

说出来你可能不信，内核这家伙在内存的使用上给自己开了个小灶！

Original 张彦飞allen 开发内功修炼 2021-01-24 16:08

收录于合集

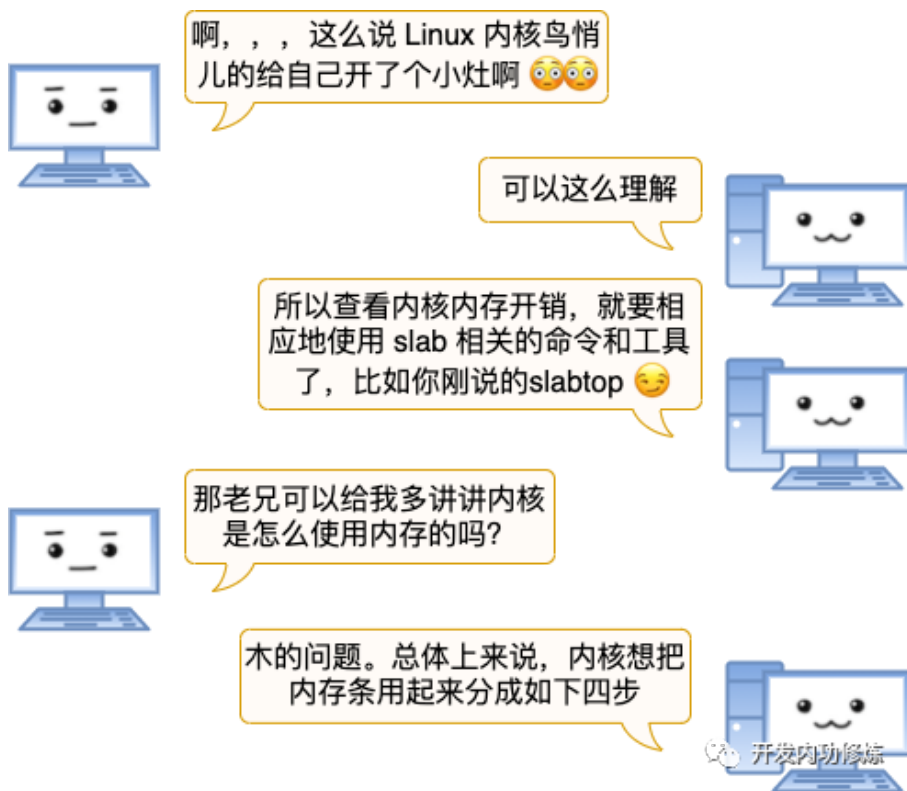
#开发内功修炼之内存篇

10个

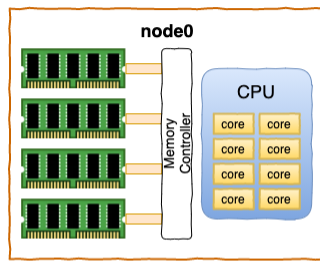
点击上方蓝字“开发内功修炼”，关注并设为星标

飞哥的硬核文章将第一时间送达~~~

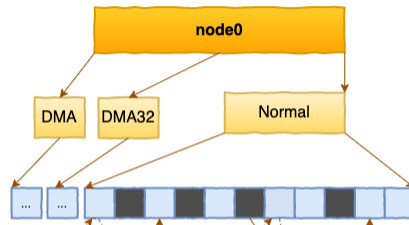
○



1. 相邻的内存条和 CPU 被划分成 node



2. 每个node被划分成多个zone, 每个zone下包含很多个页面



3. 每个 zone 下的空闲页面都通过一个伙伴系统进行管理



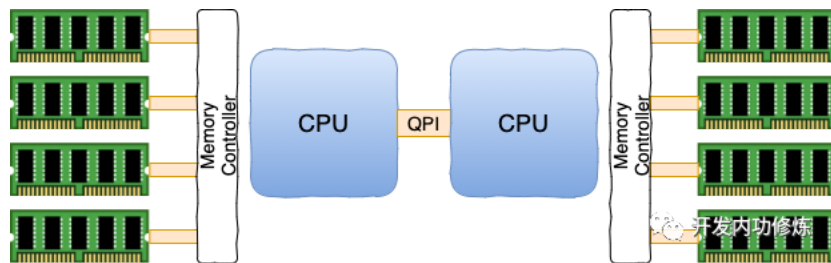
4. slab 分配器向伙伴系统申请连续整页内存存储内核对象



现在你可能还觉得node、zone、伙伴系统、slab这些东东还有那么一点点陌生。别怕，接下来我们结合动手观察，把它们逐个来展开细说。（下面的讨论都基于Linux 3.10.0版本）

一、NODE 划分

在现代的服务器上，内存和CPU都是所谓的NUMA架构



CPU往往不止是一颗。通过dmidecode命令看到你主板上插着的CPU的详细信息

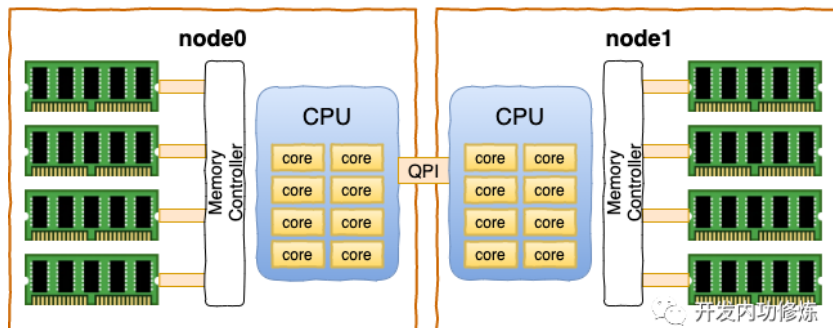
```
Processor Information //第一颗CPU
SocketDesignation: CPU1
Version: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Core Count: 8
Thread Count: 16
Processor Information //第二颗CPU
Socket Designation: CPU2
Version: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
Core Count: 8
```

内存也不只是一条。dmidecode同样可以查看到服务器上插着的所有内存条，也可以看到它是和哪个CPU直接连接的。

```
//CPU1 上总共插着四条内存
Memory Device
Size: 16384 MB
Locator: CPU1 DIMM A1
Memory Device
Size: 16384 MB
Locator: CPU1 DIMM A2
.....
//CPU2 上也插着四条
```

```
Memory Device
Size: 16384 MB
Locator: CPU2 DIMM E1
Memory Device
Size: 16384 MB
Locator: CPU2 DIMM F1
.....
```

每一个CPU以及和他直连的内存条组成了一个 **node (节点)**。

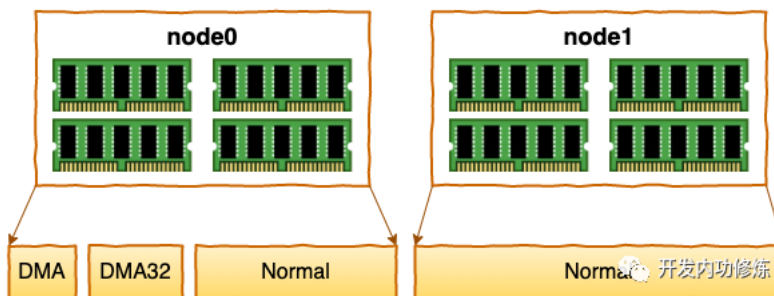


在你的机器上，你可以使用numactl你可以看到每个node的情况

```
numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 65419 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 65536 MB
```

二、ZONE 划分

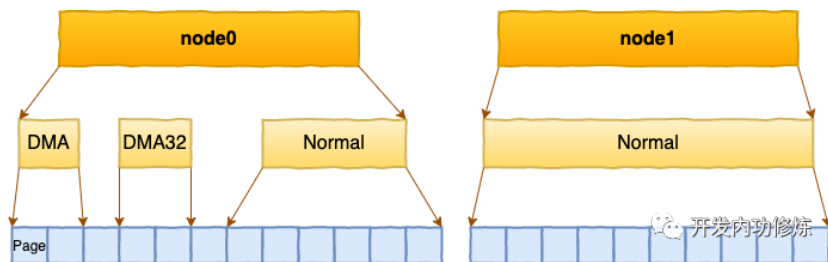
每个 node 又会划分成若干的 **zone (区域)**。zone 表示内存中的一块范围



- ZONE_DMA: 地址段最低的一块内存区域，ISA(Industry Standard Architecture)设备DMA访问
- ZONE_DMA32: 该Zone用于支持32-bits地址总线的DMA设备，只在64-bits系统里才有效
- ZONE_NORMAL: 在X86-64架构下，DMA和DMA32之外的内存全部在NORMAL的Zone里管理

为什么没有提 ZONE_HIGHMEM 这个zone? 因为这是 32 位机时代的产物。现在应该没谁在用这种古董了吧。

在每个zone下，都包含了许许多多 Page (页面)，在linux下一个Page的大小一般是 4 KB。



在你的机器上，你可以使用通过 `zoneinfo` 查看到你机器上 `zone` 的划分，也可以看到每个 `zone` 下所管理的页面有多少个。

```
# cat /proc/zoneinfo
Node 0, zone DMA
  pages free      3973
  managed         3973
Node 0, zone DMA32
  pages free     390390
  managed        427659
Node 0, zone Normal
  pages free    15021616
  managed     15990165
Node 1, zone Normal
  pages free    16012823
  managed     16514393
```

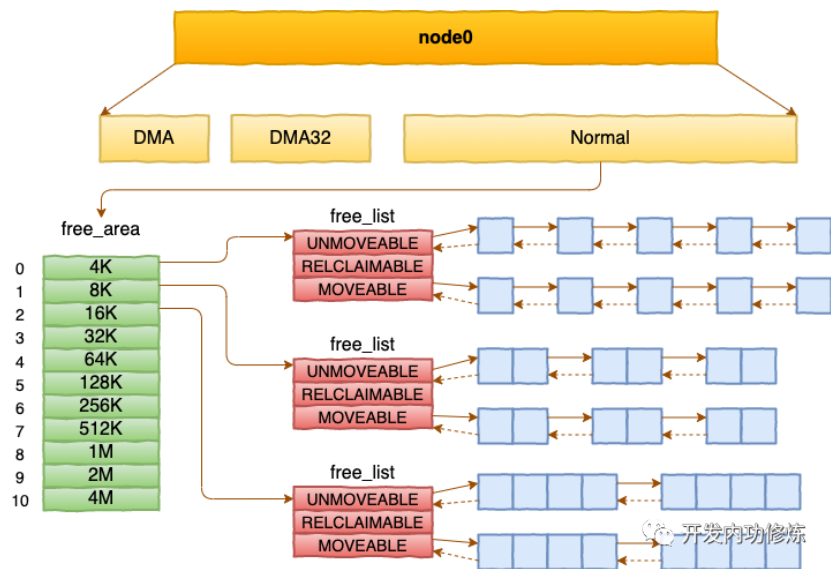
每个页面大小是4K，很容易可以计算出每个 `zone` 的大小。比如对于上面 Node1 的 Normal， $16514393 * 4K = 66 \text{ GB}$ 。

三、基于伙伴系统管理空闲页面

每个 `zone` 下面都有如此之多的页面，Linux使用**伙伴系统**对这些页面进行高效的管理。在内核中，表示 `zone` 的数据结构是 `struct zone`。其下面的一个数组 `free_area` 管理了绝大部分可用的空闲页面。这个数组就是**伙伴系统**实现的重要数据结构。

```
//file: include/linux/mmzone.h
#define MAX_ORDER 11
struct zone {
    free_area    free_area[MAX_ORDER];
    .....
}
```

`free_area`是一个11个元素的数组，在每一个数组分别代表的是空闲可分配连续4K、8K、16K、.....、4M内存链表。



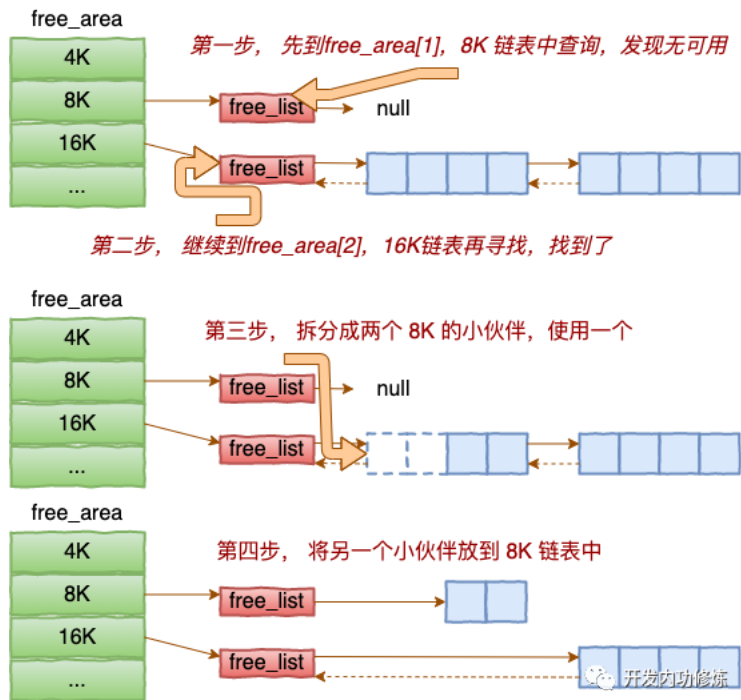
通过 `cat /proc/pagetypeinfo`, 你可以看到当前系统里伙伴系统里各个尺寸的可用连续内存块数量。

Free pages	count	per	migrate	type	at	order	0	1	2	3	4	5	6	7	8	9	10
Node 0, zone	DMA, type	Unmovable	1	0	1	0	2	1	1	0	1	0	0	0	0	0	0
Node 0, zone	DMA, type	Reclaimable	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	DMA, type	Movable	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
Node 0, zone	DMA, type	Reserve	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Node 0, zone	DMA, type	CMA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	DMA, type	Isolate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	DMA32, type	Unmovable	153	126	44	11	15	18	15	18	13	10	64				
Node 0, zone	DMA32, type	Reclaimable	1	0	48	61	67	43	30	10	8	4	1				
Node 0, zone	DMA32, type	Movable	250	1478	617	174	100	48	14	3	1	1	280				
Node 0, zone	DMA32, type	Reserve	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Node 0, zone	DMA32, type	CMA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	DMA32, type	Isolate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	Normal, type	Unmovable	688	193	563	772	673	549	474	434	377	305	1459				
Node 0, zone	Normal, type	Reclaimable	592	869	1499	926	981	674	455	281	165	80	20				
Node 0, zone	Normal, type	Movable	0	24481	12880	3555	1378	950	385	131	76	51	12352				
Node 0, zone	Normal, type	Reserve	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Node 0, zone	Normal, type	CMA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Node 0, zone	Normal, type	Isolate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

内核提供分配器函数 `alloc_pages` 到上面的多个链表中寻找可用连续页面。

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

`alloc_pages` 是怎么工作的呢？我们举个简单的小例子。假如要申请8K-连续两个页框的内存。为了描述方便，我们先暂时忽略UNMOVABLE、RELCLAIMABLE等不同类型



伙伴系统中的伙伴指的是两个内存块，大小相同，地址连续，同属于一个大块区域。

基于伙伴系统的内存分配中，有可能需要将大块内存拆分成两个小伙伴。在释放中，可能会将两个小伙伴合并再次组成更大块的连续内存。

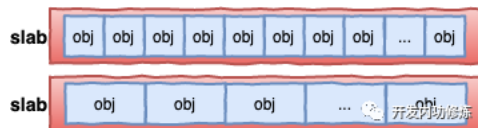
四、SLAB管理器

说到现在，不知道你注意到没有。目前我们介绍的内存分配都是以**页面（4KB）为单位**的。

对于各个内核运行中实际使用的对象来说，多大的对象都有。有的对象有1K多，但有的对象只有几百、甚至几十个字节。如果都直接分配一个 4K的页面 来存储的话也太败家了，所以伙伴系统并不能直接使用。

在伙伴系统之上，**内核又给自己搞了一个专用的内存分配器，叫slab或slub**。这两个词老混用，为了省事，接下来我们就统一叫 slab 吧。

这个分配器最大的特点就是，一个slab内只分配特定大小、甚至是特定的对象。这样当一个对象释放内存后，另一个同类对象可以直接使用这块内存。通过这种办法极大地降低了碎片发生的几率。



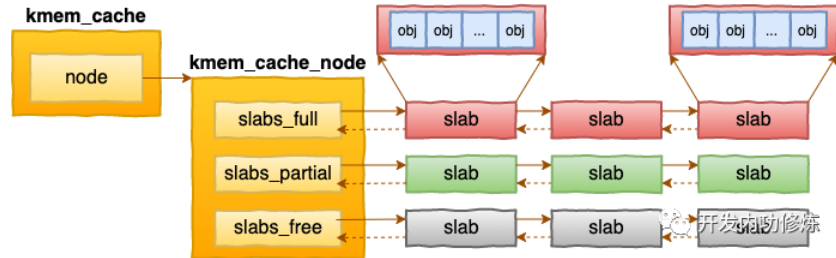
slab相关的内核对象定义如下：

```
//file: include/linux/slab_def.h
struct kmem_cache {
    struct kmem_cache_node **node
    .....
}
```

```
//file: mm/slab.h
struct kmem_cache_node {
    struct list_head slabs_partial;
    struct list_head slabs_full;
    struct list_head slabs_free;
    .....
}
```

每个cache都有满、半满、空三个链表。每个链表节点都对应一个 slab，一个 slab 由 1 个或者多个内存页组成。

在每一个 slab 内都保存的是同等大小的对象。一个cache的组成示意图如下：



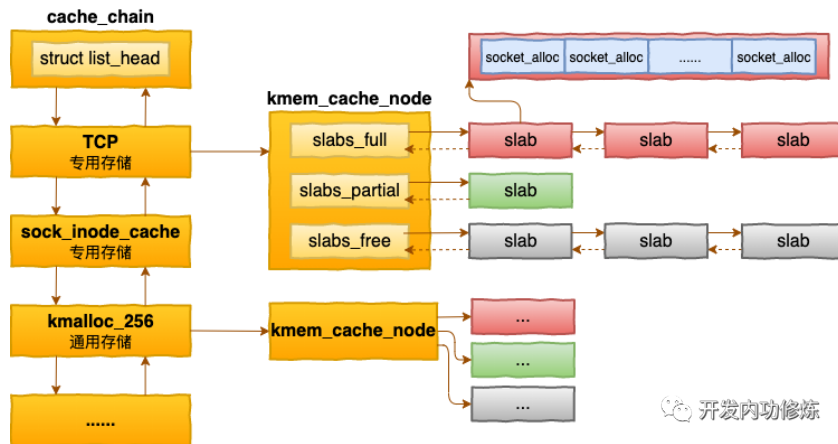
当 cache 中内存不够的时候，会调用基于伙伴系统的分配器（__alloc_pages函数）请求整页连续内存的分配。

```
//file: mm/slab.c
static void *kmem_getpages(struct kmem_cache *cachep,
    gfp_t flags, int nodeid)
{
    .....
    flags |= cachep->allocflags;
    if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
        flags |= __GFP_RECLAIMABLE;

    page = alloc_pages_exact_node(nodeid, ...);
    .....
}
```

```
//file: include/linux/gfp.h
static inline struct page *alloc_pages_exact_node(int nid,
    gfp_t gfp_mask, unsigned int order)
{
    return __alloc_pages(gfp_mask, order, node_zonelist(nid, gfp_mask));
}
```

内核中会有很多个 **kmem_cache** 存在。它们是在linux初始化，或者是运行的过程中分配出来的。它们有的是专用的，有的是通用的。



上图中，我们看到 socket_alloc 内核对象都存在 TCP 的专用 kmem_cache 中。

通过查看 [/proc/slabinfo](#) 我们可以查看到所有的 kmem cache。

```
slabinfo - version: 2.1
# name      topic3      <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
kvm_vcpu    delay_cpu0 mem0.0 0 0 16256 2 8 : tunables 0 0 0 : slab
kvm_mmu_page_header delay_cpu0 mem0.0 0 0 168 48 2 : tunables 0 0 0 : slab
xfs_dqtrx   delay_nobind.cov 992 992 528 31 4 : tunables 0 0 0 : slab
xfs_dquot   index.md 68 68 472 34 4 : tunables 0 0 0 : slab
xfs_icr     index.md 560 560 144 28 1 : tunables 0 0 0 : slab
xfs_ili     index2.md 8480 8692 152 53 2 : tunables 0 0 0 : slab
xfs_inode   37.png 6837 7050 1088 30 8 : tunables 0 0 0 : slab
xfs_efd_item 37.png 2480 2800 400 40 4 : tunables 0 0 0 : slab
xfs_da_state 102 102 480 34 4 : tunables 0 0 0 : slab
xfs_btree_cur 1248 1248 208 39 2 : tunables 0 0 0 : slab
xfs_log_ticket 6996 6996 184 44 2 : tunables 0 0 0 : slab
nfsd4_openowners 0 0 440 37 4 : tunables 0 0 0 : slab
```

另外 linux 还提供了一个特别方便的命令 slabtop 来按照占用内存从大往小进行排列。这个命令用来分析 slab 内存开销非常的方便。

```
Active / Total Objects (% used) : 375042 / 375274 (99.9%)
Active / Total Slabs (% used)   : 15235 / 15235 (100.0%)
Active / Total Caches (% used)  : 79 / 112 (70.5%)
Active / Total Size (% used)    : 183646.03K / 183829.13K (99.9%)
Minimum / Average / Maximum Object : 0.01K / 0.49K / 8.00K
```

OBJS	ACTIVE	USE	OBJ SIZE	SLABS	OBJ/SLAB	CACHE SIZE	NAME
73080	73080	100%	0.19K	3480	21	13920K	dentry
59776	59776	100%	0.06K	934	64	3736K	kmalloc-64
51728	51728	100%	0.25K	3233	16	12932K	kmalloc-256
50175	50175	100%	0.62K	2007	25	32112K	sock_inode cache
50064	50064	100%	1.94K	3129	16	100128K	TCP
15876	15876	100%	0.11K	441	36	1764K	kernfs_node cache

无论是 [/proc/slabinfo](#)，还是 slabtop 命令的输出。里面都包含了每个 cache 中 slab 的如下两个关键信息。

- objsize: 每个对象的大小
- objperslab: 一个 slab 里存放的对象的数量

在 [/proc/slabinfo](#) 还多输出了一个pagesperslab。展示了一个slab 占用的页面的数量，每个页面4K，这样也就能算出每个 slab 占用的内存大小。

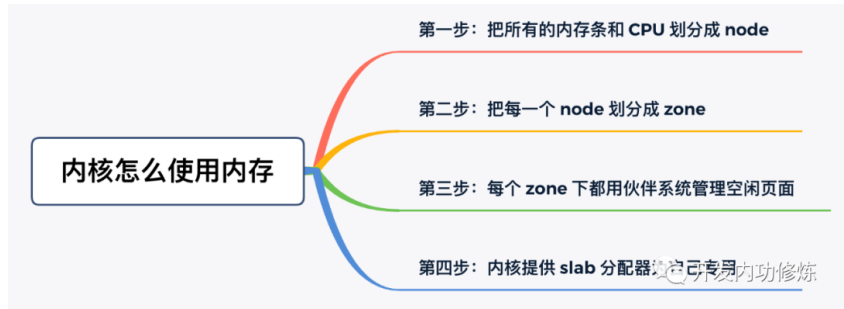
最后，slab 管理器组件提供了若干接口函数，方便自己使用。举三个例子：

- kmem_cache_create: 方便地创建一个基于 slab 的内核对象管理器。
- kmem_cache_alloc: 快速为某个对象申请内存
- kmem_cache_free: 归还对象占用的内存给 slab 管理器

在内核的源码中，可以大量见到 kmem_cache 开头函数的使用。

总结

通过上面描述的几个步骤，内核高效地把内存用了起来。



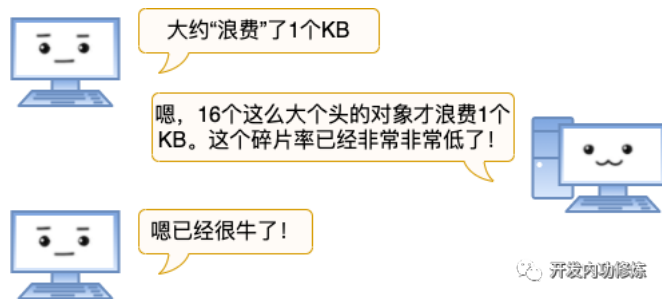
前三步是基础模块，为应用程序分配内存时的请求调页组件也能够用到。但第四步，就算是内核的小灶了。内核根据自己的使用场景，量身打造的一套自用的高效内存分配管理机制。



```
# cat /proc/slabinfo | grep TCP
TCP                288    384   1984    16     8
```

“可以看到 TCP cache下每个 slab 占用 8 个 Page，也就是 $8 * 4096 = 32768\text{KB}$ 。该对象的单个大小是 1984 字节，每个 slab 内放了 16 个对象。 $1984 * 16 = 31744$ ”

“这个时候再多放一个 TCP 对象又放不下，剩下的 1K 内存就只好“浪费”掉了。但是鉴于 slab 机制整体提供的高性能、以及低碎片的效果，这一点点的额外开销还是很值得的。”



飞哥Github出炉，访问请复制下面网址

网址：<https://github.com/yanfeizhang/coder-kung-fu>

附项目预览图如下：

README.md

公众号: 开发内功修炼

在我十年的工作生涯中，我虽然从事的是应用层的开发，但仍然一直保持着对底层的好奇。把工作中遇到的一些问题，进行深度思考。总结出来，在这里分享给有缘的你！

持续更新ing...

一、网络篇

1.1 内核收包原理

- [图解Linux网络包接收过程](#)
- [Linux网络包接收过程的监控与调优](#)

1.2 TCP连接时间开销

- [聊聊TCP连接耗时的那些事儿](#)

1.3 TCP连接内存开销

- [漫画 | 一台Linux服务器最多能支撑多少个TCP连接](#)
- [漫画 | 理解了TCP连接的实现以后，客户端的并发也爆发了！](#)
- [漫画 | 花了七天时间测试，我彻底搞明白了 TCP 的这些内存开销！](#)

二、硬盘篇


2.1 硬件工作原理

- 磁盘开篇：扒开机械硬盘坚硬的外衣！
- 磁盘分区也是隐含了技术技巧的
- 我们怎么解决机械硬盘既慢又容易坏的问题？
- 拆解固态硬盘结构

2.2 文件系统浅析

- 新建一个空文件占用多少磁盘空间？
- 只有1个字节的文件实际占用多少磁盘空间
- 文件过多时ls命令为什么会卡住？
- 理解格式化原理

2.3 文件读写性能

- read文件一个字节实际会发生多大的磁盘IO？
- write文件一个字节后何时发起写磁盘IO？
- 机械硬盘随机IO慢的超乎你的想象
- 搭载固态硬盘的服务器究竟比搭机械硬盘快多少？ 开发内功修炼



开发内功修炼



长按二维码识别关注

据说转发、点赞、点在看的都会变的更帅！

收录于合集 #开发内功修炼之内存篇 10

上一篇

开发内功修炼内存篇汇总

下一篇

容器进程调度时是该优先考虑CPU资源还是内存资源？

People who liked this content also liked

只有我一个人对ChatGPT感到蕉绿吗？

李rumor



看完这篇还不懂高并发中的线程与线程池你来打我(内含20张图)

码农的荒岛求生

