件，处理所有.o文件节区在目标文件中的布局。
- 动态链接：将.so或a.out文件加载到内存，处理加载文件在的内存中的布局。

什么是重定位

重定位就是把程序的逻辑地址空间变换成内存中的实际物理地址空间的过程。它是实现多道程序在内存中同时运行的基础。重定位有两种，分别是动态重定位与静态重定位。

- 1。静态重定位：即在程序装入内存的过程中完成，是指在程序开始运行前，程序中的各个地址有关的项均已完成重定位，地址变换通常是在装入时一次完成的，以后不再改变，故称为静态重定位。也就是在生成可执行/共享目标文件的同时已完成地址的静态定位，它解决了可执行文件/共享目标文件的内部矛盾。
- 2。动态重定位：它不是在程序装入内存时完成的，而是CPU每次访问内存时 由动态地址变换机构（硬件）自动进行把相对地址转换为绝对地址。动态重定位需要软件和硬件相互配合完成。也就是说可执行文件/共享目标文件的外部矛盾需要外部环境解决，它向外提供了一份入住地球村的外交说明。即本篇最后部分内容。

重定位十种类型

- 重定位有10种类型，在实际中请对号入座，这些类型部分在本篇能见到，如下:

类型	公式	具体描述
R_X86_64_32	公式:S+A S:重定项中VALUE成员所指符号的内存地址 A:被重定位处原值，表示"引用符号的内存地址"与S的偏移	全局变量，在不加-fPIC编译生成的.o文件中，每个引用处对应一个R_X86_64_32重定位项，非static全局变量，在不加-fPIC编译生成的.so文件中，每个引用处对应一个R_X86_64_32重定位项。

R_X86_64_PC32	公式:S+A-P S:重定项中VALUE成员所指符号的内存地址 A:被重定位处原值，表示"被重定位处"与"下一条指令"的偏移 P:被重定位处的内存地址	非static函数，在不加-fPIC编译生成的.o和.so文件中，每个调用处对应一个R_X86_64_PC32重定位项
R_X86_64_PLT32	公式:L+A-P L:<重定项中VALUE成员所指符号@plt>的内存地址 A:被重定位处原值，表示"被重定位处"相对于"下一条指令"的偏移 P:被重定位处的内存地址	非static函数，在加-fPIC编译生成的.o文件中，每个调用处对应一个R_386_PLT32重定位项。
R_X86_64_RELATIVE	公式:B+A B:.so文件加载到内存中的基地址 A:被重定位处原值，表示引用符号在.so文件中的偏移	static全局变量，在不加-fPIC编译生成的.so文件中，每个引用处对应一个R_X86_64_RELATIVE重定位项。
R_X86_64_GOT32	公式:G G:引用符号的地址指针，相对于GOT的偏移	非static全局变量，在加-fPIC编译生成的.o文件中，每个引用处对应一个R_X86_64_GOT32重定位项
R_X86_64_GOTOFF	公式:S-GOT S:重定项中VALUE成员所指符号的内存地址 GOT:运行时，.got段的结束地址	static全局变量，在加-fPIC编译生成的.o文件中，每个引用处对应一个R_X86_64_GOTOFF重定位项。
R_X86_64_GOLB_DAT	公式:S S:重定项中VALUE成员所指符号的内存地址	非static全局变量，在加-fPIC编译生成的.so文件中，每个引用处对应一个R_X86_64_GOLB_DAT重定位项。
R_X86_64_COPY	公式:无	.out中利用extern引用.so中的变量，每个引用处对应一个R_X86_64_COPY重定位项。
R_X86_64_JUMP_SLOT	公式:S（与R_386_GLOB_DAT的公式一样，但对于动态ld，R_386_JMP_SLOT类型与R_386_RELATIVE等价） S:重定项中VALUE成员所指符号的内存地址	非static函数，在加-fPIC编译生成的.so文件中，每个调用处对应一个R_X86_64_JMP_SLOT重定位项。
R_X86_64_GOTPC	公式:GOT+A-P GOT:运行时，.got段的结束地址 A:被重定位处原值，表示"被重定位处"在机器码中的偏移 P:被重定位处的内存地址	全局变量，在加-fPIC编译生成的.o文件中，会额外生成R_X86_64_PC32和R_X86_64_GOTPC重定位项，非static函数，在加-fPIC编译生成的.o文件中，也会额外 生成R_X86_64_PC32和R_X86_64_GOTPC重定位项。

解读

- fPIC的全称是 Position Independent Code， 用于生成位置无关代码。

objdump命令

objdump命令是Linux下的反汇编目标文件或者可执行文件的命令，它以一种可阅读的格式让你更多地了解二进制文件可能带有的附加信息。本篇将用它说明静态重定位的实现细节和动态重定位前置条件准备。先整体走读下 `objdump` 命令

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump
Usage: objdump <option(s)> <file(s)>
 Display information from object <file(s)>.
 At least one of the following switches must be given:
  -a , --archive-headers    Display archive header information
  -f , --file-headers       Display the contents of the overall file header
  -p , --private-headers    Display object format specific file header contents
  -P , --private=OPT , OPT... Display object format specific contents
  -h , --[section-]headers  Display the contents of the section headers
  -x , --all-headers        Display the contents of all headers
  -d , --disassemble        Display assembler contents of executable sections
  -D , --disassemble-all    Display assembler contents of all sections
     --disassemble=<sym>  Display assembler contents from <sym>
```

```
  -S ,  --source             Intermix source code with disassembly
     --source-comment[=<txt>] Prefix lines of source code with <txt>
  -s ,  --full-contents      Display the full contents of all sections requested
  -g ,  --debugging          Display debug information in object file
  -e ,  --debugging-tags     Display debug information using ctags style
  -G ,  --stabs              Display (in raw form) any STABS info in the file
  -W[lLiaprmfFsoRtUuTgAckK] or
  --dwarf[=rawline ,=decodedline ,=info ,=abbrev ,=pubnames ,=aranges ,=macro ,=frames ,
         =frames-interp ,=str ,=loc ,=Ranges ,=pubtypes ,
         =gdb_index ,=trace_info ,=trace_abbrev ,=trace_aranges ,
         =addr ,=cu_index ,=links ,=follow-links]
                        Display DWARF info in the file
  --ctf=SECTION          Display CTF info from SECTION
  -t ,  --syms           Display the contents of the symbol table(s)
  -T ,  --dynamic-syms       Display the contents of the dynamic symbol table
  -r ,  --reloc          Display the relocation entries in the file
  -R ,  --dynamic-reloc      Display the dynamic relocation entries in the file
  @<file>                Read options from <file>
  -v ,  --version            Display this program's version number
  -i ,  --info           List object formats and architectures supported
  -H ,  --help           Display this information
```

objdump -S ./obj/main.o

main.o是个可重定位文件，通过 readelf 可知

```
root@5e3abe332c5a:/home/docker/test4harmony/54# readelf -h ./obj/main.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement ,  little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
```

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -S ./obj/main.o
./obj/main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
#include <stdio.h>
#include "part.h"
extern int g_int;
extern char *g_str;

int main() {
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp , %rbp
   8:   48 83 ec 10             sub    $0x10 , %rsp
       int loc_int = 53;
   c:   c7 45 f4 35 00 00 00    movl   $0x35 , -0xc(%rbp)
       char *loc_str = "harmony os";
  13:   48 8d 05 00 00 00 00    lea    0x0(%rip) , %rax        # 1a <main+0x1a>
  1a:   48 89 45 f8             mov    %rax , -0x8(%rbp)
       printf("main 开始 - 全局 g_int = %d ,  全局 g_str = %s.\n" ,  g_int ,  g_str);
  1e:   48 8b 15 00 00 00 00    mov    0x0(%rip) , %rdx        # 25 <main+0x25>
  25:   8b 05 00 00 00 00       mov    0x0(%rip) , %eax        # 2b <main+0x2b>
  2b:   89 c6                   mov    %eax , %esi
  2d:   48 8d 3d 00 00 00 00    lea    0x0(%rip) , %rdi        # 34 <main+0x34>
  34:   b8 00 00 00 00          mov    $0x0 , %eax
  39:   e8 00 00 00 00          callq  3e <main+0x3e>
       func_int(loc_int);
  3e:   8b 45 f4                mov    -0xc(%rbp) , %eax
  41:   89 c7                   mov    %eax , %edi
```

```
43:   e8 00 00 00 00          callq  48 <main+0x48>
    func_str(loc_str);
48:   48 8b 45 f8             mov    -0x8(%rbp), %rax
4c:   48 89 c7                mov    %rax, %rdi
4f:   e8 00 00 00 00          callq  54 <main+0x54>
    printf("main 结束 - 全局 g_int = %d， 全局 g_str = %s.\n"， g_int， g_str);
54:   48 8b 15 00 00 00 00    mov    0x0(%rip), %rdx        # 5b <main+0x5b>
5b:   8b 05 00 00 00 00       mov    0x0(%rip), %eax        # 61 <main+0x61>
61:   89 c6                   mov    %eax, %esi
63:   48 8d 3d 00 00 00 00    lea    0x0(%rip), %rdi        # 6a <main+0x6a>
6a:   b8 00 00 00 00          mov    $0x0, %eax
6f:   e8 00 00 00 00          callq  74 <main+0x74>
    return 0;
74:   b8 00 00 00 00          mov    $0x0, %eax
79:   c9                      leaveq
7a:   c3                      retq
```

解读

- 注意那些 `00 00 00 00` 部分，这些都是编译器暂时无法确定的内容。肉眼计算下此时 `OFFSET` 偏移位为 `0x16` ， `0x21` ，即下表内容

objdump -r ./obj/main.o

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -r ./obj/main.o
./obj/main.o:     file format elf64-x86-64
RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000016 R_X86_64_PC32     .rodata-0x0000000000000004
0000000000000021 R_X86_64_PC32     g_str-0x0000000000000004
0000000000000027 R_X86_64_PC32     g_int-0x0000000000000004
0000000000000030 R_X86_64_PC32     .rodata+0x000000000000000c
000000000000003a R_X86_64_PLT32    printf-0x0000000000000004
0000000000000044 R_X86_64_PLT32    func_int-0x0000000000000004
0000000000000050 R_X86_64_PLT32    func_str-0x0000000000000004
0000000000000057 R_X86_64_PC32     g_str-0x0000000000000004
000000000000005d R_X86_64_PC32     g_int-0x0000000000000004
0000000000000066 R_X86_64_PC32     .rodata+0x0000000000000044
0000000000000070 R_X86_64_PLT32    printf-0x0000000000000004
```

解读

- `0x16` ， `0x21` 对应的这些值都是 0，也就是说对于编译器不能确定的地址都这设置为空(0x000000)，同时编译器都生成一一对应的记录，该记录告诉链接器在进行链接时要修正这条指令中函数的内存地址，并告知是什么重定位类型，要去哪里找数据填充。
- 外部全局变量重定位 `g_str` ， `g_int`

  ```
  0000000000000021 R_X86_64_PC32     g_str-0x0000000000000004
  0000000000000027 R_X86_64_PC32     g_int-0x0000000000000004
  ---
  1e:   48 8b 15 00 00 00 00    mov    0x0(%rip), %rdx        # 25 <main+0x25>
  25:   8b 05 00 00 00 00       mov    0x0(%rip), %eax        # 2b <main+0x2b>
  ```

  编译器连g_str在哪个.o文件都不知道，当然更不知道g_str运行时的地址，所以在g.o文件中设置一个重定位，要求后续过程根据"S(g_str内存地址)-A(0x04)"，修改main.o镜像中0x21偏移处的值.

- 函数重定位，重定位类型为 `R_X86_64_PLT32`

  ```
  000000000000003a R_X86_64_PLT32    printf-0x0000000000000004
  0000000000000044 R_X86_64_PLT32    func_int-0x0000000000000004
  0000000000000050 R_X86_64_PLT32    func_str-0x0000000000000004
  0000000000000070 R_X86_64_PLT32    printf-0x0000000000000004
  ---
  39:   e8 00 00 00 00          callq  3e <main+0x3e>
  43:   e8 00 00 00 00          callq  48 <main+0x48>
  ```

  同样编译器连``func_int ， printf`在哪个.o文件都不知道，当然更不知道它们的运行时的地址，所以在main.o文件中设置一个重定位，后续将 修

改main.o镜像中3a偏移处的值。

- 另一部分数据由本.o自己提供，如下

objdump -sj .rodata ./obj/main.o

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -sj .rodata ./obj/main.o
./obj/main.o:     file format elf64-x86-64
Contents of section .rodata:
 0000 6861726d 6f6e7920 6f730000 00000000  harmony os......
 0010 6d61696e 20e5bc80 e5a78b20 2d20e585  main ...... - ..
 0020 a8e5b180 20675f69 6e74203d 2025642c  .... g_int = %d ,
 0030 20e585a8 e5b18020 675f7374 72203d20   ...... g_str =
 0040 25732e0a 00000000 6d61696e 20e7bb93  %s......main ...
 0050 e69d9f20 2d20e585 a8e5b180 20675f69  ... - ...... g_i
 0060 6e74203d 2025642c 20e585a8 e5b18020  nt = %d ,  ......
 0070 675f7374 72203d20 25732e0a 00        g_str = %s...
```

解读

- 内部变量重定位。

  ```
  13:   48 8d 05 00 00 00 00    lea    0x0(%rip) , %rax        # 1a <main+0x1a>
  ---
  0000000000000016 R_X86_64_PC32     .rodata-0x0000000000000004
  ```

  因为是局部变量，编译器知道数据放在了 `.rodata` 区，要求后续过程根据 "S(main.o镜像中.rodata的内存地址)-A(0x04)"，修改main.o镜像中0x16偏移处的值。

再分析经过静态链接之后的可执行文件

objdump -S ./bin/weharmony

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -S ./bin/weharmony
Disassembly of section .text:
0000000000001188 <func_str>:
void func_str(char *str) {
    1188:       f3 0f 1e fa             endbr64
    118c:       55                    push   %rbp
    118d:       48 89 e5                mov    %rsp , %rbp
    1190:       48 83 ec 10             sub    $0x10 , %rsp
    1194:       48 89 7d f8             mov    %rdi , -0x8(%rbp)
        g_str = str;
    1198:       48 8b 45 f8             mov    -0x8(%rbp) , %rax
    119c:       48 89 05 75 2e 00 00    mov    %rax , 0x2e75(%rip)        # 4018 <g_str>
        printf("func_str g_str = %s.\n" ,  g_str);
    11a3:       48 8b 05 6e 2e 00 00    mov    0x2e6e(%rip) , %rax        # 4018 <g_str>
    11aa:       48 89 c6                mov    %rax , %rsi
    11ad:       48 8d 3d 83 0e 00 00    lea    0xe83(%rip) , %rdi        # 2037 <_IO_stdin_used+0x37>
    11b4:       b8 00 00 00 00          mov    $0x0 , %eax
    11b9:       e8 92 fe ff ff          callq  1050 <printf@plt>
    11be:       90                    nop
    11bf:       c9                    leaveq
    11c0:       c3                    retq

00000000000011c1 <main>:
#include <stdio.h>
#include "part.h"
extern int g_int;
extern char *g_str;

int main() {
    11c1:       f3 0f 1e fa             endbr64
    11c5:       55                    push   %rbp
    11c6:       48 89 e5                mov    %rsp , %rbp
    11c9:       48 83 ec 10             sub    $0x10 , %rsp
        int loc_int = 53;
```

```
   11cd:       c7 45 f4 35 00 00 00    movl   $0x35 , -0xc(%rbp)
     char *loc_str = "harmony os";
   11d4:       48 8d 05 75 0e 00 00    lea    0xe75(%rip) , %rax        # 2050 <_IO_stdin_used+0x50>
   11db:       48 89 45 f8             mov    %rax , -0x8(%rbp)
     printf("main 开始 - 全局 g_int = %d ， 全局 g_str = %s.\n" ， g_int ， g_str);
   11df:       48 8b 15 32 2e 00 00    mov    0x2e32(%rip) , %rdx        # 4018 <g_str>
   11e6:       8b 05 24 2e 00 00       mov    0x2e24(%rip) , %eax        # 4010 <g_int>
   11ec:       89 c6                   mov    %eax , %esi
   11ee:       48 8d 3d 6b 0e 00 00    lea    0xe6b(%rip) , %rdi        # 2060 <_IO_stdin_used+0x60>
   11f5:       b8 00 00 00 00          mov    $0x0 , %eax
   11fa:       e8 51 fe ff ff          callq  1050 <printf@plt>
     func_int(loc_int);
   11ff:       8b 45 f4                mov    -0xc(%rbp) , %eax
   1202:       89 c7                   mov    %eax , %edi
   1204:       e8 40 ff ff ff          callq  1149 <func_int>
     func_str(loc_str);
   1209:       48 8b 45 f8             mov    -0x8(%rbp) , %rax
   120d:       48 89 c7                mov    %rax , %rdi
   1210:       e8 73 ff ff ff          callq  1188 <func_str>
     printf("main 结束 - 全局 g_int = %d ， 全局 g_str = %s.\n" ， g_int ， g_str);
   1215:       48 8b 15 fc 2d 00 00    mov    0x2dfc(%rip) , %rdx        # 4018 <g_str>
   121c:       8b 05 ee 2d 00 00       mov    0x2dee(%rip) , %eax        # 4010 <g_int>
   1222:       89 c6                   mov    %eax , %esi
   1224:       48 8d 3d 6d 0e 00 00    lea    0xe6d(%rip) , %rdi        # 2098 <_IO_stdin_used+0x98>
   122b:       b8 00 00 00 00          mov    $0x0 , %eax
   1230:       e8 1b fe ff ff          callq  1050 <printf@plt>
     return 0;
   1235:       b8 00 00 00 00          mov    $0x0 , %eax
   123a:       c9                      leaveq
   123b:       c3                      retq
   123c:       0f 1f 40 00             nopl   0x0(%rax)
```

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -s ./bin/weharmony
...省略部分

Contents of section .plt.got:
 1040 f30f1efa f2ff25ad 2f00000f 1f440000  ......%./....D..
Contents of section .plt.sec:
 1050 f30f1efa f2ff2575 2f00000f 1f440000  ......%u/....D..

Contents of section .data:
 4000 00000000 00000000 08400000 00000000  .........@......
 4010 33000000 00000000 08200000 00000000  3........ ......

Contents of section .rodata:
 2000 01000200 00000000 68656c6c 6f20776f  ........hello wo
 2010 726c6400 00000000 66756e63 5f696e74  rld.....func_int
 2020 20675f69 6e74203d 2025642c 746d7020   g_int = %d , tmp
 2030 3d202564 2e0a0066 756e635f 73747220  = %d...func_str
 2040 675f7374 72203d20 25732e0a 00000000  g_str = %s......
 2050 6861726d 6f6e7920 6f730000 00000000  harmony os......
 2060 6d61696e 20e5bc80 e5a78b20 2d20e585  main ...... - ..
 2070 a8e5b180 20675f69 6e74203d 2025642c  .... g_int = %d ,
 2080 20e585a8 e5b18020 675f7374 72203d20   ...... g_str =
 2090 25732e0a 00000000 6d61696e 20e7bb93  %s......main ...
 20a0 e69d9f20 2d20e585 a8e5b180 20675f69  ... - ...... g_i
 20b0 6e74203d 2025642c 20e585a8 e5b18020  nt = %d ,  ......
 20c0 675f7374 72203d20 25732e0a 00        g_str = %s...
```

解读

- main.o中被重定位的部分不再是 `00 00 00 00` 都已经有了实际的数据，例如:

  ```
  char *loc_str = "harmony os";
  11d4:       48 8d 05 75 0e 00 00    lea    0xe75(%rip) , %rax        # 2050 <_IO_stdin_used+0x50>
  ```

  对应的 `# 2050 <_IO_stdin_used+0x50>` 地址数据正是 `.rodata` 2050位置的 `harmony os`

- 看main()中的

```
  1209:      48 8b 45 f8           mov    -0x8(%rbp),%rax
  120d:      48 89 c7              mov    %rax,%rdi
  1210:      e8 73 ff ff ff        callq  1188 <func_str>
```

`callq 1188` `1188`正是 `func_str`的入口地址

```
void func_str(char *str) {
   1188:      f3 0f 1e fa           endbr64
```

- 看全局变量 `g_str``g_int`对应的链接地址 `0x4018`和 `0x4010`

```
1215:      48 8b 15 fc 2d 00 00    mov    0x2dfc(%rip),%rdx       # 4018 <g_str>
121c:      8b 05 ee 2d 00 00       mov    0x2dee(%rip),%eax       # 4010 <g_int>
```

由`.data`区提供

```
  4000 00000000 00000000 08400000 00000000  .........@......
  4010 33000000 00000000 08200000 00000000  3........ ......
```

`0x4010` = 0x33 = 51

- main函数中调用 `printf`代码为 `callq 1050`

```
  1230:      e8 1b fe ff ff        callq  1050 <printf@plt>
```

内容由`.plt.sec`区提供，并反汇编该区为

```
  Contents of section .plt.sec:
  1050 f30f1efa f2ff2575 2f00000f 1f440000  ......%u/....D..

  Disassembly of section .plt.sec:
  0000000000001050 <printf@plt>:
     1050:      f3 0f 1e fa           endbr64
     1054:      f2 ff 25 75 2f 00 00    bnd jmpq *0x2f75(%rip)       # 3fd0 <printf@GLIBC_2.2.5>
     105b:      0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

注意`3fd0`，需要运行时环境提供，加载器动态重定位实现。

- 总结下来就是 weharmony 已完成了所有.o文件的静态重定位部分， 组合成一个新的可执行文件，其中只还有动态链接部分尚未完成，因为那需要运行时重定位地址。如下:

objdump -R ./bin/weharmony

```
root@5e3abe332c5a:/home/docker/test4harmony/54# objdump -R ./bin/weharmony

./bin/weharmony:     file format elf64-x86-64

DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000003db8 R_X86_64_RELATIVE  *ABS*+0x0000000000001140
0000000000003dc0 R_X86_64_RELATIVE  *ABS*+0x0000000000001100
0000000000004008 R_X86_64_RELATIVE  *ABS*+0x0000000000004008
0000000000004018 R_X86_64_RELATIVE  *ABS*+0x0000000000002008
0000000000003fd8 R_X86_64_GLOB_DAT  _ITM_deregisterTMCloneTable
0000000000003fe0 R_X86_64_GLOB_DAT  __libc_start_main@GLIBC_2.2.5
0000000000003fe8 R_X86_64_GLOB_DAT  __gmon_start__
0000000000003ff0 R_X86_64_GLOB_DAT  _ITM_registerTMCloneTable
0000000000003ff8 R_X86_64_GLOB_DAT  __cxa_finalize@GLIBC_2.2.5
0000000000003fd0 R_X86_64_JUMP_SLOT  printf@GLIBC_2.2.5
```

解读

- 这是 `weharmony` 对运行时环境提交的一份外交说明，有了它就可以与国际接轨，入住地球村。
- 这份说明其他部分很陌生，看个熟悉的 `3fd0` ，其动态链接重定位类型为 `R_X86_64_JUMP_SLOT` ，它在告诉动态加载器，在运行时环境中找到 `printf` 并完成动态重定位。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

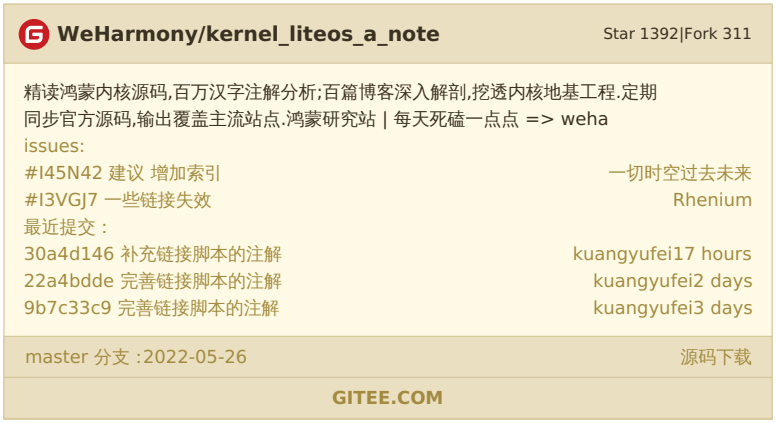
按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断

编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

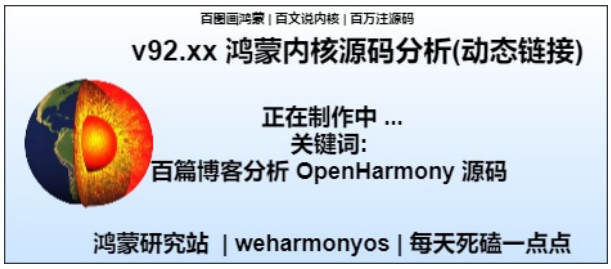


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

92_动态链接篇

本篇关键词：动态链接、静态链接、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

公用厕所

与动态链接对应的是静态链接 ，可翻看 **(静态链接篇)** ，静态链接的缺点也就是动态链接的优点，主要集中在两个方面：

- 节省空间

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， `v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

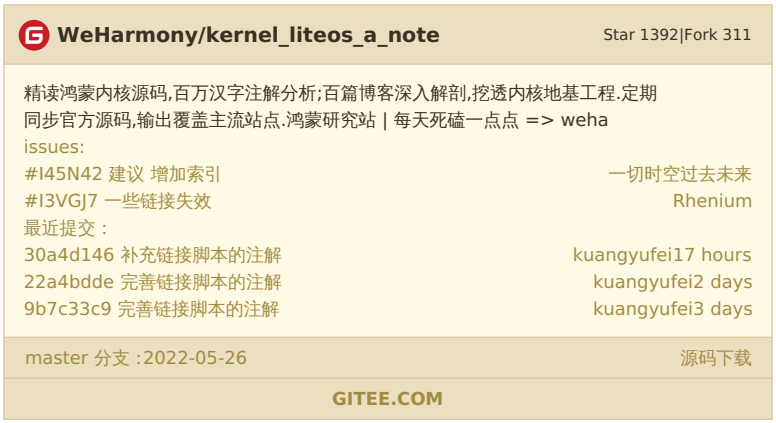
-

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

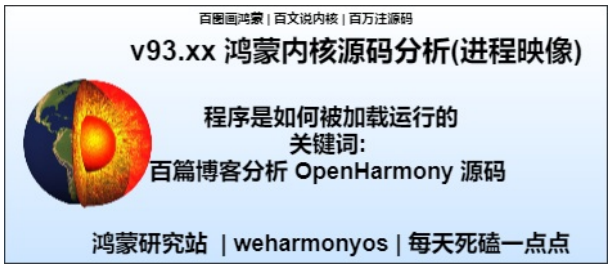


关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

93_进程映像篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

可执行文件和共享目标文件（动态连接库）是程序的静态存储形式。要执行一个程序，系统要先把相应的可执行文件和动态连接库装载到进程空间中，这样形成一个可运行的进程的内存空间布局，也可以称它为"进程映像"。

本篇结合源码介绍鸿蒙加载和运行shell进程的整个过程，因本篇涉及代码较多，所以删减了一些不相干的代码。 鸿蒙加载和运行ELF的函数为 `LOS_DoExecveFile`

LOS_DoExecveFile

根文件系统已经提供shell，fileName为 "/bin/shell"

```
//运行用户态进程 ELF格式，运行在内核态
INT32 LOS_DoExecveFile(const CHAR *fileName， CHAR * const *argv， CHAR * const *envp)
{
    ELFLoadInfo loadInfo = { 0 };
    CHAR kfileName[PATH_MAX + 1] = { 0 };//此时已陷入内核态，所以局部变量都在内核空间
    INT32 ret;
    loadInfo.newSpace = OsCreateUserVmSapce();//创建用户虚拟空间
    if (loadInfo.newSpace == NULL) {
        PRINT_ERR("%s %d， failed to allocate new vm space\n"， __FUNCTION__， __LINE__);
        return -ENOMEM;
    }
    loadInfo.argv = argv;//参数数组
    loadInfo.envp = envp;//环境数组
    ret = OsLoadELFFile(&loadInfo);//加载ELF文件
    if (ret != LOS_OK) {
        return ret;
    }
   //对当前进程旧虚拟空间和文件进行回收
    ret = OsExecRecycleAndInit(OsCurrProcessGet()， loadInfo.fileName， loadInfo.oldSpace， loadInfo.oldFiles);
    if (ret != LOS_OK) {
        (VOID)LOS_VmSpaceFree(loadInfo.oldSpace);//释放虚拟空间
        goto OUT;
    }
```

```
    ret = OsExecve(&loadInfo);//运行ELF内容
    if (ret != LOS_OK) {
        goto OUT;
    }
    return loadInfo.stackTop;
OUT:
    (VOID)LOS_Exit(OS_PRO_EXIT_OK);
    return ret;
}
```

解读

- 创建了一个新的用户进程空间，每个应用进程都有自己独立的进程空间，也称虚拟空间。这个空间和内核空间是隔离的，用户空间的虚拟地址范围为 0x00000000 ~ 0x3FFFFFFF，内核空间是0x3FFFFFFF ~ 0xFFFFFFFF
- 加载ELF文件，注意 SysExecve -> LOS_DoExecveFile，而 `SysExecve` 是个系统调用，所以 `LOS_DoExecveFile` 是运行在内核空间。加载过程由内核完成，包括申请的动态内存都是由内核空间提供。
- 加载成功后，当前进程会被腾龙换鸟，把原有内脏挖空后留给新的 `shell` 使用，原用进程空间和文件都会被保存下来。
- 运行shell，代码段，数据段装载完成后，设置好运行栈，运行就变得很简单，将用户栈保存到内核栈中，程序就会切到shell入口地址 `0x1000` 执行，正式开始了 shell 之旅

如何加载？

ELF一体两面，面对不同的场景扮演不同的角色，这是理解ELF的关键，链接器只关注1(ELF头信息)，3(区)，4(区头表) 的内容，加载器只关注1(ELF头信息)，2(段头表)，3(段)的内容，本篇说加载过程，所以不会出现区(sections)这个概念。 先看 `shell` 1，2，3(段)的内容，这些内容看过

- v53.xx 鸿蒙内核源码分析(ELF解析篇)
- v51.xx 鸿蒙内核源码分析(ELF格式篇)

的不会陌生，对照着代码去看很容易理解。

```
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries/bin# readelf -h shell
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement， little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x1000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          25268 (bytes into file)
  Flags:                             0x5000200， Version5 EABI， soft-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         11
  Size of section headers:           40 (bytes)
  Number of section headers:         27
  Section header string table index: 26
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries/bin# readelf -l shell

Elf file type is DYN (Shared object file)
Entry point 0x1000
There are 11 program headers， starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00160 0x00160 R   0x4
  INTERP         0x000194 0x00000194 0x00000194 0x00016 0x00016 R   0x1
      [Requesting program interpreter: /lib/ld-musl-arm.so.1]
  LOAD           0x000000 0x00000000 0x00000000 0x00e64 0x00e64 R   0x1000
  LOAD           0x001000 0x00001000 0x00001000 0x03690 0x03690 R E 0x1000
  LOAD           0x005000 0x00005000 0x00005000 0x001b8 0x001b8 RW  0x1000
  LOAD           0x006000 0x00006000 0x00006000 0x00034 0x00060 RW  0x1000
  DYNAMIC        0x005008 0x00005008 0x00005008 0x000c8 0x000c8 RW  0x4
  GNU_RELRO      0x005000 0x00005000 0x00005000 0x001b8 0x01000 R   0x1
```

```
  GNU_EH_FRAME   0x000e54 0x00000e54 0x00000e54 0x0000c 0x0000c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  EXIDX          0x000928 0x00000928 0x00000928 0x00010 0x00010 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .dynsym .gnu.hash .hash .dynstr .rel.dyn .ARM.exidx .rel.plt .rodata .eh_frame_hdr .eh_frame
   03     .text .init .fini .plt
   04     .init_array .fini_array .dynamic .got .got.plt
   05     .data .bss
   06     .dynamic
   07     .init_array .fini_array .dynamic .got .got.plt .bss.rel.ro
   08     .eh_frame_hdr
   09
   10     .ARM.exidx
```

解读

- INTERP 段，说明ELF需要加载另一个动态链接库 `/lib/ld-musl-arm.so.1` 。
- GNU_STACK 段，指的就是栈，没有它内核无法构建栈，而且必须是RW
- LOAD 段，指加载段，即.bss，.data，.text都属于加载段，加载它们到指定位置就是加载器的工作，而ELF本身已经提供了指令/数据的相对位置。加载器只需提供一个加载开始地址就能计算出指令/数据在虚拟空间中的最终地址。

ELFLoadInfo

理解 `ELFLoadInfo` 是理解鸿蒙加载ELF运行的关键。代码都已经注释。

```
typedef struct {//加载ELF信息结构体
    ELFInfo      execInfo; //可执行文件信息
    ELFInfo      interpInfo;//解析器文件信息 lib/libc.so
    const CHAR   *fileName;//文件名称
    CHAR         *execName;//程序名称
    INT32        argc; //参数个数
    INT32        envc; //环境变量个数
    CHAR *const  *argv; //参数数组
    CHAR *const  *envp; //环境变量数组
    UINTPTR      stackTop;//栈底位置，递减满栈下，stackTop是高地址位
    UINTPTR      stackTopMax;//栈最大上限
    UINTPTR      stackBase;//栈顶位置
    UINTPTR      stackParamBase;//栈参数空间，放置启动ELF时的外部参数，大小为 USER_PARAM_BYTE_MAX 4K
    UINT32       stackSize;//栈大小
    INT32        stackProt;//LD_PT_GNU_STACK栈的权限 ，例如(RW)
    UINTPTR      loadAddr; //加载地址
    UINTPTR      elfEntry; //装载点地址 即: _start 函数地址
    UINTPTR      topOfMem; //虚拟空间顶部位置，loadInfo->topOfMem = loadInfo->stackTopMax - sizeof(UINTPTR);
    UINTPTR      oldFiles; //旧空间的文件映像
    LosVmSpace   *newSpace;//新虚拟空间
    LosVmSpace   *oldSpace;//旧虚拟空间
#ifdef LOSCFG_ASLR
    INT32        randomDevFD;
#endif
} ELFLoadInfo;
```

解读

- 一个程序要运行需要两个必不可少的硬性条件。
  -
    - 指令在哪里，由 `elfEntry` ，它是 `.text` 的开始位置，直接在 elf头中可以读到。
  -
    - 拿到指令后在哪里运行，即栈在哪里， `ELFLoadInfo` 有7个变量在描述栈信息。足以说明栈的重要性。栈的构建对应的是ELF的 `GNU_STACK` 段，权限必须是(R + W)
- `interpInfo` 对应的是ELF的 `INTERP` 段，不是所有的ELF都会有 `INTERP` 段，如下:

```
INTERP         0x000194 0x00000194 0x00000194 0x00016 0x00016 R   0x1
    [Requesting program interpreter: /lib/ld-musl-arm.so.1]
```

这个段的意思就是需要加载动态链接库， `/lib/ld-musl-arm.so.1` 是 `libc.so` 的一个软链，具体位置在根文件系统 /rootfs/lib/libc.so 位置。

- `argv` ， `envc` 命令行参数和环境变量内核会专门开辟4K空间，保存在栈底位置，一起保存的还有ELF的辅助向量表 `auxVector` 。
- `loadAddr` 通过 `LOS_MMap` 将各 `LOAD` 段并做好的虚拟地址和物理地址的映射关系保存在了映射区。
  - 从代码看对 `.bss` 区做了匿名映射，见于 `OsSetBss()` ，不清楚为何内核要区别对待 `.bss` 区。
  - 其余各区做了文件映射。

加载过程(OsLoadELFFile)

源码位置: ..\kernel\extended\dynload\src\los_load_elf.c

```
//加载ELF格式文件
INT32 OsLoadELFFile(ELFLoadInfo *loadInfo)
{
    INT32 ret;
    OsLoadInit(loadInfo);//初始化加载信息
    ret = OsReadEhdr(loadInfo->fileName， &loadInfo->execInfo， TRUE);//读ELF头信息
    if (ret != LOS_OK) {
        goto OUT;
    }
    ret = OsReadPhdrs(&loadInfo->execInfo， TRUE);//读ELF程序头信息，构建进程映像所需信息。
    if (ret != LOS_OK) {
        goto OUT;
    }
    ret = OsReadInterpInfo(loadInfo);//读取段 INTERP 解析器信息
    if (ret != LOS_OK) {
        goto OUT;
    }
    ret = OsSetArgParams(loadInfo， loadInfo->argv， loadInfo->envp);//设置外部参数内容
    if (ret != LOS_OK) {
        goto OUT;
    }
    OsFlushAspace(loadInfo);//擦除空间
    ret = OsLoadELFSegment(loadInfo);//加载段信息
    if (ret != LOS_OK) {//加载失败时
        OsCurrProcessGet()->vmSpace = loadInfo->oldSpace;//切回原有虚拟空间
        LOS_ArchMmuContextSwitch(&OsCurrProcessGet()->vmSpace->archMmu);//切回原有MMU
        goto OUT;
    }
    OsDeInitLoadInfo(loadInfo);//ELF和.so 加载完成后释放内存
    return LOS_OK;

OUT:
    OsDeInitFiles(loadInfo);
    (VOID)LOS_VmSpaceFree(loadInfo->newSpace);
    (VOID)OsDeInitLoadInfo(loadInfo);
    return ret;
}
```

解读

- `OsReadPhdrs` 读取程序头(段头)，共11个段头。
- `OsReadInterpInfo` 读取动态链接库 `lib/libc.so` 段头信息。
- `OsSetArgParams` 将外部参数(命令行和环境变量)保存在栈底位置
- `OsFlushAspace` 切换进程空间，新进程空间重置堆区，映射区，MMU切换。映射区一旦变化意味着MMU的L1，L2表的变化。
- `OsLoadELFSegment` 加载ELF `.bss，.data，.text` 区，这些区统一叫 `LOAD` 段，建立新的虚拟地址和物理地址映射关系

```
LOAD           0x000000 0x00000000 0x00000000 0x00e64 0x00e64 R   0x1000
```

```
LOAD           0x001000 0x00001000 0x00001000 0x03690 0x03690 R E 0x1000
LOAD           0x005000 0x00005000 0x00005000 0x001b8 0x001b8 RW  0x1000
LOAD           0x006000 0x00006000 0x00006000 0x00034 0x00060 RW  0x1000
四个加载段的内容对应以下各区，这些区都会加载到用户空间指定位置。
 02     .interp .dynsym .gnu.hash .hash .dynstr .rel.dyn .ARM.exidx .rel.plt .rodata .eh_frame_hdr .eh_frame
 03     .text .init .fini .plt
 04     .init_array .fini_array .dynamic .got .got.plt
 05     .data .bss
```

- 经过以上操作， shell在虚拟内存中真实样子如下:

```
|  内存映像    | 虚拟地址范围 | 大小  |备注|
|stack 向下生长|USER_ASPACE_TOP_MAX ~ USER_MAP_SIZE + USER_MAP_BASE||
|mmap 向上生长 |USER_MAP_SIZE + USER_MAP_BASE ~ USER_MAP_BASE| USER_MAP_SIZE|USER_MAP_BASE = (USER_ASPACE_TOP_MAX >> 1
|heap 向上生长 |USER_MAP_BASE ~ USER_HEAP_BASE||USER_HEAP_BASE = USER_ASPACE_TOP_MAX >> 2
|.data .bss   |0x06060 ~ 0x006000|0x00060|
|.init_array .fini_array .dynamic .got .got.plt| 0x051b8 ~ 0x005000|0x001b8|
|.text .init .fini .plt|0x04690 ~ 0x001000|0x03690|
|.interp .dynsym .gnu.hash .hash .dynstr .rel.dyn .ARM.exidx .rel.plt .rodata .eh_frame_hdr .eh_frame|0x00e64 ~ 0x000000 |0x00e64|
```

但注意:其中不包含 /lib/libc.so的信息，动态链接部分会单独一篇去说明。

- 用户地址空间在 mmap处 一切为二， 堆区独占1/4， 所有区(.bbs，.text，..)共占1/4，映射区和栈区共占1/2，二者相立而行，向中间靠拢。

如何运行？

由 ..\kernel\extended\dynload\src\los_exec_elf.c 提供，很简单。

```
//运行ELF
STATIC INT32 OsExecve(const ELFLoadInfo *loadInfo)
{
    if ((loadInfo == NULL) || (loadInfo->elfEntry == 0)) {
        return LOS_NOK;
    }
 //任务运行的两个硬性要求:1.提供入口指令 2.运行栈空间。
    return OsExecStart((TSK_ENTRY_FUNC)(loadInfo->elfEntry)， (UINTPTR)loadInfo->stackTop，
                       loadInfo->stackBase， loadInfo->stackSize);
}

//执行用户态任务， entry为入口函数 ，其中 创建好task，task上下文 等待调度真正执行， sp:栈指针 mapBase:栈底 mapSize:栈大小
LITE_OS_SEC_TEXT UINT32 OsExecStart(const TSK_ENTRY_FUNC entry， UINTPTR sp， UINTPTR mapBase， UINT32 mapSize)
{
    UINT32 intSave;

    if (entry == NULL) {
        return LOS_NOK;
    }

    if ((sp == 0) || (LOS_Align(sp， LOSCFG_STACK_POINT_ALIGN_SIZE) != sp)) {//对齐
        return LOS_NOK;
    }
 //注意 sp此时指向栈底，栈底地址要大于栈顶
    if ((mapBase == 0) || (mapSize == 0) || (sp <= mapBase) || (sp > (mapBase + mapSize))) {//参数检查
        return LOS_NOK;
    }

    LosTaskCB *taskCB = OsCurrTaskGet();//获取当前任务
    SCHEDULER_LOCK(intSave);//拿自旋锁

    taskCB->userMapBase = mapBase;//用户态栈顶位置
    taskCB->userMapSize = mapSize;//用户态栈
    taskCB->taskEntry = (TSK_ENTRY_FUNC)entry;//任务的入口函数
 //初始化内核态栈
    TaskContext *taskContext = (TaskContext *)OsTaskStackInit(taskCB->taskID， taskCB->stackSize，
                                                              (VOID *)taskCB->topOfStack， FALSE);
    OsUserTaskStackInit(taskContext， (UINTPTR)taskCB->taskEntry， sp);//初始化用户栈，将内核栈中上下文的 context->R[0] = sp ，context->sp = s
    //这样做的目的是将用户栈SP保存到内核栈中，
    SCHEDULER_UNLOCK(intSave);//解锁
```

```
    return LOS_OK;
}
```

解读

- 运行shell出奇的简单，设置好执行指令的入口地址(PC)寄出器和栈指针(SP)就可以了，这些内容在系列篇中已经反复说过，请自行翻看。
- 因shell为用户态进程，所以会有内核态和用户态两个栈，初始化内核栈 `OsTaskStackInit` 和用户栈 `OsUserTaskStackInit` 过程在线程概念篇中也已有描述。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 `debug` 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

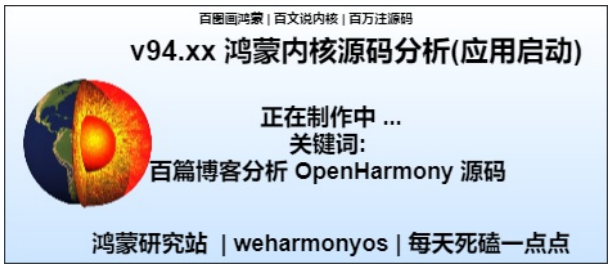
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

94_应用启动篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

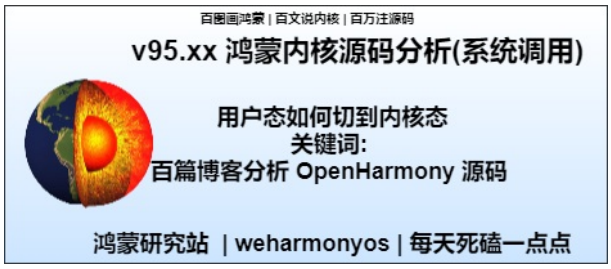
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

95_系统调用篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

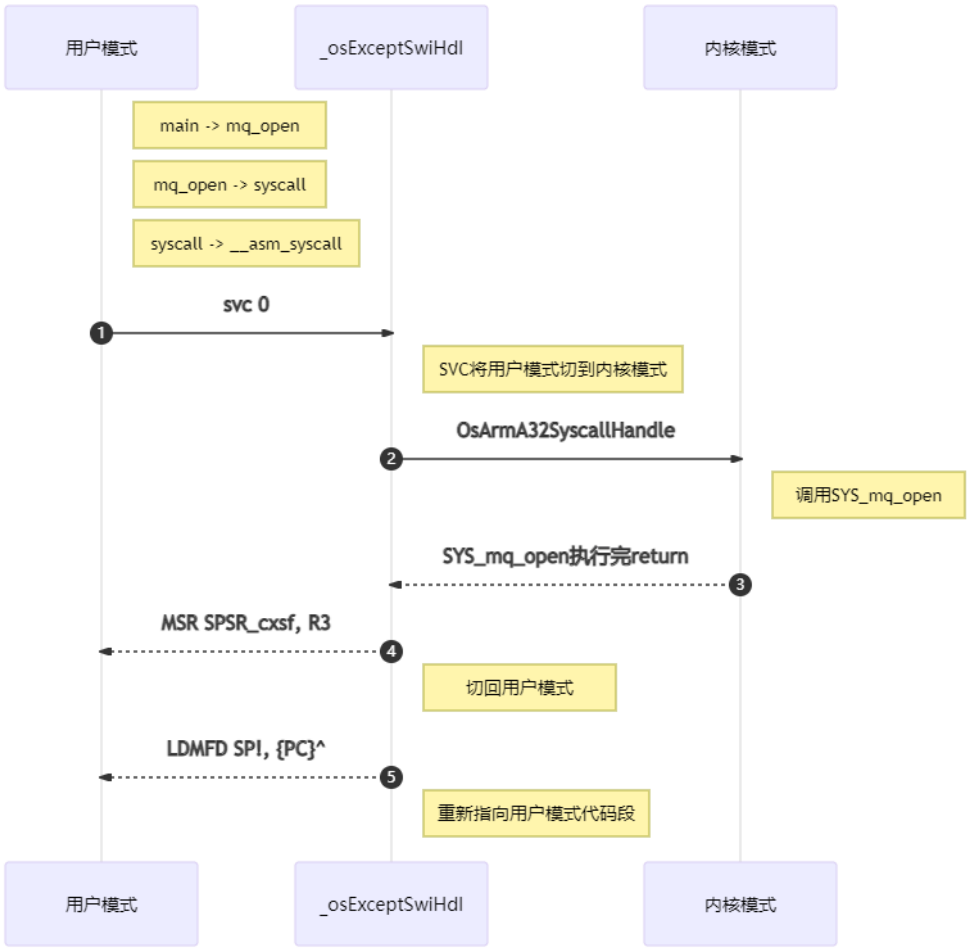
编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

本篇说清楚系统调用

读本篇之前建议先读v08。xx 鸿蒙内核源码分析(总目录) 工作模式篇。

本篇通过一张图和七段代码详细说明系统调用的整个过程，代码一捅到底，直到汇编层再也捅不下去。 先看图，这里的模式可以理解为空间，因为模式不同运行的栈空间就不一样。



过程解读

- 在应用层 `main` 中使用系统调用 `mq_open` (posix标准接口)
- `mq_open` 被封装在库中，这里直接看库里的代码。
- `mq_open` 中调用 `syscall` ，将参数传给寄出器 `R7`，R0~R6
- `SVC 0` 完成用户模式到内核模式(SVC)的切换
- `_osExceptSwiHdl` 运行在svc模式下。
- PC寄存器直接指向 `_osExceptSwiHdl` 处取指令。
- `_osExceptSwiHdl` 是汇编代码，先保存用户模式现场(R0~R12寄存器)，并调用 `OsArmA32SyscallHandle` 完成系统调用
- `OsArmA32SyscallHandle` 中通过系统调用号(保存在R7寄存器)查询对应的注册函数 `SYS_mq_open`
- `SYS_mq_open` 是本次系统调用的实现函数，完成后return回到 `OsArmA32SyscallHandle`
- `OsArmA32SyscallHandle` 再return回到 `_osExceptSwiHdl`
- `_osExceptSwiHdl` 恢复用户模式现场(R0~R12寄存器)
- 从内核模式(SVC)切回到用户模式，PC寄存器也切回用户现场。
- 由此完成整个系统调用全过程

七段追踪代码，逐个分析

1.应用程序 main

```
int main(void)
{
 char mqname[NAMESIZE] ， msgrv1[BUFFER] ， msgrv2[BUFFER];
 const char *msgptr1 = "test message1";
 const char *msgptr2 = "test message2 with differnet length";
 mqd_t mqdes;
 int prio1 = 1 ， prio2 = 2;
```

```
 struct timespec ts;
 struct mq_attr attr;
 int unresolved = 0 , failure = 0;
 sprintf(mqname , "/" FUNCTION "_" TEST "_%d" , getpid());
 attr.mq_msgsize = BUFFER;
 attr.mq_maxmsg = BUFFER;
 mqdes = mq_open(mqname , O_CREAT | O_RDWR , S_IRUSR | S_IWUSR , &attr);
 if (mqdes == (mqd_t)-1) {
  perror(ERROR_PREFIX "mq_open");
  unresolved = 1;
 }
 if (mq_send(mqdes , msgptr1 , strlen(msgptr1) , prio1) != 0) {
  perror(ERROR_PREFIX "mq_send");
  unresolved = 1;
 }
 printf("Test PASSED\n");
 return PTS_PASS;
}
```

2. mq_open 发起系统调用

```
mqd_t mq_open(const char *name , int flags , ...)
{
 mode_t mode = 0;
 struct mq_attr *attr = 0;
 if (*name == '/') name++;
 if (flags & O_CREAT) {
  va_list ap;
  va_start(ap , flags);
  mode = va_arg(ap , mode_t);
  attr = va_arg(ap , struct mq_attr *);
  va_end(ap);
 }
 return syscall(SYS_mq_open , name , flags , mode , attr);
}
```

解读

- `SYS_mq_open` 是真正的系统调用函数，对应一个系统调用号 `__NR_mq_open` ，通过宏 `SYSCALL_HAND_DE F`将 `SysMqOpen` 注册到 `g_syscallHandle` 中。

```
static UINTPTR g_syscallHandle[SYS_CALL_NUM] = {0}; //系统调用入口函数注册
static UINT8 g_syscallNArgs[(SYS_CALL_NUM + 1) / NARG_PER_BYTE] = {0};//保存系统调用对应的参数数量
#define SYSCALL_HAND_DEF(id , fun , rType , nArg)                                    \
    if ((id) < SYS_CALL_NUM) {                                                       \
        g_syscallHandle[(id)] = (UINTPTR)(fun);                                      \
        g_syscallNArgs[(id) / NARG_PER_BYTE] |= ((id) & 1) ? (nArg) << NARG_BITS : (nArg); \
    }                                                                                \

    #include "syscall_lookup.h"
#undef SYSCALL_HAND_DEF

SYSCALL_HAND_DEF(__NR_mq_open , SysMqOpen , mqd_t , ARG_NUM_4)
```

- `g_syscallNArgs` 为注册函数的参数个数，也会一块记录下来。
- 四个参数为 SYS_mq_open的四个参数，后续将保存在R0~R3寄存器中

3. syscall

```
long syscall(long n , ...)
{
 va_list ap;
 syscall_arg_t a , b , c , d , e , f;
 va_start(ap , n);
 a=va_arg(ap , syscall_arg_t);
 b=va_arg(ap , syscall_arg_t);
```

```
 c=va_arg(ap， syscall_arg_t);
 d=va_arg(ap， syscall_arg_t);
 e=va_arg(ap， syscall_arg_t);
 f=va_arg(ap， syscall_arg_t);//最多6个参数
 va_end(ap);
 return __syscall_ret(__syscall(n，a，b，c，d，e，f));
}
//4个参数的系统调用时底层处理
static inline long __syscall4(long n， long a， long b， long c， long d)
{
 register long a7 __asm__("a7") = n; //将系统调用号保存在R7寄存器
 register long a0 __asm__("a0") = a; //R0
 register long a1 __asm__("a1") = b; //R1
 register long a2 __asm__("a2") = c; //R2
 register long a3 __asm__("a3") = d; //R3
 __asm_syscall("r"(a7)， "0"(a0)， "r"(a1)， "r"(a2)， "r"(a3))
}
```

解读

- 可变参数实现所有系统调用的参数的管理，可以看出，在鸿蒙内核中系统调用的参数最多不能大于6个
- R7寄存器保存了系统调用号，R0~R5保存具体每个参数
- 可变参数的具体实现后续有其余篇幅详细介绍，敬请关注。

4. svc 0

```
//切到SVC模式
#define __asm_syscall(...) do { \
 __asm__ __volatile__ ( "svc 0" \
 : "=r"(x0) : __VA_ARGS__ : "memory"， "cc"); \
 return x0; \
 } while (0)
```

看不太懂的没关系，这里我们只需要记住：系统调用号存放在r7寄存器，参数存放在r0，r1，r2寄存器中，返回值最终会存放在寄存器r0中

The ARM720T supports seven modes of operation as listed in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution state
FIQ	fiq	Designed to support a data transfer or channel process
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

Table 2-4 Exception vector addresses

High address	Low address	Exception	Mode on entry
0xFFFF0000	0x00000000	Reset	Supervisor
0xFFFF0004	0x00000004	Undefined instruction	Undefined
0xFFFF0008	0x00000008	Software interrupt	Supervisor
0xFFFF000C	0x0000000C	Abort (prefetch)	Abort
0xFFFF0010	0x00000010	Abort (data)	Abort
0xFFFF0014	0x00000014	Reserved	Reserved
0xFFFF0018	0x00000018	IRQ	IRQ
0xFFFF001C	0x0000001C	FIQ	FIQ

```
b   reset_vector          @开机代码
b   _osExceptUndefInstrHdl  @异常处理之CPU碰到不认识的指令
b   _osExceptSwiHdl   @异常处理之:软中断
b   _osExceptPrefetchAbortHdl @异常处理之:取指异常
b   _osExceptDataAbortHdl  @异常处理之:数据异常
b   _osExceptAddrAbortHdl  @异常处理之:地址异常
b   OsIrqHandler    @异常处理之:硬中断
b   _osExceptFiqHdl    @异常处理之:快中断
```

解读

- `svc` 全称是 SuperVisor Call，完成工作模式的切换。不管之前是7个模式中的哪个模式，统一都切到 `SVC` 管理模式。但你也许会好奇，ARM软中断不是用 `SWI` 吗，这里怎么变成了 `SVC` 了，请看下面一段话，是从ARM官网翻译的：

  > SVC 超级用户调用。 语法 SVC{cond} #immed 其中： cond 是一个可选的条件代码（请参阅条件执行）。 immed 是一个表达式，其取值为以下范围内的一个整数： 在 ARM 指令中为 0 到 224–1（24 位值） 在 16 位 Thumb 指令中为 0-255（8 位值）。 用法 SVC 指令会引发一个异常。 这意味着处理器模式会更改为超级用户模式，CPSR 会保存到超级用户模式 SPSR，并且执行会跳转到 SVC 向量（请参阅《开发指南》中的第 6 章 处理处理器异常）。 处理器会忽略 immed。 但异常处理程序会获取它，借以确定所请求的服务。 Note 作为 ARM 汇编语言开发成果的一部分，SWI 指令已重命名为 SVC。 在此版本的 RVCT 中，SWI 指令反汇编为 SVC，并提供注释以指明这是以前的 SWI。 条件标记 此指令不更改标记。 体系结构 此 ARM 指令可用于所有版本的 ARM 体系结构。

- 而软中断对应的处理函数为 `_osExceptSwiHdl`，即PC寄存器将跳到 `_osExceptSwiHdl` 执行

5. _osExceptSwiHdl

```
@ Description: Software interrupt exception handler
_osExceptSwiHdl: @软中断异常处理
    @保存任务上下文(TaskContext) 开始... 一定要对照TaskContext来理解
    SUB     SP， SP， #(4 * 16) @先申请16个栈空间用于处理本次软中断
    STMIA   SP， {R0-R12} @TaskContext.R[GEN_REGS_NUM] STMIA从左到右执行，先放R0 .. R12
    MRS     R3， SPSR   @读取本模式下的SPSR值
    MOV     R4， LR    @保存回跳寄存器LR

    AND     R1， R3， #CPSR_MASK_MODE                   @ Interrupted mode 获取中断模式
    CMP     R1， #CPSR_USER_MODE                    @ User mode 是否为用户模式
    BNE     OsKernelSVCHandler                  @ Branch if not user mode 非用户模式下跳转
@ 当为用户模式时，获取SP和LR寄出去值
    @ we enter from user mode， we need get the values of  USER mode r13(sp) and r14(lr).
    @ stmia with ^ will return the user mode registers (provided that r15 is not in the register list).
```

```
    MOV     R0， SP          @获取SP值，R0将作为OsArmA32SyscallHandle的参数
    STMFD   SP!， {R3}                              @ Save the CPSR 入栈保存CPSR值 => TaskContext.regPSR
    ADD     R3， SP， #(4 * 17)                       @ Offset to pc/cpsr storage 跳到PC/CPSR存储位置
    STMFD   R3!， {R4}                              @ Save the CPSR and r15(pc) 保存LR寄存器 => TaskContext.PC
    STMFD   R3， {R13， R14}^                           @ Save user mode r13(sp) and r14(lr) 从右向左 保存 => TaskContext.LR和SP
    SUB     SP， SP， #4         @ => TaskContext.resved
    PUSH_FPU_REGS R1 @保存中断模式(用户模式模式)
  @保存任务上下文(TaskContext) 结束
    MOV     FP， #0                                 @ Init frame pointer
    CPSIE   I @开中断，表明在系统调用期间可响应中断
    BLX     OsArmA32SyscallHandle /*交给C语言处理系统调用，参数为R0，指向TaskContext的开始位置*/
    CPSID   I @执行后续指令前必须先关中断
  @恢复任务上下文(TaskContext) 开始
    POP_FPU_REGS R1          @弹出FP值给R1
    ADD     SP， SP，#4         @ 定位到保存旧SPSR值的位置
    LDMFD   SP!， {R3}                              @ Fetch the return SPSR 弹出旧SPSR值
    MSR     SPSR_cxsf， R3                          @ Set the return mode SPSR 恢复该模式下的SPSR值

    @ we are leaving to user mode， we need to restore the values of USER mode r13(sp) and r14(lr).
    @ ldmia with ^ will return the user mode registers (provided that r15 is not in the register list)

    LDMFD   SP!， {R0-R12}        @恢复R0-R12寄存器
    LDMFD   SP， {R13， R14}^                           @ Restore user mode R13/R14 恢复用户模式的R13/R14寄存器
    ADD     SP， SP， #(2 * 4)       @定位到保存旧PC值的位置
    LDMFD   SP!， {PC}^                             @ Return to user 切回用户模式运行
  @恢复任务上下文(TaskContext) 结束

OsKernelSVCHandler:@主要目的是保存ExcContext中除(R0~R12)的其他寄存器
    ADD     R0， SP， #(4 * 16) @跳转到保存PC，LR，SP的位置，此时R0位置刚好是SP的位置
    MOV     R5， R0    @由R5记录SP位置，因为R0要暂时充当SP寄存器来使用
    STMFD   R0!， {R4}                              @ Store PC => ExcContext.PC
    STMFD   R0!， {R4}         @ 相当于保存了=>  ExcContext.LR
    STMFD   R0!， {R5}         @ 相当于保存了=>  ExcContext.SP

    STMFD   SP!， {R3}                              @ Push task`s CPSR (i.e. exception SPSR). =>ExcContext.regPSR
    SUB     SP， SP， #(4 * 2)                        @ user sp and lr => =>ExcContext.USP，ULR

    MOV     R0， #OS_EXCEPT_SWI                         @ Set exception ID to OS_EXCEPT_SWI.
            @ 设置异常ID为软中断
    B       _osExceptionSwi                         @ Branch to global exception handler.
            @ 跳到全局异常处理
```

解读

- 运行到此处，已经切到SVC的栈运行，所以先保存上一个模式的现场
- 获取中断模式，软中断的来源可不一定是用户模式，完全有可能是SVC本身，比如系统调用中又发生系统调用。就变成了从SVC模式切到SVC的模式
- `MOV R0， SP` ;sp将作为参数传递给`OsArmA32SyscallHandle`
- 调用 `OsArmA32SyscallHandle` 这是所有系统调用的统一入口
- 注意看`OsArmA32SyscallHandle`的参数 `UINT32 *regs`

6. OsArmA32SyscallHandle

```
/* The SYSCALL ID is in R7 on entry.  Parameters follow in R0..R6 */
/*****************************************************************
由汇编调用，见于 los_hw_exc.S    / BLX    OsArmA32SyscallHandle
SYSCALL是产生系统调用时触发的信号，R7寄存器存放具体的系统调用ID，也叫系统调用号
regs:参数就是所有寄存器
注意:本函数在用户态和内核态下都可能被调用到
//MOV     R0， SP @获取SP值，R0将作为OsArmA32SyscallHandle的参数
*****************************************************************/
LITE_OS_SEC_TEXT UINT32 *OsArmA32SyscallHandle(UINT32 *regs)
{
    UINT32 ret;
    UINT8 nArgs;
    UINTPTR handle;
    UINT32 cmd = regs[REG_R7];//C7寄存器记录了触发了具体哪个系统调用
```

```
    if (cmd >= SYS_CALL_NUM) {//系统调用的总数
        PRINT_ERR("Syscall ID: error %d !!!\n", cmd);
        return regs;
    }

    if (cmd == __NR_sigreturn) {//收到 __NR_sigreturn 信号
        OsRestorSignalContext(regs);//恢复信号上下文
        return regs;
    }

    handle = g_syscallHandle[cmd];//拿到系统调用的注册函数，类似 SysRead
    nArgs = g_syscallNArgs[cmd / NARG_PER_BYTE]; /* 4bit per nargs */
    nArgs = (cmd & 1) ? (nArgs >> NARG_BITS) : (nArgs & NARG_MASK);//获取参数个数
    if ((handle == 0) || (nArgs > ARG_NUM_7)) {//系统调用必须有参数且参数不能大于8个
        PRINT_ERR("Unsupport syscall ID: %d nArgs: %d\n", cmd, nArgs);
        regs[REG_R0] = -ENOSYS;
        return regs;
    }
//regs[0-6] 记录系统调用的参数，这也是由R7寄存器保存系统调用号的原因
    switch (nArgs) {//参数的个数
        case ARG_NUM_0:
        case ARG_NUM_1:
            ret = (*(SyscallFun1)handle)(regs[REG_R0]);//执行系统调用，类似 SysUnlink(pathname);
            break;
        case ARG_NUM_2://如何是两个参数的系统调用，这里传三个参数也没有问题，因被调用函数不会去取用R2值
        case ARG_NUM_3:
            ret = (*(SyscallFun3)handle)(regs[REG_R0], regs[REG_R1], regs[REG_R2]);//类似 SysExecve(fileName, argv, envp);
            break;
        case ARG_NUM_4:
        case ARG_NUM_5:
            ret = (*(SyscallFun5)handle)(regs[REG_R0], regs[REG_R1], regs[REG_R2], regs[REG_R3],
                                         regs[REG_R4]);
            break;
        default: //7个参数的情况
            ret = (*(SyscallFun7)handle)(regs[REG_R0], regs[REG_R1], regs[REG_R2], regs[REG_R3],
                                         regs[REG_R4], regs[REG_R5], regs[REG_R6]);
    }

    regs[REG_R0] = ret;//R0保存系统调用返回值
    OsSaveSignalContext(regs);//保存信号上下文现场

    /* Return the last value of curent_regs.  This supports context switches on return from the exception.
     * That capability is only used with theSYS_context_switch system call.
     */
    return regs;//返回寄存器的值
}
```

解读

- 参数是 `regs` 对应的就是R0~Rn
- R7保存的是系统调用号，R0~R3保存的是 `SysMqOpen` 的四个参数
- `g_syscallHandle[cmd]` 就能查询到 `SYSCALL_HAND_DEF(__NR_mq_open, SysMqOpen, mqd_t, ARG_NUM_4)` 注册时对应的 `SysMqOpen` 函数
- `*(SyscallFun5)handle` 此时就是 `SysMqOpen`
- 注意看 SysMqOpen 的参数是最开始的 `main` 函数中的 `mqdes = mq_open(mqname, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR, &attr);` 由此完成了真正系统调用的过程

7. SysMqOpen

```
mqd_t SysMqOpen(const char *mqName, int openFlag, mode_t mode, struct mq_attr *attr)
{
    mqd_t ret;
    int retValue;
    char kMqName[PATH_MAX + 1] = { 0 };

    retValue = LOS_StrncpyFromUser(kMqName, mqName, PATH_MAX);
    if (retValue < 0) {
        return retValue;
```

```
    }
    ret = mq_open(kMqName， openFlag， mode， attr);//一个消息队列可以有多个进程向它读写消息
    if (ret == -1) {
        return (mqd_t)-get_errno();
    }
    return ret;
}
```

解读

- 此处的`mq_open`和main函数的`mq_open`其实是两个函数体实现。一个是给应用层的调用，一个是内核层使用，只是名字一样而已。
- `SysMqOpen`是返回到`OsArmA32SyscallHandle` `regs[REG_R0] = ret;`
- `OsArmA32SyscallHandle`再返回到`_osExceptSwiHdl`
- `_osExceptSwiHdl`后面的代码是用于恢复用户模式现场和`SPSR`，`PC`等寄存器。

以上为鸿蒙系统调用的整个过程。
关于寄存器(R0~R15)在每种模式下的使用方式，后续将由其他篇详细说明，敬请关注。

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断`debug`一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx`代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。
- 

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表	调度故事	任务控制块	内存规则
内核概念	进程控制块	并发并行	物理内存
源码结构	进程空间	就绪队列	内存概念
地址空间	线性区	调度机制	虚实映射
计时单位	红黑树	任务管理	页表管理
优雅的宏	进程管理	用栈方式	静态分配
钩子框架	Fork进程	软件定时器	TLFS算法
位图管理	进程回收	控制台	内存池管理
POSIX	Shell编辑	远程登录	原子操作
main函数	Shell解析	协议栈	圆整对齐
通讯机制	文件系统	硬件架构	内核汇编

通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期
同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha
issues:
#I45N42 建议 增加索引 一切时空过去未来
#I3VGJ7 一些链接失效 Rhenium
最近提交：
30a4d146 补充链接脚本的注解 kuangyufei17 hours
22a4bdde 完善链接脚本的注解 kuangyufei2 days
9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

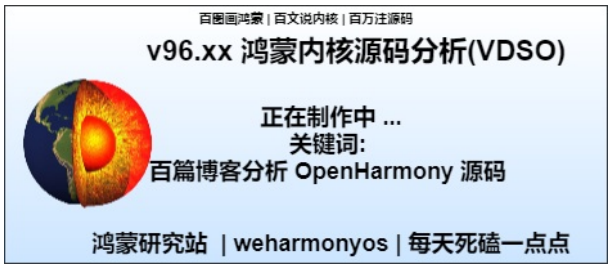
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

96_VDSO篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

编译运行相关篇为:

- v84.02 鸿蒙内核源码分析(编译过程) | 简单案例说透中间过程
- v85.03 鸿蒙内核源码分析(编译构建) | 编译鸿蒙防掉坑指南
- v86.04 鸿蒙内核源码分析(GN语法) | 如何构建鸿蒙系统
- v87.03 鸿蒙内核源码分析(忍者无敌) | 忍者的特点就是一个字
- v88.04 鸿蒙内核源码分析(ELF格式) | 应用程序入口并非main
- v89.03 鸿蒙内核源码分析(ELF解析) | 敢忘了她姐俩你就不是银
- v90.04 鸿蒙内核源码分析(静态链接) | 一个小项目看中间过程
- v91.04 鸿蒙内核源码分析(重定位) | 与国际接轨的对外发言人
- v92.01 鸿蒙内核源码分析(动态链接) | 正在制作中 ...
- v93.05 鸿蒙内核源码分析(进程映像) | 程序是如何被加载运行的
- v94.01 鸿蒙内核源码分析(应用启动) | 正在制作中 ...
- v95.06 鸿蒙内核源码分析(系统调用) | 开发者永远的口头禅
- v96.01 鸿蒙内核源码分析(VDSO) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理
双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

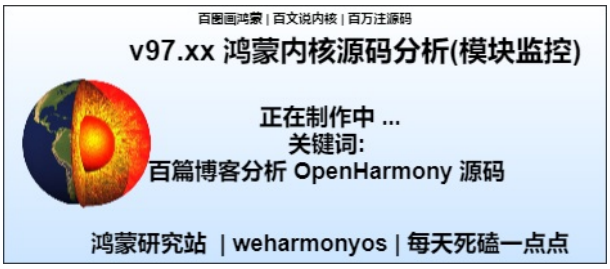
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

97_模块监控篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

调测工具相关篇为:

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引	一切时空过去未来
#I3VGJ7 一些链接失效	Rhenium

最近提交：

30a4d146 补充链接脚本的注解	kuangyufei17 hours
22a4bdde 完善链接脚本的注解	kuangyufei2 days
9b7c33c9 完善链接脚本的注解	kuangyufei3 days

master 分支 :2022-05-26 源码下载

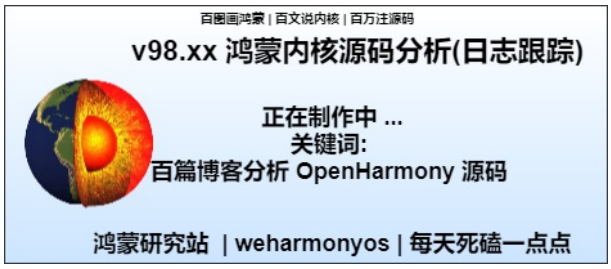
GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

98_日志跟踪篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

调测工具相关篇为:

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新，`v**.xx` 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

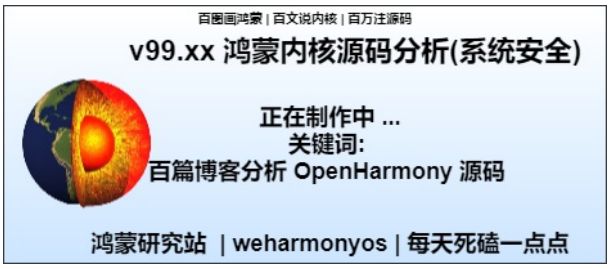
- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)

99_系统安全篇

本篇关键词：、、、



下载 >> 离线文档.鸿蒙内核源码分析(百篇博客分析.挖透鸿蒙内核).pdf

调测工具相关篇为:

- v97.01 鸿蒙内核源码分析(模块监控) | 正在制作中 ...
- v98.01 鸿蒙内核源码分析(日志跟踪) | 正在制作中 ...
- v99.01 鸿蒙内核源码分析(系统安全) | 正在制作中 ...
- v100.01 鸿蒙内核源码分析(测试用例) | 正在制作中 ...

站长正在努力制作中 ...，请客官稍等时日，可前往其他篇幅观看

百文说内核 | 抓住主脉络

- 百文相当于摸出内核的肌肉和器官系统，让人开始丰满有立体感，因是直接从注释源码起步，在加注释过程中，每每有心得处就整理,慢慢形成了以下文章。内容立足源码，常以生活场景打比方尽可能多的将内核知识点置入某种场景，具有画面感，容易理解记忆。说别人能听得懂的话很重要! 百篇博客绝不是百度教条式的在说一堆诘屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切。
- 与代码需不断 debug 一样，文章内容会存在不少错漏之处，请多包涵，但会反复修正，持续更新， v**.xx 代表文章序号和修改的次数，精雕细琢，言简意赅，力求打造精品内容。
- 百文在 < 鸿蒙研究站 | 开源中国 | 博客园 | 51cto | csdn | 知乎 | 掘金 > 站点发布，百篇博客系列目录如下。

按功能模块:

基础知识	进程管理	任务管理	内存管理

双向链表 内核概念 源码结构 地址空间 计时单位 优雅的宏 钩子框架 位图管理 POSIX main函数	调度故事 进程控制块 进程空间 线性区 红黑树 进程管理 Fork进程 进程回收 Shell编辑 Shell解析	任务控制块 并发并行 就绪队列 调度机制 任务管理 用栈方式 软件定时器 控制台 远程登录 协议栈	内存规则 物理内存 内存概念 虚实映射 页表管理 静态分配 TLFS算法 内存池管理 原子操作 圆整对齐
通讯机制	文件系统	硬件架构	内核汇编
通讯总览 自旋锁 互斥锁 快锁使用 快锁实现 读写锁 信号量 事件机制 信号生产 信号消费 消息队列 消息封装 消息映射 共享内存	文件概念 文件故事 索引节点 VFS 文件句柄 根文件系统 挂载机制 管道文件 文件映射 写时拷贝	芯片模式 ARM架构 指令集 协处理器 工作模式 寄存器 多核管理 中断概念 中断管理	编码方式 汇编基础 汇编传参 链接脚本 内核启动 进程切换 任务切换 中断切换 异常接管 缺页中断
编译运行	调测工具		
编译过程 编译构建 GN语法 忍者无敌 ELF格式 ELF解析 静态链接 重定位 动态链接 进程映像 应用启动 系统调用 VDSO	模块监控 日志跟踪 系统安全 测试用例		

百万注源码 | 处处扣细节

- 百万汉字注解内核目的是要看清楚其毛细血管，细胞结构，等于在拿放大镜看内核。内核并不神秘，带着问题去源码中找答案是很容易上瘾的，你会发现很多文章对一些问题的解读是错误的，或者说不深刻难以自圆其说，你会慢慢形成自己新的解读，而新的解读又会碰到新的问题，如此层层递进，滚滚向前，拿着放大镜根本不愿意放手。
- < gitee | github | coding | gitcode > 四大码仓推送 | 同步官方源码。

WeHarmony/kernel_liteos_a_note Star 1392|Fork 311

精读鸿蒙内核源码,百万汉字注解分析;百篇博客深入解剖,挖透内核地基工程.定期同步官方源码,输出覆盖主流站点.鸿蒙研究站 | 每天死磕一点点 => weha

issues:

#I45N42 建议 增加索引 一切时空过去未来

#I3VGJ7 一些链接失效 Rhenium

最近提交：

30a4d146 补充链接脚本的注解 kuangyufei17 hours

22a4bdde 完善链接脚本的注解 kuangyufei2 days

9b7c33c9 完善链接脚本的注解 kuangyufei3 days

master 分支 :2022-05-26 源码下载

GITEE.COM

关注不迷路 | 代码即人生

据说喜欢 **点赞 + 分享** 的,后来都成了大神。:)