

目录

目录	1
00_说明	16
几点说明	16
如何确认本文档为最新版本?	16
在线确认	16
致敬内核开发者	16
为何要精读内核源码?	17
01_双向链表篇	18
谁是鸿蒙内核最重要的结构体?	18
基本概念	19
功能接口	19
强大的宏	20
LOS_OFF_SET_OF 和 LOS_DL_LIST_ENTRY	20
OsGetTopTask	21
结构体的最爱	22
百篇博客.往期回顾	22
关于 51 .c .h .o	24
百万汉字注解.百篇博客分析	24
关注不迷路.代码即人生	24
02_进程管理篇	25
官方基本概念	25
进程状态说明 :	25
使用场景	26
开始正式分析	26
进程模块是如何初始化的	28
内核态根进程创建过程	29
用户态根进程创建过程	30
百篇博客.往期回顾	31
关于 51 .c .h .o	32
百万汉字注解.百篇博客分析	33
关注不迷路.代码即人生	33
03_时钟任务篇	34
时钟概念	34
时钟节拍的实现方式	34
第一:检查当前任务的时间片,任务执行一次分配多少时间呢?答案是2个时间片,即 20ms.	35
第二:扫描任务,主要是检查被阻塞的任务是否可以被重新调度	35
第三:定时器扫描,看是否有超时的定时器	36
最后看调度算法的实现	37
百篇博客.往期回顾	38
关于 51 .c .h .o	39
百万汉字注解.百篇博客分析	40
关注不迷路.代码即人生	40
04_任务调度篇	41
任务即线程	41
官方是怎么描述线程的	41
执行task命令	42
task长得什么样子	42
Task怎么管理	43
什么是任务池?	44

就绪队列是怎么回事	44
任务栈是怎么回事	45
任务栈初始化	46
Task函数集	46
使用场景和功能	46
创建任务的过程	47
百篇博客.往期回顾	49
关于 51 .c .h .o	51
百万汉字注解.百篇博客分析	51
关注不迷路.代码即人生	51
05_任务管理篇	52
任务即线程	52
官方是怎么描述线程的	52
执行task命令	53
task长得什么样子	53
Task怎么管理	54
什么是任务池？	55
就绪队列是怎么回事	55
任务栈是怎么回事	56
任务栈初始化	57
Task函数集	57
使用场景和功能	57
创建任务的过程	58
百篇博客.往期回顾	60
关于 51 .c .h .o	62
百万汉字注解.百篇博客分析	62
关注不迷路.代码即人生	62
06_调度队列篇	63
为何单独讲调度队列？	63
涉及函数	63
位图调度器	64
进程就绪队列机制	64
几个常用函数	64
同一个进程下的线程的优先级可以不一样吗？	65
task调度器	66
百篇博客.往期回顾	67
关于 51 .c .h .o	68
百万汉字注解.百篇博客分析	68
关注不迷路.代码即人生	69
07_调度机制篇	70
为什么学个东西要学那么多的概念？	70
进程和线程的状态迁移图	70
谁来触发调度工作？	71
源码告诉你调度过程是怎样的	72
请读懂OsGetTopTask()	73
百篇博客.往期回顾	74
关于 51 .c .h .o	75
百万汉字注解.百篇博客分析	75
关注不迷路.代码即人生	76
08_总目录	77

百篇博客.往期回顾	77
关于 51 .c .h .o	78
百万汉字注解.百篇博客分析	79
关注不迷路.代码即人生	79
09_调度故事篇	80
有个场馆	80
表演走什么流程？	80
西门大官人什么时候表演？	81
西门好事被破坏了怎么办了？	81
表演给谁看呢？	82
张大爷团队做什么的？	82
王场馆是做什么的？	82
李后勤是做什么的？	82
故事想说什么呢？	82
内核和故事的关系映射	82
请牢记这个故事	82
百篇博客.往期回顾	82
关于 51 .c .h .o	84
百万汉字注解.百篇博客分析	84
关注不迷路.代码即人生	84
10_内存主奴篇	86
主子和奴才	86
先说如果没有内存管理会怎样？	86
内存管理在管什么？	86
MMU是干什么事的？	86
举例说明	87
百篇博客.往期回顾	88
关于 51 .c .h .o	90
百万汉字注解.百篇博客分析	90
关注不迷路.代码即人生	90
11_内存分配篇	91
百篇博客.往期回顾	91
关于 51 .c .h .o	92
百万汉字注解.百篇博客分析	93
关注不迷路.代码即人生	93
12_内存管理篇	94
初始化整个内存	94
鸿蒙虚拟内存整体布局图	94
内核空间是怎么初始化的？	95
Page是如何初始化的？	96
进程是如何申请内存的？	97
task是如何申请内存的？	98
百篇博客.往期回顾	99
关于 51 .c .h .o	101
百万汉字注解.百篇博客分析	101
关注不迷路.代码即人生	101
13_源码注释篇	102
同步官方源码历史	102
几点说明	102
为何要精读内核源码？	103

热爱是所有的理由和答案	103
(ㄥ・■■■)・■■■)ゞ鸿蒙内核开发者	103
理解内核的三个层级	103
加注方式是怎样的？	103
有哪些特殊的记号	105
新增zzz目录	105
百篇博客.往期回顾	105
关于 51 .c .h .o	107
百万汉字注解.百篇博客分析	107
关注不迷路.代码即人生	107
14_内存汇编篇	108
ARM-CP15协处理器	108
先拆解一段汇编代码	108
CP15有哪些寄存器	109
TTB寄存器(Translation table base)	109
mmu上下文	110
TLB (translation lookaside buffer)	111
asid寄存器	112
百篇博客.往期回顾	113
关于 51 .c .h .o	114
百万汉字注解.百篇博客分析	115
关注不迷路.代码即人生	115
15_内存映射篇	116
MMU的本质	116
一级页表L1	116
LOS_ArchMmuQuery	116
二级页表L2	117
映射初始化的过程	118
OsSetKSectionAttr 内核空间的设置和映射	118
LOS_ArchMmuMap	120
百篇博客.往期回顾	121
关于 51 .c .h .o	122
百万汉字注解.百篇博客分析	122
关注不迷路.代码即人生	122
16_内存规则篇	124
主子 and 奴才	124
先说如果没有内存管理会怎样？	124
内存管理在管什么？	124
MMU是干什么事的？	124
举例说明	125
百篇博客.往期回顾	126
关于 51 .c .h .o	128
百万汉字注解.百篇博客分析	128
关注不迷路.代码即人生	128
17_物理内存篇	129
如何初始化物理内存？	129
如何分配/回收物理内存？ 答案是伙伴算法	130
百篇博客.往期回顾	133
关于 51 .c .h .o	135
百万汉字注解.百篇博客分析	135

关注不迷路.代码即人生	135
18_源码结构篇	136
鸿蒙内核源码注解分析	136
百篇博客.往期回顾	138
关于 51 .c .h .o	139
百万汉字注解.百篇博客分析	140
关注不迷路.代码即人生	140
19_位图管理篇	141
为何进程和线程都是32个优先级？	141
应用开发和内核开发有哪些区别？	141
什么是位图管理器？	142
位图在哪些地方应用？	142
编程实例	143
结果验证	143
百篇博客.往期回顾	144
关于 51 .c .h .o	145
百万汉字注解.百篇博客分析	145
关注不迷路.代码即人生	146
20_用栈方式篇	147
百篇博客.往期回顾	149
关于 51 .c .h .o	150
百万汉字注解.百篇博客分析	151
关注不迷路.代码即人生	151
21_线程概念篇	152
本篇说清楚任务的问题	152
第一大块:多核CPU相关块	153
第二大块:栈空间	153
第三大块:资源竞争/同步	155
第四大块:任务调度	155
第五大块:任务间通讯	155
第六大块:辅助工具	156
百篇博客.往期回顾	156
关于 51 .c .h .o	157
百万汉字注解.百篇博客分析	158
关注不迷路.代码即人生	158
22_汇编基础篇	159
汇编很简单	159
square(c -> 汇编)	159
fp(c -> 汇编)	160
main(c -> 汇编)	160
文件全貌	161
先看最短的那个	161
入参方式	161
追问三个问题	162
百篇博客.往期回顾	162
关于 51 .c .h .o	164
百万汉字注解.百篇博客分析	164
关注不迷路.代码即人生	164
23_汇编传参篇	165
汇编如何传复杂的参数？	165

入参方式	166
memcpy汇编调用	166
逐句分析 framePoint	167
总结	167
百篇博客.往期回顾	167
关于 51 .c .h .o	169
百万汉字注解.百篇博客分析	169
关注不迷路.代码即人生	169
24_进程概念篇	171
本篇说清楚进程	171
官方基本概念	171
官方概念解读	171
ProcessCB真身	171
第一大块:和任务(线程)关系	172
第二大块:和其他进程的关系	173
第三大块:进程的五种状态	173
第四大块:和内存的关系	174
第五大块:和文件的关系	174
第六大块:辅助工具	174
百篇博客.往期回顾	175
关于 51 .c .h .o	176
百万汉字注解.百篇博客分析	177
关注不迷路.代码即人生	177
25_并行并发篇	178
本篇说清楚并发并行	178
理解并发概念	178
理解并行概念	178
理解协程概念	178
内核如何描述CPU	179
LOSCFG_KERNEL_SMP	179
多CPU核支持	179
1.OsMplnit	180
2.次级CPU的初始化	180
多CPU核还有哪些问题?	181
百篇博客.往期回顾	181
关于 51 .c .h .o	183
百万汉字注解.百篇博客分析	183
关注不迷路.代码即人生	183
26_自旋锁篇	185
本篇说清楚自旋锁	185
概述	185
自旋锁长什么样?	186
自旋锁使用流程	186
几个关键函数	186
ArchSpinLock 汇编代码	186
ArchSpinTrylock 汇编代码	187
ArchSpinUnlock 汇编代码	187
汇编指令之 WFI / WFE / SEV	187
汇编指令之 LDREX / STREX	188
编程实例	188

运行结果	190
总结	190
百篇博客.往期回顾	190
关于 51 .c .h .o	191
百万汉字注解.百篇博客分析	192
关注不迷路.代码即人生	192
27_互斥锁篇	193
本篇说清楚互斥锁	193
概述	193
互斥锁长什么样?	194
初始化	194
三种申请模式	194
申请互斥锁主函数 OsMuxPendOp	195
释放锁的主体函数 OsMuxPostOp	195
编程实例	196
结果验证	198
总结	198
百篇博客.往期回顾	198
关于 51 .c .h .o	200
百万汉字注解.百篇博客分析	200
关注不迷路.代码即人生	200
28_进程通讯篇	201
进程间为何要通讯?	201
大致有以下几种通讯需求:	201
内核目录和系列篇更新	202
进程间九种通讯方式	202
1.管道pipe(fs_syscall.c)	202
2.信号(los_signal.c)	202
3.消息队列(los_queue.c)	203
4.共享内存(shm.c)	203
5.信号量(los_sem.c)	203
6.互斥锁 (los_mux.c) :	204
7.快锁 (los_futex.c)	204
8.事件 (los_event.c)	204
9.文件消息队列 (hm_liteipc.c)	204
百篇博客.往期回顾	204
关于 51 .c .h .o	206
百万汉字注解.百篇博客分析	206
关注不迷路.代码即人生	206
29_信号量篇	208
本篇说清楚信号量	208
基本概念	208
信号量运作原理	208
信号量长什么样?	209
初始化信号量模块	209
创建信号量	209
申请信号量	210
释放信号量	211
编程示例	212
实例运行结果:	214

百篇博客.往期回顾	214
关于 51 .c .h .o	216
百万汉字注解.百篇博客分析	216
关注不迷路.代码即人生	216
30_事件控制篇	217
本篇说清楚事件 (Event)	217
官方概述	217
再看事件图	217
事件控制块长什么样?	218
事件控制块<>事件<>任务 三者关系	218
函数列表	219
事件初始化 -> LOS_EventInit	219
事件生产过程 -> OsEventWrite	220
事件消费过程 -> OsEventRead	221
编程实例	222
运行结果	223
百篇博客.往期回顾	224
关于 51 .c .h .o	225
百万汉字注解.百篇博客分析	225
关注不迷路.代码即人生	226
31_定时器篇	227
本篇说清楚定时器的实现	227
运作机制	227
定时器长什么样?	227
定时器分类	228
定时器怎么管理?	228
初始化 -> OsSwtmrInit	228
定时任务 -> 最高优先级	230
队列消费者 -> OsSwtmrTask	230
队列生产者 -> OsSwtmrScan	231
总结	232
百篇博客.往期回顾	232
关于 51 .c .h .o	234
百万汉字注解.百篇博客分析	234
关注不迷路.代码即人生	234
32_CPU篇	235
本篇说清楚CPU	235
Percpu	235
百篇博客.往期回顾	238
关于 51 .c .h .o	239
百万汉字注解.百篇博客分析	240
关注不迷路.代码即人生	240
33_消息队列篇	241
本篇说清楚消息队列	241
基本概念	241
队列特性	241
消息队列长什么样?	241
初始化队列	242
创建队列	243
关键函数OsQueueOperate	244

编程实例	245
结果验证	247
总结	247
百篇博客.往期回顾	247
关于 51 .c .h .o	249
百万汉字注解.百篇博客分析	249
关注不迷路.代码即人生	249
34_原子操作篇	250
本篇说清楚原子操作	250
基本概念	250
ArchSpinLock 申请锁代码	250
ArchSpinUnlock 释放锁代码	251
运作机制	251
功能列表	251
LOS_AtomicAdd	252
LOS_AtomicSub	253
volatile	253
编程实例	253
结果验证	254
百篇博客.往期回顾	254
关于 51 .c .h .o	256
百万汉字注解.百篇博客分析	256
关注不迷路.代码即人生	256
35_时间管理篇	258
本篇说清楚时间概念	258
Tick硬中断函数	259
功能函数	260
编程示例	260
百篇博客.往期回顾	261
关于 51 .c .h .o	263
百万汉字注解.百篇博客分析	263
关注不迷路.代码即人生	263
36_工作模式篇	264
本篇说清楚CPU的工作模式	264
七种模式	264
1.异常模式栈空间怎么申请?	266
2.异常模式入口地址在哪?	266
开机代码	268
异常的优先级	268
3.异常模式怎么切换?	268
CPSR 寄存器	268
SPSR 寄存器	269
百篇博客.往期回顾	270
关于 51 .c .h .o	271
百万汉字注解.百篇博客分析	271
关注不迷路.代码即人生	271
37_系统调用篇	273
本篇说清楚系统调用	273
七段追踪代码, 逐个分析	274
1.应用程序 main	274

2. mq_open 发起系统调用	274
3. syscall	275
4. svc 0	275
5. _osExceptSwiHdl	277
6. OsArmA32SyscallHandle	278
7. SysMqOpen	279
百篇博客.往期回顾	279
关于 51 .c .h .o	281
百万汉字注解.百篇博客分析	281
关注不迷路.代码即人生	281
38_寄存器篇	282
本篇说清楚寄存器	282
寄存器的本质	282
七种工作模式	283
R0~R7 寄存器	284
R7 寄存器	286
fp(R11) 寄存器	286
SP(R13) 寄存器	286
LR(R14) 寄存器	286
PC(R15) 寄存器	286
CPSR 寄存器	287
SPSR 寄存器	288
留个问题	288
百篇博客.往期回顾	288
关于 51 .c .h .o	289
百万汉字注解.百篇博客分析	290
关注不迷路.代码即人生	290
39_异常接管篇	291
为何要有异常接管?	291
七种工作模式	291
官方概念	292
进入和退出异常方式	293
栈帧	294
六种异常模式实现代码	295
地址异常处理(Address abort)	295
快中断处理(fiq)	295
取指异常(Prefectch abort)	296
数据访问异常(Data abort)	296
软中断处理(swi)	296
普通中断处理(irq)	297
未定义异常处理(undef)	297
异常分发统一处理	298
非常重要的ARM37个寄存器	298
结尾	299
百篇博客.往期回顾	299
关于 51 .c .h .o	301
百万汉字注解.百篇博客分析	301
关注不迷路.代码即人生	301
40_汇编汇总篇	302
汇编其实很可爱	302

汇编目录	302
hw_user_get.S	302
reset_vector_mp.S 和 reset_vector_up.S	303
los_dispatch.S 和 los_hw_exc.S	304
jmp.S	304
los_hw_runstop.S	304
cache.S	304
百篇博客.往期回顾	305
关于 51 .c .h .o	307
百万汉字注解.百篇博客分析	307
关注不迷路.代码即人生	307
41_任务切换篇	308
本篇说清楚线程环境下的任务切换	308
前置条件	309
TaskContext 任务上下文	309
OsSchedResched 调度算法	310
OsTaskSchedule 汇编实现	310
百篇博客.往期回顾	311
关于 51 .c .h .o	313
百万汉字注解.百篇博客分析	313
关注不迷路.代码即人生	313
42_中断切换篇	315
中断环境下的任务切换	315
普通中断模式相关寄存器	315
TaskIrqContext 和 TaskContext	316
普通中断处理程序	317
百篇博客.往期回顾	319
关于 51 .c .h .o	321
百万汉字注解.百篇博客分析	321
关注不迷路.代码即人生	321
43_中断概念篇	322
中断概念	322
什么是中断?	322
中断相关的硬件介绍	322
中断源	323
中断类型	324
中断请求	324
中断触发	324
中断优先级	324
中断处理程序	324
中断向量表	324
用户中断服务程序(ISR)	325
中断嵌套	325
中断共享	325
核间中断	325
功能API	326
百篇博客.往期回顾	326
关于 51 .c .h .o	328
百万汉字注解.百篇博客分析	328
关注不迷路.代码即人生	328

44_中断管理篇	329
编译开关	329
中断初始化	329
中断相关的结构体	332
注册硬中断	332
中断怎么触发的?	334
中断统一处理入口函数 HallrqHandler	334
百篇博客.往期回顾	336
关于 51 .c .h .o	337
百万汉字注解.百篇博客分析	337
关注不迷路.代码即人生	338
45_fork篇	339
fork是什么	339
运行结果	340
为什么是fork	341
fork怎么实现的?	341
百篇博客.往期回顾	344
关于 51 .c .h .o	346
百万汉字注解.百篇博客分析	346
关注不迷路.代码即人生	346
46_特殊进程篇	348
三个进程	348
家族式管理	348
2号进程 KProcess	349
0 号进程 KIdle	350
1号进程 init	352
百篇博客.往期回顾	354
关于 51 .c .h .o	356
百万汉字注解.百篇博客分析	356
关注不迷路.代码即人生	356
47_进程回收篇	357
进程关系链	357
进程正常死亡过程	357
孤儿进程	359
僵尸进程	359
waitpid	359
百篇博客.往期回顾	363
关于 51 .c .h .o	365
百万汉字注解.百篇博客分析	365
关注不迷路.代码即人生	365
48_信号生产篇	366
信号生产	366
信号分类	366
信号来源	367
信号与进程的关系	367
信号与任务的关系	368
信号发送过程	369
代码细节	369
信号相关函数	372
百篇博客.往期回顾	372

关于 51 .c .h .o	374
百万汉字注解.百篇博客分析	374
关注不迷路.代码即人生	374
49_信号消费篇	375
信号消费	375
sig_switch_context	376
OsArmA32SyscallHandle 系统调用总入口	376
OsSaveSignalContext 保存信号上下文	378
OsRestorSignalContext 恢复信号上下文	379
百篇博客.往期回顾	381
关于 51 .c .h .o	382
百万汉字注解.百篇博客分析	382
关注不迷路.代码即人生	383
50_编译环境篇	384
几点说明	384
编译鸿蒙	384
编译环境	385
编译前提	386
编译过程 安装 hb	386
编译命令 hb set	387
编译命令 hb env	388
编译命令 hb build	388
编译命令 hb clean	389
编译输出 out 目录	389
用户进程	392
百篇博客.往期回顾	393
关于 51 .c .h .o	395
百万汉字注解.百篇博客分析	395
关注不迷路.代码即人生	395
51_ELF格式篇	396
阅读之前的说明	396
示例代码	397
名正才言顺	398
ELF历史	398
ELF整体布局	398
ELF头信息	399
段(Segment)头信息	400
区表	402
String Table	404
符号表 Symbol Table	404
反汇编代码区	406
百篇博客.往期回顾	407
关于 51 .c .h .o	409
百万汉字注解.百篇博客分析	409
关注不迷路.代码即人生	409
52_静态站点篇	410
几点说明	410
OpenHarmony开发者文档	410
侧边栏	411
主题色	412

搜索极为便利，国内外双站点	412
中英文切换	413
带私货	413
百篇博客.往期回顾	414
关于 51 .c .h .o	416
百万汉字注解.百篇博客分析	416
关注不迷路.代码即人生	416
53_ELF解析篇	417
readelf -S app	417
区名称 Section Head Name	419
区类型 Section Head Type	420
区标签 Section Head Flag	420
readelf -s app	421
符号表绑定 Symbol Table Bind	422
符号表类型 Symbol Table Type	423
符号表可见性 Symbol Table Visibility	423
符号表索引 Symbol Table Ndx	423
百篇博客.往期回顾	423
关于 51 .c .h .o	425
百万汉字注解.百篇博客分析	425
关注不迷路.代码即人生	425
54_静态链接篇	427
准备工作	427
目录结构	427
cat .c .h	428
cat Makefile	428
编译.链接.运行.看结果	429
开始分析	429
readelf 大S小s ./obj/main.o	429
readelf 大S小s ./obj/part.o	430
readelf 大S小s ./bin/weharmony	431
百篇博客.往期回顾	433
关于 51 .c .h .o	434
百万汉字注解.百篇博客分析	435
关注不迷路.代码即人生	435
55_重定位篇	436
什么是重定位	436
重定位十种类型	436
objdump命令	437
objdump -S ./obj/main.o	438
objdump -r ./obj/main.o	439
objdump -sj .rodata ./obj/main.o	439
objdump -S ./bin/weharmony	440
objdump -R ./bin/weharmony	442
百篇博客.往期回顾	442
关于 51 .c .h .o	444
百万汉字注解.百篇博客分析	444
关注不迷路.代码即人生	444
56_进程映像篇	446
LOS_DoExecveFile	446

解析，装载过程	447
ELFLoadInfo	448
加载过程(OsLoadELFFile)	448
运行	450
百篇博客.往期回顾	450
关于 51 .c .h .o	452
百万汉字注解.百篇博客分析	452
关注不迷路.代码即人生	452
readme	454
关于 51 .c .h .o	455
百万汉字注解.百篇博客分析	455
关注不迷路.代码即人生	455

00_说明

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v00.xx 鸿蒙内核源码分析(版本说明篇) | 如何获取最新文档 | 51 .c .h .o

几点说明

- 百万汉字注解仓库: `kernel_liteos_a_note` 是在 OpenHarmony 的 `kernel_liteos_a` (鸿蒙轻内核项目)基础上给源码加上中文注解的版本.加注版与官方最新源码保持同步.
- 百篇博客分析地址:
 - 国内: <https://weharmony.gitee.io/weharmony>
 - 国外: <https://weharmony.github.io/weharmony>
- OpenHarmony开发者文档 是对官方文档 docs 做的非常炫酷的静态站点,支持侧边栏/面包屑/搜索/中英文,非常方便的查看官方文档,大大提高学习和开发效率.
 - 国内: <https://weharmony.gitee.io/openharmony>
 - 国外: <https://weharmony.github.io/openharmony>
- 鸿蒙全部源码仓库: OpenHarmony 是HarmonyOS 110+个子项目的所有源码汇总. 因HarmonyOS使用 repo 管理众多 git 项目, repo 在 linux 下很方便,但在 windows 上使用会有相当的困难,所以将所有子项目整合成一个.git工程,如此windows用户使用熟悉的 git 方式便能下载整个鸿蒙系统源码,方便学习使用.仓库与官方仓库保持同步.已编译通过.

```
....
[OHOS INFO] [1587/1590] STAMP obj/test/xts/acts/build_lite/acts_generate_module_data.stamp
[OHOS INFO] [1588/1590] ACTION //test/xts/acts/build_lite:acts//build/lite/toolchain:linux_x86_64_ohos_clang
[OHOS INFO] [1589/1590] STAMP obj/test/xts/acts/build_lite/acts.stamp
[OHOS INFO] [1590/1590] STAMP obj/build/lite/ohos.stamp
[OHOS INFO] ipcamera_hispanic_aries build success
root@5e3abe332c5a:/home/harmony#
```

- 下载.鸿蒙源码分析.离线文档 < 国内 | 国外 >
- 加入兴趣小组.微信群聊 < 国内 | 国外 >

如何确认本文档为最新版本?

- 本文档版本: 2021/05/28

在线确认

- 查询历史版本 < gitee | github >
- 链接一:<https://weharmony.gitee.io/history/>
- 链接二:<https://weharmony.github.io/history/>

致敬内核开发者

- 感谢开放原子开源基金会,致敬鸿蒙内核开发者.
- 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 计算机组成原理 五门课程的教学项目.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.

为何要精读内核源码?

- 每位码农的学职生涯,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,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,高密度的记忆网,训练大脑肌肉记忆,将整个内核知识点变成大脑常量,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.

01_双向链表篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- 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 | 51 .c .h .o

谁是鸿蒙内核最重要的结构体？

答案一定是：LOS_DL_LIST (双向链表)，它长这样.

```
typedef struct LOS_DL_LIST { //双向链表，内核最重要结构体
    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;
```

结构体够简单了吧，只有前后两个指向自己的指针，但恰恰是因为太简单，所以才太不简单. 就像氢原子一样，宇宙中无处不在，占比最高，原因是因为它最简单，最稳定!

内核的各自模块都能看到双向链表的身影，下图是各处初始化双向链表的操作，因为太多了，只截取了部分：

	los_list.h (kernel\include)
LOS_ListInit	vid.c (security\vid)
VidMapListInit	fs_file_mapping.c (fs\vfs\operation)
init_file_mapping	los_vm_page.c (kernel\base\vm)
OsVmPageInit	los_multipldlinkhead.c (kernel\base\mem\bestfit)
OsDlnkInitMultiHead	los_event.c (kernel\base\ipc)
LOS_EventInit	los_sortlink.c (kernel\base\core)
OsSortLinkInit	los_priqueue.c (kernel\base\sched\sched_sq)
OsPriQueueInit	los_sem.c (kernel\base\ipc)
OsSemInit	mtd_partition.c (fs\vfs\multi_partition\src)
MtdNorParamAssign	los_queue.c (kernel\base\ipc)
OsQueueInit	los_vm_phys.c (kernel\base\vm)
OsVmPhysLrulInit	los_futex.c (kernel\base\ipc)
OsFutexInit	fs_file_mapping.c (fs\vfs\operation)
add_mapping	los_sem.c (kernel\base\ipc)
OsSemCreate	los_task.c (kernel\base\core)
OsSetMainTask	shm.c (kernel\base\vm)
ShmInit	los_swtmr.c (kernel\base\core)
OsSwtmrInit	los_vm_map.c (kernel\base\vm)
OsVmSpacelnitCommon	los_queue.c (kernel\base\ipc)
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)

很多人问图怎么来的，source insight 4.0 是阅读大型 C/C++ 工程的必备工具，要用4.0否则中文有乱码.

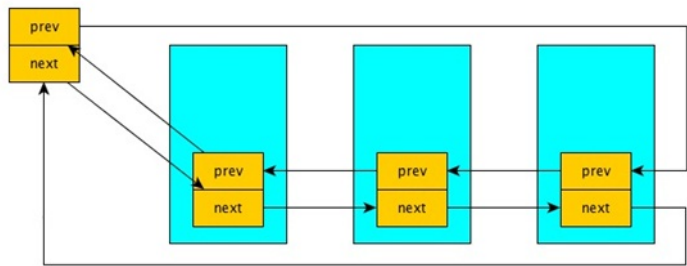
可以豪不夸张的说理解 LOS_DL_LIST 及相关函数是读懂鸿蒙内核的关键。前后指针(注者后续将比喻成一对左右触手)灵活的指挥着系统精准的运行，

越是深入分析内核源码，越能感受到内核开发者对 LOS_DL_LIST 非凡的驾驭能力，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！这么重要的结构体还是需详细讲解一下。

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，还增加一个指向其前一个节点的指针。其头指针 head 是唯一确定的。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，但需要注意前后方向的操作。

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。



功能接口

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	LOSListAdd	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

请结合下面的代码和图去理解双向链表，不管花多少时间，一定要理解它的插入/删除动作， 否则后续内容将无从谈起。

```
//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;
}

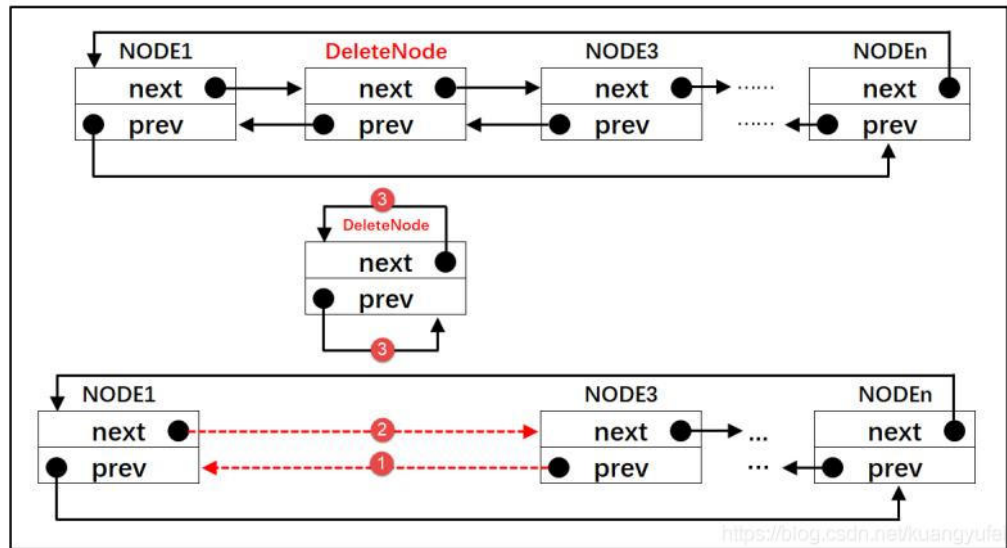
//将指定节点从链表中删除，并使用该节点初始化链表
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListDelInit(LOS_DL_LIST *list)
{

```

```

list->pstNext->pstPrev = list->pstPrev;
list->pstPrev->pstNext = list->pstNext;
LOS_ListInit(list);
}

```



此处仅列出 `LOS_ListDelInit` 图

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 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. 以 ProcessCB 结构体举例

```
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
    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 the priority hash table */ //进
    LOS_DL_LIST    waitList;          /**< The process holds the waitLists to support wait/waitpid */ //进程持有等待链表以支持wait/waitpid
} 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 8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8双(16只)触手。

```
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 ，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!

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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， .xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o

- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o

- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要 CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

02_进程管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

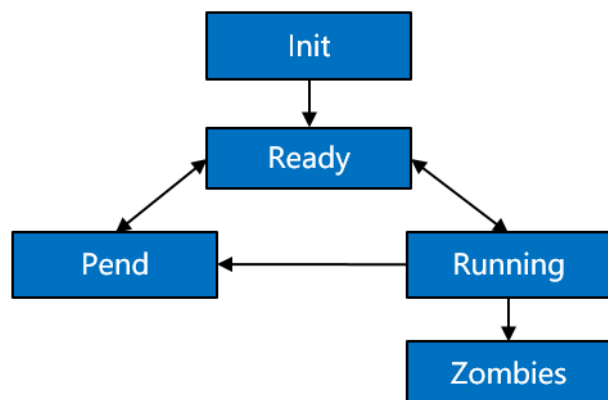
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51 .c .h .o

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。
- OpenHarmony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。
- OpenHarmony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。
- OpenHarmony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。
- 用户态根进程init由内核态创建，其它用户态进程均由init进程fork而来。

进程状态说明：

- 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。



- 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：
 - 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。
- 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。

使用场景

进程创建后，用户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）。用户态允许进程挂起，恢复，延时等操作，同时也可以设置用户态进程调度优先级和调度策略，获取进程调度优先级和调度策略。进程结束的时候，进程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。

开始正式分析

对应张大爷的故事，进程就是那些在场馆外32个队列里排队的，那些队列就是进程的就绪队列。

请注意 进程是资源管理单元，而非最终调度单元，调度单元是谁？是 `Task`，看下官方对应状态定义

```
#define OS_PROCESS_STATUS_INIT      0x0010U  //进程初始状态
#define OS_PROCESS_STATUS_READY    0x0020U  //进程就绪状态
#define OS_PROCESS_STATUS_RUNNING  0x0040U  //进程运行状态
#define OS_PROCESS_STATUS_PEND     0x0080U  //进程阻塞状态
#define OS_PROCESS_STATUS_ZOMBIES  0x100U   //进程僵死状态
```

一个进程从创建到消亡过程，在内核肯定是极其复杂的。为了方便理解进程，整个系列文章笔者会用张大爷的故事打比方，从生活中的例子来将神秘的系统内核外化解剖出来给大家看。一件这么复杂的事情肯定会有个复杂的结构体来承载，它就是LosProcessCB(进程控制块)，代码很长但必须全部拿出来。

```
LITE_OS_SEC_BSS LosProcessCB *g_runProcess[LOSCFG_KERNEL_CORE_NUM]; //用一个指针数组记录进程运行，LOSCFG_KERNEL_CORE_NUM 为 CPU的
LITE_OS_SEC_BSS LosProcessCB *g_processCBArray = NULL; //进程池，最大进程数为 64个
LITE_OS_SEC_DATA_INIT STATIC LOS_DL_LIST g_freeProcess; //记录空闲的进程链表
LITE_OS_SEC_DATA_INIT STATIC LOS_DL_LIST g_processRecyleList; //记录回收的进程列表

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表示了二层逻辑 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,就由此节点挂到等待链
    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)
    ProclpClnfo  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;

```

进程的模式有两种，内核态和用户态，能想到 `main` 函数中肯定会创建一个内核态的最高优先级进程，他就是 `KProcess`

通过 `task` 命令查看任务运行状态，可以看到 `KProcess` 进程，看名字就知道是一个内核进程，在系统启动时创建，图中可以看到 `KProcess` 的 `task` 运行情况，从表里可以看到 `KProcess` 内有 10 几个 `task`

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	tcip_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	0x46a39	shellTask	
22	3	0x3	-1	Pend	0x3000	0x924	0x450	ShellEntry	

进程模块是如何初始化的

KProcess 在张大爷的故事里相当于场馆的工作人员，他们也要接受张大爷的调度排队进场，但他们的优先级是最高的0级，他们进场后需完成场馆的准备工作，再开门做生意。如果需要多个工作人员怎么办，就是通过fork,简单说就是复制一个，复制的前提是需要有一个，鸿蒙里就是KProcess，其他工作人员都是通过它fork的。那用户怎么来的呢？就是真正要排队的人也是一样，先创建一个用户祖先，其他用户皆由祖先fork来的。注意用户进程和内核进程的祖先是不同的，有各自的祖先根。分别是g_userInitProcess(1号) 和 g_kernelInitProcess(2号)

```

/*****
并发（Concurrent）：多个线程在单个核心运行，同一时间一个线程运行，系统不停切换线程，
看起来像同时运行，实际上是线程不停切换
并行（Parallel）每个线程分配给独立的CPU核心，线程同时运行
单核CPU多个进程或多个线程内能实现并发（微观上的串行，宏观上的并行）
多核CPU线程间可以实现宏观和微观上的并行
LITE_OS_SEC_BSS 和 LITE_OS_SEC_DATA_INIT 是告诉编译器这些全局变量放在哪个数据段
*****/

LITE_OS_SEC_BSS LosProcessCB *g_runProcess[LOSCFG_KERNEL_CORE_NUM]; // CPU内核个数,超过一个就实现了并行
LITE_OS_SEC_BSS LosProcessCB *g_processCBArray = NULL; // 进程池数组
LITE_OS_SEC_DATA_INIT STATIC LOS_DL_LIST g_freeProcess; // 空闲状态下的进程链表, 个人觉得应该取名为 g_freeProcessList @note_thinking
LITE_OS_SEC_DATA_INIT STATIC LOS_DL_LIST g_processRecyleList; // 需要回收的进程列表
LITE_OS_SEC_BSS UINT32 g_userInitProcess = OS_INVALID_VALUE; // 用户态的初始init进程,用户态下其他进程由它 fork
LITE_OS_SEC_BSS UINT32 g_kernelInitProcess = OS_INVALID_VALUE; // 内核态初始Kprocess进程,内核态下其他进程由它 fork
LITE_OS_SEC_BSS UINT32 g_kernelIdleProcess = OS_INVALID_VALUE; // 内核态idle进程,由Kprocess fork
LITE_OS_SEC_BSS UINT32 g_processMaxNum; // 进程最大数量,默认64个
LITE_OS_SEC_BSS ProcessGroup *g_processGroup = NULL; // 全局进程组,负责管理所有进程组

//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]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_FROM_I
}

g_userInitProcess = 1; /* 1: The root process ID of the user-mode process is fixed at 1 *///用户模式的根进程
LOS_ListDelete(&g_processCBArray[g_userInitProcess].pendList);// 清空g_userInitProcess pend链表

g_kernelInitProcess = 2; /* 2: The root process ID of the kernel-mode process is fixed at 2 *///内核模式的根进程
LOS_ListDelete(&g_processCBArray[g_kernelInitProcess].pendList);// 清空g_kernelInitProcess pend链表

return LOS_OK;
}

```

代码已经很清楚, 创建了一个进程池, 默认64个进程, 也就是不改宏 LOSCFG_BASE_CORE_PROCESS_LIMIT 的情况下 系统最多是64个进程, 但有两个进程先被占用, 用户态和内核态各一个, 他们是后续创建进程的根, 所以最多留给外面的只有 62个进程可创建, 代码的最后两个根进程的task阻塞链表被清空了, 因为没有阻塞任务当然要清空。

内核态根进程创建过程

创建"Kprocess"进程, 也就是线程池中的2号进程g_kernelInitProcess, 设为最高优先级 0

```

//初始化 2号进程,即内核根进程
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级为内核
    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
}

//创建一个名叫"Idle"的进程,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"Idle"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, "Idle", (TSK_ENTRY_FUNC)OsIdleTask, LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE);
    if (ret < 0) {
        return LOS_NOK;
    }
    g_kernelIdleProcess = (UINT32)ret;//返回进程ID

    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;
}

```

用户态根进程创建过程

创建 Init 进程，也就是线程池中的1号进程 g_userInitProcess，优先级为 28,好低啊

```

/**
 * @ingroup los_process
 * User state root process default priority
 */
#define OS_PROCESS_USERINIT_PRIORITY 28

//所有的用户进程都是使用同一个用户代码段描述符和用户数据段描述符，它们是__USER_CS和__USER_DS，也就是每个进程处于用户态时，它们的CS寄存器和DS寄存器
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,所有进程
    CHAR *userInitBssStart = (CHAR *)&__user_init_bss;// 未初始化数据区（BSS）。在运行时改变其值
    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);
    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程
    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 -> userText
    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);// 初始化堆栈区
    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;// 指向栈顶
    param.userParam.userMapBase = (UINTPTR)stack;// 栈底
    param.userParam.userMapSize = size;// 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
    OsDelInitPCB(processCB); //删除PCB块
    return ret;
}

```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o

- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c .h .o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o ，我要CHO ，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

03_时钟任务篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51 .c .h .o

时钟概念

- 时间是非常重要的概念，我们整个学生阶段有个东西很重要,就是校园铃声. 它控制着上课,下课,吃饭,睡觉的节奏.没有它学校的管理就乱套了,老师拖课想拖多久就多久,那可不行,下课铃声一响就是在告诉老师时间到了,该停止了让学生HAPPY去了.
- 操作系统也一样，需要通过时间来规范其任务的执行，操作系统中最小的时间单位是时钟节拍 (OS Tick)。任何操作系统都需要提供一个时钟节拍，以供系统处理所有和时间有关的事件，如线程的延时、线程的时间片轮转调度以及定时器超时等。时钟节拍是特定的周期性中断，这个中断可以看做是系统心跳，中断之间的时间间隔取决于不同的应用，一般是 1ms–100ms，时钟节拍率越快，系统的实时响应越快，但是系统的额外开销就越大，从系统启动开始计数的时钟节拍数称为系统时间。
- 在鸿蒙内核中，时钟节拍的长度可以根据 LOSCFG_BASE_CORE_TICK_PER_SECOND 的定义来调整，等于 1/LOSCFG_BASE_CORE_TICK_PER_SECOND 秒。

时钟节拍的实现方式

时钟节拍由配置为中断触发模式的硬件定时器产生，当中断到来时，将调用一次：void OsTickHandler(void)，通知操作系统已经过去一个系统时钟；不同硬件定时器中断实现都不同，

```
/**
 * @ingroup los_config
 * Number of Ticks in one second
 */
#ifndef LOSCFG_BASE_CORE_TICK_PER_SECOND
#define LOSCFG_BASE_CORE_TICK_PER_SECOND 100 //默认每秒100次触发，当然这是可以改的
#endif
```

每秒100个tick，时间单位为10毫秒，即每秒调用时钟中断处理程序100次。

```
/*
 * Description : Tick interruption handler
 *///节拍中断处理函数,鸿蒙默认10ms触发一次
LITE_OS_SEC_TEXT VOID OsTickHandler(VOID)
{
    //...
    OsTimesliceCheck(); //进程和任务的时间片检查
    OsTaskScan(); /* task timeout scan */ //任务扫描
    #if (LOSCFG_BASE_CORE_SWTMR == YES)
        OsSwtmrScan(); //定时器扫描,看是否有超时的定时器
    #endif
}
```

它主要干了三件事情

第一:检查当前任务的时间片,任务执行一次分配多少时间呢?答案是2个时间片,即 20ms.

```
#ifndef LOSCFG_BASE_CORE_TIMESLICE_TIMEOUT
#define LOSCFG_BASE_CORE_TIMESLICE_TIMEOUT 2 //2个时间片,20ms
#endif
//检查进程和任务的时间片,如果没有时间片了直接调度
LITE_OS_SEC_TEXT VOID OsTimesliceCheck(VOID)
{
    LosTaskCB *runTask = NULL;
    LosProcessCB *runProcess = OsCurrProcessGet();//获取当前进程
    if (runProcess->policy != LOS_SCHED_RR) { //进程调度算法是否是抢占式
        goto SCHED_TASK; //进程不是抢占式调度直接去检查任务的时间片
    }

    if (runProcess->timeSlice != 0) { //进程还有时间片吗?
        runProcess->timeSlice--; //进程时间片减少一次
        if (runProcess->timeSlice == 0) { //没有时间片了
            LOS_Schedule(); //进程时间片用完,发起调度
        }
    }
}

SCHED_TASK:
runTask = OsCurrTaskGet();//获取当前任务
if (runTask->policy != LOS_SCHED_RR) { //任务调度算法是否是抢占式
    return; //任务不是抢占式调度直接结束检查
}

if (runTask->timeSlice != 0) { //任务还有时间片吗?
    runTask->timeSlice--; //任务时间片也减少一次
    if (runTask->timeSlice == 0) { //没有时间片了
        LOS_Schedule(); //任务时间片用完,发起调度
    }
}
}
```

第二:扫描任务,主要是检查被阻塞的任务是否可以被重新调度

```
LITE_OS_SEC_TEXT VOID OsTaskScan(VOID)
{
    SortLinkList *sortList = NULL;
    LosTaskCB *taskCB = NULL;
    BOOL needSchedule = FALSE;
    UINT16 tempStatus;
    LOS_DL_LIST *listObject = NULL;
    SortLinkAttribute *taskSortLink = NULL;

    taskSortLink = &OsPercpuGet()->taskSortLink; //获取任务的排序链表
    taskSortLink->cursor = (taskSortLink->cursor + 1) & OS_TSK_SORTLINK_MASK;
    listObject = taskSortLink->sortLink + taskSortLink->cursor; //只处理这个游标上的链表,因为系统对超时任务都已经规链表了.
    //当任务因超时而挂起时,任务块处于超时排序链接上,(每个cpu)和ipc(互斥锁、扫描电镜等)的块同时被唤醒
    /*不管是超时还是相应的ipc,它都在等待。现在使用synchronize sortlink precedure, 因此整个任务扫描需要保护,防止另一个核心同时删除sortlink.
    * When task is pended with timeout, the task block is on the timeout sortlink
    * (per cpu) and ipc(mutex,sem and etc.)'s block at the same time, it can be waken
    * up by either timeout or corresponding ipc it's waiting.
    */
    * Now synchronize sortlink precedure is used, therefore the whole task scan needs
    * to be protected, preventing another core from doing sortlink deletion at same time.
    */
    LOS_SpinLock(&g_taskSpin);

    if (LOS_ListEmpty(listObject)) {
        LOS_SpinUnlock(&g_taskSpin);
        return;
    }
    sortList = LOS_DL_LIST_ENTRY(listObject->pstNext, SortLinkList, sortLinkNode); //拿本次Tick对应链表的SortLinkList的第一个节点sortLinkNode
    ROLLNUM_DEC(sortList->idxRollNum); //滚动数--

    while (ROLLNUM(sortList->idxRollNum) == 0) { //找到时间到了节点,注意这些节点都是由定时器产生的,
```



```

    LOS_ListDelete(&sortList->sortLinkNode);
    taskCB = LOS_DL_LIST_ENTRY(sortList, LosTaskCB, sortList); //拿任务,这里的任务都是超时任务
    taskCB->taskStatus &= ~OS_TASK_STATUS_PEND_TIME;
    tempStatus = taskCB->taskStatus;
    if (tempStatus & OS_TASK_STATUS_PEND) {
        taskCB->taskStatus &= ~OS_TASK_STATUS_PEND;
    }
    #if (LOSCFG_KERNEL_LITEIPC == YES)
        taskCB->ipcStatus &= ~IPC_THREAD_STATUS_PEND;
    #endif
    taskCB->taskStatus |= OS_TASK_STATUS_TIMEOUT;
    LOS_ListDelete(&taskCB->pendList);
    taskCB->taskSem = NULL;
    taskCB->taskMux = NULL;
} else {
    taskCB->taskStatus &= ~OS_TASK_STATUS_DELAY;
}

if (!(tempStatus & OS_TASK_STATUS_SUSPEND)) {
    OS_TASK_SCHED_QUEUE_ENQUEUE(taskCB, OS_PROCESS_STATUS_PEND);
    needSchedule = TRUE;
}

if (LOS_ListEmpty(listObject)) {
    break;
}

sortList = LOS_DL_LIST_ENTRY(listObject->pstNext, SortLinkList, sortLinkNode);
}

LOS_SpinUnlock(&g_taskSpin);

if (needSchedule != FALSE) { //需要调度
    LOS_MpSchedule(OS_MP_CPU_ALL); //核间通讯,给所有CPU发送调度信号
    LOS_Schedule(); //开始调度
}
}

```

第三:定时器扫描,看是否有超时的定时器

```

/*
 * Description: Tick interrupt interface module of software timer
 * Return    : LOS_OK on success or error code on failure
 *///OsSwtmrScan 由系统时钟中断处理函数调用
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);
}

```

最后看调度算法的实现

```

//调度算法的实现
VOID OsSchedResched(VOID)
{
    LosTaskCB *runTask = NULL;
    LosTaskCB *newTask = NULL;
    LosProcessCB *runProcess = NULL;
    LosProcessCB *newProcess = NULL;
    LOS_ASSERT(LOS_SpinHeld(&g_taskSpin));//必须持有任务自旋锁,自旋锁是不是进程层面去抢锁,而是CPU各自核之间去争夺锁

    if (!OsPreemptableInSched()) { //是否置了重新调度标识位
        return;
    }
    runTask = OsCurrTaskGet();//获取当前任务
    newTask = OsGetTopTask();//获取优先级最最高的任务
    /* always be able to get one task */
    LOS_ASSERT(newTask != NULL);//不能没有需调度的任务
    if (runTask == newTask) { //当前任务就是最高任务,那还调度个啥的,直接退出.
        return;
    }
    runTask->taskStatus &= ~OS_TASK_STATUS_RUNNING;//当前任务状态位置成不在运行状态
    newTask->taskStatus |= OS_TASK_STATUS_RUNNING;//最高任务状态位置成在运行状态
    runProcess = OS_PCB_FROM_PID(runTask->processID);//通过进程ID索引拿到进程实体
    newProcess = OS_PCB_FROM_PID(newTask->processID);//同上
    OsSchedSwitchProcess(runProcess, newProcess);//切换进程,里面主要涉及进程空间的切换,也就是MMU的上下文切换.
#ifdef LOSCFG_KERNEL_SMP == YES //CPU多核的情况
    /* mask new running task's owner processor */
    runTask->currCpu = OS_TASK_INVALID_CPUID;//当前任务不占用CPU
    newTask->currCpu = ArchCurrCpuID();//让新任务占用CPU
#endif
}

```

```

    (VOID)OsTaskSwitchCheck(runTask, newTask); //切换task的检查
    #if (LOSCFG_KERNEL_SCHED_STATISTICS == YES)
        OsSchedStatistics(runTask, newTask);
    #endif
    if ((newTask->timeSlice == 0) && (newTask->policy == LOS_SCHED_RR)) { //没有时间片且是抢占式调度的方式,注意 非抢占式都不需要时间片的.
        newTask->timeSlice = LOSCFG_BASE_CORE_TIMESLICE_TIMEOUT; //给新任务时间片 默认 20ms
    }
    OsCurrTaskSet((VOID*)newTask); //设置新的task为CPU核的当前任务
    if (OsProcessIsUserMode(newProcess)) { //用户模式下会怎么样?
        OsCurrUserTaskSet(newTask->userArea); //设置task栈空间
    }
    /* 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
}

```

百篇博客.往期回顾

在加注过程中,整理出以下文章。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.确实有难度,自不量力,但已经出发,回头已是不可能的了。 :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o

- [v32.xx 鸿蒙内核源码分析\(CPU篇\) | 整个内核就是一个死循环 | 51.c.h.o](#)
- [v31.xx 鸿蒙内核源码分析\(定时器篇\) | 哪个任务的优先级最高 | 51.c.h.o](#)
- [v30.xx 鸿蒙内核源码分析\(事件控制篇\) | 任务间多对多的同步方案 | 51.c.h.o](#)
- [v29.xx 鸿蒙内核源码分析\(信号量篇\) | 谁在负责解决任务的同步 | 51.c.h.o](#)
- [v28.xx 鸿蒙内核源码分析\(进程通讯篇\) | 九种进程间通讯方式速览 | 51.c.h.o](#)
- [v27.xx 鸿蒙内核源码分析\(互斥锁篇\) | 比自旋锁丰满的互斥锁 | 51.c.h.o](#)
- [v26.xx 鸿蒙内核源码分析\(自旋锁篇\) | 自旋锁当立贞节牌坊 | 51.c.h.o](#)
- [v25.xx 鸿蒙内核源码分析\(并发并行篇\) | 听过无数遍的两个概念 | 51.c.h.o](#)
- [v24.xx 鸿蒙内核源码分析\(进程概念篇\) | 进程在管理哪些资源 | 51.c.h.o](#)
- [v23.xx 鸿蒙内核源码分析\(汇编传参篇\) | 如何传递复杂的参数 | 51.c.h.o](#)
- [v22.xx 鸿蒙内核源码分析\(汇编基础篇\) | CPU在哪里打卡上班 | 51.c.h.o](#)
- [v21.xx 鸿蒙内核源码分析\(线程概念篇\) | 是谁在不断的折腾CPU | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 `oschina gitee`，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- `51cto`
- `csdn`
- `harmony`

- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 `51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

04_任务调度篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51 .c .h .o

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。

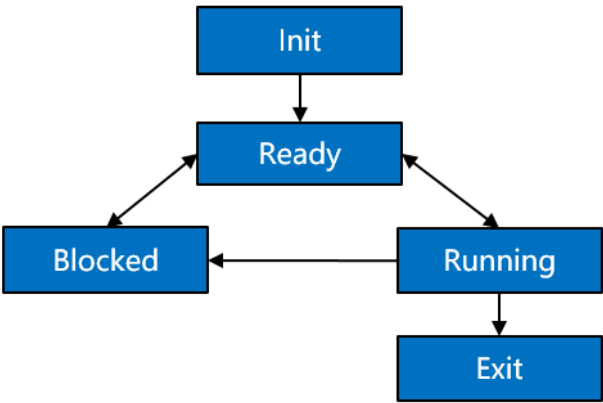
就绪（Ready）：该线程在就绪列表中，等待CPU调度。

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	0x46a38	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 */ //非用户模式下的栈指针
    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;          /**< Task 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
#endif
#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
}

    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_taskCBArry = (LosTaskCB *)LOS_MemAlloc(m_aucSysMem0, size);//任务池 常驻内存,不被释放
    if (g_taskCBArry == NULL) {
        return LOS_ERRNO_TSK_NO_MEMORY;
    }
    (VOID)memset_s(g_taskCBArry, 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_taskCBArry[index].taskStatus = OS_TASK_STATUS_UNUSED;
        g_taskCBArry[index].taskId = index;//任务ID最大默认127
        LOS_ListTailInsert(&g_losFreeTask, &g_taskCBArry[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_taskCBArry 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，所以工作要提前做好，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，不清楚的去 [鸿蒙系统源码分析\(总目录\)](#)里面翻。

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，一秒种内最多要来回切换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 {

    #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;

```

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 0xCCCCCCC "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

```
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));//注意看上下文将存放在栈的底部

    /* initialize the task context */
#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 */
    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 */
    }

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

#ifdef !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (index = 0; index < FP_REGS_NUM; index++) {

        taskContext->D[index] = 0xAAA000000000000LL + 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。

功能分类	接口名	描述
任务的创建和删除	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。

任务状态控制	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。
任务调度的控制	LOS_TaskLock	锁任务调度。
	LOS_TaskUnlock	解锁任务调度。
任务优先级的控制	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。
任务信息获取	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
#ifdef LOSCFG_KERNEL_SMP == YES
    UINT16          usCpuAffiMask; /**< Task cpu affinity mask */ //任务cpu亲和掩码
#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！

百篇博客.往期回顾

在加注过程中，整理出以下文章。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话屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o

- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o

- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

05_任务管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51 .c .h .o

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。

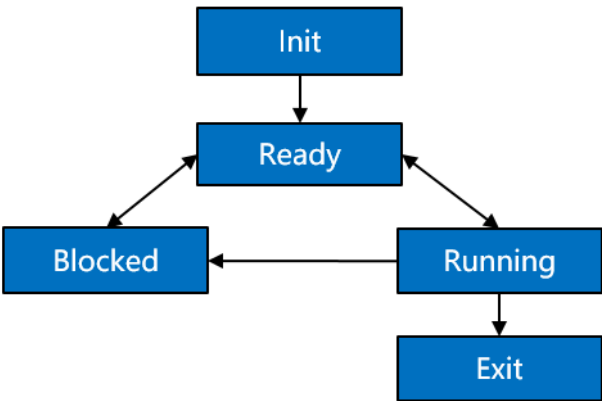
就绪（Ready）：该线程在就绪列表中，等待CPU调度。

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	0x46a38	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 */ //非用户模式下的栈指针
    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亲和掩码，最多支持16核，亲和性很重要，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
#endif
#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
}
    userArea; //使用区域，由运行时划定，根据运行态不同而不同
    userMapBase; //用户模式下的栈底位置
    userMapSize; /**< user thread stack size , real size : userMapSize + USER_STACK_MIN_SIZE */
    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，不清楚的去 [鸿蒙系统源码分析\(总目录\)](#)里面翻。

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，一秒种内最多要来回切换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 {

    #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;

```

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 0xCCCCCCC "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

```
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));//注意看上下文将存放在栈的底部

    /* initialize the task context */
#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 */
    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 */
    }

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

#ifdef !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;
}
```

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。

功能分类	接口名	描述
任务的创建和删除	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。

任务状态控制	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。
任务调度的控制	LOS_TaskLock	锁任务调度。
	LOS_TaskUnlock	解锁任务调度。
任务优先级的控制	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。
任务信息获取	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
#ifdef LOSCFG_KERNEL_SMP == YES
    UINT16          usCpuAffiMask; /**< Task cpu affinity mask      */ //任务cpu亲和掩码
#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！

百篇博客.往期回顾

在加注过程中，整理出以下文章。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话屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o

- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o

- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

06_调度队列篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51 .c .h .o

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：就绪队列就是在外排队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。


```

VOID OsPriQueueEnqueueHead(LOS_DL_LIST *priQueueList, UINT32 *bitMap, LOS_DL_LIST *prqueueltem, UINT32 priority)
{
    /*
     * Task control blocks are initied 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(prqueueltem->pstNext == NULL);

    if (LOS_ListEmpty(&priQueueList[priority])) {
        *bitMap |= PRIQUEUE_PRIOR0_BIT >> priority;//对应优先级位 置1
    }

    LOS_ListHeadInsert(&priQueueList[priority], prqueueltem);
}

VOID OsPriQueueDequeue(LOS_DL_LIST *priQueueList, UINT32 *bitMap, LOS_DL_LIST *prqueueltem)
{
    LosTaskCB *task = NULL;
    LOS_ListDelete(prqueueltem);

    task = LOS_DL_LIST_ENTRY(prqueueltem, 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：

- 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。

- 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 */
```

乍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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o

- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

07_调度机制篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o

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

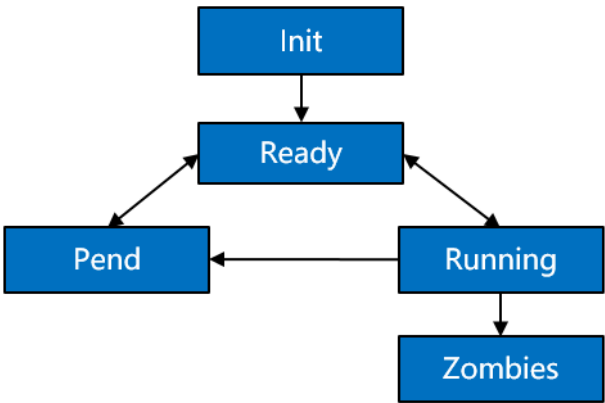
先看看task从哪些渠道产生：

□

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。
- □

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？[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，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
}

```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很简单，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速撷 | 51.c.h.o

- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

08_总目录

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- [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 | 51 .c .h .o](#)

百篇博客.往期回顾

在加注过程中,整理出以下文章。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.确实有难度,自不量力,但已经出发,回头已是不可能的了。 :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51 .c .h .o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51 .c .h .o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51 .c .h .o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51 .c .h .o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51 .c .h .o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51 .c .h .o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51 .c .h .o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百,依然活力十足 | 51 .c .h .o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51 .c .h .o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51 .c .h .o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用,两次返回 | 51 .c .h .o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51 .c .h .o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51 .c .h .o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51 .c .h .o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51 .c .h .o](#)
- [v40.xx 鸿蒙内核源码分析\(汇编汇总篇\) | 汇编可爱如邻家女孩 | 51 .c .h .o](#)
- [v39.xx 鸿蒙内核源码分析\(异常接管篇\) | 社会很单纯,复杂的是人 | 51 .c .h .o](#)
- [v38.xx 鸿蒙内核源码分析\(寄存器篇\) | 小强乃宇宙最忙存储器 | 51 .c .h .o](#)
- [v37.xx 鸿蒙内核源码分析\(系统调用篇\) | 开发者永远的口头禅 | 51 .c .h .o](#)

- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

09_调度故事篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51 .c .h .o

通俗易懂故事说内核 生活场景式理解 鸿蒙内核调度过程

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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o

- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。`51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

10_内存主奴篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51 .c .h .o

主子和奴才

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 and 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 over 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)

	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：

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

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用，两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51.c.h.o](#)

- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o

- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

11_内存分配篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51 .c .h .o

百篇博客.往期回顾

在加注过程中,整理出以下文章。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.确实有难度,自不量力,但已经出发,回头已是不可能的了。 :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51 .c .h .o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51 .c .h .o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51 .c .h .o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51 .c .h .o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51 .c .h .o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51 .c .h .o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51 .c .h .o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51 .c .h .o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51 .c .h .o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51 .c .h .o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51 .c .h .o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51 .c .h .o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51 .c .h .o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51 .c .h .o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51 .c .h .o

- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

12_内存管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51 .c .h .o

初始化整个内存

OsSysMemInit	los_vm_boot.c (kernel\base\vm)
OsSysMemInit	los_vm_boot.h (kernel\base\include)
OsMain	los_config.c (kernel\common)
OsMain	los_config.h (kernel\common)
main	main.c (platform)

从main()跟踪可看内存部分初始化是在OsSysMemInit()中完成的。

```
UINT32 OsSysMemInit(VOID)
{
    STATUS_T ret;

    OsKSpaceInit();//内核空间初始化

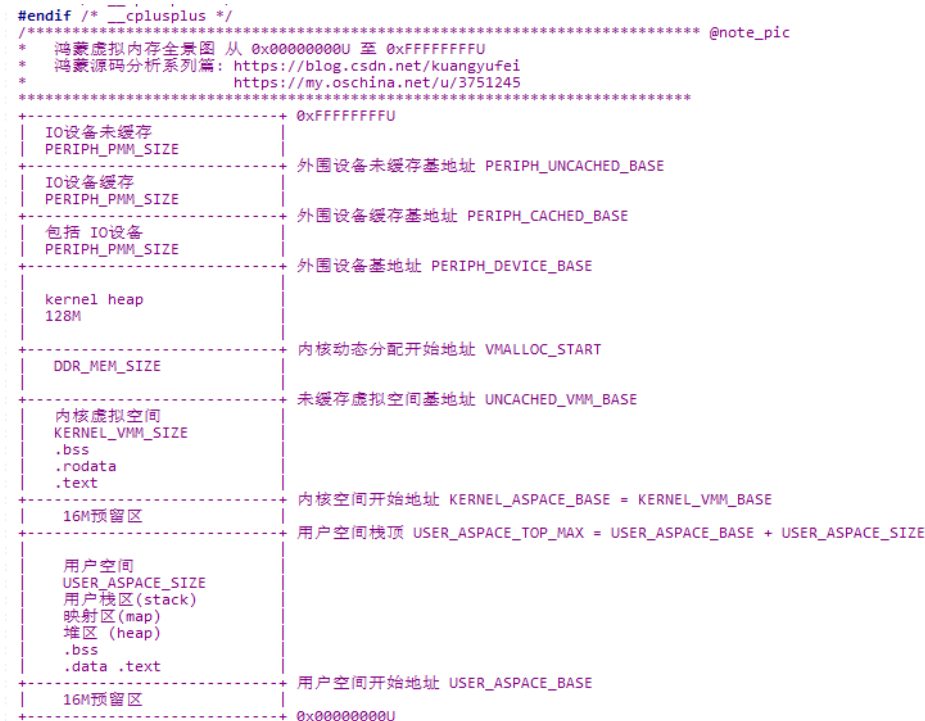
    ret = OsKHeapInit(OS_KHEAP_BLOCK_SIZE);// 内核动态内存初始化 512K
    if (ret != LOS_OK) {
        VM_ERR("OsKHeapInit fail");
        return LOS_NOK;
    }

    OsVmPageStartup();// page初始化
    OsInitMappingStartUp();// 映射初始化

    ret = ShmInit();// 共享内存初始化
    if (ret < 0) {
        VM_ERR("ShmInit fail");
        return LOS_NOK;
    }

    return LOS_OK;
}
```

鸿蒙虚拟内存整体布局图



```
// HarmonyOS 内核空间包含以下各段:
extern CHAR __int_stack_start; // 运行系统函数栈的开始地址
extern CHAR __rodata_start; // ROM开始地址 只读
extern CHAR __rodata_end; // ROM结束地址
extern CHAR __bss_start; // bss开始地址
extern CHAR __bss_end; // bss结束地址
extern CHAR __text_start; // 代码区开始地址
extern CHAR __text_end; // 代码区结束地址
extern CHAR __ram_data_start; // RAM开始地址 可读可写
extern CHAR __ram_data_end; // RAM结束地址
extern UINT32 __heap_start; // 堆区开始地址
extern UINT32 __heap_end; // 堆区结束地址
```

内存一开始一张白纸，这些extern就是给它画大界线的，从哪到哪是属于什么段。这些值大小取决实际项目内存条的大小，不同的内存条，地址肯定不会一样，所以必须由外部提供，鸿蒙内核采用了Linux的段管理方式。结合上图对比以下的解释自行理解下位置。

BSS段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。该段用于存储未初始化的全局变量或者是默认初始化为0的全局变量，它不占用程序文件的大小，但是占用程序运行时的内存空间。

data段 该段用于存储初始化的全局变量，初始化为0的全局变量出于编译优化的策略还是被保存在BSS段。

细心的读者可能发现了，鸿蒙内核几乎所有的全局变量都没有赋初始化值或NULL，这些变量经过编译后是放在了BSS段的，运行时占用内存空间，如此编译出来的ELF包就变小了。

.rodata段，该段也叫常量区，用于存放常量数据，ro就是Read Only之意。

text段 是用于存放程序代码的，编译时确定，只读。更进一步讲是存放处理器的机器指令，当各个源文件单独编译之后生成目标文件，经连接器链接各个目标文件并解决各个源文件之间函数的引用，与此同时，还得将所有目标文件中的.text段合在一起。

stack栈段，是由系统负责申请释放，用于存储参数变量及局部变量以及函数的执行。

heap段 它由用户申请和释放，申请时至少分配虚存，当真正存储数据时才分配相应的实存，释放时也并非立即释放实存，而是可能被重复利用。

内核空间是怎么初始化的？

```
LosMux g_vmSpaceListMux;//虚拟空间互斥锁，一般和g_vmSpaceList配套使用
LOS_DL_LIST_HEAD(g_vmSpaceList);//g_vmSpaceList把所有虚拟空间挂在一起，
LosVmSpace g_kVmSpace; //内核空间地址
```

```

LosVmSpace g_vMallocSpace;//虚拟分配空间地址

//鸿蒙内核空间有两个(内核进程空间和内核动态分配空间)，共用一张L1页表
VOID OsKSpaceInit(VOID)
{
    OsVmMapInit();// 初始化互斥量
    OsKernVmSpaceInit(&g_kVmSpace, OsGFirstTableGet());// 初始化内核虚拟空间，OsGFirstTableGet 为L1表基地址
    OsVMallocSpaceInit(&g_vMallocSpace, OsGFirstTableGet());// 初始化动态分配区虚拟空间，OsGFirstTableGet 为L1表基地址
}
//g_kVmSpace g_vMallocSpace 共用一个L1页表
//初始化内核堆空间
STATUS_T OsKHeapInit(size_t size)
{
    STATUS_T ret;
    VOID *ptr = NULL;
    /*
     * roundup to MB aligned in order to set kernel attributes. kernel text/code/data attributes
     * should page mapping, remaining region should section mapping. so the boundary should be
     * MB aligned.
     */
    //向上舍入到MB对齐是为了设置内核属性。内核文本/代码/数据属性应该是页映射，其余区域应该是段映射，所以边界应该对齐。
    UINTPTR end = ROUNDUP(g_vmBootMemBase + size, MB);//用M是因为采用section mapping 鸿蒙内核源码分析(内存映射篇)有阐述
    size = end - g_vmBootMemBase;
    //ROUNDUP(0x00000200+512, 1024) = 1024 ROUNDUP(0x00000201+512, 1024) = 2048 此处需细品!
    ptr = OsVmBootMemAlloc(size);//因刚开机，使用引导分配器分配
    if (!ptr) {
        PRINT_ERR("vmm_kheap_init boot_alloc_mem failed! %d\n", size);
        return -1;
    }

    m_aucSysMem0 = m_aucSysMem1 = ptr;//内存池基地址，取名auc还用0和1来标识有何深意，一直没整明白， 哪位大神能告诉下?
    ret = LOS_MemInit(m_aucSysMem0, size);//初始化内存池
    if (ret != LOS_OK) {
        PRINT_ERR("vmm_kheap_init LOS_MemInit failed!\n");
        g_vmBootMemBase -= size;//分配失败时需归还size， g_vmBootMemBase是很野蛮粗暴的
        return ret;
    }
    LOS_MemExpandEnable(OS_SYS_MEM_ADDR);//地址可扩展
    return LOS_OK;
}

```

内核空间用了三个全局变量，其中一个互斥LosMux，IPC部分会详细讲，这里先不展开。比较有意思的是LOS_DL_LIST_HEAD，看内核源码过程中经常会为这样的代码点头称赞，会心一笑。点赞！

```
#define LOS_DL_LIST_HEAD(list) LOS_DL_LIST list = { &(list), &(list) }
```

Page是如何初始化的？

page是映射的最小单位，是物理地址<--->虚拟地址映射的数据结构的基础

```

// page初始化
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);//申请页表存放区域

    OsVmPhysAreaSizeAdjust(ROUNDUP(g_vmPageArraySize, PAGE_SIZE));// g_physArea 变小

    OsVmPhysSegAdd();// 段页绑定

```

```

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++) {
    seg = &g_vmPhysSeg[segID];
    nPage = seg->size >> PAGE_SHIFT;
    for (page = seg->pageBase, pa = seg->start; page <= seg->pageBase + nPage;
        page++, pa += PAGE_SIZE) {
        OsVmPageInit(page, pa, segID);//page初始化
    }
    OsVmPageOrderListInit(seg->pageBase, nPage);// 页面分配的排序
}
}
}

```

进程是如何申请内存的？

进程的主体是来自进程池，进程池是统一分配的，怎么创建进程池的去翻系列篇里的文章，所以创建一个进程的时候只需要分配虚拟内存 LosVmSpace，这里要分内核模式和用户模式下的申请。

```

//初始化进程的 用户空间 或 内核空间
//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;//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);//初始化任务/线程链表
    LOS_ListInit(&processCB->childrenList);    //初始化孩子链表
    LOS_ListInit(&processCB->exitChildList);    //初始化记录哪些孩子退出了的链表
    LOS_ListInit(&(processCB->waitList));    //初始化等待链表

    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]);
    }

    if (OsProcessIsUserMode(processCB)) { // 是否为用户态进程
        space = LOS_MemAlloc(m_aucSysMem0, sizeof(LosVmSpace));
        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);
        retVal = OsUserVmSpaceInit(space, ttb);//初始化虚拟空间和本进程 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;
}
LosVmSpace *LOS_GetKVmSpace(VOID)
{
    return &g_kVmSpace;
}

```

从代码可以看出，内核空间固定只有一个g_kVmSpace，而每个用户进程的虚拟内存空间都是独立的。请细品！

task是如何申请内存的？

task的主体是来自进程池，task池是统一分配的，怎么创建task池的去翻系列篇里的文章。这里task只需要申请stack空间，还是直接上看源码吧，用OsUserInitProcess函数看应用程序的main()是如何被内核创建任务和运行的。

```

//所有的用户进程都是使用同一个用户代码段描述符和用户数据段描述符，它们是__USER_CS和__USER_DS，也就是每个进程处于用户态时，它们的CS寄存器和DS寄存器
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，所有进程
    CHAR *userInitBssStart = (CHAR *)&__user_init_bss;//未初始化数据区（BSS）。在运行时改变其值
    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);
    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程
    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 -> userText
    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);// 初始化堆栈区
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;// 指向栈底
param.userParam.userMapBase = (UINTPTR)stack;// 栈顶
param.userParam.userMapSize = size;// 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
OsDelInitPCB(processCB);//删除PCB块
return ret;
}
```

所有的用户进程都是通过init进程 fork来的， 可以看到创建进程的同时创建了一个task， 入口函数就是代码区的第一条指令，也就是应用程序 main 函数。这里再说下stack的大小，不同空间下的task栈空间是不一样的，鸿蒙内核中有三种栈空间size，如下

```
#define LOSCFG_BASE_CORE_TSK_IDLE_STACK_SIZE SIZE(0x800)//内核进程，运行在内核空间2K
#define OS_USER_TASK_SYSCALL_SATCK_SIZE 0x3000 //用户进程，通过系统调用创建的task运行在内核空间的 12K
#define OS_USER_TASK_STACK_SIZE 0x100000//用户进程运行在用户空间的1M
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o

- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o

- [v06.xx 鸿蒙内核源码分析\(调度队列篇\)](#) | 内核有多少个调度队列 | [51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\)](#) | 任务池是如何管理的 | [51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\)](#) | 任务是内核调度的单元 | [51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\)](#) | 触发调度谁的贡献最大 | [51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\)](#) | 谁在管理内核资源 | [51.c.h.o](#)
- [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 | [51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 `51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

13_源码注释篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o

同步官方源码历史

- 每月至少同步一次
- 2021/5/28 -- 本次官方改动不大,主要针对一些错误单词拼写纠正
- 2021/5/13 -- 本次官方对系统调用,任务切换,信号处理,异常接管,汇编文件,文件管理,shell 做了较大更新,代码结构更清晰,部分代码需重新加注.
- 2021/4/21 -- 官方优化了很多之前吐槽的地方,点赞.
- 2020/9/16 -- 中文注解版起点.

几点说明

- `kernel_liteos_a_note` 是在 OpenHarmony 的 `kernel_liteos_a` (鸿蒙轻内核项目)基础上给源码加上中文注解的版本.加注版与官方版本保持每月同步.
- 注解分析博客地址:
 - 国内: <https://weharmony.gitee.io/weharmony>
 - 国外: <https://weharmony.github.io/weharmony>
- 博客站点更新速度:
 - 注解分析地址
 - oschina
 - 51cto
 - csdn
- OpenHarmony开发者文档是对官方文档 `docs` 做的非常炫酷的静态站点,支持侧边栏/面包屑/搜索/中英文,非常方便的查看官方文档,大大提高学习和开发效率.
 - 国内: <https://weharmony.gitee.io/openharmony>
 - 国外: <https://weharmony.github.io/openharmony>
- OpenHarmony全量代码仓库 是 OpenHarmony 的110+个子项目的所有代码. OpenHarmony 使用 `repo` 管理众多 `git` 项目, `repo` 在 `linux` 下很方便,但在 `windows` 上谁用谁知道,使用会有相当的困难,所以将所有项目整合成一个`.git`工程,如此使用 `git` 便能下载整个鸿蒙系统源码,方便学习使用.与官方仓库保持每月同步.OpenHarmony全量代码仓库已编译通过.

```
....
[OHOS INFO] [1587/1590] STAMP obj/test/xts/acts/build_lite/acts_generate_module_data.stamp
[OHOS INFO] [1588/1590] ACTION //test/xts/acts/build_lite:acts(//build/lite/toolchain:linux_x86_64_ohos_clang)
[OHOS INFO] [1589/1590] STAMP obj/test/xts/acts/build_lite/acts.stamp
[OHOS INFO] [1590/1590] STAMP obj/build/lite/ohos.stamp
[OHOS INFO] ipcamera_hispanic_aries build success
root@5e3abe332c5a:/home/harmony#
```

- 下载:鸿蒙源码分析-离线文档
 - 国内: <https://weharmony.gitee.io/history/>
 - 国外: <https://weharmony.github.io/history/>

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网,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.跟骑自行车一样,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,如何让更多对内核感兴趣的朋友减少阅读时间,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.
- 到 2021/3/10 刚好半年,对内核源码的注解已完成了 70%,对内核源码的博客分析已完成了40篇,每天都很充实,很兴奋,连做梦内核代码都在鱼贯而入.如此疯狂地做一件事还是当年谈恋爱的时候,只因热爱,热爱是所有的理由和答案.: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.精读内核源码加注并整理成档是件很有挑战的事,时间上要以月甚至年为单位,但正因为很难才值得去做!干困难事,方有所得;专注聚焦,必有所获.
- 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,计算机组成原理 五门课程的教学项目.如此宝库,不深入研究实在是暴殄天物,于心不忍,注者坚信鸿蒙大势所趋,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 [鸿蒙内核源码分析\(总目录\)之故事篇](#) 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,尤其是那些资本大鳄,加大对基础软件的投入.
- **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 [鸿蒙内核源码分析\(总目录\)百篇博客](#) 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,希望鸿蒙内核源码注解分析能让更多爱好者快速的理解内核,共同进步.

加注方式是怎样的?

- 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试手段，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：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。栈帧从高地指向低地址生长。

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。

- 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->mutexList.pstPrev == NULL) || (mutex->mutexList.pstNext == NULL)) { //列表为空时的处理
        /* This is for mutex macro initialization. */
        mutex->mutexCount = 0; //锁计数器清0
        mutex->owner = NULL; //锁没有归属任务
        LOS_ListInit(&mutex->mutexList); //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->mutexCount == 0) { //无task用锁时,肯定能拿到锁了.在里面返回
        mutex->mutexCount++; //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mutexCount>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->mutexCount++; //递归锁计数器加1,递归锁的目的是防止死锁,鸿蒙默认的就是递归锁(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); //在等待锁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) { //这行代码是和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;
}
```

- 另外画了一些字符图方便理解，直接嵌入到头文件中，比如虚拟内存的全景图，因没有这些图是很难理解虚拟内存是如何管理的。



有哪些特殊的记号

- 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_good` 是给源码点赞的地方

新增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，里面放了一些文件，它与内核代码无关，大家可以忽略它，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 :|P

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :|P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51 .c .h .o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51 .c .h .o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51 .c .h .o

- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o

- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要 CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

14_内存汇编篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

- 内核源码注解分析 →
- OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51 .c .h .o

ARM-CP15协处理器

ARM处理器使用协处理器15(CP15)的寄存器来控制cache、TCM和存储器管理。CP15的寄存器只能被MRC和MCR（Move to Coprocessor from ARM Register）指令访问，包含16个32位的寄存器，其编号为0~15。本篇重点讲解其中的 C7，C2，C13三个寄存器。

先拆解一段汇编代码

上来看段汇编，读懂内核源码不会点汇编是不行的，但不用发怵，没那么恐怖，由浅入深，内核其实挺好玩的。见于 arm.h，里面全是这些玩意。

```
#define DSB __asm__ volatile("dsb" ::: "memory")
#define ISB __asm__ volatile("isb" ::: "memory")
#define DMB __asm__ volatile("dmb" ::: "memory")

STATIC INLINE VOID OsArmWriteBpiallis(UINT32 val)
{
    __asm__ volatile("mcr p15, 0, %0, c7, c1, 6" ::"r"(val));
    __asm__ volatile("isb" ::: "memory");
}
```

指令	说 明	语法格式
mcr	将ARM处理器的寄存器中的数据写到CP15中的寄存器中	mcr{<cond>} p15, <opcode_1>, <rd>, <crm>, <crm>, {<opcode_2>
mrc	将CP15中的寄存器中的数据读到ARM处理器的寄存器中	mrc{<cond>} p15, <opcode_1>, <rd>, <crm>, <crm>, {<opcode_2>

这句汇编的指令字面意思是: 将ARM寄存器R0的数据写到CP15中编号为7的寄存器中，值由外面传进来。

例如 OsArmWriteBpiallis(0) 做了4个动作

- 1.把0值写入R0寄存器，注意这个寄存器是ARM即CPU的寄存器，::"r"(val) 意思代表向GCC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。
- 2.volatile的意思还是告诉编译器，不要去优化这段代码，原封不动的生成目标指令。

- 3."isb" ::: "memory" 还是告诉编译器内存的内容可能被更改了，需要无效所有Cache，并访问实际的内容，而不是Cache！
- 4.再把R0的值写入到C7中，C7是CP15协处理器的寄存器。C7寄存器是负责什么的？对照下面的表。

CP15有哪些寄存器

寄存器编号	基本作用	在MMU中的作用	在PU中的作用
0	ID编码（只读）	ID编码和cache类型	
1	控制位（可读写）	各种控制位	
2	存储保护和控制	地址转换表基地址	Cachability的控制位
3	存储保护和控制	域访问控制位	Bufferability控制位
4	存储保护和控制	保留	保留
5	存储保护和控制	内存失效状态	访问权限控制位
6	存储保护和控制	内存失效地址	保护区域控制
7	高速缓存和写缓存	高速缓存和写缓存控制	
8	存储保护和控制	TLB控制	保留
9	高速缓存和写缓存	高速缓存锁定	
10	存储保护和控制	TLB锁定	保留
11	保留		
12	保留		
13	进程标识符	进程标识符	
14	保留		
15	因不同设计而异	因不同设计而异	因不同设计而异

这句话真正的意思是：关闭高速缓存和写缓存控制！，其他部分寄存器下面会讲，先有个大概印象。

mmu从哪里获取 page table 的信息？答案是: TTB

TTB寄存器(Translation table base)

参考上表可知TTB寄存器是CP15协处理器的C2寄存器，存页表的基地址，即一级映射描述符表的基地址。围绕着TTB鸿蒙提供了以下读取函数。简单说就是内核从外面不断的修改和读取寄存器值，而MMU只会直接通过硬件读取这个寄存器的值，以达到MMU获取不一样的页表进行进程虚拟地址和物理地址的转换。还记得吗？每个进程的页表都是独立的！

□

那么什么情况下会修改里面的值呢？换页表意味着 mmu在进行上下文的切换！还是直接看代码吧。

mmu上下文



只被这一个函数调用。毫无疑问LOS_ArchMmuContextSwitch是关键函数。

```

typedef struct ArchMmu {
    LosMux      mtx;          /**< arch mmu page table entry modification mutex lock */
    VADDR_T     *virtTtb;     /**< translation table base virtual addr */
    PADDR_T     physTtb;      /**< translation table base phys addr */
    UINT32      asid;         /**< TLB asid */
    LOS_DL_LIST ptList;       /**< page table vm page list */
} LosArchMmu;

// mmu 上下文切换
VOID LOS_ArchMmuContextSwitch(LosArchMmu *archMmu)
{
    UINT32 ttbr;
    UINT32 ttbcrr = OsArmReadTtbcrr(); // 读取TTB寄存器的状态值
    if (archMmu) {
        ttbr = MMU_TTBRR_FLAGS | (archMmu->physTtb); // 进程TTB物理地址值
        /* enable TTBR0 */
        ttbcrr &= ~MMU_DESCRIPTOR_TTBRR_PD0; // 使能TTBR0
    } else {
        ttbr = 0;
        /* disable TTBR0 */
        ttbcrr |= MMU_DESCRIPTOR_TTBRR_PD0;
    }

    /* 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;
    OsArmWriteTtbr0(ttbr); // 通过r0寄存器将进程页面基址写入TTB
    ISB;
    OsArmWriteTtbcrr(ttbcrr); // 写入TTB状态位
    ISB;
    if (archMmu) {
        OsArmWriteContextidr(archMmu->asid); // 通过R0寄存器写入进程标识符至C13寄存器
        ISB;
    }
}

// c13 asid(Address Space ID)进程标识符
STATIC INLINE VOID OsArmWriteContextidr(UINT32 val)
{
    __asm__ volatile("mcr p15, 0, %0, c13, c0, 1" ::: "r"(val));
    __asm__ volatile("isb" ::: "memory");
}
  
```

再看下那些地方会调用 LOS_ArchMmuContextSwitch，下图一目了然。

□

有四个地方会切换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
    }

    #ifndef 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，还有一个是因task切换引起的CPU的上下文切换，还能想起来吗？

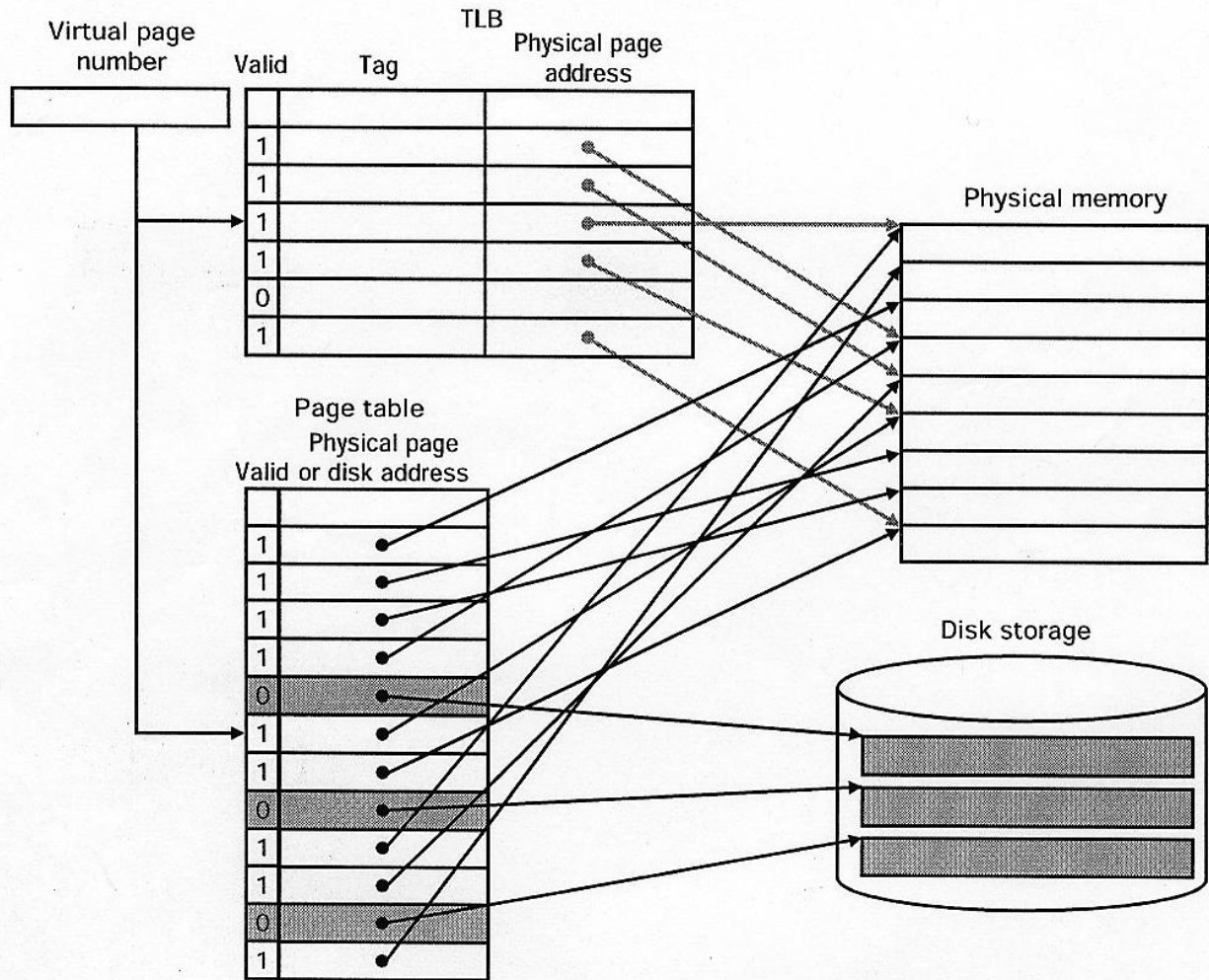
第二：是加载ELF文件的时候会切换mmu，一个崭新的进程诞生了，具体将在 鸿蒙内核源码分析(启动加载篇) 会细讲，敬请关注系列篇动态。

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。

TLB (translation lookaside buffer)

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。
1. 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。
 2. 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。
 3. 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。

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;
}
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o

- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

15_内存映射篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51 .c .h .o

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.鸿蒙内核源码注释 < G | G | C >

一级页表L1

L1页表将全部的4G地址空间划分为4096个1M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

□

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)
116 / 457
```



```

{ //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__);
}

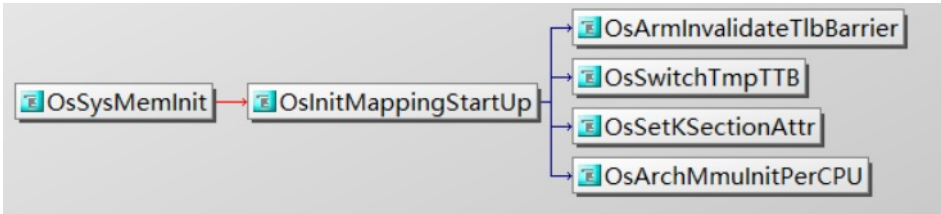
```

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(rodEnd - rodataStart, MMU_DESCRIPTOR_L2_SMALL_SIZE), //4K对齐
        .flags = VM_MAP_REGION_FLAG_PERM_READ, //常量段只读
        .name = "kernel_rodEnd"
    },
    {
        .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")))
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_BASI
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——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_TTBx_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 = OsMapL2PageContinuous(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 = OsMapL2PageContinuous(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 OsMapL2PageContinuous(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;
}
```

□

OsMapL2PageContinuous 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。

百篇博客·往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作日 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o

- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件, 这就很有意思了, 冥冥之中似有天数, 将这四个宝贝以这种方式融合在一起. 51.c.h.o, 我要CHO, 嗯嗯, hin 顺口:)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码, 中文注解分析, 深挖地基工程, 大脑永久记忆, 四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核, 问答式导读, 生活式比喻, 表格化说明, 图形化展示, 主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

16_内存规则篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51 .c .h .o

主子和奴才

看本篇之前建议先看 [鸿蒙内核源码分析\(调度故事篇\)](#) | [通俗易懂故事说内核](#). 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 and 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?

看下某地对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)

	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：

	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

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用，两次返回 | 51.c.h.o](#)

- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o

- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

17_物理内存篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51 .c .h .o

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.这些模块不在此处展开说, 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_ASAPCE_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 $2^{\text{max_order}}$ 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，运行一段时间后你还能买到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间后，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.

百篇博客.往期回顾

在加注过程中, 整理出以下文章. 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. 确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样, 文章和注解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, .xx 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容.

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百, 依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用, 两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)

- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o

- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | [51.c.h.o](#)
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | [51.c.h.o](#)
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | [51.c.h.o](#)
- 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 | [51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。`51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

18_源码结构篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51 .c .h .o

鸿蒙内核源码注解分析

点击文件查看源码

- kernel_liteos_a_note
 - kernel
 - base
 - core -> -> 这个core指的是与CPU core相关的文件
 - los_bitmap.c -> -> 位图管理器有什么作用？在内核常应用于哪些场景？
 - los_process.c -> 鸿蒙内核源码分析(进程管理篇) -> 进程是内核的资源管理单元，它是如何管理 任务，内存，文件的？进程间是如何协作的？
 - los_sortlink.c -> -> 排序链表的实现，它的应用场景是怎样的？
 - los_swtmr.c -> -> 内核的定时器是如何实现和管理的？
 - los_sys.c -> -> 几个跟tick相关的转化函数
 - los_task.c -> 鸿蒙内核源码分析(Task管理篇) -> Task是内核调度的单元，它解决了什么问题？如何调度？
 - los_tick.c -> 鸿蒙内核源码分析(时钟管理篇) -> 是谁在一直触发调度？硬时钟中断都干了些什么事？
 - los_timeslice.c -> -> 进程和任务能一直占有CPU吗？怎么合理的分配时间？
 - ipc -> -> 进程间通讯有哪些方式？请说出三种？是如何实现的？
 - los_event.c -> -> 事件解决了什么问题？怎么管理的？
 - los_futex.c -> -> futex 是Fast Userspace muTexes的缩写(快速用户空间互斥体)，它有什么作用？
 - los_ipcdebug.c -> -> 进程间通讯如何调试？
 - los_mux.c -> -> 互斥量，有你没我的零和博弈，为什么需要互斥量？是如何实现的？
 - los_queue.c -> -> 内核消息队列是如何实现的？对长度和大小有限制吗？
 - los_queue_debug.c -> -> 如何调试消息队列？
 - los_sem.c -> -> 信号量解决了什么问题？它的本质是什么？
 - los_sem_debug.c -> -> 如何调试信号量？
 - los_signal.c -> -> 信号解决了什么问题？你知道哪些信号？
 - mem -> -> 内存管理模块管理系统的内存资源，它是操作系统的核心模块之一
 - bestfit -> -> 动态内存管理的优点是按需分配，那缺点又是什么？
 - los_memory.c -> -> 鸿蒙内核中动态内存池由哪三个部分组成？
 - los_multipldlinkhead.c -> -> 什么是最佳适应算法？是如何实现？
 - bestfit_little -> -> bestfit_little算法是在最佳适应算法的基础上加入slab机制形成的算法。
 - los_heap.c -> -> slab算法机制是怎样的？又是如何实现？
 - common -> ->
 - membox -> -> 静态内存池的优点是分配和释放效率高，无碎片，那缺点呢？
 - los_membox.c -> -> 静态内存有什么用？是如何实现的？
 - misc -> ->
 - kill_shellcmd.c -> -> shell命令kill的实现，熟悉的 kill 9 18 的背后发生了什么？

- `los_misc.c` -> ->
- `los_stackinfo.c` -> -> 栈有哪些信息？如何检测栈是否异常？
- `mempt_shellcmd.c` -> -> 和内存相关的shell命令有哪些？
- `swtmr_shellcmd.c` -> -> 和软时钟相关的shell命令有哪些？
- `sysinfo_shellcmd.c` -> -> 和系统信息相关的shell命令有哪些？
- `task_shellcmd.c` -> -> 和任务相关的shell命令有哪些？
- `vm_shellcmd.c` -> -> 和虚拟内存相关的shell命令有哪些？
- `mp` -> -> MP指支持多处理器的模块
 - `los_lockdep.c` -> -> 死锁是怎么发生的？如何检测死锁？
 - `los_mp.c` -> -> 鸿蒙最大支持多少个CPU？它们是如何工作的？CPU之间是如何通讯的？
 - `los_percpu.c` -> -> CPU有哪些信息？
 - `los_stat.c` -> -> CPU的运行信息如何统计？
- `om` -> ->
 - `los_err.c` -> ->
- `sched/sched_sq` -> ->
 - `los_pqueue.c` -> 鸿蒙内核源码分析(调度队列篇) -> 为什么只有就绪状态才会有队列？
 - `los_sched.c` -> 鸿蒙内核源码分析(调度机制篇) -> 哪些情况下会触发调度？调度算法是怎样的？
- `vm` -> 鸿蒙内核源码分析(内存规则篇) -> 什么是虚拟内存？虚拟内存全景图是怎样的？
 - `los_vm_boot.c` -> -> 开机阶段内存是如何初始化的？
 - `los_vm_dump.c` -> -> 如何 dump 内存数据？
 - `los_vm_fault.c` -> -> 为什么会缺页？缺页怎么处理？
 - `los_vm_filemap.c` -> -> 文件和内存是如何映射？什么是 写时拷贝技术(cow)？
 - `los_vm_iomap.c` -> -> 设备和内存是如何映射？
 - `los_vm_map.c` -> 鸿蒙内核源码分析(内存映射篇) -> 内核空间，用户空间，线性区是如何分配的，虚拟内存<-->物理内存是如何映射的？
 - `los_vm_page.c` -> -> 什么是物理页框，哪些地方会用到它？
 - `los_vm_phys.c` -> 鸿蒙内核源码分析(物理内存篇) -> 段页式管理，物理内存是如何分配和回收的？
 - `los_vm_scan.c` -> -> LRU算法是如何运作的？
 - `los_vm_syscall.c` -> -> 系统调用之内存，用户进程如何申请内存？底层发生了什么？
 - `oom.c` -> -> 内存溢出是如何检测的？
 - `shm.c` -> -> 共享内存是如何实现的？
- `common` -> ->
 - `console.c` -> -> 熟悉的控制台是如何实现的？
 - `hwi_shell.c` -> -> 如何查询硬件中断？
 - `los_cir_buf.c` -> -> 环形缓冲区的读写是如何实现的？常用于什么场景下？
 - `los_config.c` -> -> 内核有哪些配置信息？
 - `los_exc_interaction.c` -> -> 任务出现异常如何检测？
 - `los_excinfo.c` -> -> 异常有哪些信息？如何记录异常信息？
 - `los_hilog.c` -> -> 内核是如何封装日志的？
 - `los_magickey.c` -> -> 魔法键有什么作用？
 - `los_printf.c` -> -> 内核对 printf 做了哪些封装？
 - `los_rootfs.c` -> -> 什么是根文件系统？为什么需要它？
 - `los_seq_buf.c` -> ->
 - `virtual_serial.c` -> -> 如何实现访问串口如同访问文件一样方便？
- `extended` -> ->
 - `cppsupport` -> ->
 - `los_cppsupport.c` -> -> 对C++是如何支持的？
 - `cpup` -> ->
 - `cpup_shellcmd.c` -> -> 如何实时查询系统CPU的占用率？
 - `los_cpup.c` -> -> 内核如何做到实时统计CPU性能的？
 - `dynload/src` -> ->
 - `los_exec_elf.c` -> -> 鸿蒙如何运行ELF？什么是腾笼换鸟技术？
 - `los_load_elf.c` -> -> 鸿蒙如何动态加载 ELF？
 - `liteipc` -> ->
 - `hm_liteipc.c` -> -> 如何用文件的方式读取消息队列？liteipc和普通消息队列区别有哪些？
 - `tickless` -> ->

- [los_tickless.c](#) -> -> 新定时机制新在哪里？它解决了哪些问题？
- [trace](#) -> ->
 - [los_trace.c](#) -> -> 如何实现跟踪？内核在跟踪什么？
- [vdso](#) -> -> 用户空间访问内核空间有哪些途径？
 - [src](#) -> ->
 - [los_vdso.c](#) -> -> VDSO(Virtual Dynamically-linked Shared Object) 是如何实现的？
 - [los_vdso_text.S](#) -> ->
 - [usr](#) -> ->
 - [los_vdso_sys.c](#) -> ->
- [user/src](#) -> ->
- [los_user_init.c](#) -> ->

百篇博客.往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用，两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51.c.h.o](#)
- [v40.xx 鸿蒙内核源码分析\(汇编汇总篇\) | 汇编可爱如邻家女孩 | 51.c.h.o](#)
- [v39.xx 鸿蒙内核源码分析\(异常接管篇\) | 社会很单纯，复杂的是人 | 51.c.h.o](#)
- [v38.xx 鸿蒙内核源码分析\(寄存器篇\) | 小强乃宇宙最忙存储器 | 51.c.h.o](#)
- [v37.xx 鸿蒙内核源码分析\(系统调用篇\) | 开发者永远的口头禅 | 51.c.h.o](#)
- [v36.xx 鸿蒙内核源码分析\(工作模式篇\) | CPU是韦小宝，七个老婆 | 51.c.h.o](#)
- [v35.xx 鸿蒙内核源码分析\(时间管理篇\) | 谁是内核基本时间单位 | 51.c.h.o](#)
- [v34.xx 鸿蒙内核源码分析\(原子操作篇\) | 谁在为原子操作保驾护航 | 51.c.h.o](#)
- [v33.xx 鸿蒙内核源码分析\(消息队列篇\) | 进程间如何异步传递大数据 | 51.c.h.o](#)
- [v32.xx 鸿蒙内核源码分析\(CPU篇\) | 整个内核就是一个死循环 | 51.c.h.o](#)

- [v31.xx 鸿蒙内核源码分析\(定时器篇\) | 哪个任务的优先级最高 | 51.c.h.o](#)
- [v30.xx 鸿蒙内核源码分析\(事件控制篇\) | 任务间多对多的同步方案 | 51.c.h.o](#)
- [v29.xx 鸿蒙内核源码分析\(信号量篇\) | 谁在负责解决任务的同步 | 51.c.h.o](#)
- [v28.xx 鸿蒙内核源码分析\(进程通讯篇\) | 九种进程间通讯方式速揽 | 51.c.h.o](#)
- [v27.xx 鸿蒙内核源码分析\(互斥锁篇\) | 比自旋锁丰满的互斥锁 | 51.c.h.o](#)
- [v26.xx 鸿蒙内核源码分析\(自旋锁篇\) | 自旋锁当立贞节牌坊 | 51.c.h.o](#)
- [v25.xx 鸿蒙内核源码分析\(并发并行篇\) | 听过无数遍的两个概念 | 51.c.h.o](#)
- [v24.xx 鸿蒙内核源码分析\(进程概念篇\) | 进程在管理哪些资源 | 51.c.h.o](#)
- [v23.xx 鸿蒙内核源码分析\(汇编传参篇\) | 如何传递复杂的参数 | 51.c.h.o](#)
- [v22.xx 鸿蒙内核源码分析\(汇编基础篇\) | CPU在哪里打卡上班 | 51.c.h.o](#)
- [v21.xx 鸿蒙内核源码分析\(线程概念篇\) | 是谁在不断的折腾CPU | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 .c .h .o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o ，我要CHO ，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

19_位图管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半用 | 51 .c .h .o

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 0011000111100101010100111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

000000000010000 = 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%x\n", flag);

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

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

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

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

百篇博客·往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o

- [v27.xx 鸿蒙内核源码分析\(互斥锁篇\) | 比自旋锁丰满的互斥锁 | 51.c.h.o](#)
- [v26.xx 鸿蒙内核源码分析\(自旋锁篇\) | 自旋锁当立贞节牌坊 | 51.c.h.o](#)
- [v25.xx 鸿蒙内核源码分析\(并发并行篇\) | 听过无数遍的两个概念 | 51.c.h.o](#)
- [v24.xx 鸿蒙内核源码分析\(进程概念篇\) | 进程在管理哪些资源 | 51.c.h.o](#)
- [v23.xx 鸿蒙内核源码分析\(汇编传参篇\) | 如何传递复杂的参数 | 51.c.h.o](#)
- [v22.xx 鸿蒙内核源码分析\(汇编基础篇\) | CPU在哪里打卡上班 | 51.c.h.o](#)
- [v21.xx 鸿蒙内核源码分析\(线程概念篇\) | 是谁在不断的折腾CPU | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 [51.c.h.o](#)，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 [.c.h.o](#) 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 [51.c.h.o](#)，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

20_用栈方式篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v20.xx 鸿蒙内核源码分析(用栈方式篇) | 程序运行场地谁提供的 | 51 .c .h .o

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:
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:
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:
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:
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保存的是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?

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常作品 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很简单，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o

- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 oschina gitee, 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c .h .o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 `51.c.h.o` ，我要CHO ，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

21_线程概念篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o

什么叫任务(LosTaskCB) ? 剖开五脏六腑看看长啥样 ? 栈空间的本质是什么?

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 | 51.c.h.o 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;           /**< Task 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亲和力和掩码,最多支持16核,亲和力和力很重要,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
#endif
#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亲和力和掩码,最多支持16核,亲和力和力很重要,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
#endif
#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, THUMB-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

#ifdef !defined(LOSCFG_ARCH_FPU_DISABLE)
    /* 0xAAA0000000000000LL : float reg initialed magic word */
    for (index = 0; index < FP_REGS_NUM; index++) {
        taskContext->D[index] = 0xAAA000000000000LL + 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;          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,没印象的看这篇 [鸿蒙内核源码分析\(双向链表篇\) | 谁是内核最重要结构体?](#),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
```

是简单的先后来后(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？就是它并不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！

百篇博客:往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作日 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o

- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c .h .o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. `51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

22_汇编基础篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51 .c .h .o

栈是CPU上班的地方 如何快速读懂汇编语言？ 鸿蒙用了多少汇编语言？

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。——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:
    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:
    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:
    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:
    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. 寄存器<->寄存器，内存<->寄存器，内存<->内存 搬运指令都不一样。

```
str    r1, [sp]    @ 寄存器->内存
ldr    r1, [sp, #4] @ 内存->寄存器
```

这又是一对，用于 内存<->寄存器之间，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CPU吗？

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o

- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o

- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

23_汇编传参篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51 .c .h .o

汇编怎么传结构体? 颠覆对栈的认知 读懂鸿蒙汇编前奏

汇编如何传复杂的参数?

- 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 | 51 .c .h .o 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.pc = 2
ldr    r2, [fp, #404] @取值cpu.Rn[0] = 1
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

结构体中共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, 这些值正好是memcpy没有拷贝到变量剩余的



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.pc = 2
    ldr    r2, [fp, #404] @取值cpu.Rn[0] = 1
    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, 因此SP位置要减4.
- 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 m
- 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.pc = 2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.Rn[0] = 1, 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fram
- 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.

百篇博客.往期回顾

在加注过程中, 整理出以下文章. 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. 确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样, 文章和注解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, `.xx` 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容.

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)

- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o

- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

[热爱是所有的理由和答案 - turing](#)

原创不易，欢迎转载，但麻烦请注明出处.

24_进程概念篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51 .c .h .o

进程(LosProcessCB) 进程负责管理什么? 进程间如何通讯?

本篇说清楚进程

读本篇之前建议先读[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表示了二层逻辑 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)
    ProclpcInfo 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[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表示了二层逻辑 数量和状态，点赞！

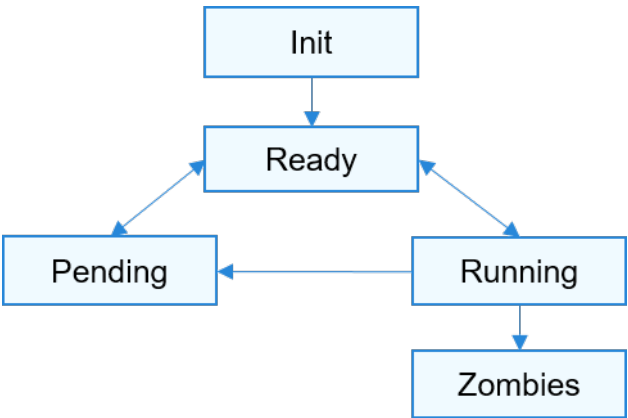
UINT16    priority;           /**< process priority */ //进程优先级
UINT16    policy;             /**< process policy */ //进程的调度方式，默认抢占式
UINT16    timeSlice;          /**< Remaining time slice *///进程时间片，默认2个tick
UINT16    consoleID;          /**< The console id of task belongs *///任务的控制台id归属
UINT16    processMode;        /**< 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 (LOS_CFG_KERNEL_LITEIPC == YES)
ProclpCInfo 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, 叫作进程空间, 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[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！

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o

- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

25_并行并发篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51 .c .h .o

并发是高速单车道 并行是高速多车道 鸿蒙如何支持并行处理

本篇说清楚并发并行

读本篇之前建议先读

- 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 | 51 .c .h .o 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(并发串行).

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，需要重新调度的了
    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
    HallrqSendIpi(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 OsMplnit(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.OsMplnit

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
        HallrqInitPercpu();//CPU硬件中断初始化

        OsCurrProcessSet(OS_PCB_FROM_PID(OsGetKernlInitProcessID())); //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 HallrqInitPercpu
- 初始化定时器队列 OsSwtmrInit
- 创建空闲任务 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 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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)

- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o

- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

26_自旋锁篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

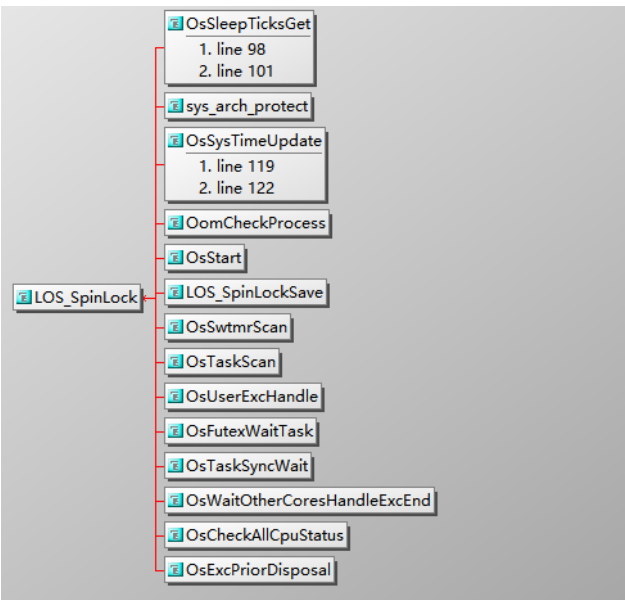
百篇博客系列篇.本篇为:

- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51 .c .h .o

自旋锁(SpinLock) 自旋锁解决了什么问题? 实现自旋锁为什么要用汇编?

本篇说清楚自旋锁

读本篇之前建议先读鸿蒙内核源码分析(总目录)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
    strxeq 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。笔者代码注释到这里那真是心潮澎湃，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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o

- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn

- [harmony](#)
- [oschina](#)

而巧合的是 .c .h .o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51 .c .h .o ，我要CHO ，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

27_互斥锁篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

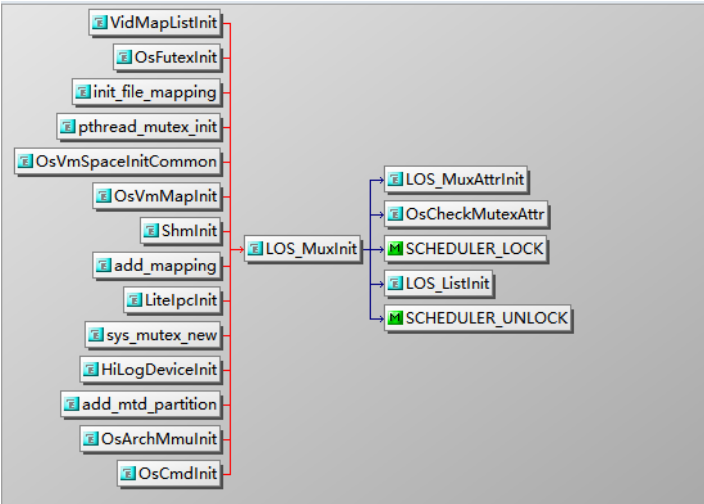
内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51 .c .h .o

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，先后来后，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了自己接着执行.
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.prioc ceiling) && (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.prioc ceiling); //2.把高优先级的mutex->attr.prioc ceiling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; //就走去再次调度流程
}

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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)

- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o

- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

28_进程通讯篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o

进程间为何需要通讯? IPC有哪些通讯方式? 每种IPC注解的源码在哪里?

进程间为何要通讯?

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，让各自过好各自的日子，但大家毕竟在一口锅里吃饭，不可能不与外界联系，联系就得有渠道，有规矩.

举两个应用场景说明下通讯的必要性:

一.被动式 广为熟知的shell命令 kill 9 13，是通过 shell任务给 13号进程发送一个干掉它的信号.

```
#define SIGKILL 9 //常用的命令 kill 9 13
```

这是被动式通讯的场景，至于为什么要干掉你，原因可能很多啊，很可能是检测到13占用内存太多了，也可能13太低调长期不活跃，启动新进程发现没位置了，得先收了你.总之系统必须得有对付你的抓手，可以随时登门查水电表.

二.主动式的，比如要访问某些公共资源(全局变量，消息队列)，而资源有限或具有排他性，别人正在使用导致你不能，所以需统一管理，要用就必须先申请，按规矩办事，毕竟和谐社会没规矩不成方圆.如果申请失败了就需要排队了，同时还要让出CPU给别人占用了，否则占着茅坑不办事这样对大家都不好撒.

大致有以下几种通讯需求:

- (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）。
- (4).资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供锁和同步机制。
- (5).进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

内核目录和系列篇更新

内核有个专门的IPC目录，详见如下. 可直接点击查看注解源码.

 los_event.c	规范和补充中断，定时器，互斥量，信号量模块注解.
 los_futex.c	关于内核这些问题你知道多少? .
 los_ipcdebug.c	鸿蒙内核源码分析系列篇 https://blog.csdn.net/kuangyufei
 los_mux.c	静态内存管理中节点处的实现很巧妙,点赞!
 los_queue.c	规范和补充中断，定时器，互斥量，信号量模块注解.
 los_queue_debug.c	Shell的本质是向外界提供一个窗口窥视内核.SHELLCMD_ENTRY(l, cmdType, cmdKey, paraNum, cmd...
 los_sem.c	规范和补充中断，定时器，互斥量，信号量模块注解.
 los_sem_debug.c	Shell的本质是向外界提供一个窗口窥视内核.SHELLCMD_ENTRY(l, cmdType, cmdKey, paraNum, cmd...
 los_signal.c	1.主从CPU是如何初始化的 2.搞明白了变量前缀 uc:UINT8 us:UINT16 uw:UINT32 代表的意思

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。

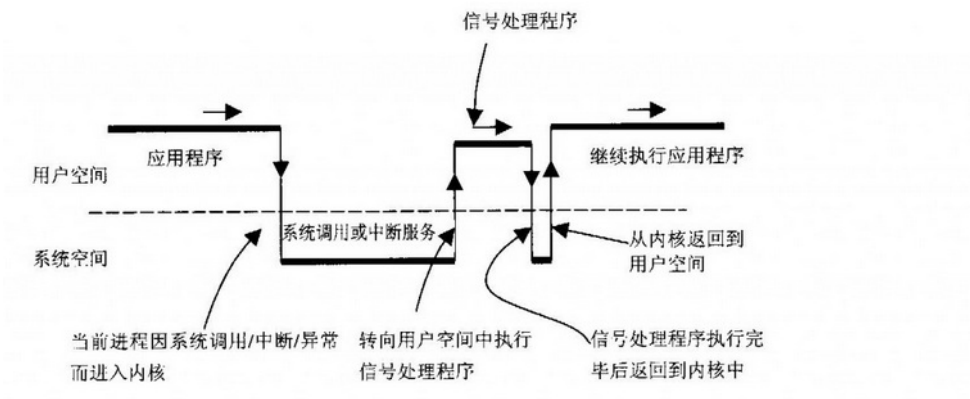
这部分后系列篇文件相关篇中会重点讲，敬请关注. 详细看 SysPipe 函数.

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来完成相应的处理(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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o

- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

29_信号量篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51 .c .h .o

信号量(Semaphore) 信号量解决任务同步问题 官方案例理解信号量

本篇说清楚信号量

读本篇之前建议先读[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 [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task
        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 .
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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， .xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51 .c .h .o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51 .c .h .o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51 .c .h .o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51 .c .h .o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51 .c .h .o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51 .c .h .o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51 .c .h .o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51 .c .h .o

- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o

- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

30_事件控制篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51 .c .h .o

本篇说清楚事件 (Event)

读本篇之前建议先读[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。

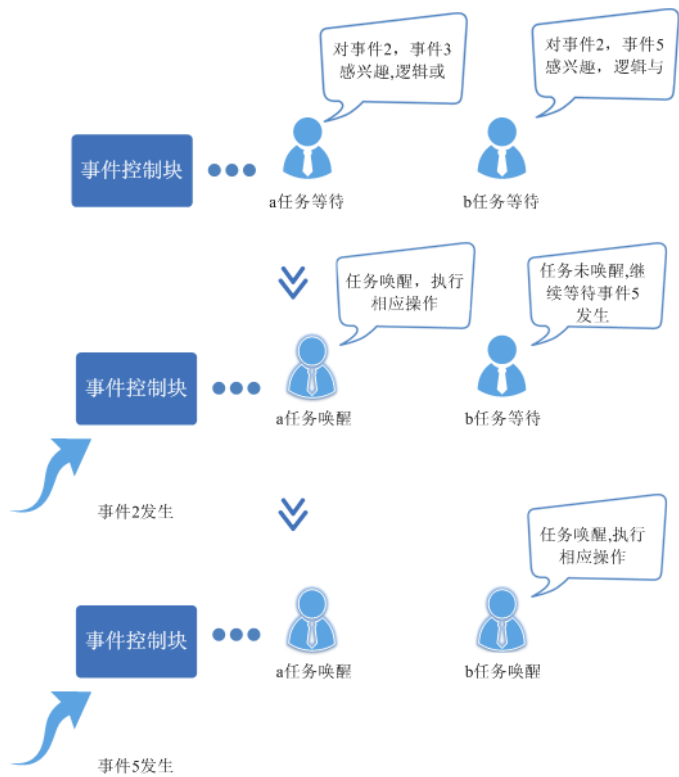
多任务环境下, 任务之间往往需要同步操作, 一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。

- 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 [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：
 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;
```

```

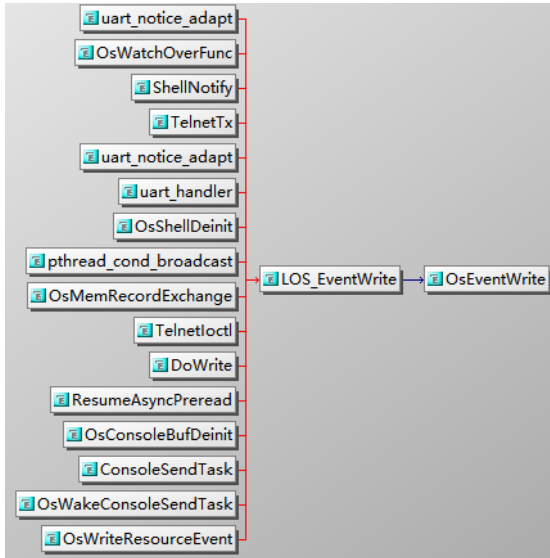
}

```

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;

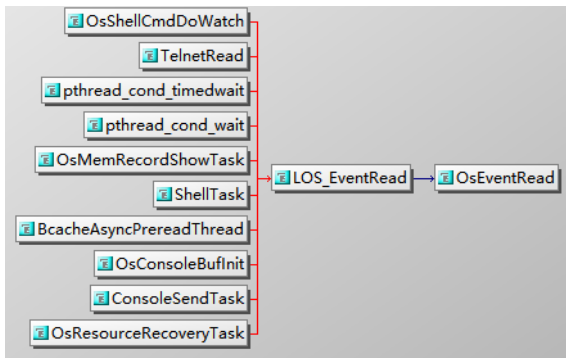
        resumedTask->taskEvent = NULL;
        OsTaskWake(resumedTask);//唤醒任务，加入就绪队列
    }

    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
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o

- [v27.xx 鸿蒙内核源码分析\(互斥锁篇\) | 比自旋锁丰满的互斥锁 | 51.c.h.o](#)
- [v26.xx 鸿蒙内核源码分析\(自旋锁篇\) | 自旋锁当立贞节牌坊 | 51.c.h.o](#)
- [v25.xx 鸿蒙内核源码分析\(并发并行篇\) | 听过无数遍的两个概念 | 51.c.h.o](#)
- [v24.xx 鸿蒙内核源码分析\(进程概念篇\) | 进程在管理哪些资源 | 51.c.h.o](#)
- [v23.xx 鸿蒙内核源码分析\(汇编传参篇\) | 如何传递复杂的参数 | 51.c.h.o](#)
- [v22.xx 鸿蒙内核源码分析\(汇编基础篇\) | CPU在哪里打卡上班 | 51.c.h.o](#)
- [v21.xx 鸿蒙内核源码分析\(线程概念篇\) | 是谁在不断的折腾CPU | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 [51.c.h.o](#)，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 [.c.h.o](#) 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 [51.c.h.o](#)，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

31_定时器篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51 .c .h .o

内核如何实现定时器? 各CPU核有各自的定时任务 最高优先级任务竟然是它!

本篇说清楚定时器的实现

读本篇之前建议先读[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_swtmrCBArry = 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

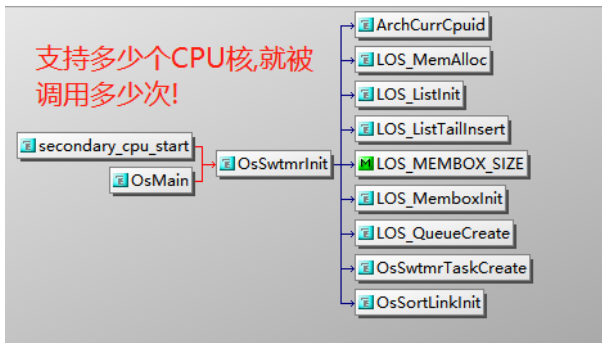
typedef struct { //处理软件定时器超时的回调函数的结构体
    SWTMR_PROC_FUNC handler; /**< Callback function that handles software timer timeout */ //处理软件定时器超时的回调函数
    UINTPTR arg; /* Parameter passed in when the callback function
                that handles software timer timeout is called */ //调用处理软件计时器超时的回调函数时传入的参数
} 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_swtmrCBArry` 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` 上挂的是一个一个的 `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;
}

```

代码解读:

- 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;
}
```

代码解读:

- 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
            }
        }
    }
}
```



```
}
```

代码解读:

- 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);
}

```

代码解读:

- 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_swtmrCBArrary 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.

百篇博客.往期回顾

在加注过程中, 整理出以下文章.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.确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样, 文章和注解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, .xx 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容.

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百, 依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用, 两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o

- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o

- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

32_CPU篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51 .c .h .o

内核如何描述CPU? 任务是理解CPU的主线 CPU空闲时在干什么?

本篇说清楚CPU

读本篇之前建议先读[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只不过是撸草打兔子. 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
```

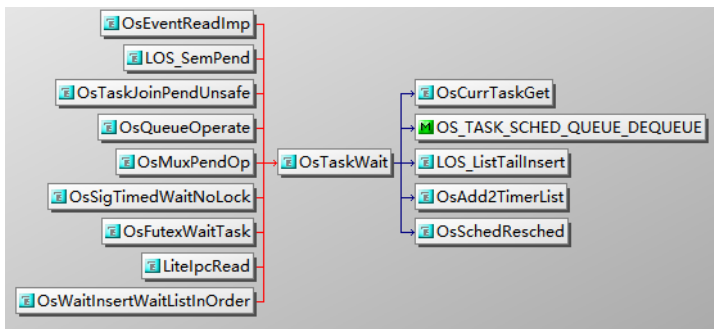
```

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，，这一步很关键，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[
        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，可前往以下查看。v31.xx (定时器篇) | 内核最高优先级任务是谁？看完后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[鸿蒙内核源码分析\(总目录\)](#)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. 这句话能理解吗?

- 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。

```

#ifdef (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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o

- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功, 也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。 `51.c.h.o` ，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

33_消息队列篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51 .c .h .o

进程间如何传递大数据? 设计原则 - 解耦 通信双方约定消息规则

本篇说清楚消息队列

读本篇之前建议先读[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_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，注意软件时钟定时器是由 g_swtmrCBArry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; //可写资源计数 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);

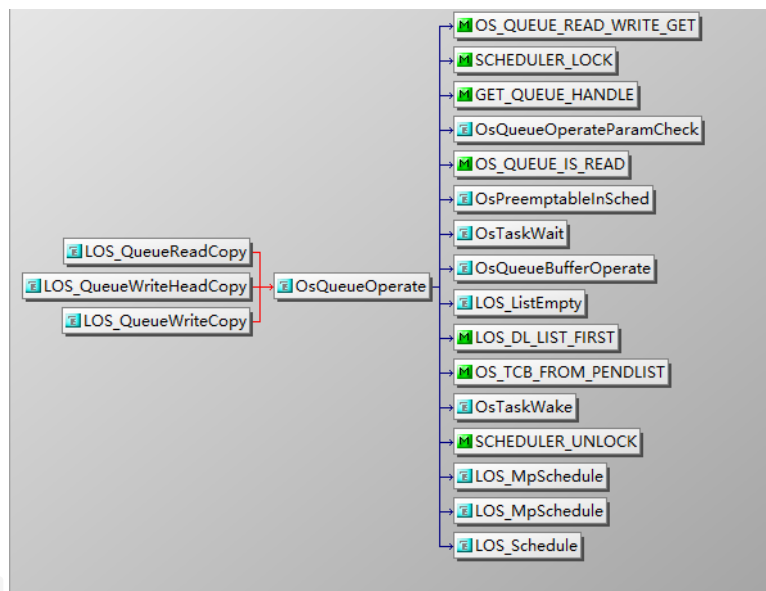
*queueID = queueCB->queueID; //带走队列ID
return LOS_OK;
}

```

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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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， .xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o

- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o

- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

34_原子操作篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51 .c .h .o

修改内存区域数据三步骤 读取-修改-写入 如何防止步骤中间被打断?

本篇说清楚原子操作

读本篇之前建议先读[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”操作在其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[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 `val` `status` `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 `val` `status` `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 = 1失败的可能性很大，不成功就重新来一遍，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GCC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
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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. 确实有难度, 自不量力, 但已经出发, 回头已是不可能的了。 :P

与代码有bug需不断debug一样, 文章和注解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, `.xx` 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百, 依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用, 两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯, 复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝, 七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o

- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件, 这就很有意思了, 冥冥之中似有天数, 将这四个宝贝以这种方式融合在一起. 51.c.h.o, 我要CHO, 嗯嗯, hin 顺口:)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码, 中文注解分析, 深挖地基工程, 大脑永久记忆, 四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核, 问答式导读, 生活式比喻, 表格化说明, 图形化展示, 主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

35_时间管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51 .c .h .o

时钟/机器/指令周期的含义 cpu各核独自统计Tick数 为何Tick不能用作准确时间?

本篇说清楚时间概念

读本篇之前建议先读[鸿蒙内核源码分析\(总目录\)](#)其他篇.

时间概念太重要了,在鸿蒙内核又是如何管理和使用时间的呢?

时间管理以系统时钟 `g_sysClock` 为基础,给应用程序提供所有和时间有关的服务。

- 用户以秒、毫秒为单位计时.
- 操作系统以Tick为单位计时,这个认识很重要. 每秒的tick大小很大程度上决定了内核调度的次数多少.
- 当用户需要对系统进行操作时,例如任务挂起、延时等,此时需要时间管理模块对Tick和秒/毫秒进行转换。

熟悉两个概念:

- Cycle(周期):系统最小的计时单位。Cycle的时长由系统主时钟频率决定,系统主时钟频率就是每秒钟的Cycle数。
- Tick(节拍):Tick是操作系统的基本时间单位,由用户配置的每秒Tick数决定,可大可小。

怎么去理解他们之间的关系呢?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,意味着正常情况下每秒100次检查
#endif
#define OS_CYCLE_PER_TICK (g_sysClock / LOSCFG_BASE_CORE_TICK_PER_SECOND) //每个tick多少机器周期
```

时钟周期(振荡周期)

在鸿蒙 `g_sysClock` 表示时钟周期,是CPU的赫兹,也就是上面说的 `Cycle`,这是固定不变的,由硬件晶振的频率决定的. `OsMain` 是内核运行的第一个C函数,首个子函数就是 `osRegister`,完成对 `g_sysClock` 的赋值


```
LITE_OS_SEC_TEXT_INIT VOID osRegister(VOID)
{
    g_sysClock = OS_SYS_CLOCK; //获取CPU HZ
    g_tickPerSecond = LOSCFG_BASE_CORE_TICK_PER_SECOND; //每秒节拍数 默认100 即一个tick = 10ms
    return;
}
```

CPU周期也叫(机器周期)

在鸿蒙宏 `OS_CYCLE_PER_TICK` 表示机器周期，Tick 由用户根据实际情况配置. 例如:主频为1G的CPU，其振荡周期为: 1吉赫 (GHz 10⁹ Hz) = 1 000 000 000 Hz 当Tick为100时，则1 000 000 000/100 = 10000000，即一个tick内可产生1千万个CPU周期.CPU就是用这1千万个周期去执行指令的.

指令周期

指令周期是执行一条指令所需要的时间，一般由若干个机器周期组成。指令不同，所需的机器周期数也不同。对于一些简单的单字节指令，在取指令周期中，指令取出到指令寄存器后，立即译码执行，不再需要其它的机器周期。对于一些比较复杂的指令，例如转移指令、乘法指令，则需要两个或者两个以上的机器周期。通常含一个机器周期的指令称为单周期指令，包含两个机器周期的指令称为双周期指令。

Tick硬中断函数

```
LITE_OS_SEC_BSS volatile UINT64 g_tickCount[LOSCFG_KERNEL_CORE_NUM] = {0}; //tick计数器，系统一旦启动，一直在++，为防止溢出，这是一个 UI
LITE_OS_SEC_DATA_INIT UINT32 g_sysClock; //系统时钟，是绝大部分部件工作的时钟源，也是其他所有外设的时钟的来源
LITE_OS_SEC_DATA_INIT UINT32 g_tickPerSecond; //每秒Tick数，鸿蒙默认是每秒100次，即:10ms
LITE_OS_SEC_BSS DOUBLE g_cycle2NsScale; //周期转纳秒级

/* spinlock for task module */
LITE_OS_SEC_BSS SPIN_LOCK_INIT(g_tickSpin); //节拍器自旋锁
#define TICK_LOCK(state)      LOS_SpinLockSave(&g_tickSpin, &(state))
/*
 * Description : Tick interruption handler
 */
/*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);
    g_tickCount[ArchCurrCpuId()]++; //当前CPU核计数器
    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

    #if (LOSCFG_BASE_CORE_SWTMR == YES)
        OsSwtmrScan(); //定时器扫描，看是否有超时的定时器
    #endif
}

#ifdef __cplusplus
    #if __cplusplus
}
```

解读

- `g_tickCount` 记录每个CPU核tick的数组，每次硬中断都触发 `OsTickHandler`，每个CPU核单独计数。
- `OsTickHandler` 是内核调度的动力，其中会检查任务时间片是否用完，定时器是否超时.主动delay的任务是否需要被唤醒，其本质是个硬中断，在 `HalClockInit` 硬时钟初始化时创建的，具体在硬中断篇中会详细讲解。
- `TICK_LOCK` 是tick操作的自旋锁，宏原型 `LOS_SpinLockSave` 在自旋锁篇中已详细介绍。

功能函数

```
#define OS_SYS_MS_PER_SECOND 1000 //一秒多少毫秒
//获取自系统启动以来的Tick数
LITE_OS_SEC_TEXT_MINOR UINT64 LOS_TickCountGet(VOID)
{
    UINT32 intSave;
    UINT64 tick;

    /*
     * use core0's tick as system's timeline ,
     * the tick needs to be atomic.
     */
    TICK_LOCK(intSave);
    tick = g_tickCount[0]; //使用CPU core0作为系统的 tick数
    TICK_UNLOCK(intSave);

    return tick;
}
//每个Tick多少Cycle数
LITE_OS_SEC_TEXT_MINOR UINT32 LOS_CyclePerTickGet(VOID)
{
    return g_sysClock / LOSCFG_BASE_CORE_TICK_PER_SECOND;
}
//毫秒转换成Tick
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;
}
//Tick转化为毫秒
LITE_OS_SEC_TEXT_MINOR UINT32 LOS_Tick2MS(UINT32 tick)
{
    return ((UINT64)tick * OS_SYS_MS_PER_SECOND) / LOSCFG_BASE_CORE_TICK_PER_SECOND;
}
```

说明

- 在CPU篇中讲过，0号CPU核默认为主核，默认获取自系统启动以来的Tick数使用的是 `g_tickCount[0]`
- 因每个CPU核的tick是独立计数的，所以 `g_tickCount` 中各值是不一样的。
- 系统的Tick数在关中断的情况下不进行计数，因为 `OsTickHandler` 本质是由硬中断触发的，屏蔽硬中断的情况下就不会触发 `OsTickHandler`，自然也就不会有 `g_tickCount[ArchCurrCpuId()]++` 的计数，所以系统Tick数不能作为准确时间使用。
- 追问下，什么情况下硬中断会被屏蔽？

编程示例

前提条件：

- 使用每秒的Tick数`LOSCFG_BASE_CORE_TICK_PER_SECOND`的默认值100。
- 配好`OS_SYS_CLOCK`系统主时钟频率。

时间转换

```
VOID Example_TransformTime(VOID)
{
    UINT32 ms;
    UINT32 tick;
```

```
tick = LOS_MS2Tick(10000); // 10000ms转换为tick
dprintf("tick = %d \n", tick);
ms = LOS_Tick2MS(100); // 100tick转换为ms
dprintf("ms = %d \n", ms);
}
```

时间转换结果

```
tick = 1000
ms = 1000
```

时间统计和时间延迟

```
LITE_OS_SEC_TEXT UINT32 LOS_TaskDelay(UINT32 tick);
VOID Example_GetTime(VOID)
{
    UINT32 cyclePerTick;
    UINT64 tickCount;

    cyclePerTick = LOS_CyclePerTickGet();
    if(0 != cyclePerTick) {
        dprintf("LOS_CyclePerTickGet = %d \n", cyclePerTick);
    }

    tickCount = LOS_TickCountGet();
    if(0 != tickCount) {
        dprintf("LOS_TickCountGet = %d \n", (UINT32)tickCount);
    }

    LOS_TaskDelay(200); //延迟200个tick
    tickCount = LOS_TickCountGet();
    if(0 != tickCount) {
        dprintf("LOS_TickCountGet after delay = %d \n", (UINT32)tickCount);
    }
}
```

时间统计和时间延迟结果

```
LOS_CyclePerTickGet = 495000 //取决于CPU的频率
LOS_TickCountGet = 1 //实际情况不一定是1的
LOS_TickCountGet after delay = 201 //实际情况不一定是201，但二者的差距会是200
```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o

- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o

- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要 CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

36_工作模式篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51 .c .h .o

CPU在哪七种模式下工作? 开机代码放哪了?长什么样? 系统调用靠什么实现的?

系列篇硬件部分说明基于ARM720T.pdf文档.

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[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

先看一张图,图来源于 [ARM720T.pdf](#)第43页,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，内核肯定立马闪崩.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？

再看一张图，图来源于 [ARM720T.pdf](#) 第56页

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 _osExceptUndefinHdl @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 _osExceptPrefetchHdl @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 OslrqHandler      @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 含义见 ARM720T.PDF 第64页
/* 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).

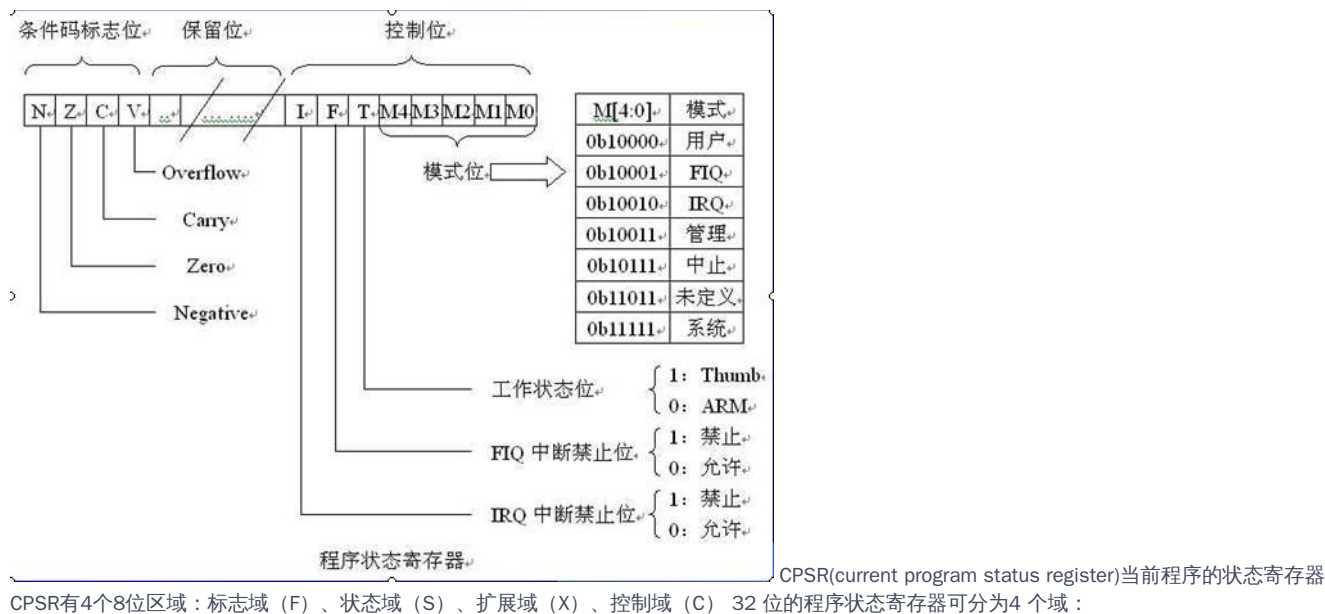
可以看出swi的优先级最低,swi就是软中断,系统调用就是通过它来实现的.

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



- 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

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o

- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件, 这就很有意思了, 冥冥之中似有天数, 将这四个宝贝以这种方式融合在一起. 51.c.h.o, 我要CHO, 嗯嗯, hin 顺口:)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码, 中文注解分析, 深挖地基工程, 大脑永久记忆, 四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核, 问答式导读, 生活式比喻, 表格化说明, 图形化展示, 主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

37_系统调用篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

- 内核源码注解分析 →
- OpenHarmony开发者文档 →

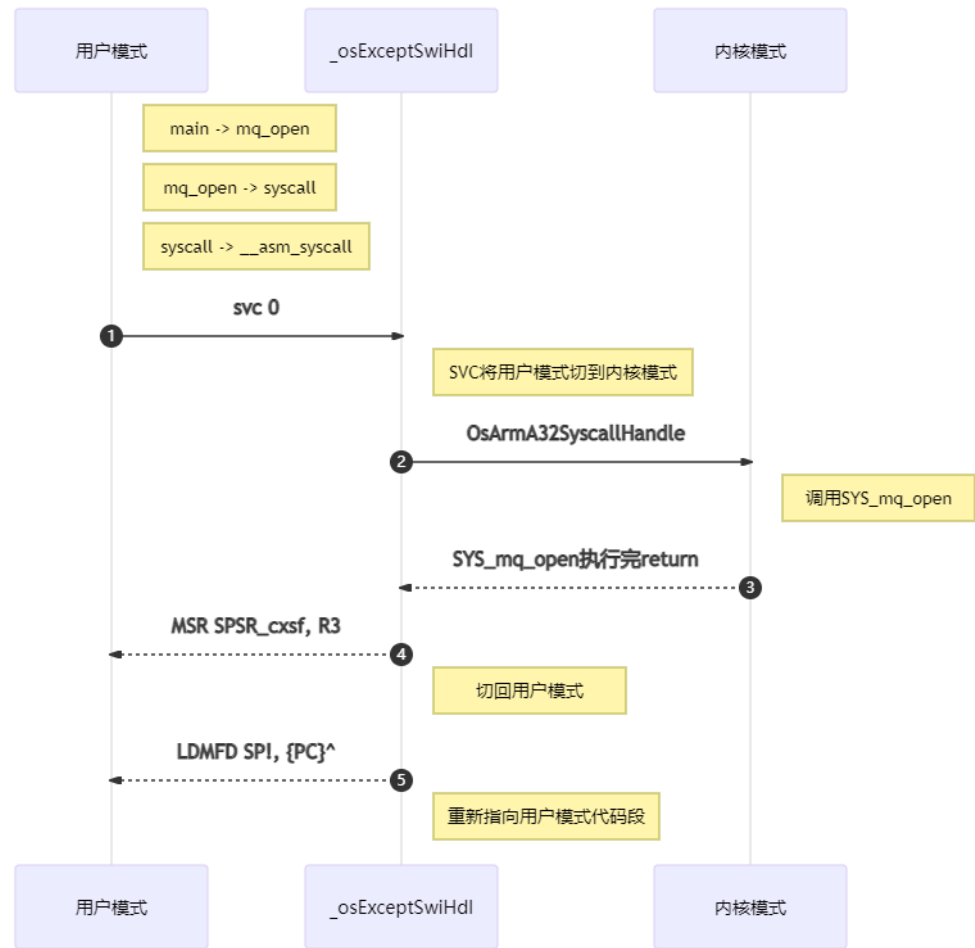
百篇博客系列篇.本篇为:

- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51 .c .h .o

本篇说清楚系统调用

读本篇之前建议先读鸿蒙内核源码分析(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，将参数传给寄存器 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], msgptr1[BUFFER], msgptr2[BUFFER];
    const char *msgptr1 = "test message1";
    const char *msgptr2 = "test message2 with different 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_HANDLER` 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_HANDLER(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_HANDLER

SYSCALL_HANDLER(__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

```
//切换到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 _osExceptUndefinHdl @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 OslrqHandler      @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。但你也可能会好奇，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.USR, 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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51 .c .h .o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51 .c .h .o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o

- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要 CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

38_寄存器篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51 .c .h .o

本篇说清楚寄存器

ARM系列篇基于[ARM720T.pdf](#)文档.

读本篇之前建议先读[鸿蒙内核源码分析\(总目录\)](#)arm体系系列篇.

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,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](https://gitee.com/weharmony)

ARM state program status registers [weharmony.github.io](https://github.com/weharmony)

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 [鸿蒙内核源码分析\(总目录\)](#)之工作模式篇中有详细, 可自行前往查看. 此处只简单说明下. 下图来源于 [ARM720T.pdf](#)第43页, 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.

RO~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 含义见 ARM720T.PDF 第64页
/* 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[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;
    UIPTPTR 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。

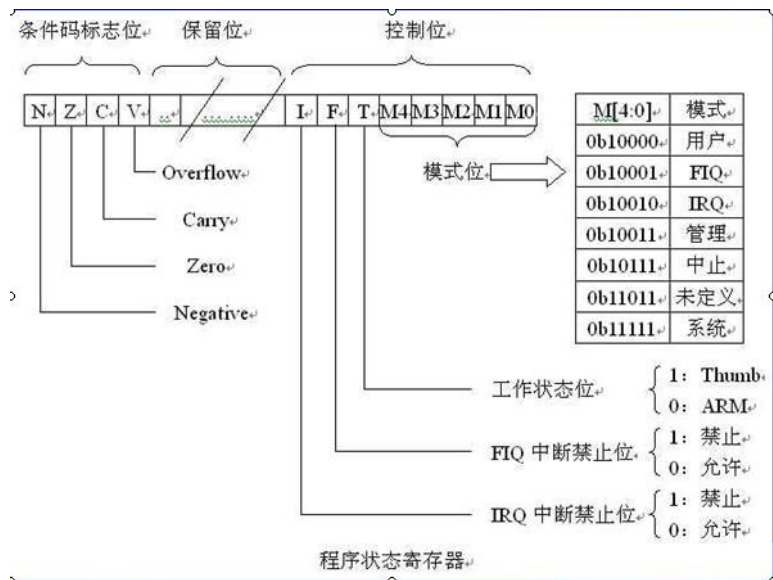
PC(R15) 寄存器

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R12夹在中间不上不下的，这又是为什么呢？

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o

- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

39_异常接管篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o 系列篇ARM部分说明基于ARM720T.pdf文档.

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,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

图来源于 [ARM720T.pdf](#)第43页,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.

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：

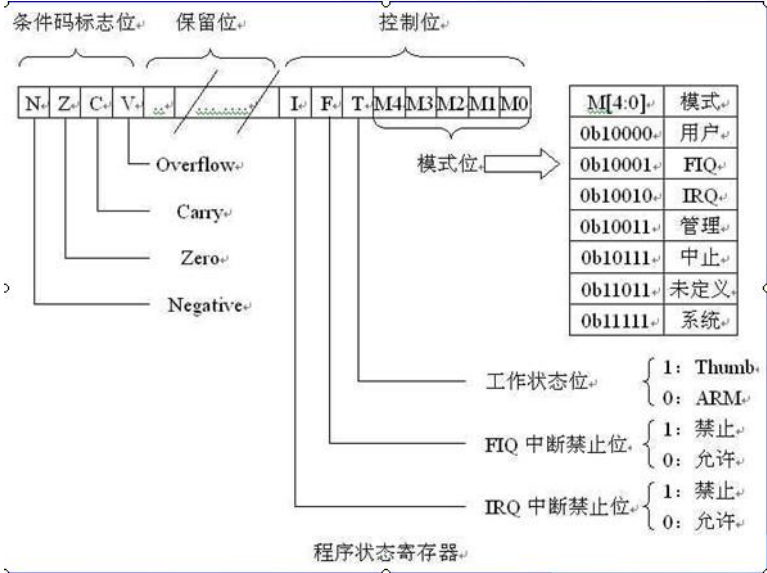
- 一个是代码要切到哪个位置，也就是要重置PC寄存器，每种异常模式下的切换方式如图：

weharmony.gitee.io 来源:鸿蒙内核源码分析

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
UDEFa	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:

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

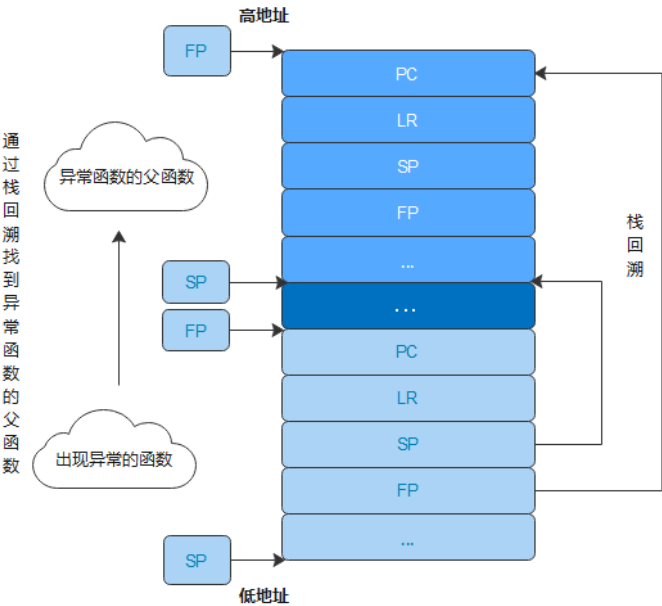
Table 2-2 PSR mode bit values (continued)

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(Prefetch abort)

```

@ Description: Prefetch 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 OsUndefIncExchHandleEntry
#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](https://gitee.com/weharmony)

ARM state program status registers [weharmony.github.io](https://github.com/weharmony)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

详见 系列篇之寄存器篇

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!

百篇博客.往期回顾

在加注过程中，整理出以下文章。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语屈聱牙的概念，那没什么意思。更希望让内核变得栩栩如生，倍感亲切.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o

- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很简单,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o

- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

40_汇编汇总篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51 .c .h .o

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，真的是很傻很简单。

鸿蒙内核源码分析系列篇至少已经有五篇涉及到了汇编，请自行翻看，但还是远远不够，要写十五篇，彻底摸透，现在才刚刚开始，本篇先整理鸿蒙内核所有汇编文件和大概说明文件的作用，后续一块一块来剥，不把这些汇编剥个精光不罢休。

汇编目录

鸿蒙所有汇编文件如下: 直接点击可以查看注解源码，有些站点会把链接去除，没办法，可直接去各大站点搜"鸿蒙内核源码分析"，找到源码注解。

- \arch\arm\arm\src
 - startup 启动相关
 - reset_vector_mp.S 多核CPU下启动代码，大文件
 - reset_vector_up.S 单核CPU下启动代码，大文件
 - armv7a
 - cache.S 缓存相关的两个函数
 - los_dispatch.S 异常分发处理，大文件.
 - los_hw_exc.S 硬件异常相关，大文件.
 - los_hw_runstop.S OsSRSaveRegister 和 OsSRRestoreRegister 汇编实现
 - jmp.S 两个简单的跳转函数
 - hw_user_get.S 拷贝用户空间数据到内核空间
 - hw_user_put.S 拷贝内核空间数据到用户空间

hw_user_get.S

将用户空间数据src 拷贝到内核空间 dst

```
// errno_t_arm_get_user(void *dst, const void *src, size_t dstTypeLen, size_t srcTypeLen)
FUNCTION(_arm_get_user)
    stmdb    sp!, {r0, r1, r2, r3, lr} @四个参数入栈，保存LR
    cmp     r2, #0 @r2 和 0比较
    beq     .Lget_user_return @相等 跳到Lget_user_return 直接返回
    cmp     r2, r3 @r2 和 r3比较
    bne     .Lget_user_err @不等，说明函数要返回错误
```

```

    cmp    r2, #1 @r2 和 1比较
    bhi    .Lget_user_half @if(dstTypeLen>1) 跳转到Lget_user_half
.Lget_user_byte: @按字节拷贝数据
0: ldrbt   r3, [r1], #0 @r3=*r1
1: strb    r3, [r0], #0 @*r0=r3
    b      .Lget_user_return
.Lget_user_half:
    cmp    r2, #2 @r2 和 2比较
    bhi    .Lget_user_word @if(dstTypeLen>2) Lget_user_word
2: ldrht   r3, [r1], #0 @完成最后一个字节的拷贝
3: strh    r3, [r0], #0 @完成最后一个字节的拷贝
    b      .Lget_user_return
.Lget_user_word:
    cmp    r2, #4 @r2 和 4比较
    bhi    .Lget_user_err @if(dstTypeLen>4) 跳转到Lget_user_err
4: ldrt    r3, [r1], #0
5: str     r3, [r0], #0
.Lget_user_return: @返回锚点
    ldmia   sp!, {r0, r1, r2, r3, lr} @保存的内容出栈, 恢复各寄存器值
    mov     r0, 0 @r0保存返回值为0
    bx      lr @跳回调用函数继续执行, _arm_get_user到此结束!
.Lget_user_err:
    ldmia   sp!, {r0, r1, r2, r3, lr} @保存的内容出栈, 恢复各寄存器值
    mov     r0, #-14 @r0保存返回值为-14
    bx      lr @跳回调用函数继续执行, _arm_get_user到此结束!

.pushsection __exc_table, "a"
.long 0b, .Lget_user_err
.long 1b, .Lget_user_err
.long 2b, .Lget_user_err
.long 3b, .Lget_user_err
.long 4b, .Lget_user_err
.long 5b, .Lget_user_err
.popsection

```

解读

- 用户空间和内核空间的数据为什么需要拷贝? 这是个经典问题, 看了网上的一些回答, 没毛病:

内核不能信任任何用户空间的指针。必须对用户空间的指针指向的数据进行验证。如果只做验证不做拷贝的话, 那么在随后的运行中要随时受到其它进/线程可能修改用户空间数据的威胁。所以必须做拷贝。

在内存系列篇中已经反复的说过, 每个用户进程都有自己独立的用户空间, 但这个用户空间是通过MMU映射出来的, 是表面上繁花似锦, 背后都共用着真正的物理内存, 所以在高频率的任务切换过程中, 原有的用户空间地址内容很容易被覆盖掉.举个例子说明下:

- 用户A有个美女西施放在万聪酒店21号房说要献给内核大佬, 如果内核不直接把美女接回家, 而仅仅是做个记录, 写着西施在万聪酒店21号房, 内核大佬立马跑去, 还不会错能拿对人, 但如果被其他事给耽搁了呢?
- 耽搁的这功夫, 调度算法把万聪酒店21号房给了用户B使用, 当然用户B使用之前, 酒店管理人员会把西施置换个地方(以至于用户A再回到酒店时, 原来的东西该怎么还咋样还原). 等21号房空出来了, B肯定不知道原来的房间是A在用, 而且里面曾经还过有个美女西施, 更不可能晓得A把西施献给内核大佬这回事了.因为B的业务需要, 很可能往21号房整了个东施进来.
- 此时如果内核大佬事忙完了, 想起用户A献美女的事了, 是时候了.因为只记录了地址, 直接去万聪酒店21号房抓人, 可这会抓出来那是咱东施小姐呀.这可不把事给搞砸啦.
- 所以需要跨空间拷贝, 直接把美女接回家找个地方关起来先.

reset_vector_mp.S 和 reset_vector_up.S

鸿蒙开机代码根据 CPU多核还是单核分成了两个独立文件处理. mp 就是多处理器(multiprocessing)的意思: 多CPU核的操作系统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。

up (unit processing)的意思, 单个CPU, 虽然没mp的复杂, 但文件也很大 500行汇编, 一小节讲不完, 需要单独的一篇专讲 **reset_vector**

这里只列出up情况下的开机代码

```

reset_vector: @鸿蒙单核cpu 开机代码
/* do some early cpu setup: i/d cache disable , mmu disabled */
mrc    p15 , 0 , r0 , c1 , c0 , 0
bic    r0 , #(1<<12)
bic    r0 , #(1<<2 | 1<<0)
mcr    p15 , 0 , r0 , c1 , c0 , 0

/* r11: delta of physical address and virtual address */
adr    r11 , pa_va_offset
ldr    r0 , [r11]
sub    r11 , r11 , r0

/* if we need to relocate to proper location or not */
adr    r4 , __exception_handlers      /* r4: base of load address */
ldr    r5 , =SYS_MEM_BASE             /* r5: base of physical address */
subs   r12 , r4 , r5                  /* r12: delta of load address and physical address */
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 */

```

los_dispatch.S 和 los_hw_exc.S

异常模式处理入口和统一分发现实，之前也有提到过，很复杂，1000多行，后续单独细说实现过程。

jmp.S

两个简单的函数 `longjmp` `setjmp` 的实现，加注解部分请前往 [鸿蒙内核源码注解分析](#) 查看

```

FUNCTION(longjmp)
    ldmbd    r0 , {r4-r12}
    add      r0 , #(4 * 9)
    ldr      r13 , [r0]
    add      r0 , #4
    ldr      r14 , [r0]
    cmp      r1 , #0
    moveq    r1 , #1
    mov      r0 , r1
    mov      pc , lr

FUNCTION(setjmp)
    stmea    r0 , {r4-r12}
    add      r0 , #(4 * 9)
    str      r13 , [r0]
    add      r0 , #4
    str      r14 , [r0]
    mov      r0 , #0
    mov      pc , lr

```

los_hw_runstop.S

```

.global OsSRSaveRegister
.global OsSRRestoreRegister

```

两个函数的汇编现实，有点复杂，后续单独说明。

cache.S

这是缓存部分的两个函数实现，此处没有加注解，试着看明白这两个函数的实现.加注解部分请前往 [鸿蒙内核源码注解分析](#) 查看

```

.macro DCACHE_LINE_SIZE , reg , tmp
    mrc    p15 , 0 , \tmp , c0 , c0 , 1

```

```

    lsr    \tmp, \tmp, #16
    and    \tmp, \tmp, #0xf
    mov     \reg, #4
    mov     \reg, \reg, lsl \tmp
.endm

FUNCTION(arm_inv_cache_range)
    push    {r2, r3}
    DCACHE_LINE_SIZE r2, r3
    sub     r3, r2, #1
    tst     r0, r3
    bic     r0, r0, r3

    mcrne   p15, 0, r0, c7, c14, 1

    tst     r1, r3
    bic     r1, r1, r3
    mcrne   p15, 0, r1, c7, c14, 1
1:
    mcr     p15, 0, r0, c7, c6, 1
    add     r0, r0, r2
    cmp     r0, r1
    blo     1b
    dsb
    pop     {r2, r3}
    mov     pc, lr

FUNCTION(arm_clean_cache_range)
    push    {r2, r3}
    DCACHE_LINE_SIZE r2, r3
    sub     r3, r2, #1
    bic     r0, r0, r3

1:
    mcr     p15, 0, r0, c7, c10, 1
    add     r0, r0, r2
    cmp     r0, r1
    blo     1b
    dsb
    pop     {r2, r3}
    mov     pc, lr

```

百篇博客.往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o

- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o

- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

41_任务切换篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51 .c .h .o

任务上下文 TaskContext 代码实现细节是怎样的? 汇编逐步跟踪切换过程

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[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 [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。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 `STMFDP 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打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打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.

百篇博客.往期回顾

在加注过程中,整理出以下文章.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话屈辱的概念,那没什么意思.更希望让内核变得栩栩如生,倍感亲切.确实有难度,自不量力,但已经出发,回头已是不可能的了. :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容.

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o

- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o

- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

42_中断切换篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51 .c .h .o

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[鸿蒙内核源码分析\(总目录\)](#)查看.
- **中断管理篇** 从中断初始化 `HallrqInit` 开始,到注册中断的 `LOS_HwiCreate` 函数,到消费中断函数的 `HallrqHandler`,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[鸿蒙内核源码分析\(总目录\)](#)查看.
- **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[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。

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](https://gitee.com/weharmony)

ARM state program status registers [weharmony.github.io](https://github.com/weharmony)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

- 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 `HallIrqHandler`，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
#ifdef (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 OsrIrqHandler 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

```

OsrIrqHandler: @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    OsrIrqFromKernel @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

OsrIrqFromKernel: @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 HallIrqHandler /* 调用硬中断处理程序，无参，说明HallIrqHandler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，接下来是TaskIrqContext.USR，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.USR，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.USR，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) .....

```

- 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: `STMTFD 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   HallIrqHandler /* 调用硬中断处理程序, 无参, 说明HallIrqHandler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 `HallIrqHandler` `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

百篇博客.往期回顾

在加注过程中, 整理出以下文章. 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. 确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样, 文章和注解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, `.xx` 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容.

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o

- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o

- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要 CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

43_中断概念篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51 .c .h .o

关于中断部分系列篇将用三篇详细说明整个过程.

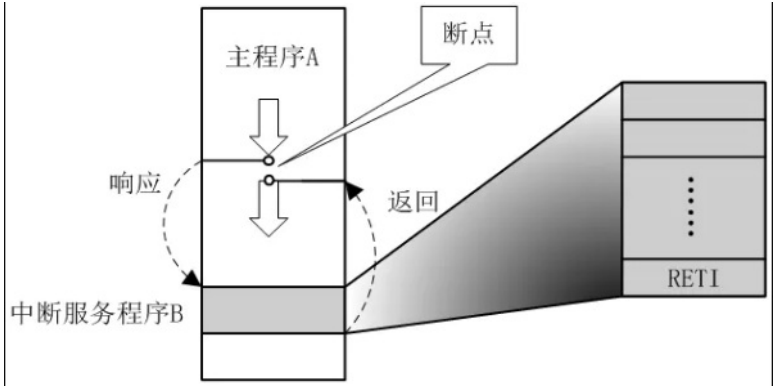
- **中断概念篇** 中断概念很多, 比如中断控制器, 中断源, 中断向量, 中断共享, 中断处理程序等等.本篇做一次整理.先了解透概念才好理解中断过程.本篇的主角是海公公, 用海公公打比方说明白中断各个概念.
- **中断管理篇** 从中断初始化 `HallrqInit` 开始, 到注册中断的 `LOS_HwiCreate` 函数, 到消费中断函数的 `HallrqHandler`, 剖析鸿蒙内核实现中断的过程, 很像设计模式中的观察者模式. 可前往[鸿蒙内核源码分析\(总目录\)](#)查看.
- **中断切换篇** 用自下而上的方式, 从中断源头纯汇编代码往上跟踪代码细节.说清楚保存和恢复中断现场 `TaskIrqContext` 过程.可前往[鸿蒙内核源码分析\(总目录\)](#)查看.

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?

- 中断是指出现需要时, CPU暂停执行当前程序, 转而执行新程序的过程.即在程序运行过程中, 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。

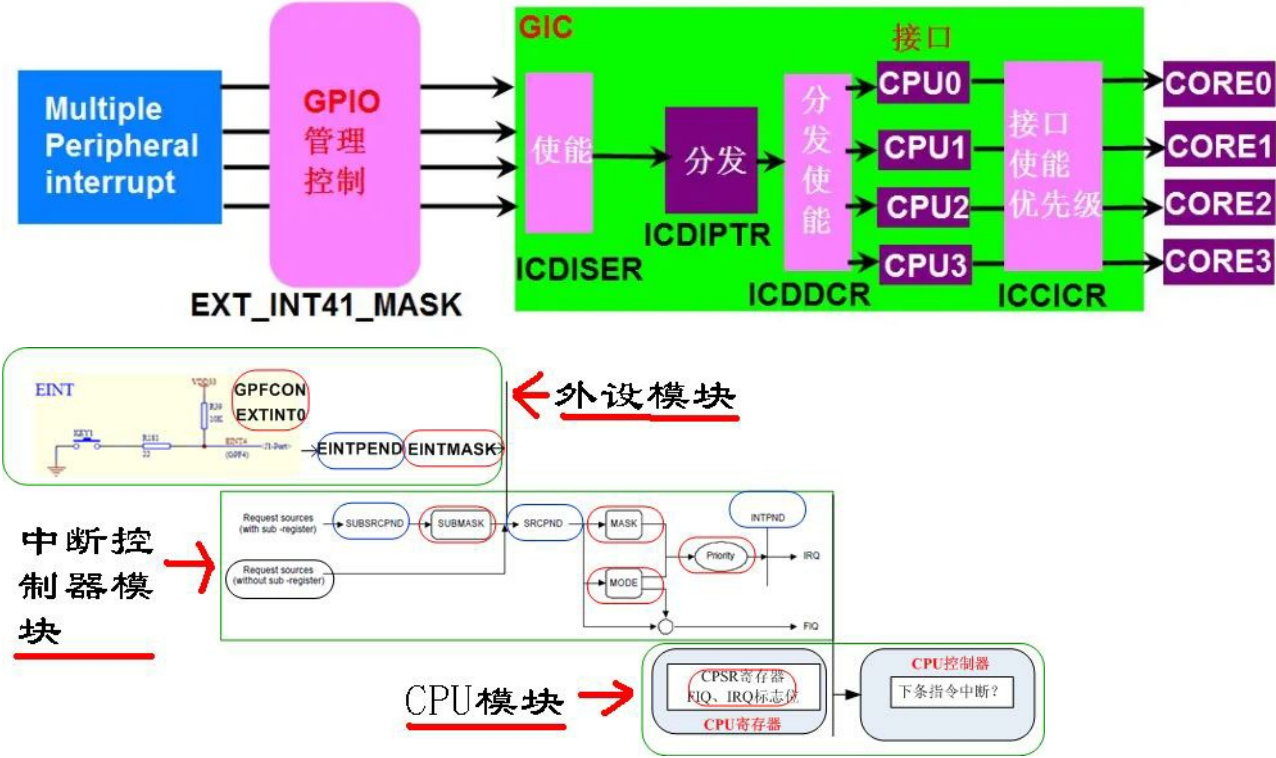
中断相关的硬件介绍

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.

```
#ifndef 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_HPIR (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超限报警等事件。
- 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，如存储器超限、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。

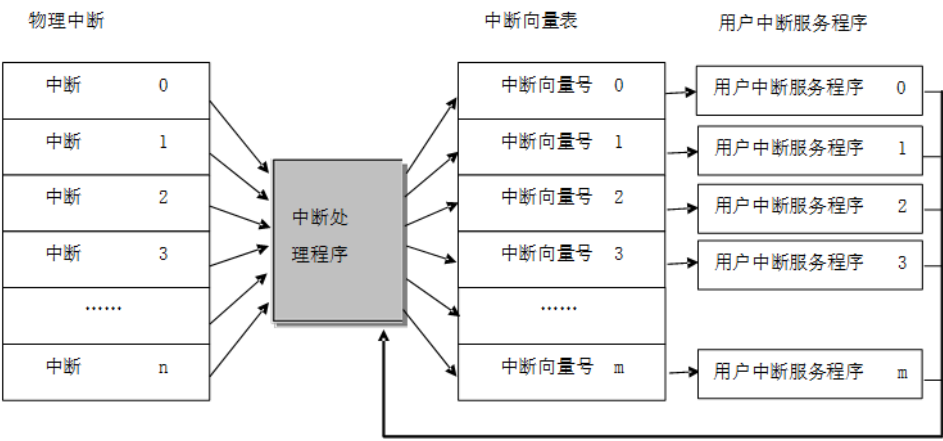
中断处理程序

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)

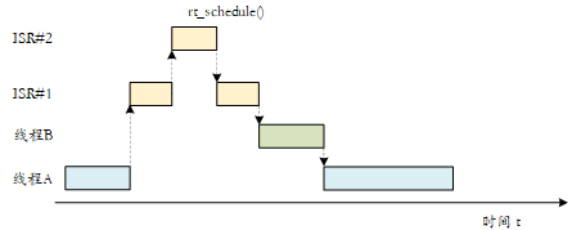
在用户中断服务程序（ISR）中，分为两种情况:

- 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
打开和关闭所有中断	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
使能和屏蔽指定中断	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）
清除中断寄存器状态	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）

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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， `.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)

- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o

- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

44_中断管理篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51 .c .h .o

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[鸿蒙内核源码分析\(总目录\)](#)查看.
- **中断管理篇(本篇)** 从中断初始化 `HallrqInit` 开始, 到注册中断的 `LOS_HwiCreate` 函数, 到消费中断函数的 `HallrqHandler`, 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. 可前往[鸿蒙内核源码分析\(总目录\)](#)查看.

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 HallrqInit(VOID)
{
    UINT32 i;

    /* set external 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 external 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;
}

HallrqInitPercpu();//初始化当前CPU中断信息

/* enable gic distributor control */
GIC_REG_32(GICC_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 HallrqInitPercpu(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	<i>Aliased Highest Priority Pending Interrupt Register; GICC_AHPPIR on page 4-148</i>
GICC_BPR	<i>Binary Point Register; GICC_BPR on page 4-133</i>
GICC_CTLR	<i>CPU Interface Control Register; GICC_CTLR on page 4-125</i>
GICC_DIR	<i>Deactivate Interrupt Register; GICC_DIR on page 4-153</i>
GICC_EOIR	<i>End of Interrupt Register; GICC_EOIR on page 4-138</i>
GICC_HPPIR	<i>Highest Priority Pending Interrupt Register; GICC_HPPIR on page 4-143</i>
GICC_IAR	<i>Interrupt Acknowledge Register; GICC_IAR on page 4-135</i>
GICC_IIDR	<i>CPU Interface Identification Register; GICC_IIDR on page 4-152</i>
GICC_NSAPRn	<i>Non-secure Active Priorities Registers; GICC_NSAPRn on page 4-151</i>
GICC_PMR	<i>Interrupt Priority Mask Register; GICC_PMR on page 4-131</i>
GICC_RPR	<i>Running Priority Register; GICC_RPR on page 4-142</i>
GICD_CPENDSGIRn	<i>SGI Clear-Pending Registers; GICD_CPENDSGIRn on page 4-115</i>
GICD_CTLR	<i>Distributor Control Register; GICD_CTLR on page 4-85</i>
GICD_ICACTIVERn	<i>Interrupt Clear-Active Registers; GICD_ICACTIVERn on page 4-103</i>
GICD_ICENABLERn	<i>Interrupt Clear-Enable Registers; GICD_ICENABLERn on page 4-95</i>
GICD_ICFGRn	<i>Interrupt Configuration Registers; GICD_ICFGRn on page 4-109</i>
GICD_ICPENDRn	<i>Interrupt Clear-Pending Registers; GICD_ICPENDRn on page 4-99</i>
GICD_IGROUPRn	<i>Interrupt Group Registers; GICD_IGROUPRn on page 4-91</i>
GICD_IIDR	<i>Distributor Implementer Identification Register; GICD_IIDR on page 4-90</i>
GICD_IPRIORITYRn	<i>Interrupt Priority Registers; GICD_IPRIORITYRn on page 4-104</i>
GICD_ISACTIVERn	<i>Interrupt Set-Active Registers; GICD_ISACTIVERn on page 4-102</i>
GICD_ISENABLERn	<i>Interrupt Set-Enable Registers; GICD_ISENABLERn on page 4-93</i>
GICD_ISPENDRn	<i>Interrupt Set-Pending Registers; GICD_ISPENDRn on page 4-97</i>
GICD_ITARGETSRn	<i>Interrupt Processor Targets Registers; GICD_ITARGETSRn on page 4-106</i>

- 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 pfHook; //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 = OsHwiCplrqParam(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->pstNext != NULL))
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_DMACE    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 [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 `HallrqHandler`

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 _osExceptUndefInsrHdl @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
    HallrqSendIpi(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 HallrqSendIpi(UINT32 target, UINT32 ipi)
{
    GicWriteSgi(ipi, target, 0);
}
```

中断统一处理入口函数 HallrqHandler

```
//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 HallrqHandler(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_curlrqNum = vector; //记录当前中断ID号
OsInterrupt(vector); //调用上层中断处理函数
/* use original 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
    OsCpuplrqStart(); //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
#ifdef 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
        }
    }
#ifdef 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
    OsCpuplrqEnd(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.

百篇博客.往期回顾

在加注过程中,整理出以下文章.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.确实有难度,自不量力,但已经出发,回头已是不可能的了。 :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o

- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

45_fork篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o

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.

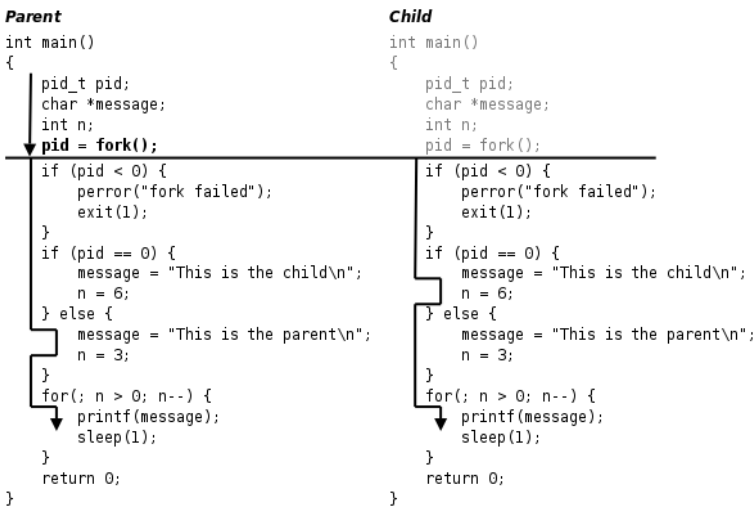
```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

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予进程的新任务栈中。

- 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(&proc
    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予任务
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.

百篇博客.往期回顾

在加注过程中, 整理出以下文章.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.确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o

- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 `oschina gitee`，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- `51cto`
- `csdn`
- `harmony`
- `oschina`

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。`51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

46_特殊进程篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

- 内核源码注解分析 →
- OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51 .c .h .o

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	Affinity	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	tcip_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	0x46a39	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 *///获取用户态进程的根进程，所有用户进程都是g_processCBArry[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_kernellInitProcess(2号进程).

```
g_kernellInitProcess = 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_kernellInitProcess] fork来的，包括g_processCBArray[g_kernellIdleProcess]进程
LITE_OS_SEC_TEXT UINT32 OsGetKernellInitProcessID(VOID)
{
    return g_kernellInitProcess;
}
```

- 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：绝对有可能!!! 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 = OsKernellInitProcess();// 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 OsKernellInitProcess(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_kernellInitProcess);// 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号
    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`
 - `tcip_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
#ifdef 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();
        }
    }
#ifdef WFI
    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_kernellInitProcess = 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_kernellInitProcess].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 = OsKernellInitProcess();// 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程
    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 -> userText
    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;
}

```

- 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.也就是说用户态老祖宗的创建是在 `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:



百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o

- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o

- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

47_进程回收篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o

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 the priority hash table */ //进程的:
    LOS_DL_LIST    waitList;          /**< The process holds the waitList 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 (!OsProcessExitCodeSignallsSet(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` 或 `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_WUNTRACE D	不支持	报告终止或停止的子进程的状态
LOS_WAIT_WCONTINU ED	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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o

- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o

- [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 | [51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> [精读鸿蒙源码](#)，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> [故事说内核](#)，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

48_信号生产篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百, 依然活力十足 | 51.c.h.o

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:

- 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

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, 没明白为啥要这么搞? 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,噁瑟.
- 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
|
|-----|
|               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;
    }
    #ifndef 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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o

- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很简单，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o

- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

49_信号消费篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

- 内核源码注解分析 →
- OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

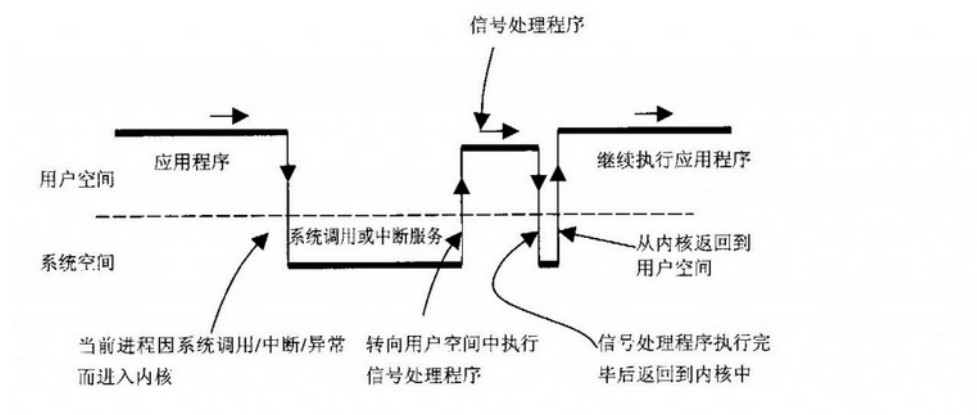
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51 .c .h .o

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:

- 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

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 \(工作模式篇\)](#) | [cpu是韦小宝, 有哪七个老婆?](#) 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)FindFirstSetBit(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.USR = 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.USR;//用户栈堆栈指针, USR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.USR;//用户栈堆栈指针, USR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. USR 和 ULR 指向的是用户栈的位置.一旦 PC , USR , 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
```

具体也可看这两篇:

[v42.xx \(中断切换篇\) | 系统因中断活力四射](#)

[v41.xx \(任务切换篇\) | 看汇编如何切换任务](#)

百篇博客.往期回顾

在加注过程中, 整理出以下文章。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.确实有难度, 自不量力, 但已经出发, 回头已是不可能的了。 :P

与代码有bug需不断debug一样, 文章和注解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, .xx 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百, 依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用, 两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51.c.h.o](#)
- [v40.xx 鸿蒙内核源码分析\(汇编汇总篇\) | 汇编可爱如邻家女孩 | 51.c.h.o](#)
- [v39.xx 鸿蒙内核源码分析\(异常接管篇\) | 社会很单纯, 复杂的是人 | 51.c.h.o](#)
- [v38.xx 鸿蒙内核源码分析\(寄存器篇\) | 小强乃宇宙最忙存储器 | 51.c.h.o](#)
- [v37.xx 鸿蒙内核源码分析\(系统调用篇\) | 开发者永远的口头禅 | 51.c.h.o](#)
- [v36.xx 鸿蒙内核源码分析\(工作模式篇\) | CPU是韦小宝, 七个老婆 | 51.c.h.o](#)
- [v35.xx 鸿蒙内核源码分析\(时间管理篇\) | 谁是内核基本时间单位 | 51.c.h.o](#)
- [v34.xx 鸿蒙内核源码分析\(原子操作篇\) | 谁在为原子操作保驾护航 | 51.c.h.o](#)
- [v33.xx 鸿蒙内核源码分析\(消息队列篇\) | 进程间如何异步传递大数据 | 51.c.h.o](#)
- [v32.xx 鸿蒙内核源码分析\(CPU篇\) | 整个内核就是一个死循环 | 51.c.h.o](#)
- [v31.xx 鸿蒙内核源码分析\(定时器篇\) | 哪个任务的优先级最高 | 51.c.h.o](#)
- [v30.xx 鸿蒙内核源码分析\(事件控制篇\) | 任务间多对多的同步方案 | 51.c.h.o](#)

- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件, 这就很有意思了, 冥冥之中似有天数, 将这四个宝贝以这种方式融合在一起. 51.c.h.o, 我要CHO, 嗯嗯, hin 顺口:)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

50_编译环境篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

几点说明

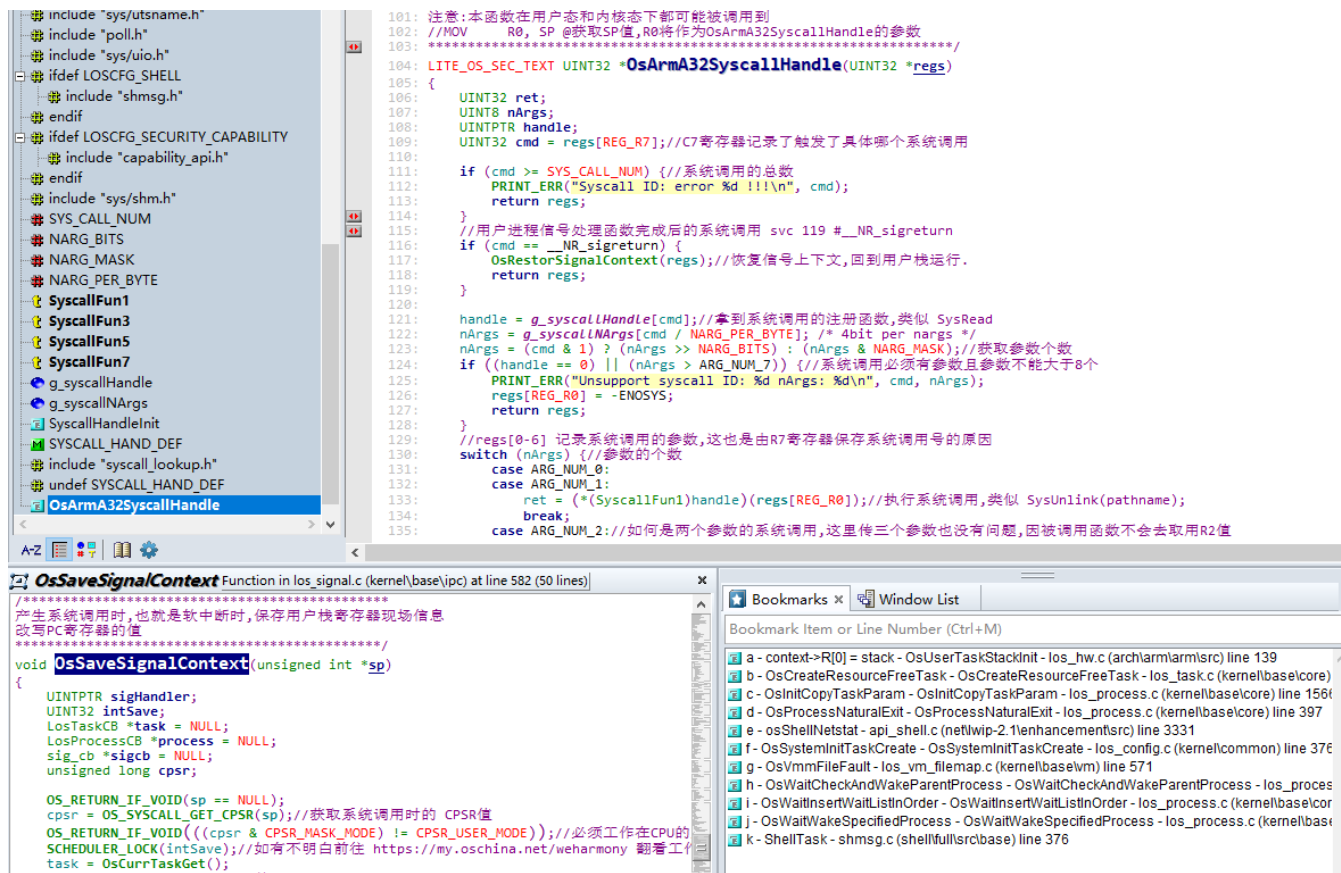
- 百万汉字注解仓库: `kernel_liteos_a_note` 是在 OpenHarmony 的 `kernel_liteos_a` (鸿蒙轻内核项目)基础上给源码加上中文注解的版本.加注版与官方最新源码保持同步.
- 百篇博客分析地址:
 - 国内: <https://weharmony.gitee.io/weharmony>
 - 国外: <https://weharmony.github.io/weharmony>
- OpenHarmony开发者文档 是对官方文档 docs 做的非常炫酷的静态站点,支持侧边栏/面包屑/搜索/中英文,非常方便的查看官方文档,大大提高学习和开发效率.
 - 国内: <https://weharmony.gitee.io/openharmony>
 - 国外: <https://weharmony.github.io/openharmony>
- 鸿蒙全部源码仓库: OpenHarmony 是HarmonyOS 110+个子项目的所有源码汇总. 因HarmonyOS使用 repo 管理众多 git 项目, repo 在 linux 下很方便,但在 windows 上使用会有相当的困难,所以将所有子项目整合成一个.git工程,如此windows用户使用熟悉的 git 方式便能下载整个鸿蒙系统源码,方便学习使用.仓库与官方仓库保持同步.已编译通过.

```
....
[OHOS INFO] [1587/1590] STAMP obj/test/xts/acts/build_lite/acts_generate_module_data.stamp
[OHOS INFO] [1588/1590] ACTION //test/xts/acts/build_lite:acts//build/lite/toolchain:linux_x86_64_ohos_clang
[OHOS INFO] [1589/1590] STAMP obj/test/xts/acts/build_lite/acts.stamp
[OHOS INFO] [1590/1590] STAMP obj/build/lite/ohos.stamp
[OHOS INFO] ipcamera_hispanic_aries build success
root@5e3abe332c5a:/home/harmony#
```

- [下载.鸿蒙源码分析.离线文档 < 国内 | 国外 >](#)
- [加入兴趣小组.微信群聊 < 国内 | 国外 >](#)

编译鸿蒙

因 `sourecinsight` 多年的使用习惯,爱不释手并为电脑性能计,所以选择以windows + docker方式编译鸿蒙.



本篇记录编译鸿蒙的过程，以备后续不用再去一大堆无效的误导性软文中搜寻芝麻大点有用的信息，那样真挺费时费心力。

编译环境

先安装 Docker Desktop 下载windows版本一直下一步.

在windows下拉取 openharmony-docker 官方镜像，Docker方式获取编译环境 强烈推荐这么做.

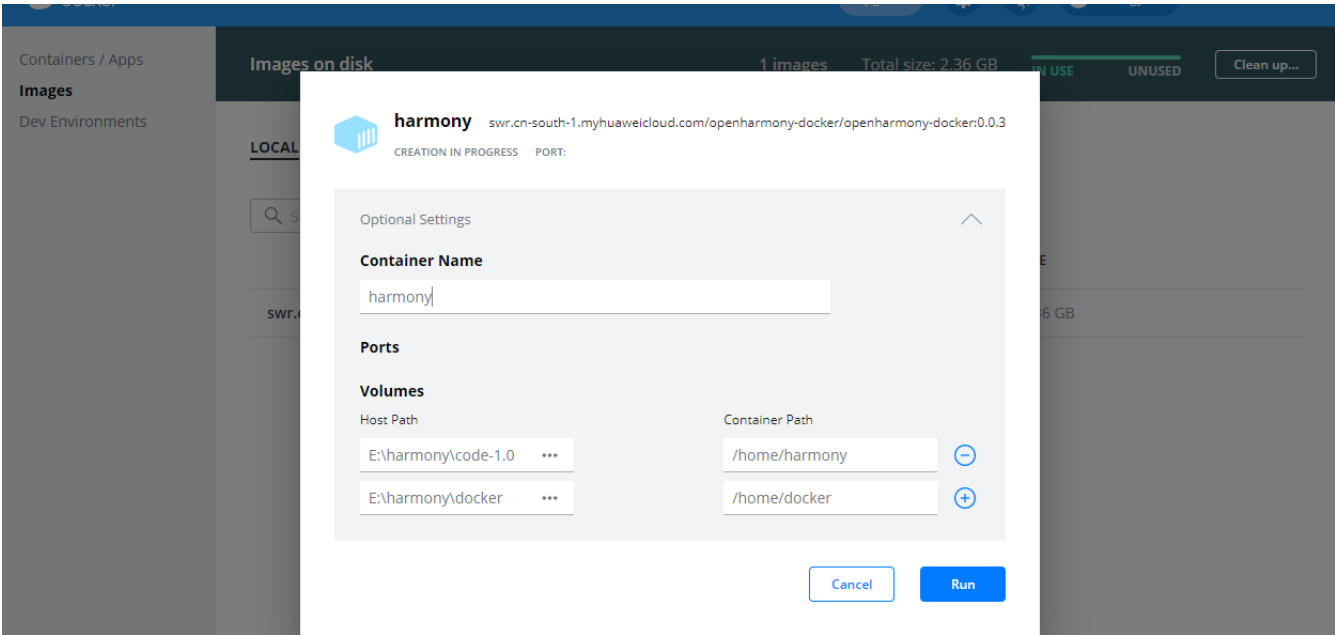
```
docker pull swr.cn-south-1.myhuaweicloud.com/openharmony-docker/openharmony-docker:0.0.3
```

2.36G，拉取看网速，大概10分钟后成功了，有了镜像

```
PS E:\harmony\kernel_liteos_a_note> docker images
REPOSITORY                                TAG      IMAGE ID      CREATED      SIZE
swr.cn-south-1.myhuaweicloud.com/openharmony-docker/openharmony-docker  0.0.3    50d0aa6ea9ba  2 weeks ago  2.36GB
```

vscode对docker的管理插件非常的强大，管理镜像和容器的工作就交给它了.

启动docker，创建好容器，本文的选择是这样的，当然大家可以灵活处理，命名. 本文使用 `E:\harmony\code-1.0` `E:\harmony\docker`，这两个文件夹一定要先创建好.



容器创建成功后可以在 vscode 右键容器 inspect 查看到绑定的目录.

```
"HostConfig": {
  "Binds": [
    "E:\\harmony\\code-1.0:/home/harmony",
    "E:\\harmony\\docker:/home/docker"
  ],
}
```

本文这样做的目的是为了在windows上能方便的查看文件. `harmony` 目录用于下载编译源码目录, 另外一个 `docker` 放其他无关文件 repo方式下载源码进入容器, vscode右键容器 `Attach shell`, 再进入 `/home/harmony` 目录

```
root@5e3abe332c5a:/home/harmony#repo init -u https://gitee.com/openharmony/manifest.git -b master --no-repo-verify
....
root@5e3abe332c5a:/home/harmony#repo sync -c
```

这个过程也是比网速.慢慢等吧. 下载完成之后的样子

```
Updating files: 100% (379/379), done.third_party_ltpUpdating files: 52% (200/379)
Updating files: 100% (1015/1015), done.ird_party_lwipUpdating files: 26% (273/1015)
Updating files: 100% (3118/3118), done.ird_party_mtd_utilsUpdating files: 8% (258/3118)
Updating files: 100% (3098/3098), done.hird_party_muslUpdating files: 8% (253/3098)
Updating files: 100% (198/198), done.third_party_opensslUpdating files: 4% (9/198)
Updating files: 100% (525/525), done.third_party_rt_threadUpdating files: 56% (299/525)
Updating files: 100% (904/904), done.third_party_unityUpdating files: 24% (221/904)
Updating files: 100% (261/261), done.third_party_wpa_supplicantUpdating files: 92% (241/261)
Checking out projects: 100% (112/112), done.
repo sync has finished successfully.
root@5e3abe332c5a:/home/harmony# ls
applications base build build.py developertools device docs domains drivers foundation kernel prebuilts test third_party utils vendor
```

编译前提

- Linux服务器, Ubuntu16.04及以上64位系统版本。
- Python 3.7.4及以上。
- OpenHarmony源码build_lite仓下载成功。

按照上面的方法, 能到这里说明这三个条件都已经满足了.下面安装编译工具

编译过程 | 安装 hb

hb 全称应该是 harmonyos build (猜的哈) 在源码的根目录执行:

```
root@5e3abe332c5a:/home/harmony#python3 -m pip install --user build/lite
```

执行hb -h有相关帮助信息，有打印信息即表示安装成功：

```
root@5e3abe332c5a:/home/harmony/# hb -v
[OHOS INFO] hb version 0.3.8
root@5e3abe332c5a:/home/harmony/# hb -h
usage: hb

OHOS build system

positional arguments:
  {build, set, env, clean, deps}
  build                Build source code
  set                  OHOS build settings
  env                  Show OHOS build env
  clean                Clean output
  deps                 OHOS components deps

optional arguments:
  -h, --help            show this help message and exit
  -v, --version          show program's version number and exit
```

另外 hb 的卸载方法如下:

```
python3 -m pip uninstall ohos-build
```

安装编译工具 hb 成功后，开始下一步设置编译路径。

编译命令 | hb set

先看看它的帮助信息

```
root@5e3abe332c5a:/home/harmony#hb set -h
usage: hb set [-h] [-root [ROOT_PATH]] [-p]

optional arguments:
  -h, --help            Show this help message and exit.
  -root [ROOT_PATH], --root_path [ROOT_PATH]
                        Set OHOS root path.
  -p, --product          Set OHOS board and kernel.
```

- hb set 后无参数，进入默认设置流程
- hb set -root [ROOT_PATH] 直接设置代码根目录
- hb set -p --product 设置要编译的产品

本篇选择当前路径。

```
root@5e3abe332c5a:/home/harmony#hb set
[OHOS INFO] Input code path: .
OHOS Which product do you need? (Use arrow keys)

hisilicon
> ipcamera_hispark_aries
wifiiot_hispark_pegasus
ipcamera_hispark_taurus
```

直接回车，本篇选择了ipcamera_hispark_aries，这三个对应平台的关系如下

```
Hi3518 : ipcamera_hispark_aries@hisilicon
Hi3861 : wifiiot_hispark_pegasus@hisilicon
```

```
Hi3516 : ipcamera_hispark_taurus@hisilicon
```

编译命令 | hb env

设置路径成功后，可查看下当前设置信息

```
root@5e3abe332c5a:/home/harmony# hb env
[OHOS INFO] root path: /home/harmony
[OHOS INFO] board: hispark_aries
[OHOS INFO] kernel: liteos_a
[OHOS INFO] product: ipcamera_hispark_aries
[OHOS INFO] product path: /home/harmony/vendor/hisilicon/hispark_aries
[OHOS INFO] device path: /home/harmony/device/hisilicon/hispark_aries/sdk_liteos
```

编译命令 | hb build

同样看下它的帮助信息，可以选择全编，编单个模块，只编译ndk，看具体的开发场景。

```
root@5e3abe332c5a:/home/harmony# hb build -h
usage: hb build [-h] [-b BUILD_TYPE] [-c COMPILER] [-t [TEST [TEST ...]]] [--dmverity] [--tee] [-p PRODUCT]
               [-f] [-n] [-T [TARGET [TARGET ...]]] [-v] [-shs]
               [component [component ...]]

positional arguments:
  component              name of the component

optional arguments:
  -h, --help            show this help message and exit
  -b BUILD_TYPE, --build_type BUILD_TYPE
                        release or debug version
  -c COMPILER, --compiler COMPILER
                        specify compiler
  -t [TEST [TEST ...]], --test [TEST [TEST ...]]
                        compile test suit
  --dmverity            Enable dmverity
  --tee                Enable tee
  -p PRODUCT, --product PRODUCT
                        build a specified product with {product_name}@{company}, eg: camera@huawei
  -f, --full            full code compilation
  -n, --ndk             compile ndk
  -T [TARGET [TARGET ...]], --target [TARGET [TARGET ...]]
                        Compile single target
  -v, --verbose         show all command lines while building
  -shs, --sign_haps_by_server
                        sign haps by server
```

解读

- hb build后无参数，会按照设置好的代码路径、产品进行编译，编译选项使用与之前保持一致。
- hb build component：基于设置好的产品对应的单板、内核，单独编译组件（e.g.：hb build kv_store）。
- hb build -p PRODUCT：免set编译产品，该命令可以跳过set步骤，直接编译产品。

本文选择全编 `hb build -f`，拼电脑性能，搞开发一定要搞台好电脑，时间就是生命，内存最好32G，过程大概10分钟。

```
root@5e3abe332c5a:/home/harmony# hb build -f
...
[OHOS INFO] [1587/1590] STAMP obj/test/xts/acts/build_lite/acts_generate_module_data.stamp
[OHOS INFO] [1588/1590] ACTION //test/xts/acts/build_lite:acts(//build/lite/toolchain:linux_x86_64_ohos_clang)
[OHOS INFO] [1589/1590] STAMP obj/test/xts/acts/build_lite/acts.stamp
[OHOS INFO] [1590/1590] STAMP obj/build/lite/ohos.stamp
[OHOS INFO] ipcamera_hispark_aries build success
root@5e3abe332c5a:/home/harmony#
```

会多出一个 out 目录，每个的目录含义如下

目录名	描述
applications	应用程序样例，包括wifi-iot，camera等
base	基础软件服务子系统集&硬件服务子系统集
build	组件化编译、构建和配置脚本
docs	说明文档
domains	增强软件服务子系统集
drivers	驱动子系统
foundation	系统基础能力子系统集
kernel	内核子系统
prebuilts	编译器及工具链子系统
test	测试子系统
third_party	开源第三方组件
utils	常用的工具集
vendor	厂商提供的软件
build.py	编译脚本文件
out	编译后生成

编译命令 | hb clean

清除out目录对应产品的编译产物，仅剩下args.gn、build.log。清除指定路径可输入路径参数：hb clean xxx/out/xxx，否则将清除hb set的产品对应out路径.此处不要去敲它，它的帮助提示信息如下，

```
root@5e3abe332c5a:/home/harmony# hb clean -h
usage: hb clean [-h] [out_path]

positional arguments:
  out_path  clean a specified path.

optional arguments:
  -h, --help  show this help message and exit
```

编译输出 | out 目录

out 为编译结果输出目录

输出目录：out/hispark_aries/ipcamera_hispark_aries

```
root@5e3abe332c5a:/home/harmony# cd out/hispark_aries/ipcamera_hispark_aries
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries#
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries# ls
args.gn  build.log  bundle_daemon_tool.map  dev_tools  libs  NOTICE_FILE  OHOS_Image.asm  rootfs  suites  toggleButton
bin      build.ninja  config  etc  liteos.bin  obj  OHOS_Image.bin  rootfs_jffs2.img  test  toolchain.ninja  userfs_jff
bm_tool.map  build.ninja.d  data  foundation.map  media_server.map  OHOS_Image  OHOS_Image.map  server.map  test_info  uns
```

系列篇会详细讲解启动过程，此处进入bin目录瞅瞅都有些啥好宝贝.

```
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries/bin# ls
ai_server      module_ActsAbilityMgrTest.bin  module_ActsGraphVersionTest.bin  module_ActsJFFS2CapabilityTest.bin  module_ActsNFSTest.bin
apphilogcat    module_ActsBootstrapTest.bin  module_ActsHeapBaseTest.bin      module_ActsJFFS2DACTest.bin        module_ActsParameterTes
appspawn       module_ActsBundleMgrTest.bin  module_ActsHilogTest.bin         module_ActsJFFS2Test.bin           module_ActsPMSTest.bin
bundle_daemon  module_ActsColorTest.bin      module_ActsIoApiTest.bin         module_ActsKvStoreTest.bin         module_ActsProcessApiTest.bi
CalcSubTest.bin module_ActsDyloadTest.bin      module_ActsIpcMqTest.bin         module_ActsListTest.bin            module_ActsRectTest.bin
foundation     module_ActsFutexApiTest.bin    module_ActsIpcPipeTest.bin       module_ActsLwipTest.bin            module_ActsSamgrTest.bin
hilogcat       module_ActsGeometyr2dTest.bin  module_ActsIpcSemTest.bin        module_ActsMathApiTest.bin         module_ActsSchedApiTest.bi
init           module_ActsGraphicMathTest.bin  module_ActsIpcShmTest.bin        module_ActsMemApiTest.bin          module_ActsSecurityDataTest.l
media_server   module_ActsGraphMemApiTest.bin  module_ActsIpcSignalTest.bin     module_ActsNetTest.bin             module_ActsSoftBusTest.bi
```

这难道都是ELF格式可执行程序? 用 readelf 命令试下 shell 就知道了.果然是shell程序，加载它将创建激动人心的 shell 进程

```
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries/bin# readelf -h shell
ELF Header:
  Magic:   7f 45 4c 46 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

```

再 `readelf -l shell` 查看程序头表，也称段头表，内核是通过此表来构建进程映像的，具体构建过程在

- [v56.xx 鸿蒙内核源码分析\(进程镜像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#) 中有详细介绍，请前往翻看。

```
root@5e3abe332c5a:/home/harmony/out/hispanic_aries/ipcamera_hispanic_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...
```

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

再随便选择一个 `module_ActsListTest.bin` 看下，这些是鸿蒙用于测试的代码生成的.也是一个个的独立程序.可以在工程里找到他们的身影 `list_test.cpp` , `ActsListTest.json` 等文件

```
root@5e3abe332c5a:/home/harmony/out/hispanic_aries/ipcamera_hispanic_aries/bin# readelf -h module_ActsListTest.bin
```

```
ELF Header:
```

```

Magic: 7f 45 4c 46 01 01 01 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:  0xb000
Start of program headers: 52 (bytes into file)
Start of section headers: 172256 (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

```

解读

仔细比较下这两个ELF的头文件看哪里不一样

```

Entry point address: 0x1000
Entry point address: 0xb000

```

这是 .text 代码段的入口地址，但注意这并不是 main 函数的地址，真正的入口地址是 _start，由它再调用 main。具体看

- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o

有详细说明，readelf 的功能很强大，有兴趣的可以玩下这个命令，看看elf里面究竟装的啥。

```

root@5e3abe332c5a:/home/harmony/out/hisparc_aries/ipcamera_hisparc_aries/bin# readelf -help
readelf: option requires an argument -- 'p'
Usage: readelf <option(s)> elf-file(s)
Display information about the contents of ELF format files
Options are:
-a --all             Equivalent to: -h -l -S -r -d -V -A -l
-h --file-header     Display the ELF file header
-l --program-headers Display the program headers
--segments          An alias for --program-headers
-S --section-headers Display the sections' header
--sections          An alias for --section-headers
-g --section-groups  Display the section groups
-t --section-details Display the section details
-e --headers         Equivalent to: -h -l -S
-s --syms            Display the symbol table
--symbols           An alias for --syms
--dyn-syms          Display the dynamic symbol table
-n --notes           Display the core notes (if present)
-r --relocs          Display the relocations (if present)
-u --unwind          Display the unwind info (if present)
-d --dynamic          Display the dynamic section (if present)
-V --version-info    Display the version sections (if present)
-A --arch-specific   Display architecture specific information (if any)
-c --archive-index   Display the symbol/file index in an archive
-D --use-dynamic      Use the dynamic section info when displaying symbols
-x --hex-dump=<number|name>
                    Dump the contents of section <number|name> as bytes
-p --string-dump=<number|name>
                    Dump the contents of section <number|name> as strings
-R --relocated-dump=<number|name>
                    Dump the contents of section <number|name> as relocated bytes
-z --decompress      Decompress section before dumping it
-w[LIaprmfFsoRtUuTgAckK] or
--debug-dump[=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 the contents of DWARF debug sections
--dwarf-depth=N      Do not display DIEs at depth N or greater
--dwarf-start=N      Display DIEs starting with N, at the same depth
                    or deeper

```



```
--ctf=<number|name>    Display CTF info from section <number|name>
--ctf-parent=<number|name>
                        Use section <number|name> as the CTF parent

--ctf-symbols=<number|name>
                        Use section <number|name> as the CTF external symtab

--ctf-strings=<number|name>
                        Use section <number|name> as the CTF external strtob

-l --histogram          Display histogram of bucket list lengths
-W --wide               Allow output width to exceed 80 characters
@<file>                Read options from <file>
-H --help               Display this information
-v --version            Display the version number of readelf
```

参数中文说明

```
-a --all 显示全部信息，等价于 -h -l -S -s -r -d -V -A -l.
-h --file-header 显示elf文件开始的文件头信息。
-l --program-headers --segments 显示程序头（段头）信息(如果有的话)。
-S --section-headers --sections 显示区头信息(如果有的话)。
-g --section-groups 显示区组信息(如果有的话)。
-t --section-details 显示区的详细信息(-S 的)。
-s --syms --symbols 显示符号表段中的项（如果有的话）。
-e --headers 显示全部头信息，等价于: -h -l -S
-n --notes 显示note段（内核注释）的信息。
-r --relocs 显示可重定位段的信息。
-u --unwind 显示unwind段信息。当前只支持 IA64ELF 的 unwind 段信息。
-d --dynamic 显示动态段的信息。
-V --version-info 显示版本段的信息。
-A --arch-specific 显示CPU构架信息。
-D --use-dynamic 使用动态段中的符号表显示符号，而不是使用符号段。
-x --hex-dump= 以16进制方式显示指定段内容。number 指定段表中段的索引，或字符串指定文件中的段名。
-w[liaprmfFsoR] --debug-dump[=line, =info, =abbrev, =pubnames, =aranges, =macro, =frames, =frames-interp, =str, =loc, =Ranges]
-l --histogram 显示符号的时候，显示 bucketlist 长度的柱状图。
-v --version 显示 readelf 的版本信息。
-H --help 显示 readelf 所支持的命令行选项。
-W --wide 宽行输出。
@file 可以将选项集中到一个文件中，然后使用这个 @file 选项载入。
```

具体的加载过程和elf格式后续有专门的篇幅详细介绍，此处不做说明。

用户进程

用 grep 过滤下干扰的*.bin，剩下的就是平台(Hi3518)需要内核创建的用户态进程。

```
root@5e3abe332c5a:/home/harmony/out/hispark_aries/ipcamera_hispark_aries/bin# ls | grep -v .bin
ai_server
apphilogcat
appspawn
bundle_daemon
foundation
hilogcat
init
media_server
os_dump
shell
tftp
wms_server
```

这些服务含义和仓库如下。

ai_server	AI业务子系统	https://gitee.com/openharmony/ai_engine
apphilogcat	小型系统的流水日志功能	https://gitee.com/openharmony/hiviewdfx_hilog_lite
appspawn	应用孵化模块进程	https://gitee.com/openharmony/startup_appspawn_lite

bundle_daemon	用户程序框架内部工具接口	https://gitee.com/openharmony/appexecfwk_appexecfwk_lite
foundation	foundation系统进程	https://gitee.com/openharmony/distributedschedule_safwk_lite
hilogcat	管理日志打印	https://gitee.com/openharmony/hiviewdfx_hilog_lite
		base\hiviewdfx\hilog_lite\services\hilogcat\hiview_logcat.c
init	用户态祖宗进程	1号进程, -> 鸿蒙内核源码分析(特殊进程篇)
media_server	播放模块框架实现	https://gitee.com/openharmony/multimedia_media_lite
os_dump	备份文件系统	utils\native\lite\os_dump\os_dump.c
shell	窥视内核的窗口	3号进程, -> 鸿蒙内核源码分析(shell篇)
tftp	传输文件	third_party\curl\lib\tftp.c
wms_server	窗口管理服务	https://gitee.com/openharmony/graphic_wms

在整个项目工程中能轻易找到他们的入口函数.比如 appspawn 的启动过程

```
//base\startup\appspawn_lite\services\src\main.c
int main(int argc, char const argv[])
{
    sleep(1);
    HILOG_INFO(HILOG_MODULE_HIVIEW, "[appspawn] main, enter.");

    // 1. ipc module init
    HOS_SystemInit();

    // 2. register signal for SIGCHLD
    SignalRegist();

    // 3. keep process alive
    HILOG_INFO(HILOG_MODULE_HIVIEW, "[appspawn] main, entering wait.");
    while (1) {
        // pause only returns when a signal was caught and the signal-catching function returned.
        // pause only returns -1, no need to process the return value.
        (void)pause();
    }
}
```

鸿蒙将服务做成了组件,为最上层的应用程度提供管理/工具类的服务.但这些都是framework层的工作,超出了内核源码分析的范畴.希望后续有机会能去剖析它们.

百篇博客.往期回顾

在加注过程中,整理出以下文章.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.确实有难度,自不量力,但已经出发,回头已是不可能的了. :P

与代码有bug需不断debug一样,文章和注解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, .xx 代表修改的次数,精雕细琢,言简意赅,力求打造精品内容.

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o

- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o

- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

51_ELF格式篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o

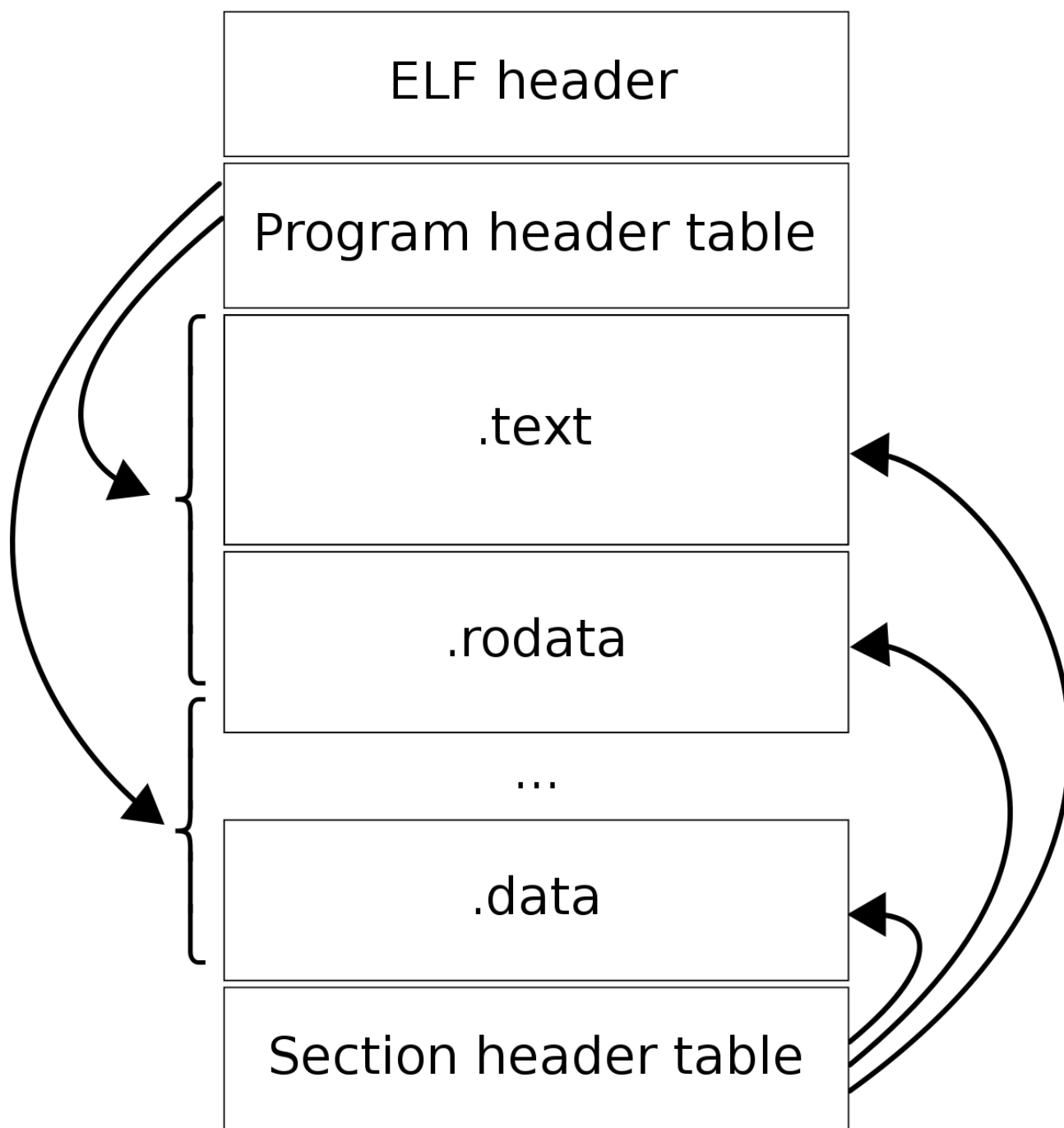
相关篇为:

- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

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.
- - ELF头定义全局性信息
- - Segment(段)头,内容描述段的名称,开始位置,类型,偏移,大小及每段由哪些区组成.
- - 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.
- - Section(区)头,内容描述区的名称,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\download\include\os_id_elf_pri.h` 文件中



示例代码

在windows目录 E:\harmony\docker\test4harmony 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 \(编译环境篇\)](#) | [编译鸿蒙看这篇或许真的够了](#)

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/test4harmony# ls
main.c
root@5e3abe332c5a:/home/docker/test4harmony# gcc -o app main.c
root@5e3abe332c5a:/home/docker/test4harmony# ls
app main.c
root@5e3abe332c5a:/home/docker/test4harmony# ./app
hello , harmony os!
```

名正言顺

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):

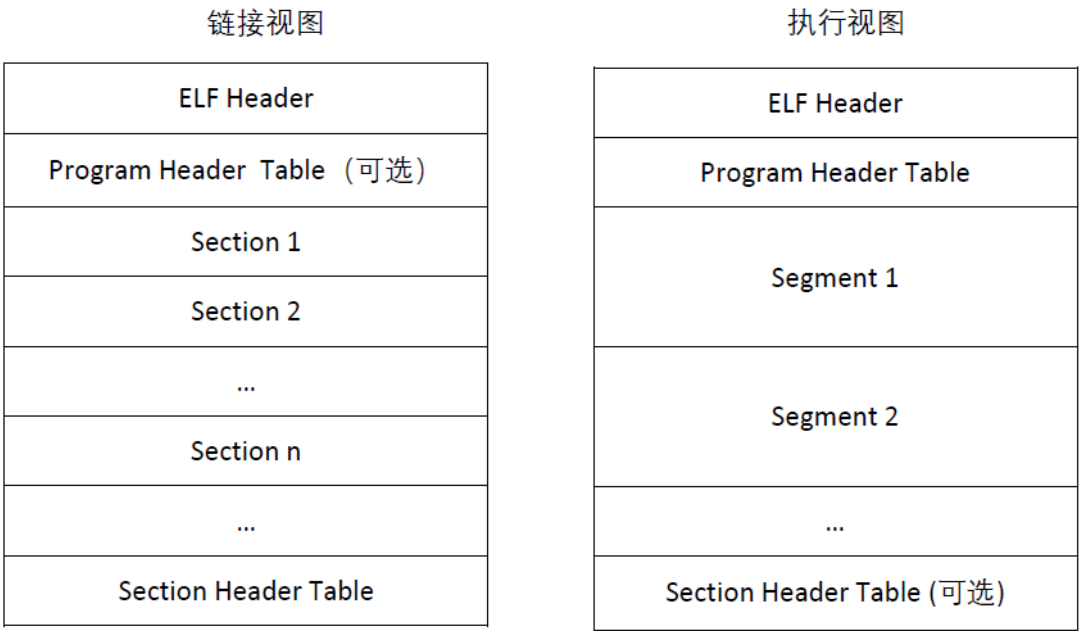
- 可重定位文件(relocatable file), 用于与其它目标文件进行连接以构建可执行文件或动态链接库.可重定位文件就是常说的目标文件, 由源文件编译而成, 但还没有连接成可执行文件.在UNIX系统下, 一般有扩展名".o".之所以称其为"可重定位", 是因为在这些文件中, 如果引用到其它目标

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/test4harmony# readelf -h app
```

ELF Header:

```
  Magic:  7f 45 4c 46 02 01 01 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/test4harmony/51# od -Ax -t x1 -N 64 app
000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
000010 03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00
000020 40 00 00 00 00 00 00 00 c0 39 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，应用程序
    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/test4harmony# 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 0x0000000000000318
               0x000000000000001c 0x000000000000001c R    0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000618 0x0000000000000618 R    0x1000
LOAD           0x0000000000000100 0x0000000000000100 0x0000000000000100 0x0000000000000100
               0x0000000000000225 0x0000000000000225 R E  0x1000
LOAD           0x0000000000000200 0x0000000000000200 0x0000000000000200 0x0000000000000200
               0x0000000000000190 0x0000000000000190 R    0x1000
LOAD           0x00000000000002db8 0x00000000000003db8 0x00000000000003db8
               0x0000000000000260 0x0000000000000268 RW  0x1000
DYNAMIC        0x00000000000002dc8 0x00000000000003dc8 0x00000000000003dc8
               0x00000000000001f0 0x00000000000001f0 RW  0x8
NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
               0x0000000000000020 0x0000000000000020 R    0x8
NOTE           0x0000000000000358 0x0000000000000358 0x0000000000000358 0x0000000000000358
               0x0000000000000044 0x0000000000000044 R    0x4
GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338 0x0000000000000338
               0x0000000000000020 0x0000000000000020 R    0x8
GNU_EH_FRAME   0x0000000000000201c 0x0000000000000201c 0x0000000000000201c
               0x000000000000004c 0x000000000000004c R    0x4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000 RW  0x10
GNU_RELRO      0x00000000000002db8 0x00000000000003db8 0x00000000000003db8
               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(initialized)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(uninitialized)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_id_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的

```

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/test4harmony# 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.propt	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
	000000000000001b5	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
	00000000000000128	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
	000000000000001f0	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
	00000000000000648	0000000000000018		29 46 8
[29]	.strtab	STRTAB	0000000000000000	00003690
	0000000000000216	0000000000000000		0 0 1
[30]	.shstrtab	STRTAB	0000000000000000	000038a6

```
0000000000000011a 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/test4harmony# 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 (DYNYSYM)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_id_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/test4harmony# 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:	00000000000001000	0	SECTION	LOCAL	DEFAULT	12	
13:	00000000000001020	0	SECTION	LOCAL	DEFAULT	13	
14:	00000000000001040	0	SECTION	LOCAL	DEFAULT	14	
15:	00000000000001050	0	SECTION	LOCAL	DEFAULT	15	
16:	00000000000001060	0	SECTION	LOCAL	DEFAULT	16	
17:	00000000000001218	0	SECTION	LOCAL	DEFAULT	17	
18:	00000000000002000	0	SECTION	LOCAL	DEFAULT	18	
19:	0000000000000201c	0	SECTION	LOCAL	DEFAULT	19	
20:	00000000000002068	0	SECTION	LOCAL	DEFAULT	20	
21:	00000000000003db8	0	SECTION	LOCAL	DEFAULT	21	
22:	00000000000003dc0	0	SECTION	LOCAL	DEFAULT	22	
23:	00000000000003dc8	0	SECTION	LOCAL	DEFAULT	23	
24:	00000000000003fb8	0	SECTION	LOCAL	DEFAULT	24	
25:	00000000000004000	0	SECTION	LOCAL	DEFAULT	25	
26:	00000000000004018	0	SECTION	LOCAL	DEFAULT	26	
27:	00000000000000000	0	SECTION	LOCAL	DEFAULT	27	
28:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
29:	00000000000001090	0	FUNC	LOCAL	DEFAULT	16	deregister_tm_clones
30:	000000000000010c0	0	FUNC	LOCAL	DEFAULT	16	register_tm_clones
31:	00000000000001100	0	FUNC	LOCAL	DEFAULT	16	__do_global_dtors_aux
32:	00000000000004018	1	OBJECT	LOCAL	DEFAULT	26	completed.8060
33:	00000000000003dc0	0	OBJECT	LOCAL	DEFAULT	22	__do_global_dtors_aux_fin
34:	00000000000001140	0	FUNC	LOCAL	DEFAULT	16	frame_dummy
35:	00000000000003db8	0	OBJECT	LOCAL	DEFAULT	21	__frame_dummy_init_array_
36:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
37:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
38:	0000000000000218c	0	OBJECT	LOCAL	DEFAULT	20	__FRAME_END__
39:	00000000000000000	0	FILE	LOCAL	DEFAULT	ABS	
40:	00000000000003dc0	0	NOTYPE	LOCAL	DEFAULT	21	__init_array_end
41:	00000000000003dc8	0	OBJECT	LOCAL	DEFAULT	23	_DYNAMIC
42:	00000000000003db8	0	NOTYPE	LOCAL	DEFAULT	21	__init_array_start
43:	0000000000000201c	0	NOTYPE	LOCAL	DEFAULT	19	__GNU_EH_FRAME_HDR
44:	00000000000003fb8	0	OBJECT	LOCAL	DEFAULT	24	_GLOBAL_OFFSET_TABLE_
45:	00000000000001000	0	FUNC	LOCAL	DEFAULT	12	_init
46:	00000000000001210	5	FUNC	GLOBAL	DEFAULT	16	__libc_csu_fini
47:	00000000000004010	8	OBJECT	GLOBAL	DEFAULT	25	my_name
48:	00000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_ITM_deregisterTMCloneTab
49:	00000000000004000	0	NOTYPE	WEAK	DEFAULT	25	data_start
50:	00000000000004018	0	NOTYPE	GLOBAL	DEFAULT	25	edata
51:	00000000000001218	0	FUNC	GLOBAL	HIDDEN	17	_fini
52:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	printf@@GLIBC_2.2.5
53:	00000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@@GLIBC_
54:	00000000000004000	0	NOTYPE	GLOBAL	DEFAULT	25	__data_start
55:	00000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
56:	00000000000004008	0	OBJECT	GLOBAL	HIDDEN	25	_dso_handle
57:	00000000000002000	4	OBJECT	GLOBAL	DEFAULT	18	_IO_stdin_used
58:	000000000000011a0	101	FUNC	GLOBAL	DEFAULT	16	__libc_csu_init
59:	00000000000004020	0	NOTYPE	GLOBAL	DEFAULT	26	_end
60:	00000000000001060	47	FUNC	GLOBAL	DEFAULT	16	_start
61:	00000000000004018	0	NOTYPE	GLOBAL	DEFAULT	26	__bss_start


```
62: 00000000000001174 30 FUNC GLOBAL DEFAULT 16 main
63: 00000000000001149 43 FUNC GLOBAL DEFAULT 16 say_hello
64: 00000000000004018 0 OBJECT GLOBAL HIDDEN 25 __TMC_END__
65: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
66: 00000000000000000 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: 00000000000001174 30 FUNC GLOBAL DEFAULT 16 main
63: 00000000000001149 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/test4harmony# objdump -j .text -l -C -S app

00000000000001149 <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

00000000000001174 <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/test4harmony# ./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
      0000000000001b5 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  $0xfffffffffff0, %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.

百篇博客.往期回顾

在加注过程中, 整理出以下文章.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.确实有难度, 自不量力, 但已经出发, 回头已是不可能的了. :P

与代码有bug需不断debug一样, 文章和注解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, .xx 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容.

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o

- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o

- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

52_静态站点篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o

几点说明

- 百万汉字注解仓库: `kernel_liteos_a_note` 是在 OpenHarmony 的 `kernel_liteos_a` (鸿蒙轻内核项目)基础上给源码加上中文注解的版本.加注版与官方最新源码保持同步.
- 百篇博客分析地址:
 - 国内: <https://weharmony.gitee.io/weharmony>
 - 国外: <https://weharmony.github.io/weharmony>
- OpenHarmony开发者文档 是对官方文档 docs 做的非常炫酷的静态站点,支持侧边栏/面包屑/搜索/中英文,非常方便的查看官方文档,大大提高学习和开发效率.
 - 国内: <https://weharmony.gitee.io/openharmony>
 - 国外: <https://weharmony.github.io/openharmony>
- 鸿蒙全部源码仓库: OpenHarmony 是HarmonyOS 110+个子项目的所有源码汇总. 因HarmonyOS使用 repo 管理众多 git 项目, repo 在 linux 下很方便,但在 windows 上使用会有相当的困难,所以将所有子项目整合成一个.git工程,如此windows用户使用熟悉的 git 方式便能下载整个鸿蒙系统源码,方便学习使用.仓库与官方仓库保持同步.已编译通过.

```
....
[OHOS INFO] [1587/1590] STAMP obj/test/xts/acts/build_lite/acts_generate_module_data.stamp
[OHOS INFO] [1588/1590] ACTION //test/xts/acts/build_lite:acts//build/lite/toolchain:linux_x86_64_ohos_clang
[OHOS INFO] [1589/1590] STAMP obj/test/xts/acts/build_lite/acts.stamp
[OHOS INFO] [1590/1590] STAMP obj/build/lite/ohos.stamp
[OHOS INFO] ipcamera_hispanic_aries build success
root@5e3abe332c5a:/home/harmony#
```

- 下载.鸿蒙源码分析.离线文档 < 国内 | 国外 >
- 加入兴趣小组.微信群聊 < 国内 | 国外 >

OpenHarmony开发者文档

- OpenHarmony开发者文档是对官方文档 docs 做的非常炫酷的静态站点,支持侧边栏/面包屑/搜索/中英文,非常方便的查看官方文档,大大提高学习和开发效率.
 - 国内: <https://weharmony.gitee.io/openharmony>
 - 国外: <https://weharmony.github.io/openharmony>

先看图:

OpenHarmony开发者文档

同步官方文档 静态站点展示 轻松学习 高效开发

[中文入口 →](#)[English Enter →](#)[官方仓库 →](#)[quick-start | 快速入门](#)[get-code | 获取源码/获取工具](#)[docker | Docker镜像构建](#)[kernel | 内核](#)[driver | 驱动](#)[subsystems | 子系统](#)[bundles | 组件开发](#)[porting | 三方库移植/三方芯片移植](#)[guide | 设备开发](#)[security | 安全](#)[contribute | 贡献](#)[glossary | 术语](#)[Release Notes | 版本更新](#)[3rd-Party-License | 第三方开源软件及许可证说明](#)

- 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整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，耽误了点时间没更新博客但换来未来的一片爽朗之声，hin 值得!
- 这里必须要感谢下这套主题的作者 Mr.hope，人非常的nice，晚上12点我们还在一起解决问题.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.
- 静态站点将每月同步官方文档，静态站点仓库已经开放，[仓库地址](#) 欢迎下载部署.

侧边栏



- 这个不用说，都恨不得多开几个屏幕，技术人没它真不行，谁用谁知道，侧边栏是按官方的目录结构来的，但目前不是最优方案，会随时调整到最佳结构。

主题色



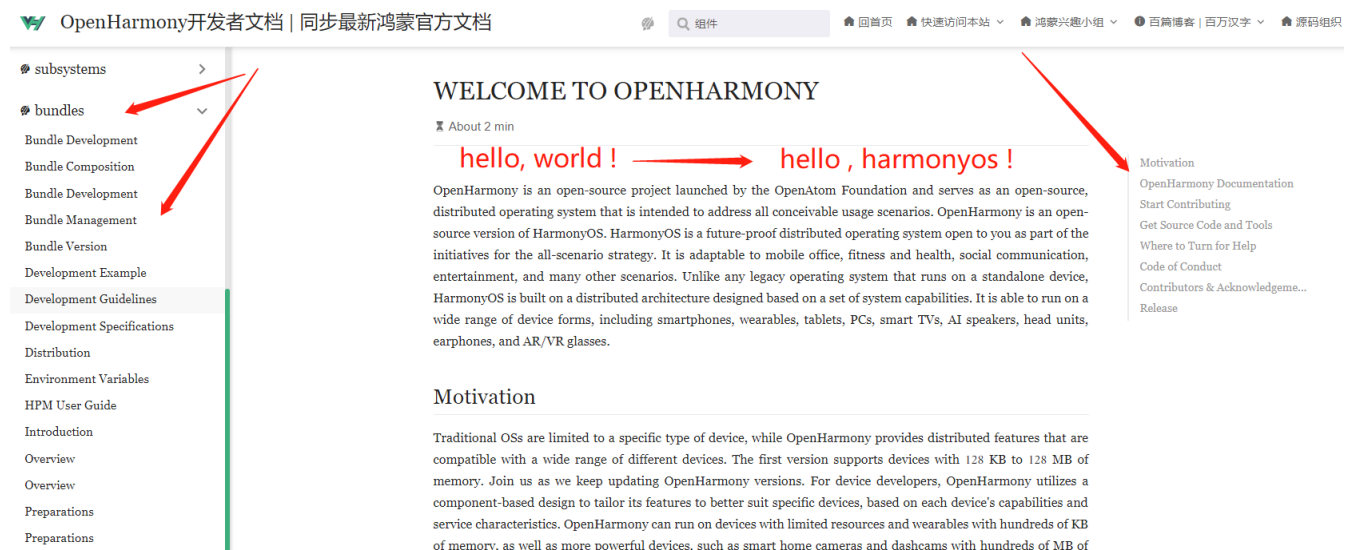
- 多种主题色，可以根据您的选择，当前时间自动切换模式，码农们得保护好视力，个人偏好白色。

搜索极为便利，国内外双站点



- 这个是很重要的功能,无搜索不技术,说到这个强烈建议不要用某度,因之前从未写过博客不清楚搜索结果是怎样的效果,现在有对比,去搜下自己的文章 ..., 不知道大家有没有这种感觉,搜了半天看到很多加工材料,最后发现了根节点,根节点不是那么容易找到的. 各搜索引擎差别真的很大,谷歌就不用去说了,有条件的推荐要用,搜狗, 360都比它好, 它虽然全但更杂,会干扰你的注意力. 哎, 以后有机会要写篇文章痛批阿度,手握流量不作为,虚假信息满屏飞.真是太耽误事了,心疼流失了这么多人宝贵的时间.

中英文切换



- 这个估计国内的各位都不会去看,希望能有更多友人了解鸿蒙.搞这行都有操作系统情怀,个人坚信鸿蒙必定成功,也必然成功,它的成功意义非凡,或许能明白的人不多,自己一定是那个摇旗呐喊者.

带私货



这是私货，是自己一直想做也正在做的事，有些事情一旦开始，就真的停不下来了，不知道你有没有这种感受。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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。确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o

- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 oschina gitee，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < gitee | github | csdn | coding >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < 51cto | csdn | harmony | osc >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

53_ELF解析篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o

系列篇将用四篇介绍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.其他相关篇幅为

- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

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/test4harmony# 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/test4harmony# 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-id 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] .rel.dyn RELA              0000000000000528 00000528
00000000000000d8 0000000000000018 A 6 0 8
[11] .rel.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
00000000000000128 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
000000000000001f0 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
00000000000000648 0000000000000018 29 46 8
[29] .strtab STRTAB             0000000000000000 00003690
00000000000000216 0000000000000000 0 0 1
[30] .shstrtab STRTAB            0000000000000000 000038a6
0000000000000011a 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/.data1	这两个区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,这两个区的名字中,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DYNSTR
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DYNSTR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/test4harmony# 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/test4harmony# 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: 00000000000001060 0 SECTION LOCAL DEFAULT 16
17: 00000000000001218 0 SECTION LOCAL DEFAULT 17
18: 00000000000002000 0 SECTION LOCAL DEFAULT 18
19: 0000000000000201c 0 SECTION LOCAL DEFAULT 19
20: 00000000000002068 0 SECTION LOCAL DEFAULT 20
21: 00000000000003db8 0 SECTION LOCAL DEFAULT 21
22: 00000000000003dc0 0 SECTION LOCAL DEFAULT 22
23: 00000000000003dc8 0 SECTION LOCAL DEFAULT 23
24: 00000000000003fb8 0 SECTION LOCAL DEFAULT 24
25: 00000000000004000 0 SECTION LOCAL DEFAULT 25
26: 00000000000004018 0 SECTION LOCAL DEFAULT 26
27: 00000000000000000 0 SECTION LOCAL DEFAULT 27
28: 00000000000000000 0 FILE LOCAL DEFAULT ABS crtstuff.c
29: 00000000000001090 0 FUNC LOCAL DEFAULT 16 deregister_tm_clones
30: 000000000000010c0 0 FUNC LOCAL DEFAULT 16 register_tm_clones
31: 00000000000001100 0 FUNC LOCAL DEFAULT 16 __do_global_ctors_aux
32: 00000000000004018 1 OBJECT LOCAL DEFAULT 26 completed.8060
33: 00000000000003dc0 0 OBJECT LOCAL DEFAULT 22 __do_global_ctors_aux_fin
34: 00000000000001140 0 FUNC LOCAL DEFAULT 16 frame_dummy
35: 00000000000003db8 0 OBJECT LOCAL DEFAULT 21 __frame_dummy_init_array_
36: 00000000000000000 0 FILE LOCAL DEFAULT ABS main.c
37: 00000000000000000 0 FILE LOCAL DEFAULT ABS crtstuff.c
38: 0000000000000218c 0 OBJECT LOCAL DEFAULT 20 __FRAME_END__
39: 00000000000000000 0 FILE LOCAL DEFAULT ABS
40: 00000000000003dc0 0 NOTYPE LOCAL DEFAULT 21 __init_array_end
41: 00000000000003dc8 0 OBJECT LOCAL DEFAULT 23 _DYNAMIC
42: 00000000000003db8 0 NOTYPE LOCAL DEFAULT 21 __init_array_start
43: 0000000000000201c 0 NOTYPE LOCAL DEFAULT 19 __GNU_EH_FRAME_HDR
44: 00000000000003fb8 0 OBJECT LOCAL DEFAULT 24 _GLOBAL_OFFSET_TABLE_
45: 00000000000001000 0 FUNC LOCAL DEFAULT 12 _init
46: 00000000000001210 5 FUNC GLOBAL DEFAULT 16 __libc_csu_fini
47: 00000000000004010 8 OBJECT GLOBAL DEFAULT 25 my_name
48: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _ITM_deregisterTMCloneTab
49: 00000000000004000 0 NOTYPE WEAK DEFAULT 25 data_start
50: 00000000000004018 0 NOTYPE GLOBAL DEFAULT 25 _edata
51: 00000000000001218 0 FUNC GLOBAL HIDDEN 17 _fini
52: 00000000000000000 0 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.2.5
53: 00000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
54: 00000000000004000 0 NOTYPE GLOBAL DEFAULT 25 __data_start
55: 00000000000000000 0 NOTYPE WEAK DEFAULT UND __gmon_start__
56: 00000000000004008 0 OBJECT GLOBAL HIDDEN 25 __dso_handle
57: 00000000000002000 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used
58: 000000000000011a0 101 FUNC GLOBAL DEFAULT 16 __libc_csu_init
59: 00000000000004020 0 NOTYPE GLOBAL DEFAULT 26 _end
60: 00000000000001060 47 FUNC GLOBAL DEFAULT 16 _start
61: 00000000000004018 0 NOTYPE GLOBAL DEFAULT 26 __bss_start
62: 00000000000001174 30 FUNC GLOBAL DEFAULT 16 main
63: 00000000000001149 43 FUNC GLOBAL DEFAULT 16 say_hello
64: 00000000000004018 0 OBJECT GLOBAL HIDDEN 25 __TMC_END__
65: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
66: 00000000000000000 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DYNYSYM
SHT_DYNYSYM 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DYNYSYM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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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. 确实有难度, 自不量力, 但已经出发, 回头已是不可能的了。 :P

与代码有bug需不断debug一样, 文章和注解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, `.xx` 代表修改的次数, 精雕细琢, 言简意赅, 力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百, 依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用, 两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51.c.h.o](#)
- [v40.xx 鸿蒙内核源码分析\(汇编汇总篇\) | 汇编可爱如邻家女孩 | 51.c.h.o](#)
- [v39.xx 鸿蒙内核源码分析\(异常接管篇\) | 社会很单纯, 复杂的是人 | 51.c.h.o](#)
- [v38.xx 鸿蒙内核源码分析\(寄存器篇\) | 小强乃宇宙最忙存储器 | 51.c.h.o](#)
- [v37.xx 鸿蒙内核源码分析\(系统调用篇\) | 开发者永远的口头禅 | 51.c.h.o](#)
- [v36.xx 鸿蒙内核源码分析\(工作模式篇\) | CPU是韦小宝, 七个老婆 | 51.c.h.o](#)
- [v35.xx 鸿蒙内核源码分析\(时间管理篇\) | 谁是内核基本时间单位 | 51.c.h.o](#)
- [v34.xx 鸿蒙内核源码分析\(原子操作篇\) | 谁在为原子操作保驾护航 | 51.c.h.o](#)
- [v33.xx 鸿蒙内核源码分析\(消息队列篇\) | 进程间如何异步传递大数据 | 51.c.h.o](#)
- [v32.xx 鸿蒙内核源码分析\(CPU篇\) | 整个内核就是一个死循环 | 51.c.h.o](#)
- [v31.xx 鸿蒙内核源码分析\(定时器篇\) | 哪个任务的优先级最高 | 51.c.h.o](#)
- [v30.xx 鸿蒙内核源码分析\(事件控制篇\) | 任务间多对多的同步方案 | 51.c.h.o](#)
- [v29.xx 鸿蒙内核源码分析\(信号量篇\) | 谁在负责解决任务的同步 | 51.c.h.o](#)
- [v28.xx 鸿蒙内核源码分析\(进程通讯篇\) | 九种进程间通讯方式速览 | 51.c.h.o](#)
- [v27.xx 鸿蒙内核源码分析\(互斥锁篇\) | 比自旋锁丰满的互斥锁 | 51.c.h.o](#)
- [v26.xx 鸿蒙内核源码分析\(自旋锁篇\) | 自旋锁当立贞节牌坊 | 51.c.h.o](#)
- [v25.xx 鸿蒙内核源码分析\(并发并行篇\) | 听过无数遍的两个概念 | 51.c.h.o](#)
- [v24.xx 鸿蒙内核源码分析\(进程概念篇\) | 进程在管理哪些资源 | 51.c.h.o](#)
- [v23.xx 鸿蒙内核源码分析\(汇编传参篇\) | 如何传递复杂的参数 | 51.c.h.o](#)

- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功,也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o, 希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符, 感谢这些站点一直以来对系列篇的支持和推荐, 尤其是 [oschina](#) [gitee](#), 很喜欢它的界面风格, 简洁大方, 让人感觉到开源的伟大!

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件, 这就很有意思了, 冥冥之中似有天数, 将这四个宝贝以这种方式融合在一起. 51.c.h.o, 我要CHO, 嗯嗯, hin 顺口:)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码, 中文注解分析, 深挖地基工程, 大脑永久记忆, 四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核, 问答式导读, 生活式比喻, 表格化说明, 图形化展示, 主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

54_静态链接篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

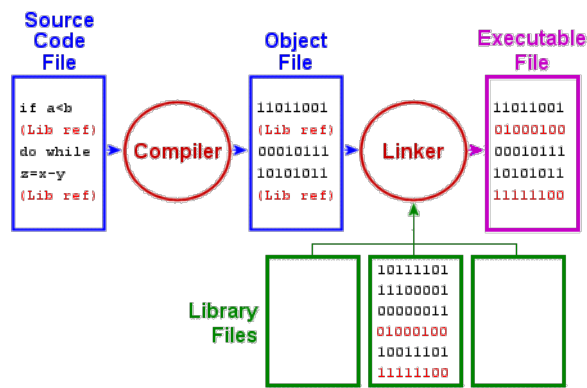
内核源码注解分析 →

OpenHarmony开发者文档 →

百篇博客系列篇.本篇为:

- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o

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，阅读之前建议先看

- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

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
.....				
[16]	.text	PROGBITS	0000000000001060	00001060
0000000000000255	0000000000000000	AX	0	0
[17]	.fini	PROGBITS	00000000000012b8	000012b8
000000000000000d	0000000000000000	AX	0	0
[18]	.rodata	PROGBITS	0000000000002000	00002000
00000000000000cd	0000000000000000	A	0	0
.....				
[25]	.data	PROGBITS	0000000000004000	00003000
0000000000000020	0000000000000000	WA	0	0
[26]	.bss	NOBITS	0000000000004020	00003020
0000000000000008	0000000000000000	WA	0	0

```
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: 00000000000003db8 0 OBJECT LOCAL DEFAULT 21 __frame_dummy_init_array_
41: 00000000000000000 0 FILE LOCAL DEFAULT ABS part.c
42: 00000000000000000 0 FILE LOCAL DEFAULT ABS main.c
43: 00000000000000000 0 FILE LOCAL DEFAULT ABS crtstuff.c
44: 0000000000000225c 0 OBJECT LOCAL DEFAULT 20 __FRAME_END__
45: 00000000000000000 0 FILE LOCAL DEFAULT ABS
46: 00000000000003dc0 0 NOTYPE LOCAL DEFAULT 21 __init_array_end
47: 00000000000003dc8 0 OBJECT LOCAL DEFAULT 23 _DYNAMIC
48: 00000000000003db8 0 NOTYPE LOCAL DEFAULT 21 __init_array_start
49: 000000000000020c0 0 NOTYPE LOCAL DEFAULT 19 __GNU_EH_FRAME_HDR
50: 00000000000003fb8 0 OBJECT LOCAL DEFAULT 24 _GLOBAL_OFFSET_TABLE_
51: 00000000000001000 0 FUNC LOCAL DEFAULT 12 _init
52: 000000000000012b0 5 FUNC GLOBAL DEFAULT 16 __libc_csu_fini
53: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _ITM_deregisterTMCloneTab
54: 00000000000004000 0 NOTYPE WEAK DEFAULT 25 data_start
55: 00000000000004020 0 NOTYPE GLOBAL DEFAULT 25 _edata
56: 000000000000012b8 0 FUNC GLOBAL HIDDEN 17 _fini
57: 00000000000000000 0 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.2.5
58: 00000000000004010 4 OBJECT GLOBAL DEFAULT 25 g_int
59: 00000000000000000 0 FUNC GLOBAL DEFAULT UND __libc_start_main@@GLIBC_
60: 00000000000004000 0 NOTYPE GLOBAL DEFAULT 25 __data_start
61: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _gmon_start__
62: 00000000000004008 0 OBJECT GLOBAL HIDDEN 25 __dso_handle
63: 00000000000004018 8 OBJECT GLOBAL DEFAULT 25 g_str
64: 00000000000002000 4 OBJECT GLOBAL DEFAULT 18 _IO_stdin_used
65: 00000000000001240 101 FUNC GLOBAL DEFAULT 16 __libc_csu_init
66: 00000000000001149 52 FUNC GLOBAL DEFAULT 16 func_int
67: 00000000000004028 0 NOTYPE GLOBAL DEFAULT 26 _end
68: 00000000000001060 47 FUNC GLOBAL DEFAULT 16 _start
69: 0000000000000117d 57 FUNC GLOBAL DEFAULT 16 func_str
70: 00000000000004020 0 NOTYPE GLOBAL DEFAULT 26 __bss_start
71: 000000000000011b6 123 FUNC GLOBAL DEFAULT 16 main
72: 00000000000004020 0 OBJECT GLOBAL HIDDEN 25 __TMC_END__
73: 00000000000000000 0 NOTYPE WEAK DEFAULT UND _ITM_registerTMCloneTable
74: 00000000000000000 0 FUNC WEAK DEFAULT UND __cxa_finalize@@GLIBC_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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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， `.xx` 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- [v56.xx 鸿蒙内核源码分析\(进程映像篇\) | ELF是如何被加载运行的? | 51.c.h.o](#)
- [v55.xx 鸿蒙内核源码分析\(重定位篇\) | 与国际接轨的对外部发言人 | 51.c.h.o](#)
- [v54.xx 鸿蒙内核源码分析\(静态链接篇\) | 完整小项目看透静态链接过程 | 51.c.h.o](#)
- [v53.xx 鸿蒙内核源码分析\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v52.xx 鸿蒙内核源码分析\(静态站点篇\) | 五一哪也没去就干了这事 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)
- [v50.xx 鸿蒙内核源码分析\(编译环境篇\) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o](#)
- [v49.xx 鸿蒙内核源码分析\(信号消费篇\) | 谁让CPU连续四次换栈运行 | 51.c.h.o](#)
- [v48.xx 鸿蒙内核源码分析\(信号生产篇\) | 年过半百，依然活力十足 | 51.c.h.o](#)
- [v47.xx 鸿蒙内核源码分析\(进程回收篇\) | 临终前如何向老祖宗托孤 | 51.c.h.o](#)
- [v46.xx 鸿蒙内核源码分析\(特殊进程篇\) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o](#)
- [v45.xx 鸿蒙内核源码分析\(Fork篇\) | 一次调用，两次返回 | 51.c.h.o](#)
- [v44.xx 鸿蒙内核源码分析\(中断管理篇\) | 江湖从此不再怕中断 | 51.c.h.o](#)
- [v43.xx 鸿蒙内核源码分析\(中断概念篇\) | 海公公的日常工作 | 51.c.h.o](#)
- [v42.xx 鸿蒙内核源码分析\(中断切换篇\) | 系统因中断活力四射 | 51.c.h.o](#)
- [v41.xx 鸿蒙内核源码分析\(任务切换篇\) | 看汇编如何切换任务 | 51.c.h.o](#)
- [v40.xx 鸿蒙内核源码分析\(汇编汇总篇\) | 汇编可爱如邻家女孩 | 51.c.h.o](#)
- [v39.xx 鸿蒙内核源码分析\(异常接管篇\) | 社会很单纯，复杂的是人 | 51.c.h.o](#)
- [v38.xx 鸿蒙内核源码分析\(寄存器篇\) | 小强乃宇宙最忙存储器 | 51.c.h.o](#)
- [v37.xx 鸿蒙内核源码分析\(系统调用篇\) | 开发者永远的口头禅 | 51.c.h.o](#)
- [v36.xx 鸿蒙内核源码分析\(工作模式篇\) | CPU是韦小宝，七个老婆 | 51.c.h.o](#)
- [v35.xx 鸿蒙内核源码分析\(时间管理篇\) | 谁是内核基本时间单位 | 51.c.h.o](#)

- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o
- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto

- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 .c .h .o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. 51 .c .h .o ， 我要CHO ，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新< [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中< [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

55_重定位篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51 .c .h .o

本篇是以下篇的延续, 建议先看

- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51 .c .h .o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51 .c .h .o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51 .c .h .o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51 .c .h .o

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_GLOB_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_GLOB_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_JMP_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[!LiaprmfFsoRtUuTgAckK] 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
  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 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 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` 和 `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:
00000000000001050 <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*+0x00000000000001140
0000000000003dc0 R_X86_64_RELATIVE *ABS*+0x00000000000001100
0000000000004008 R_X86_64_RELATIVE *ABS*+0x00000000000004008
0000000000004018 R_X86_64_RELATIVE *ABS*+0x00000000000002008
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。

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，`.xx` 代表修改的次数，精雕细琢，言

简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百，依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用，两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯，复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝，七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o

- [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 | 51.c.h.o](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花 | 51.c.h.o](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义 | 51.c.h.o](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存 | 51.c.h.o](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么 | 51.c.h.o](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里 | 51.c.h.o](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础 | 51.c.h.o](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功 | 51.c.h.o](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的 | 51.c.h.o](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式 | 51.c.h.o](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处 | 51.c.h.o](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的 | 51.c.h.o](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列 | 51.c.h.o](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的 | 51.c.h.o](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元 | 51.c.h.o](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大 | 51.c.h.o](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源 | 51.c.h.o](#)
- [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 | 51.c.h.o](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大!

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. `51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

56_进程映像篇

将 HarmonyOS | 鸿蒙 研究到底 < 国内 | 国外 >

将 HarmonyOS | 鸿蒙 研究到底

<https://gitee.com/weharmony>

[内核源码注解分析 →](#)

[OpenHarmony开发者文档 →](#)

百篇博客系列篇.本篇为:

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51 .c .h .o

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 = OsCreateUserVmSpace();//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

解析，装载过程

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，本篇说明加载过程，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\(ELF解析篇\) | 你要忘了她姐俩你就不是银 | 51.c.h.o](#)
- [v51.xx 鸿蒙内核源码分析\(ELF格式篇\) | 应用程序入口并不是main | 51.c.h.o](#)

的不会陌生，对照着代码去看很容易理解.

```
root@5e3abe332c5a:/home/harmony/out/hisparc_aries/ipcamera_hisparc_aries/bin# readelf -h shell
ELF Header:
```

```
  Magic:   7f 45 4c 46 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/hisparc_aries/ipcamera_hisparc_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...

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

```

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
    INT32      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 段加载到对应的位置, 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.

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.

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.

百篇博客.往期回顾

在加注过程中，整理出以下文章。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.确实有难度，自不量力，但已经出发，回头已是不可能的了。 :P

与代码有bug需不断debug一样，文章和注解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，.xx 代表修改的次数，精雕细琢，言简意赅，力求打造精品内容。

- v56.xx 鸿蒙内核源码分析(进程映像篇) | ELF是如何被加载运行的? | 51.c.h.o
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人 | 51.c.h.o
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程 | 51.c.h.o
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 你要忘了她姐俩你就不是银 | 51.c.h.o
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事 | 51.c.h.o
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main | 51.c.h.o
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了 | 51.c.h.o
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行 | 51.c.h.o
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足 | 51.c.h.o
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤 | 51.c.h.o
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞 | 51.c.h.o
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回 | 51.c.h.o
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断 | 51.c.h.o
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作 | 51.c.h.o
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射 | 51.c.h.o
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务 | 51.c.h.o
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩 | 51.c.h.o
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人 | 51.c.h.o
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器 | 51.c.h.o
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅 | 51.c.h.o
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆 | 51.c.h.o
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位 | 51.c.h.o
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航 | 51.c.h.o
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据 | 51.c.h.o
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环 | 51.c.h.o
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高 | 51.c.h.o
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案 | 51.c.h.o
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步 | 51.c.h.o
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速览 | 51.c.h.o
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁 | 51.c.h.o
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊 | 51.c.h.o
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念 | 51.c.h.o
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源 | 51.c.h.o
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数 | 51.c.h.o
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班 | 51.c.h.o
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU | 51.c.h.o
- 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 | 51.c.h.o

- v19.xx 鸿蒙内核源码分析(位图管理篇) | 谁能一分钱分两半花 | 51.c.h.o
- v18.xx 鸿蒙内核源码分析(源码结构篇) | 内核每个文件的含义 | 51.c.h.o
- v17.xx 鸿蒙内核源码分析(物理内存篇) | 怎么管理物理内存 | 51.c.h.o
- v16.xx 鸿蒙内核源码分析(内存规则篇) | 内存管理到底在管什么 | 51.c.h.o
- v15.xx 鸿蒙内核源码分析(内存映射篇) | 虚拟内存虚在哪里 | 51.c.h.o
- v14.xx 鸿蒙内核源码分析(内存汇编篇) | 谁是虚拟内存实现的基础 | 51.c.h.o
- v13.xx 鸿蒙内核源码分析(源码注释篇) | 鸿蒙必定成功，也必然成功 | 51.c.h.o
- v12.xx 鸿蒙内核源码分析(内存管理篇) | 虚拟内存全景图是怎样的 | 51.c.h.o
- v11.xx 鸿蒙内核源码分析(内存分配篇) | 内存有哪些分配方式 | 51.c.h.o
- v10.xx 鸿蒙内核源码分析(内存主奴篇) | 皇上和奴才如何相处 | 51.c.h.o
- v09.xx 鸿蒙内核源码分析(调度故事篇) | 用故事说内核调度过程 | 51.c.h.o
- 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 | 51.c.h.o
- v07.xx 鸿蒙内核源码分析(调度机制篇) | 任务是如何被调度执行的 | 51.c.h.o
- v06.xx 鸿蒙内核源码分析(调度队列篇) | 内核有多少个调度队列 | 51.c.h.o
- v05.xx 鸿蒙内核源码分析(任务管理篇) | 任务池是如何管理的 | 51.c.h.o
- v04.xx 鸿蒙内核源码分析(任务调度篇) | 任务是内核调度的单元 | 51.c.h.o
- v03.xx 鸿蒙内核源码分析(时钟任务篇) | 触发调度谁的贡献最大 | 51.c.h.o
- v02.xx 鸿蒙内核源码分析(进程管理篇) | 谁在管理内核资源 | 51.c.h.o
- 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 | 51.c.h.o

关于 51.c.h.o

看系列篇文章会常看到 51.c.h.o，希望这对大家阅读不会造成影响。分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- 51cto
- csdn
- harmony
- oschina

而巧合的是 .c.h.o 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起。51.c.h.o，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处.

readme

- v56.xx 鸿蒙内核源码分析(进程镜像篇) | ELF是如何被加载运行的?
- v55.xx 鸿蒙内核源码分析(重定位篇) | 与国际接轨的对外部发言人
- v54.xx 鸿蒙内核源码分析(静态链接篇) | 完整小项目看透静态链接过程
- v53.xx 鸿蒙内核源码分析(ELF解析篇) | 都这样了你还能忘了她姐俩?
- v52.xx 鸿蒙内核源码分析(静态站点篇) | 五一哪也没去就干了这事
- v51.xx 鸿蒙内核源码分析(ELF格式篇) | 应用程序入口并不是main
- v50.xx 鸿蒙内核源码分析(编译环境篇) | 编译鸿蒙看这篇或许真的够了
- v49.xx 鸿蒙内核源码分析(信号消费篇) | 谁让CPU连续四次换栈运行
- v48.xx 鸿蒙内核源码分析(信号生产篇) | 年过半百,依然活力十足
- v47.xx 鸿蒙内核源码分析(进程回收篇) | 临终前如何向老祖宗托孤
- v46.xx 鸿蒙内核源码分析(特殊进程篇) | 龙生龙凤生凤老鼠生儿会打洞
- v45.xx 鸿蒙内核源码分析(Fork篇) | 一次调用,两次返回
- v44.xx 鸿蒙内核源码分析(中断管理篇) | 江湖从此不再怕中断
- v43.xx 鸿蒙内核源码分析(中断概念篇) | 海公公的日常工作
- v42.xx 鸿蒙内核源码分析(中断切换篇) | 系统因中断活力四射
- v41.xx 鸿蒙内核源码分析(任务切换篇) | 看汇编如何切换任务
- v40.xx 鸿蒙内核源码分析(汇编汇总篇) | 汇编可爱如邻家女孩
- v39.xx 鸿蒙内核源码分析(异常接管篇) | 社会很单纯,复杂的是人
- v38.xx 鸿蒙内核源码分析(寄存器篇) | 小强乃宇宙最忙存储器
- v37.xx 鸿蒙内核源码分析(系统调用篇) | 开发者永远的口头禅
- v36.xx 鸿蒙内核源码分析(工作模式篇) | CPU是韦小宝,七个老婆
- v35.xx 鸿蒙内核源码分析(时间管理篇) | 谁是内核基本时间单位
- v34.xx 鸿蒙内核源码分析(原子操作篇) | 谁在为原子操作保驾护航
- v33.xx 鸿蒙内核源码分析(消息队列篇) | 进程间如何异步传递大数据
- v32.xx 鸿蒙内核源码分析(CPU篇) | 整个内核就是一个死循环
- v31.xx 鸿蒙内核源码分析(定时器篇) | 哪个任务的优先级最高
- v30.xx 鸿蒙内核源码分析(事件控制篇) | 任务间多对多的同步方案
- v29.xx 鸿蒙内核源码分析(信号量篇) | 谁在负责解决任务的同步
- v28.xx 鸿蒙内核源码分析(进程通讯篇) | 九种进程间通讯方式速揽
- v27.xx 鸿蒙内核源码分析(互斥锁篇) | 比自旋锁丰满的互斥锁
- v26.xx 鸿蒙内核源码分析(自旋锁篇) | 自旋锁当立贞节牌坊
- v25.xx 鸿蒙内核源码分析(并发并行篇) | 听过无数遍的两个概念
- v24.xx 鸿蒙内核源码分析(进程概念篇) | 进程在管理哪些资源
- v23.xx 鸿蒙内核源码分析(汇编传参篇) | 如何传递复杂的参数
- v22.xx 鸿蒙内核源码分析(汇编基础篇) | CPU在哪里打卡上班
- v21.xx 鸿蒙内核源码分析(线程概念篇) | 是谁在不断的折腾CPU

- [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](#)
- [v19.xx 鸿蒙内核源码分析\(位图管理篇\) | 谁能一分钱分两半花](#)
- [v18.xx 鸿蒙内核源码分析\(源码结构篇\) | 内核每个文件的含义](#)
- [v17.xx 鸿蒙内核源码分析\(物理内存篇\) | 怎么管理物理内存](#)
- [v16.xx 鸿蒙内核源码分析\(内存规则篇\) | 内存管理到底在管什么](#)
- [v15.xx 鸿蒙内核源码分析\(内存映射篇\) | 虚拟内存虚在哪里](#)
- [v14.xx 鸿蒙内核源码分析\(内存汇编篇\) | 谁是虚拟内存实现的基础](#)
- [v13.xx 鸿蒙内核源码分析\(源码注释篇\) | 鸿蒙必定成功，也必然成功](#)
- [v12.xx 鸿蒙内核源码分析\(内存管理篇\) | 虚拟内存全景图是怎样的](#)
- [v11.xx 鸿蒙内核源码分析\(内存分配篇\) | 内存有哪些分配方式](#)
- [v10.xx 鸿蒙内核源码分析\(内存主奴篇\) | 皇上和奴才如何相处](#)
- [v09.xx 鸿蒙内核源码分析\(调度故事篇\) | 用故事说内核调度过程](#)
- [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](#)
- [v07.xx 鸿蒙内核源码分析\(调度机制篇\) | 任务是如何被调度执行的](#)
- [v06.xx 鸿蒙内核源码分析\(调度队列篇\) | 内核有多少个调度队列](#)
- [v05.xx 鸿蒙内核源码分析\(任务管理篇\) | 任务池是如何管理的](#)
- [v04.xx 鸿蒙内核源码分析\(任务调度篇\) | 任务是内核调度的单元](#)
- [v03.xx 鸿蒙内核源码分析\(时钟任务篇\) | 触发调度谁的贡献最大](#)
- [v02.xx 鸿蒙内核源码分析\(进程管理篇\) | 谁在管理内核资源](#)
- [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](#)

关于 51.c.h.o

看系列篇文章会常看到 `51.c.h.o`，希望这对大家阅读不会造成影响. 分别对应以下四个站点的首个字符，感谢这些站点一直以来对系列篇的支持和推荐，尤其是 [oschina](#) [gitee](#)，很喜欢它的界面风格，简洁大方，让人感觉到开源的伟大！

- [51cto](#)
- [csdn](#)
- [harmony](#)
- [oschina](#)

而巧合的是 `.c.h.o` 是C语言的头/源/目标文件，这就很有意思了，冥冥之中似有天数，将这四个宝贝以这种方式融合在一起. `51.c.h.o`，我要CHO，嗯嗯，hin 顺口：)

百万汉字注解.百篇博客分析

百万汉字注解 >> 精读鸿蒙源码，中文注解分析，深挖地基工程，大脑永久记忆，四大码仓每日同步更新 < [gitee](#) | [github](#) | [csdn](#) | [coding](#) >

百篇博客分析 >> 故事说内核，问答式导读，生活式比喻，表格化说明，图形化展示，主流站点定期更新中 < [51cto](#) | [csdn](#) | [harmony](#) | [osc](#) >

关注不迷路.代码即人生



微信搜一搜



鸿蒙内核源码分析

热爱是所有的理由和答案 - turing

原创不易，欢迎转载，但麻烦请注明出处。

