**slub分配器**

多年以来，SLAB成为linux kernel对象缓冲区管理的主流算法，甚至长时间没有人愿意去修改，因为它实在是非常复杂，而且在大多数情况下，它的工作完成的相当不错。但是，随着大规模多处理器系统和 NUMA系统的广泛应用，SLAB 分配器逐渐暴露出自身的严重不足：

1). 缓存队列管理复杂；

2). 管理数据存储开销大；

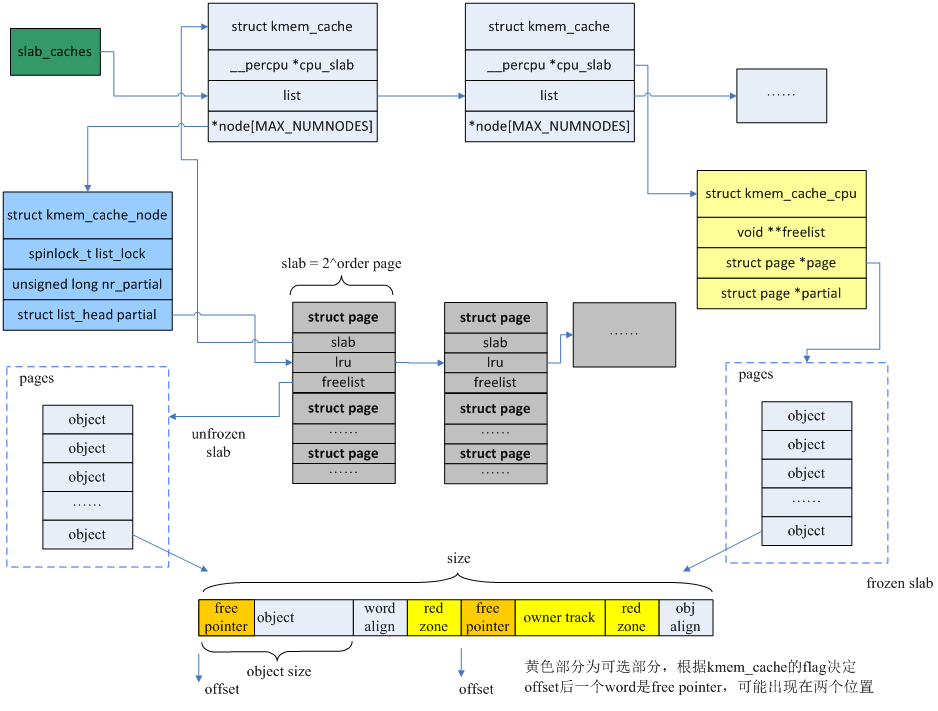
3). 对NUMA支持复杂；

4). 调试调优困难；

5). 摒弃了效果不太明显的slab着色机制；

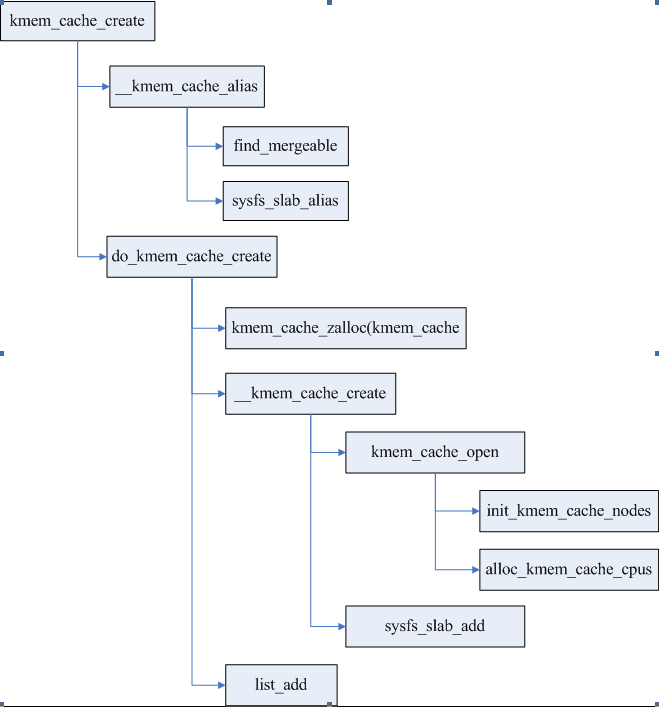
针对这些SLAB不足，内核开发人员Christoph Lameter在Linux内核2.6.22版本中引入一种新的解决方案：SLUB分配器。SLUB分配器特点是简化设计理念，同时保留SLAB分配器的基本思想：每个缓冲区由多个小的slab 组成，每个 slab 包含固定数目的对象。SLUB分配器简化kmem\_cache，slab等相关的管理数据结构，摒弃了SLAB 分配器中众多的队列概念，并针对多处理器、NUMA系统进行优化，从而提高了**性能和可扩展性并降低了内存的浪**费。为了保证内核其它模块能够无缝迁移到SLUB分配器，SLUB还保留了原有SLAB分配器所有的接口API函数。

整体数据结构关系如下图所示：



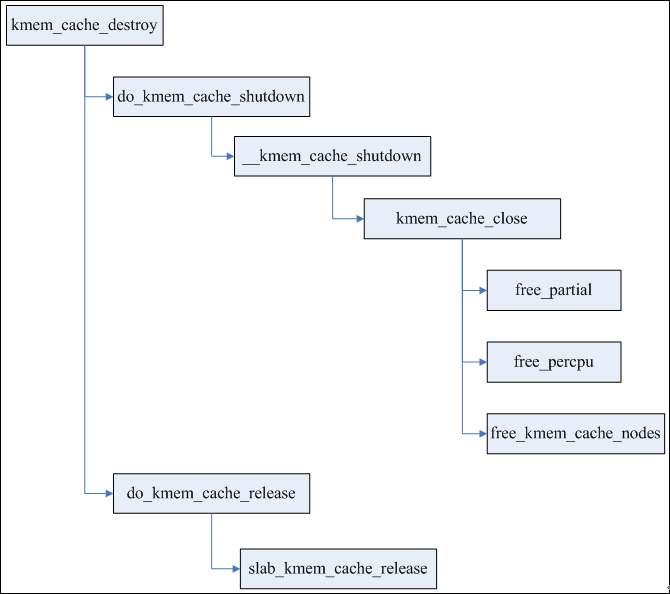
**2 缓存的创建与销毁**

**2.1 缓存的创建**



**2.2 缓存的销毁**

销毁过程比创建过程简单的多，主要工作是释放partial队列所有page，释放kmem\_cache\_cpu，释放每个node的kmem\_cache\_node，最后释放kmem\_cache本身。



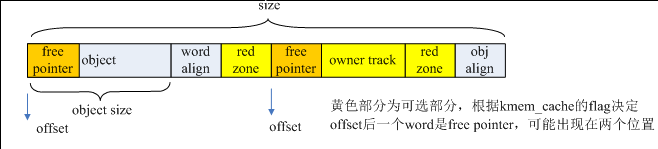
**3 申请对象**

对象是SLUB分配器中可分配的内存单元，与SLAB相比，SLUB对象的组织非常简洁，申请过程更加高效。SLUB没有任何管理区结构来管理对象，而是将对象之间的关联嵌入在对象本身的内存中，因为申请者并不关心对象在分配之前内存的内容是什么。而且各个SLUB之间的关联，也利用page自身结构进行处理。

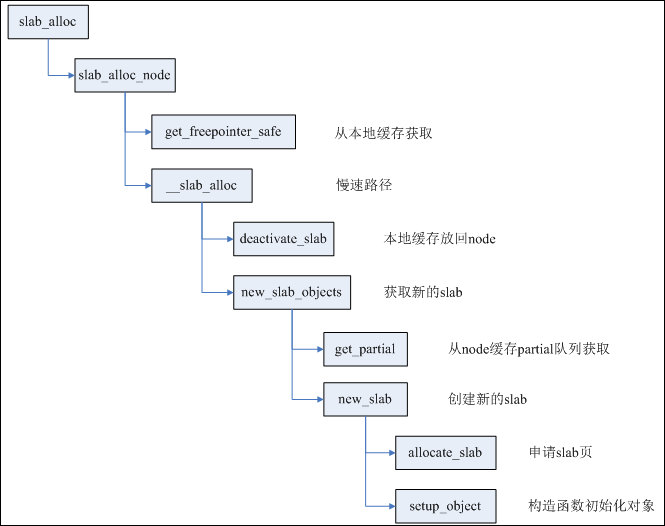
每个CPU都有一个slab作为本地高速缓存，只要slab所在的node与申请者要求的node匹配，同时该slab还有空闲对象，则直接在cpu\_slab中取出空闲对象，否则就进入慢速路径。

每个对象内存的offset偏移位置都存放着下一个空闲对象，offset通常为0，也就是复用对象内存的第一个字来保存下一个空闲对象的指针，当满足条件(flags & (SLAB\_DESTROY\_BY\_RCU | SLAB\_POISON)) 或者有对象构造函数时，offset不为0，每个对象的结构如下图。

cpu\_slab的freelist则保存着当前第一个空闲对象的地址。



如果本地CPU缓存没有空闲对象，则申请新的slab；如果有空闲对象，但是内存node不相符，则deactive当前cpu\_slab，再申请新的slab。



deactivate\_slab主要进行两步工作：

第一步，将cpu\_slab的freelist全部释放回page->freelist；

第二部，根据page(slab)的状态进行不同操作，如果该slab有部分空闲对象，则将page移到kmem\_cache\_node的partial队列；如果该slab全部空闲，则直接释放该slab；如果该slab全部占用，而且开启了CONFIG\_SLUB\_DEBUG编译选项，则将page移到full队列。

page的状态也从frozen改变为unfrozen。（frozen代表slab在cpu\_slub，unfroze代表在partial队列或者full队列）。

申请新的slab有两种情况，如果cpu\_slab的partial队列不为空，则取出队列中下一个page作为新的cpu\_slab，同时再次检测内存node是否相符，还不相符则循环处理。

如果cpu\_slab的partial队列为空，则查看本node的partial队列是否为空，如果不空，则取出page；如果为空，则看一定距离范围内其它node的partial队列，如果还为空，则需要创建新slab。

创建新slab其实就是申请对应order的内存页，用来放足够数量的对象。值得注意的是其中order以及对象数量的确定，这两者又是相互影响的。order和object数量同时存放在kmem\_cache成员kmem\_cache\_order\_objects中，低16位用于存放object数量，高位存放order。order与object数量的关系非常简单：((PAGE\_SIZE << order) - reserved) / size。

下面重点看calculate\_order这个函数。

**4 释放对象**

从上面申请对象的流程也可以看出，释放的object有几个去处：

1）cpu本地缓存slab，也就是cpu\_slab；

2）放回object所在的page（也就是slab）中；另外要处理所在的slab：

2.1）如果放回之后，slab完全为空，则直接销毁该slab；

2.2）如果放回之前，slab为满，则判断slab是否已被冻结；如果已冻结，则不需要做其他事；如果未冻结，则将其冻结，放入cpu\_slab的partial队列；如果cpu\_slab partial队列过多，则将队列中所有slab一次性解冻到各自node的partial队列中。

值得注意的是cpu partial队列的功能是个可选项，依赖于内核选项CONFIG\_SLUB\_CPU\_PARTIAL，如果没有开启，则不使用cpu partial队列，直接使用各个node的partial队列。

**图解slub**

slub是slab中的一种，slab也是slab中的一种。有时候用slab来统称slab, slub和slob。slab, slub和slob仅仅是分配内存策略不同。本篇文章中说的是slub分配器工作的原理。但是针对分配器管理的内存，下文统称为slab缓存池。所以文章中slub和slab会混用，表示同一个意思。

**2. slub数据结构**

slub的数据结构相对于slab来说要简单很多。并且对外接口和slab兼容。所以说，从slab的系统更换到slub，可以说是易如反掌。

**2.1. kmem\_cache**

现在假如从伙伴系统分配一页内存供slub分配器管理。对于slub分配器来说，就是将这段连续内存平均分成若干大小相等的object（对象）进行管理。可是我们总得知道每一个object的size吧！管理的内存页数也是需要知道的吧！不然怎么知道如何分配呢！因此需要一个数据结构管理。那就是struct kmem\_cache。kmem\_cache数据结构描述如下：

1) cpu\_slab：一个per cpu变量，对于每个cpu来说，相当于一个本地内存缓存池。当分配内存的时候优先从本地cpu分配内存以保证cache的命中率。

2) flags：object分配掩码，例如经常使用的SLAB\_HWCACHE\_ALIGN标志位，代表创建的kmem\_cache管理的object按照硬件cache 对齐，一切都是为了速度。

3) min\_partial：限制struct kmem\_cache\_node中的partial链表slab的数量。虽说是mini\_partial，但是代码的本意告诉我这个变量是kmem\_cache\_node中partial链表最大slab数量，如果大于这个mini\_partial的值，那么多余的slab就会被释放。

4) size：分配的object size

5) object\_size：实际的object size，就是创建kmem\_cache时候传递进来的参数。和size的关系就是，size是各种地址对齐之后的大小。因此，size要大于等于object\_size。

6) offset：slub分配在管理object的时候采用的方法是：既然每个object在没有分配之前不在乎每个object中存储的内容，那么完全可以在每个object中存储下一个object内存首地址，就形成了一个单链表。很巧妙的设计。那么这个地址数据存储在object什么位置呢？offset就是存储下个object地址数据相对于这个object首地址的偏移。

7) cpu\_partial：per cpu partial中所有slab的free object的数量的最大值，超过这个值就会将所有的slab转移到kmem\_cache\_node的partial链表。

8) oo：低16位代表一个slab中所有object的数量（oo & ((1 << 16) - 1)），高16位代表一个slab管理的page数量（(2^(oo 16)) pages）。

9) max：看了代码好像就是等于oo。

10) min：当按照oo大小分配内存的时候出现内存不足就会考虑min大小方式分配。min只需要可以容纳一个object即可。

11) allocflags：从伙伴系统分配内存掩码。

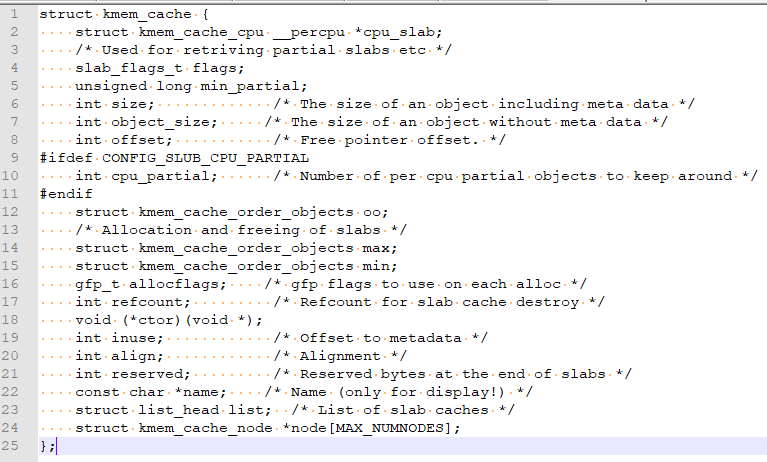
12) inuse：object\_size按照word对齐之后的大小。

13) align：字节对齐大小。

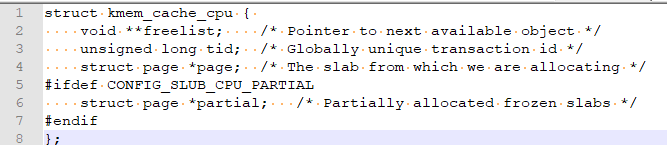
14) name：sysfs文件系统显示使用。

15) list：系统有一个slab\_caches链表，所有的slab都会挂入此链表。

16) node：slab节点。在NUMA系统中，每个node都有一个struct kmem\_cache\_node数据结构。



**2.2. kmem\_cache\_cpu**



struct kmem\_cache\_cpu是对本地内存缓存池的描述，每一个cpu对应一个结构体。其数据结构如下：

1)freelist：指向下一个可用的object。

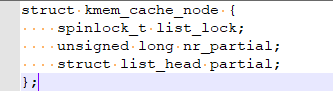
2)tid：一个神奇的数字，主要用来同步作用的。

3)page：slab内存的page指针。

4)partial：本地slab partial链表。主要是一些部分使用object的slab。

**2.3. kmem\_cache\_node**

slab节点使用struct kmem\_cache\_node结构体描述。对于slub分配器来说，成员很少，远比slab分配器简洁。



1) list\_lock： 自旋锁，保护数据。

2) nr\_partial：slab节点中slab的数量。

3) partial： slab节点的slab partial链表，和struct kmem\_cache\_cpu的partial链表功能类似。

**2.4. slub接口**

了解了基本的数据结构，再来看看slub提供的API。如果你了解slub，我想这几个接口你是再熟悉不过了。

|  |
| --- |
| struct kmem\_cache \*kmem\_cache\_create(const char \*name,  size\_t size,  size\_t align,  unsigned long flags,  void (\*ctor)(void \*));  void kmem\_cache\_destroy(struct kmem\_cache \*);  void \*kmem\_cache\_alloc(struct kmem\_cache \*cachep, int flags);  void kmem\_cache\_free(struct kmem\_cache \*cachep, void \*objp); |

1) kmem\_cache\_create是创建kmem\_cache数据结构，参数描述如下：

name：kmem\_cache的名称

size ：slab管理对象的大小

align：slab分配器分配内存的对齐字节数(以align字节对齐)

flags：分配内存掩码

ctor ：分配对象的构造回调函数

2) kmem\_cache\_destroy作用和kmem\_cache\_create相反，就是销毁创建的kmem\_cache。

3) kmem\_cache\_alloc是从cachep参数指定的kmem\_cache管理的内存缓存池中分配一个对象，其中flags是分配掩码，GFP\_KERNEL是不是很熟悉的掩码？

4) kmem\_cache\_free是kmem\_cache\_alloc的反操作

slab分配器提供的接口该如何使用呢？其实很简单，总结分成以下几个步骤：

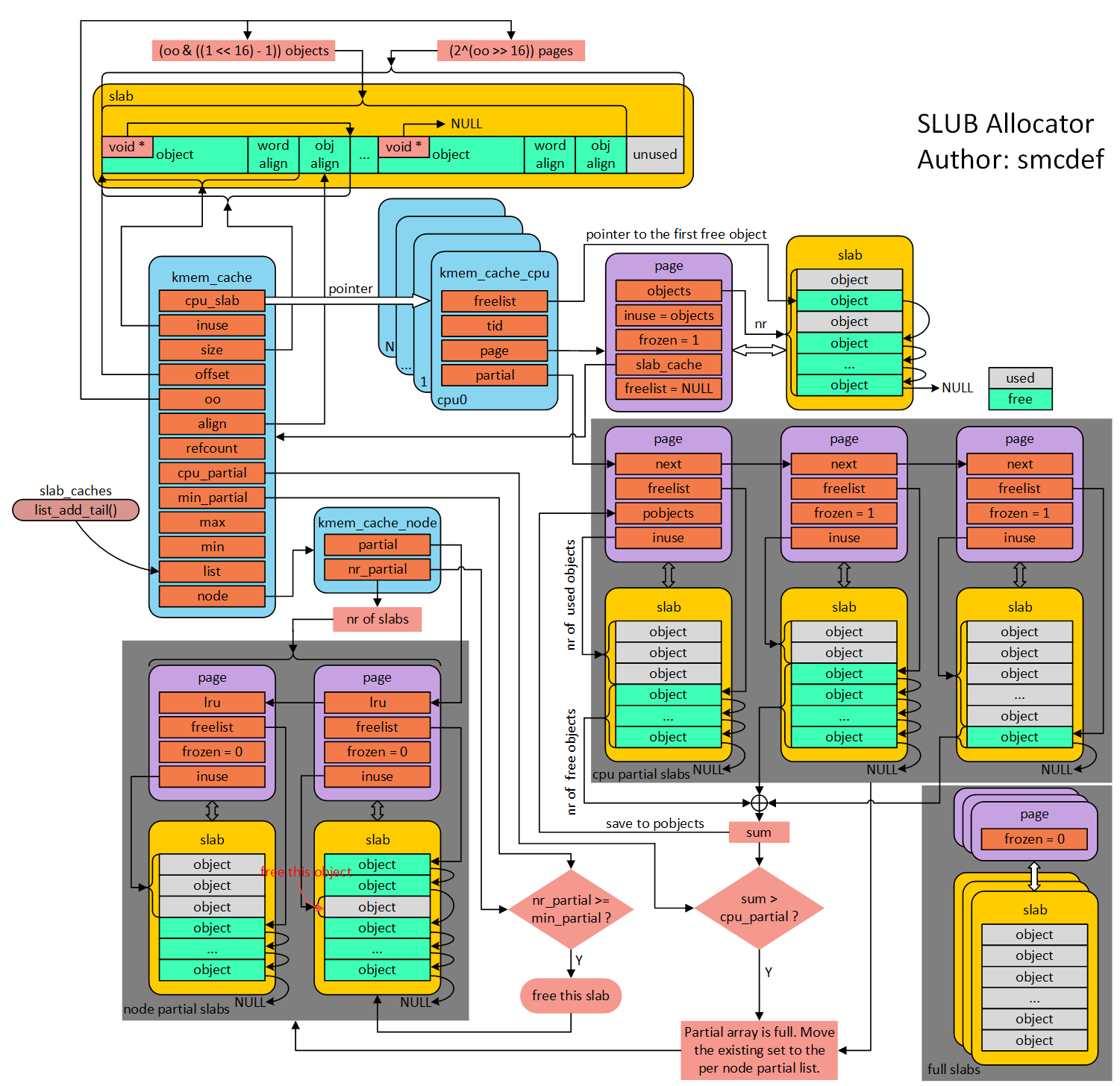
1) kmem\_cache\_create创建一个kmem\_cache数据结构。

2) 使用kmem\_cache\_alloc接口分配内存，kmem\_cache\_free接口释放内存。

3) release第一步创建的kmem\_cache数据结构。

**3. slub数据结构之间关系**

什么是slab缓存池呢？我的解释是使用struct kmem\_cache结构描述的一段内存就称作一个slab缓存池。一个slab缓存池就像是一箱牛奶，一箱牛奶中有很多瓶牛奶，每瓶牛奶就是一个object。分配内存的时候，就相当于从牛奶箱中拿一瓶。总有拿完的一天。当箱子空的时候，你就需要去超市再买一箱回来。超市就相当于partial链表，超市存储着很多箱牛奶。如果超市也卖完了，自然就要从厂家进货，然后出售给你。厂家就相当于伙伴系统。



**3.1. slub管理object方法**

在图片的左上角就是一个slub缓存池中object的分布以及数据结构和kmem\_cache之间的关系。首先一个slab缓存池包含的页数是由oo决定的。oo拆分为两部分，低16位代表一个slab缓存池中object的数量，高16位代表包含的页数。使用kmem\_cache\_create()接口创建kmem\_cache的时候需要指出obj的size和对齐align。也就是传入的参数。kmem\_cache\_create()主要是就是填充kmem\_cache结构体成员。既然从伙伴系统得到(2^(oo >> 16)) pages大小内存，按照size大小进行平分。一般来说都不会整除，因此剩下的就是图中灰色所示。由于每一个object的大小至少8字节，当然可以用来存储下一个object的首地址。就像图中所示的，形成单链表。图中所示下个obj地址存放的位置位于每个obj首地址处，在内核中称作指针内置式。同时，下个obj地址存放的位置和obj首地址之间的偏移存储在kmem\_cache的offset成员。两外一种方式是指针外置式，即下个obj的首地址存储的位置位于obj尾部，也就是在obj尾部再分配sizeof(void \*)字节大小的内存。对于外置式则offset就等于kmem\_cache的inuse成员。

**3.2. per cpu freelist**

针对每一个cpu都会分配一个struct kmem\_cacche\_cpu的结构体。可以称作是本地缓存池。当内存申请的时候，优先从本地cpu缓存池申请。在分配初期，本地缓存池为空，自然要从伙伴系统分配一定页数的内存。内核会为每一个物理页帧创建一个struct page的结构体。kmem\_cacche\_cpu中page就会指向正在使用的slab的页帧。freelist成员指向第一个可用内存obj首地址。处于正在使用的slab的struct page结构体中的freelist会置成NULL，因为没有其他地方使用。struct page结构体中inuse代表已经使用的obj数量。这地方有个很有意思的地方，在刚从伙伴系统分配的slab的 inuse在分配初期就置成obj的总数，在分配obj的时候并不会改变。你是不是觉得很奇怪，既然表示已经使用obj的数量，为什么一直是obj的总数呢？你想想，slab中的对象总有分配完的时候，那个时候就直接脱离kmem\_cache\_cpu了。此时的inuse不就名副其实了嘛！对于full slab就像图的右下角，就像无人看管的孩子，没有任何链表来管理。

**3.3. per cpu partial**

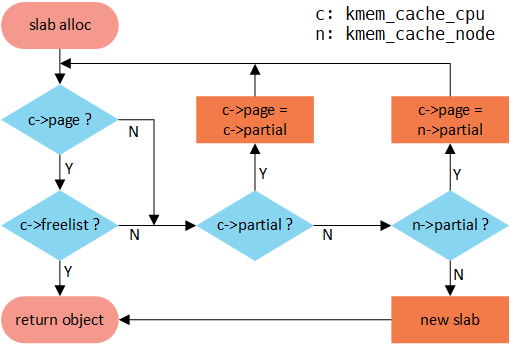
当图中右下角full slab释放obj的时候，首先就会将slab挂入per cpu partial链表管理。通过struct page中next成员形成单链表。per cpu partial链表指向的第一个page中会存放一些特殊的数据。例如：pobjects存储着per cpu partial链表中所有slab可供分配obj的总数，如图所示。当然还有一个图中没有体现的pages成员存储per cpu partial链表中所有slab缓存池的个数。pobjects到底有什么用呢？我们从full slab中释放一个obj就添加到per cpu partial链表，总不能无限制的添加吧！因此，每次添加的时候都会判断当前的pobjects是否大于kmem\_cache的cpu\_partial成员，如果大于，那么就会将此时per cpu partial链表中所有的slab移送到kmem\_cache\_node的partial链表，然后再将刚刚释放obj的slab插入到per cpu partial链表。如果不大于，则更新pobjects和pages成员，并将slab插入到per cpu partial链表。

3.4. per node partial

per node partia链表类似per cpu partial，区别是node中的slab是所有cpu共享的，而per cpu是每个cpu独占的。假如现在的slab布局如上图所示。假如现在如红色箭头指向的obj将会释放，那么就是一个empty slab，此时判断kmem\_cache\_node的nr\_partial是否大于kmem\_cache的min\_partial，如果大于则会释放该slab的内存。

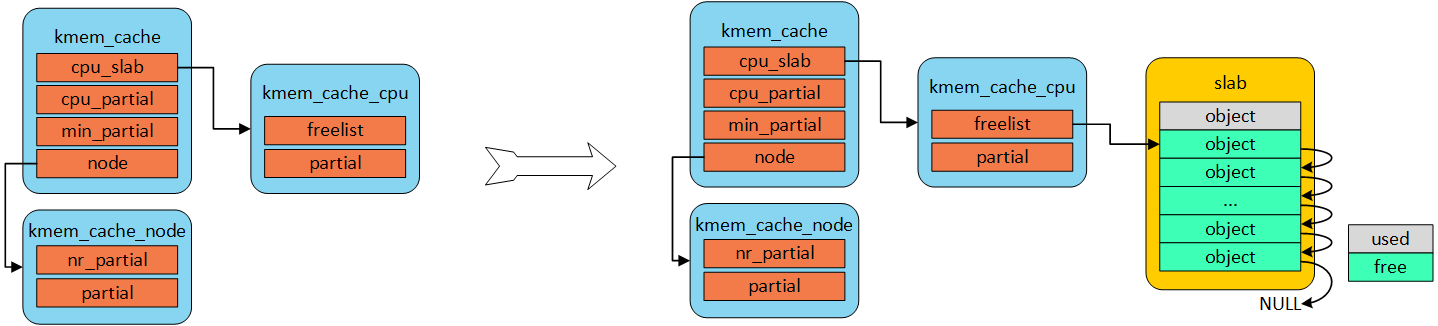
**4. slub分配内存原理**

当调用kmem\_cache\_alloc()分配内存的时候，我们可以从正在使用slab分配，也可以从per cpu partial分配，同样还可以从per node partial分配，那么分配的顺序是什么呢？我们可以用下图表示。

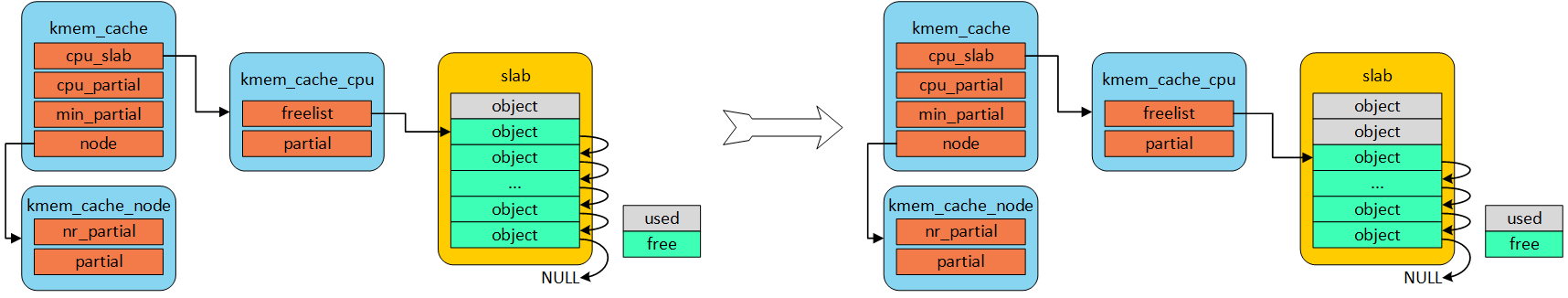


首先从cpu 本地缓存池分配，如果freelist不存在，就会转向per cpu partial分配，如果per cpu partial也没有可用对象，继续查看per node partial，如果很不幸也不没有可用对象的话，就只能从伙伴系统分配一个slab了，并挂入per cpu freelist。我们详细看一下这几种情况。

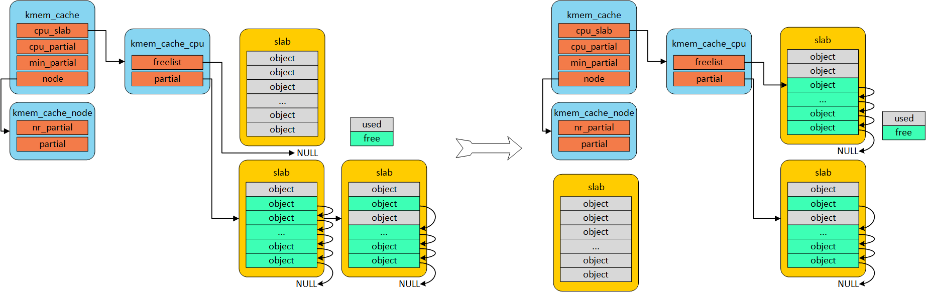
1. kmem\_cache刚刚建立，还没有任何对象可供分配，此时只能从伙伴系统分配一个slab，如下图所示。



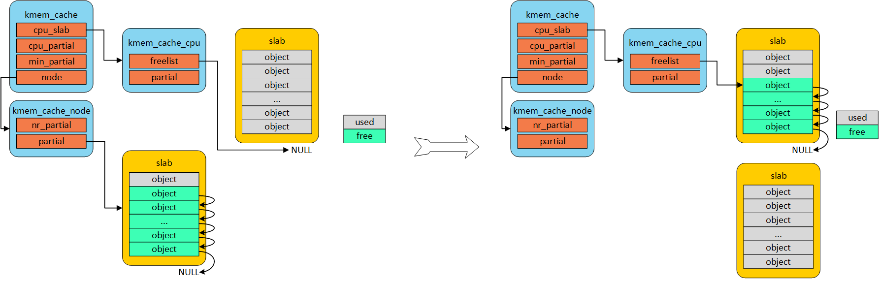
1. 如果正在使用的slab有free obj，那么就直接分配即可，这种是最简单快捷的。如下图所示。



1. 随着正在使用的slab中obj的一个个分配出去，最终会无obj可分配，此时per cpu partial链表中有可用slab用于分配，那么就会从per cpu partial链表中取下一个slab用于分配obj。如下图所示。

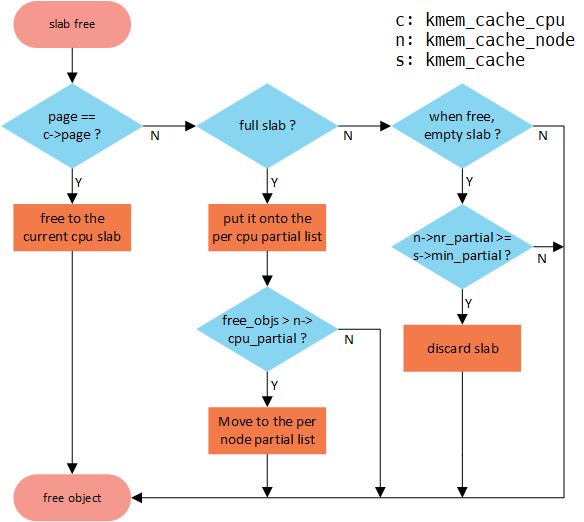


1. 随着正在使用的slab中obj的一个个分配出去，最终会无obj可分配，此时per cpu partial链表也为空，此时发现per node partial链表中有可用slab用于分配，那么就会从per node partial链表中取下一个slab用于分配obj。如下图所示。



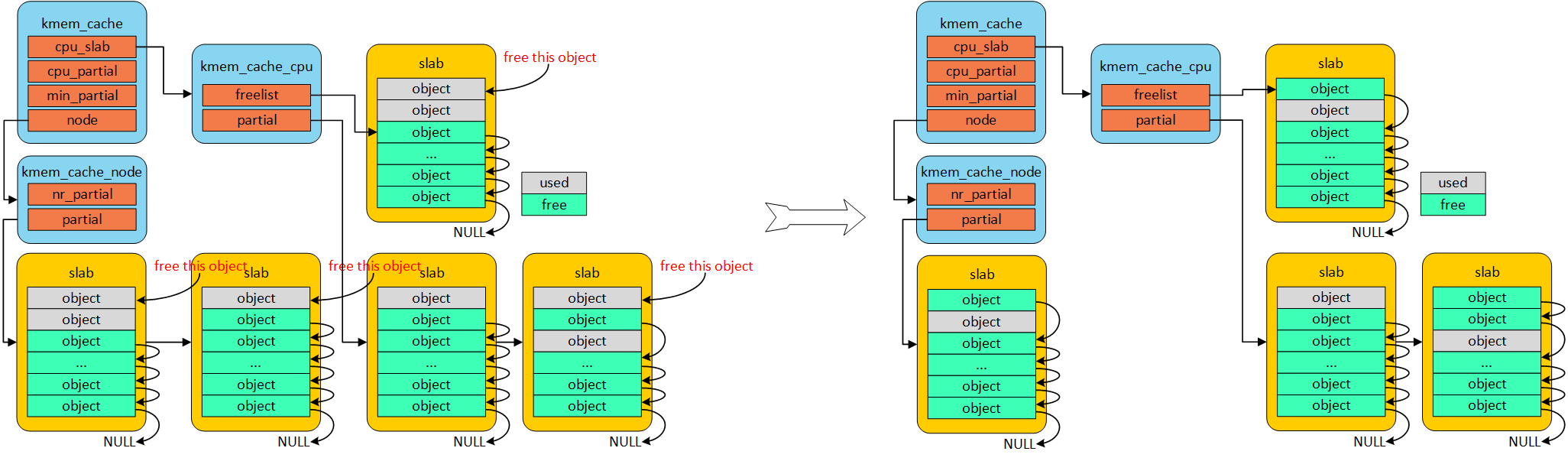
5. slub释放内存原理

我们可以通过kmem\_cache\_free()接口释放申请的obj对象。释放对象的流程如下图所示。

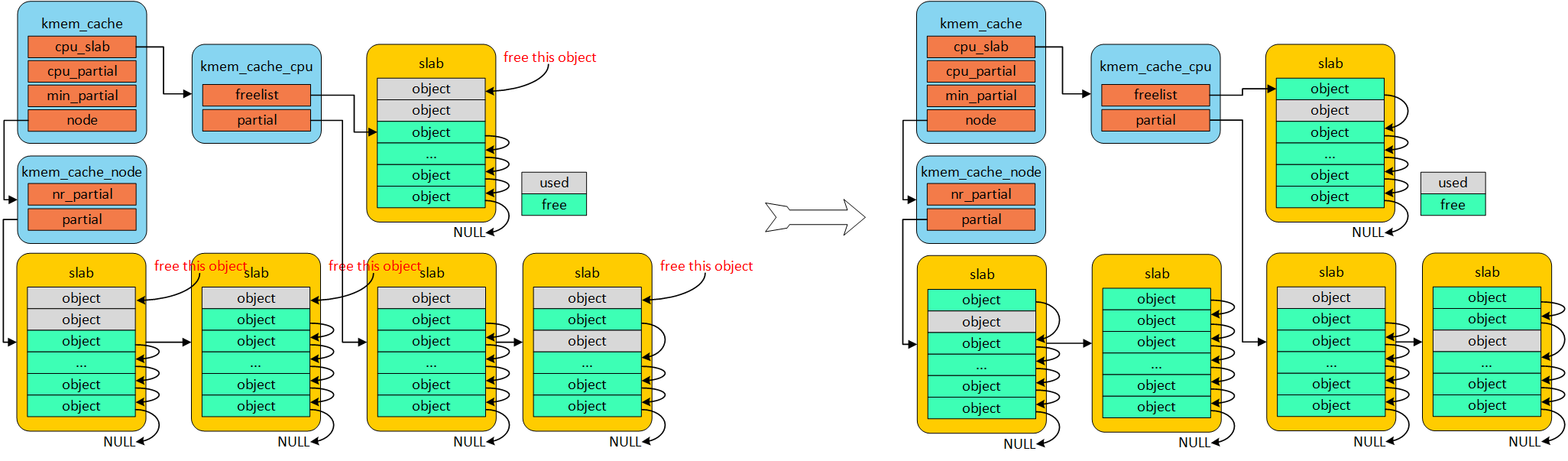


如果释放的obj就是属于正在使用cpu上的slab，那么直接释放即可，非常简单；如果不是的话，首先判断所属slub是不是full状态，因为full slab是没妈的孩子，释放之后就变成partial empty，急需要找个链表领养啊！这个妈就是per cpu partial链表。如果per cpu partial链表管理的所有slab的free object数量超过kmem\_cache的cpu\_partial成员的话，就需要将per cpu partial链表管理的所有slab移动到per node partial链表管理；如果不是full slab的话，继续判断释放当前obj后的slab是否是empty slab，如果是empty slab，那么在满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的情况下，则会释放该slab的内存。其他情况就直接释放即可。

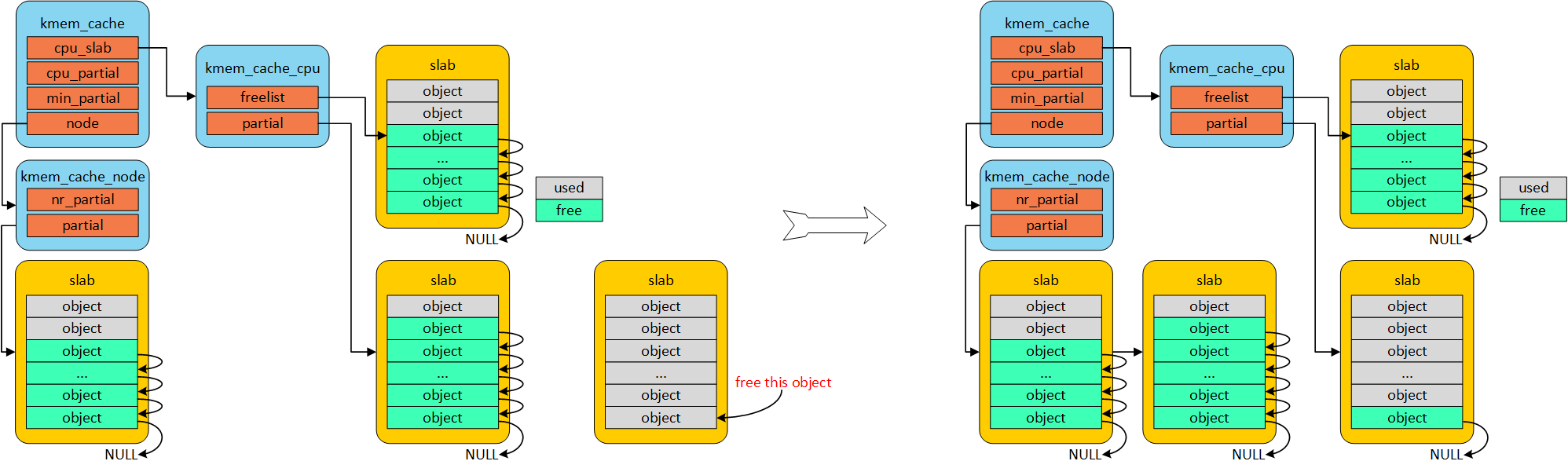
1) 假设下图左边的情况下释放obj，如果满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的话，释放情况如下图所示。



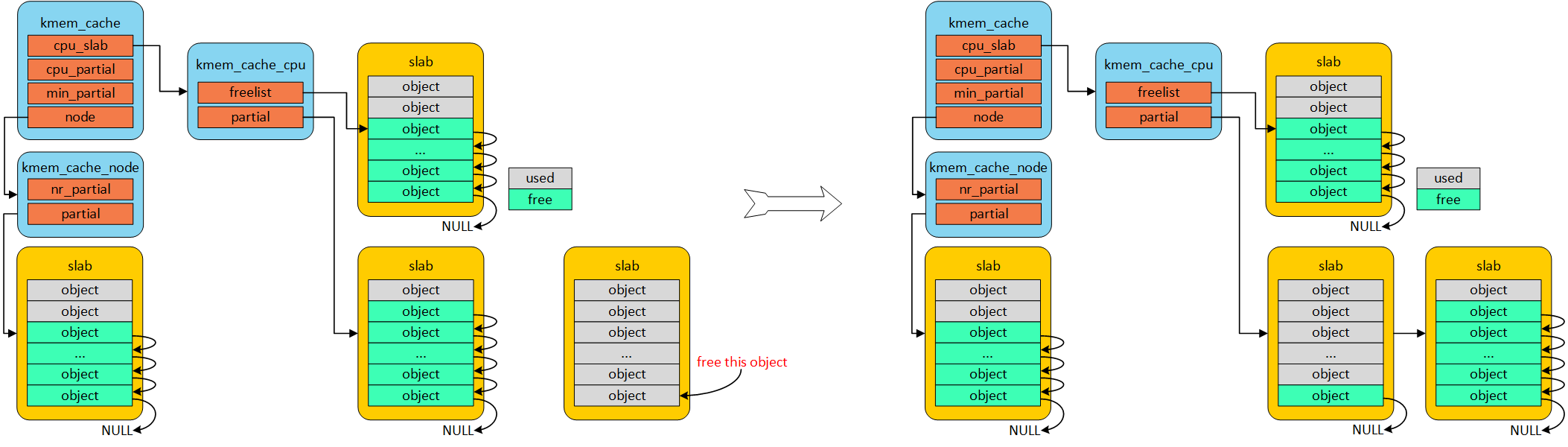
2)假设下图左边的情况下释放obj，如果不满足kmem\_cache\_node的nr\_partial大于kmem\_cache的min\_partial的话，释放情况如下图所示。



3) 假设下图从full slab释放obj的话，如果满足per cpu partial管理的所有slab的free object数量大于kmem\_cache的cpu\_partial成员的话的话，将per cpu partial链表管理的所有slab移动到per node partial链表管理，释放情况如下图所示。



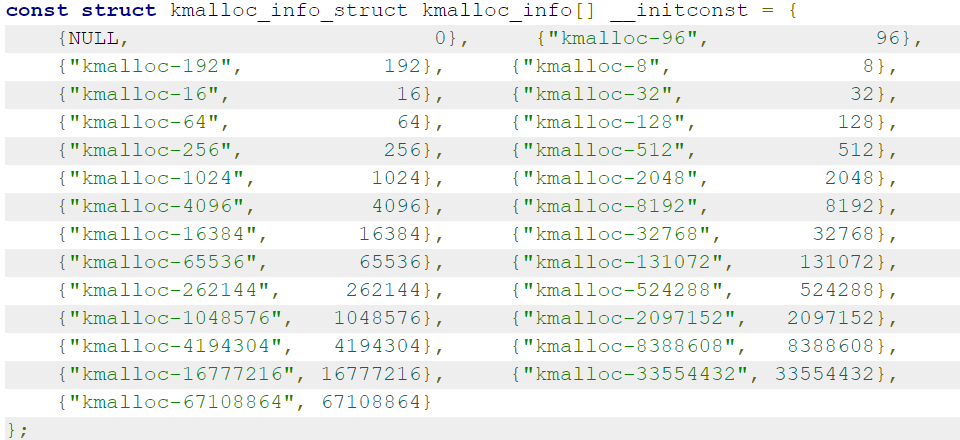
4) 假设下图从full slab释放obj的话，如果不满足per cpu partial管理的所有slab的free object数量大于kmem\_cache的cpu\_partial成员的话的话，释放情况如下图所示。



**6. kmalloc**

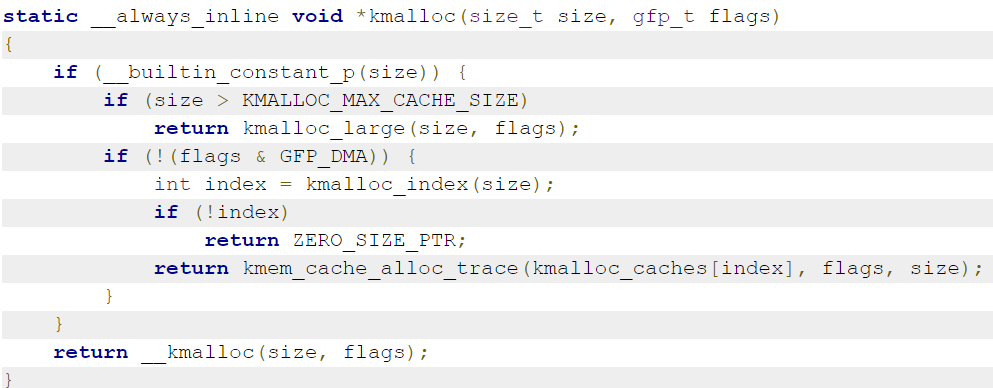
好了，说了这么多，估计你会感觉slab好像跟我们没什么关系。如果作为一个驱动开发者，是不是感觉自己写的driver从来没有使用过这些接口呢？其实我们经常使用，只不过隐藏在kmalloc的面具之下。

kmalloc的内存分配就是基于slab分配器，在系统启动初期调用create\_kmalloc\_caches ()创建多个管理不同大小对象的kmem\_cache，例如：8B、16B、32B、64B、…、64MB等大小。当然默认配置情况下，系统系统启动之后创建的最大size的kmem\_cache是kmalloc-8192。因此，通过slab接口分配的最大内存是8192 bytes。那么通过kmalloc接口申请的内存大于8192 bytes该怎么办呢？其实kmalloc会判断申请的内存是否大于8192 bytes，如果大于的话就会通过alloc\_pages接口申请内存。kmem\_cache的名称以及大小使用struct kmalloc\_info\_struct管理。所有管理不同大小对象的kmem\_cache的名称如下：



经过create\_kmalloc\_caches ()函数之后，系统通过create\_kmalloc\_cache()创建以上不同size的kmem\_cache，并将这些kmem\_cache存储在kmalloc\_caches全局变量中以备后续kmalloc分配内存。现在假如通过kmalloc(17, GFP\_KERNEL)申请内存，系统会从名称“kmalloc-32”管理的slab缓存池中分配一个对象。即使浪费了15Byte。

我们来看看kmalloc的实现方式。



1) \_\_builtin\_constant\_p是gcc工具用来判断参数是否是一个常数，毕竟有些操作对于常数来说是可以优化的。

2) 通过kmalloc\_index函数查找符合满足分配大小的最小kmem\_cache。

3) 将index作为下表从kmalloc\_caches数组中找到符合的kmem\_cache，并从slab缓存池中分配对象。

我们再看一下kmalloc\_index的实现。

