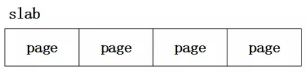
slub allocator的工作流程

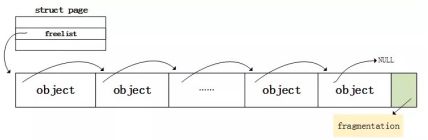
slub allocator类似于我们经常用到的malloc/free函数，主要用于分配管理内核中的小块内存。一块由2^n个连续物理页组成的一块内存称为slab。



上图是一个由四个连续物理页组成的slab。一个slab中包含多个object：

一般来讲，object无法填满slab的整个空间，总会留出一小块内存碎片，称为fragmentation。

为了将slab上的空闲object管理起来，slub allocator采用了一种很巧妙的方式：



如图所示，因为空闲object的内存不被任何人使用，所以其内部可以存放一个next指针，用来指示下一个空闲object，这样就形成了一个空闲object链表。整个slab的相关信息存放在了slab的首个物理页的struct page结构中，struct page结构中的freelist指向了对应slab的首个空闲object。

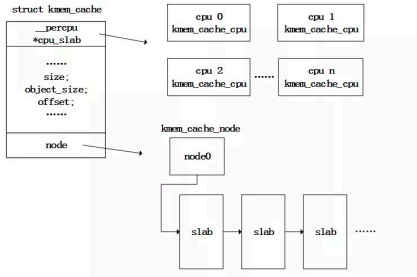
根据可用object的数目，slab可以分为三类：

全空：slab上没有在使用的object，所有object均为空闲object；

半满：slab上既有在使用的object，也有空闲object；

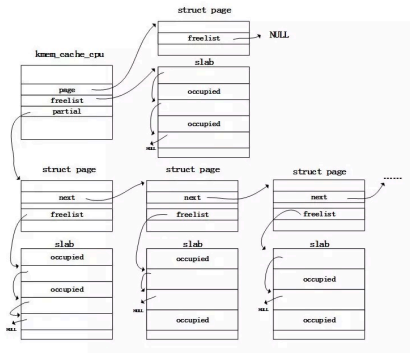
全满：slab上没有空闲object

kmem_cache是slub allocator的核心管理结构。其内部的cpu_slab成员是一个percpu类型的kmem_cache_cpu结构，每一个cpu都对应有一个kmem_cache_cpu。



kmem_cache_cpu是为加快当前cpu分配对象专门设计的。其内部缓存了多个slab。slab可以分为两类，一类是由kmem_cache_cpu的page直接指向的slab，kmem_cache_cpu的freelist则直接指向这个slab里的空闲object；另一类slab则以链表的形式组织起来，形成partial链表。凡是存放在kmem_cache_cpu中的slab，都是仅给当前cpu分配对象使用，所以，这些slab都处于“冻结”状态。

除了kmem_cache_cpu外，还有一个部分是kmem_cache_node。这个部分同样缓存了一些slab，可以供所有cpu使用，并未被冻结



对象的分配：

slub allocator分配一个对象的流程如下：

（1）若当前cpu的kmem_cache_cpu的freelist中有空闲对象，则将freelist头部的空闲对象出链，然后返回此空闲对象，这个执行路径为fast-path；否则，进行以下步骤，也就是slow-path；

（2）若当前cpu的kmem_cache_cpu的page指向的slab的freelist不为空，则将其freelist头部的空闲对象出链，作为返回值。然后将此slab的freelist赋值给kmem_cache_cpu的freelist；否则，执行下一步；

（3）若当前cpu的kmem_cache_cpu的partial中有slab，则将partial链头部的slab出链，kmem_cache_cpu的page将指向此slab，此slab的freelist头部object出链，作为返回值，然后将此slab的freelist赋值给kmem_cache_cpu的freelist；否则，进行下一步；

（4）若kmem_cache的kmem_cache_node中有slab，则将其头部slab出链，kmem_cache_cpu的page将指向此slab，此slab的freelist头部object出链，作为返回值。然后将此slab的freelist赋值给kmem_cache_cpu的freelist，最后从kmem_cache_node中取出足够的slab，放入kmem_cache_cpu的partial链表里进行管理，以加快下次分配对象的速度；否则，进行下一步；

（5）kmem_cache中没有缓存的空闲object，通过伙伴分配系统分配一个新的slab，完成初始化后，kmem_cache_cpu的page将指向此slab，此slab的freelist头部object出链，作为返回值。然后将此slab的freelist赋值给kmem_cache_cpu的freelist。

对象的释放：

处于不同位置的object，其释放过程是不一样的。可以分成以下几种情况：

（1）待释放的object处于当前cpu的kmem_cache_cpu的freelist所在的slab上，则直接将object链入kmem_cache_cpu的freelist上，此执行路径为fast-path。（以下情况均为slow-path）

（2）待释放的object的slab处于当前cpu的kmem_cache_cpu的partial链表或者处于其他cpu的kmem_cache_cpu中（包括freelist所在slab和partial链表），此slab处于被冻结状态，将此object入链到此slab的freelist中，同时更新此slab的struct page结构中的freelist。

（3）待释放的object的slab处于kmem_cache_node的partial链表中，此时的slab有两种情况：

1）释放了object之后，slab依然是一个半满的slab，将object链入slab的freelist中,更新此slab的struct page结构中的freelist;

2）释放了object之后，slab就会变为一个全空的slab，此时，除了将object链入slab的freelist中以外，还需要检查当前kmem_cache_node的partial链表中的slab数目是否超过规定值，若超过，则将此slab从kmem_cache_node的partial链表中移除，然后将此slab交由伙伴分配系统进行回收。

（4）待释放的object的slab是一个未冻结的全满slab，这样的全满slab并不被kmem_cache_cpu或者kmem_cache_node管理。释放其上面的object时，object入链到slab的freelist上，同时更新此slab的struct page结构中的freelist，此后这个slab就会成为一个半满的slab，此slab会被链入到当前cpu的kmem_cache_cpu的partial链表中。在链入到kmem_cache_cpu的partial链表前，会根据当前partial链表中包含的空闲对象的数目做不同处理：

1）若partial链表中包含的空闲对象的数目未超过一个合理值，则直接将slab链入到partial链表中；

2）若partial链表中包含的空闲对象的数目超过了一个合理值，则需要将partial链表中的所有的slab解冻，将所有半满的slab转移到kmem_cache_node中进行管理，将所有的全空slab交由伙伴分配系统进行回收。最后，将此slab链入到kmem_cache_cpu的partial链表中