http://hi.baidu.com/\_kouu/item/4c73532902a05299b73263d0

    Linux内存管理系统是对Numa计算机内存模型的抽象。Linux内存管理系统的设计框架也是把内存分为不同的节点(pglist\_data)，每个CPU对应一个本地节点，每个节点有几个区域（zone）

水平有限，描述不当之处还请之处，转载请注明出处  
blog.csdn.net/lukuen/article/details/6935068[http://blog.csdn.net/vanbreaker/article/details/7694648](http://blog.csdn.net/vanbreaker/article/details/7671618)

       Slab分配器一直处于内核内存管理的核心地位，尽管如此，它还是拥有自身的缺点，最明显的两点就是复杂性和过多的管理数据造成的内存上的开销。针对这些问题，linux引入了slub分配器，slub分配器保留了slab分配器的所有接口，实际上slub分配器的模型和slab分配的模型是基本一致的，只不过在一些地方进行了精简，这也使得slub分配器工作起来更为游刃有余。两者主要的区别如下：

* slab分配器为了增加分配速度，引入了一些管理数组，如slab管理区中的kmem\_bufctl数组和紧随本地CPU结构后面的用来跟踪最热空闲对象的数组，这些结构虽然加快了分配对象的速度，但也增加了一定的复杂性，而且随着系统变得庞大，其对内存的开销也越明显。而slub分配器则完全摒弃了这些管理数据，个人觉得这也是slub分配器最精髓的地方，至于slub分配器的具体做法是怎样的，后面再做分析；
* slab分配器针对每个缓存，根据slab的状态划分了3个链表--full,partial和free. slub分配器做了简化，去掉了free链表，对于空闲的slab,slub分配器选择直接将其释放；
* slub分配器摒弃了slab分配器中的着色概念，在slab分配器中，由于颜色的个数有限，因此着色也无法完全解决slab之间的缓存行冲突问题，考虑到着色造成了内存上的浪费，slub分配器没有引入着色；
* 在NUMA架构的支持上，slub分配器也较slab分配器做了简化。

下面来看slub分配器涉及到的主要数据结构

缓存描述结构:

struct kmem\_cache {

    /\* Used for retriving partial slabs etc \*/

    unsigned long flags; /\* cache属性的描述标识 \*/

    int size;        /\* 分配给对象的内存大小，可能大于实际对象的大小 \*/

    int objsize;         /\* 对象的实际大小 \*/

    int offset;          /\* 存放空闲对象的偏移,以字节为单位 \*/

    struct kmem\_cache\_order\_objects oo;/\* oo用来存放分配给slab的页框的阶数(高16位)和slab中的对象数量(低16位) \*/

    /\*

     \* Avoid an extra cache line for UP, SMP and for the node local to

     \* struct kmem\_cache.

     \*/

    struct kmem\_cache\_node local\_node;/\* 本地节点的slab信息 \*/

    /\* Allocation and freeing of slabs \*/

    struct kmem\_cache\_order\_objects max;

    struct kmem\_cache\_order\_objects min;

    gfp\_t allocflags;      /\* 分配时用的GFP标识 \*/

    int refcount;          /\* 缓存中存在的对象种类数目，因为slub允许缓存复用，

                                  因此一个缓存中可能存在多种对象类型 \*/

    void (\*ctor)(void \*);  /\* 创建对象的构造函数 \*/

    int inuse;             /\* 元数据的偏移 \*/

    int align;         /\* 对齐值 \*/

    unsigned long min\_partial;/\* partial slab链表中的最小slab数目 \*/

    const char \*name;     /\* 缓存名 \*/

    struct list\_head list;    /\* 用于将缓存链入slab\_caches全局缓存链表 \*/

#ifdef CONFIG\_SLUB\_DEBUG

    struct kobject kobj;      /\* For sysfs \*/

#endif

#ifdef CONFIG\_NUMA

    /\*

     \* Defragmentation by allocating from a remote node.

     \*/

    int remote\_node\_defrag\_ratio;   /\* 该值越小，越倾向于从本节点分配对象 \*/

    struct kmem\_cache\_node \*node[MAX\_NUMNODES];/\* NUMA架构下每个节点对应的slab信息 \*/

#endif

#ifdef CONFIG\_SMP

    struct kmem\_cache\_cpu \*cpu\_slab[NR\_CPUS];  /\* SMP系统下每个CPU对应的slab信息 \*/

#else

    struct kmem\_cache\_cpu cpu\_slab; /\* 单核系统下CPU对应的slab信息 \*/

#endif

};

节点的slab信息描述结构：

struct kmem\_cache\_node {

    spinlock\_t list\_lock；       /\* Protect partial list and nr\_partial \*/

    unsigned long nr\_partial;   /\* partial slab链表中slab的数量 \*/

    struct list\_head partial;   /\* partial slab链表表头\*/

    #ifdef CONFIG\_SLUB\_DEBUG

    atomic\_long\_t nr\_slabs;      /\* 节点中的slab数 \*/

    atomic\_long\_t total\_objects; /\* 节点中的对象数 \*/

    struct list\_head full;       /\* full slab链表表头 \*/

    #endif

};

本地CPU的slab信息描述结构:

struct kmem\_cache\_cpu {

    void \*\*freelist;    /\* 指向本地CPU的第一个空闲对象 \*/

    struct page \*page;  /\* 分配给本地CPU的slab的页框 \*/

    int node;           /\* 页框所处的节点，值为-1时表示DEBUG \*/

    unsigned int offset;    /\* 空闲对象指针的偏移，以字长为单位 \*/

    unsigned int objsize;   /\* 对象的大小 \*/

#ifdef CONFIG\_SLUB\_STATS

    unsigned stat[NR\_SLUB\_STAT\_ITEMS];/\*用以记录slab的状态\*/

#endif

};

用下图可以描述这些slub分配器的核心数据结构之间的关系

