SLAB

通过前面所有代码的分析和总结，已经把各个部分熟悉了一遍，在此对 Linux 内核中 slab 机制做最后的总结。

伙伴系统算法采用页作为基本内存区，这适合于大块内存的请求。对于小内存区的申请，比如说几十或几百个字节，我们用 slab 机制。

Slab 分配器把对象分组放进高速缓存。每个高速缓存都是同类型对象的一种“储备”。包含高速缓存的主内存区被划分为多个 slab，每个 slab 由一个活多个连续的页组成，这些页中既包含已分配的对象，也包含空闲的对象。

1，cache 对象管理器

Cache 对象管理器为 kmem_cache 结构，如下：

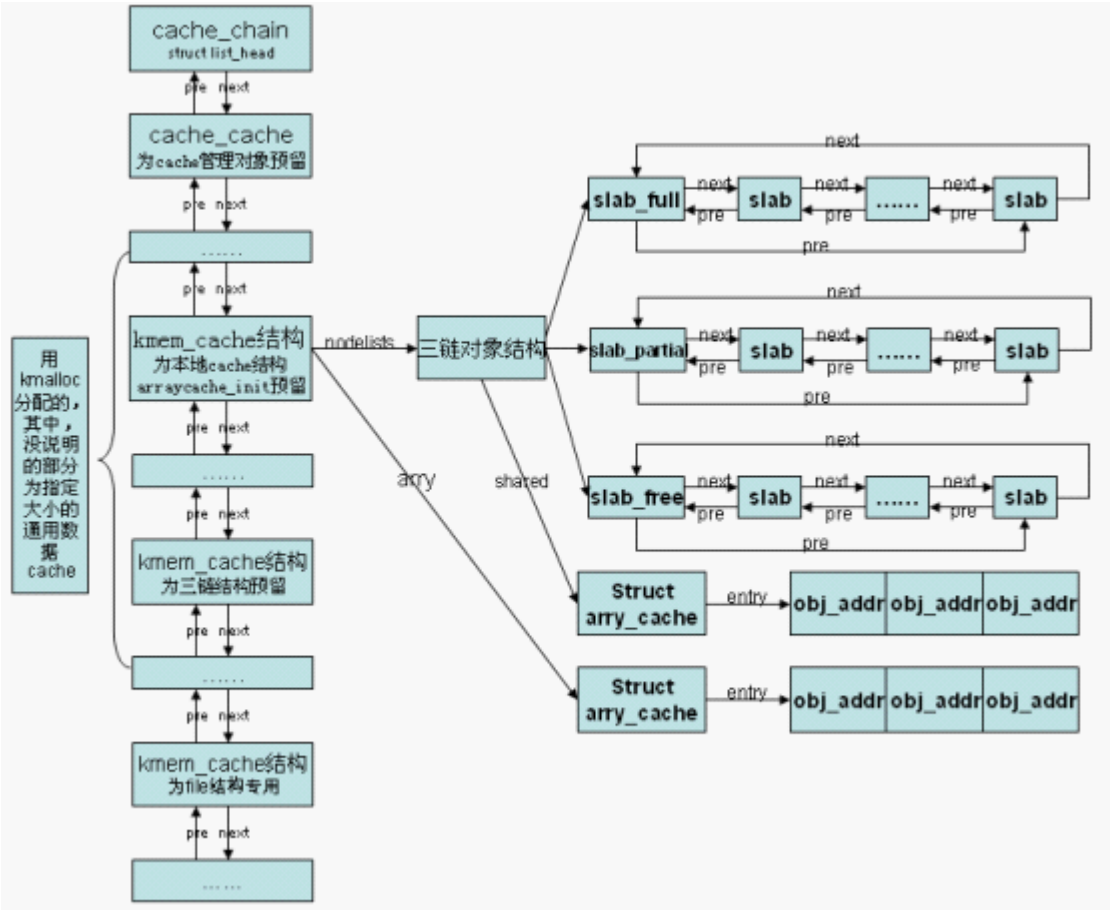
[cpp] view plaincopyprint?

```
/*
 * struct kmem_cache
 *
 * manages a cache.
 */

struct kmem_cache {
/* 1) per-cpu data, touched during every alloc/free */
    struct array_cache *array[NR_CPUS];/*local cache*/
/* 2) Cache tunables. Protected by cache_chain_mutex */
    unsigned int batchcount;
    unsigned int limit;
    unsigned int shared;

    unsigned int buffer_size;/*slab 中对象大小*/
    u32 reciprocal_buffer_size;/*slab 中对象大小的倒数*/
/* 3) touched by every alloc & free from the backend */

    unsigned int flags;     /* constant flags */
    unsigned int num;       /* # of objs per slab */

/* 4) cache_grow/shrink */
```

```
    /* order of pgs per slab (2^n) */
    unsigned int gfporder;

    /* force GFP flags, e.g. GFP_DMA */
    gfp_t gfpflags;

    size_t colour;/*着色块个数*/ /* cache colouring range */
    unsigned int colour_off;/* cache 的着色块的单位大小 */    /* colour offset */
    struct kmem_cache *slabp_cache;
    unsigned int slab_size;/*slab 管理区大小,包含 slab 对象和 kmem_bufctl_t 数组*/
    unsigned int dflags;        /* dynamic flags */

    /* constructor func */
    void (*ctor)(void *obj);

/* 5) cache creation/removal */
    const char *name;
    struct list_head next;

/* 6) statistics */
#ifdef CONFIG_DEBUG_SLAB
    unsigned long num_active;
    unsigned long num_allocations;
    unsigned long high_mark;
    unsigned long grown;
    unsigned long reaped;
    unsigned long errors;
    unsigned long max_freeable;
    unsigned long node_allocs;
    unsigned long node_frees;
    unsigned long node_overflow;
    atomic_t allochit;/*cache 命中计数，在分配中更新*/
    atomic_t allocmiss;/*cache 未命中计数，在分配中更新*/
    atomic_t freehit;
    atomic_t freemiss;

    /*
     * If debugging is enabled, then the allocator can add additional
     * fields and/or padding to every object. buffer_size contains the total
     * object size including these internal fields, the following two
     * variables contain the offset to the user object and its size.
     */
    int obj_offset;
```

```
    int obj_size;
#endif /* CONFIG_DEBUG_SLAB */

    /*
     * We put nodelists[] at the end of kmem_cache, because we want to size
     * this array to nr_node_ids slots instead of MAX_NUMNODES
     * (see kmem_cache_init())
     * We still use [MAX_NUMNODES] and not [1] or [0] because cache_cache
     * is statically defined, so we reserve the max number of nodes.
     */
    struct kmem_list3 *nodelists[MAX_NUMNODES];
    /*
     * Do not add fields after nodelists[]
     */
};
```

在初始化的时候我们看到，为 cache 对象、三链结构、本地 cache 对象预留了三个 cache 共分配。其他为通用数据 cache，整体结构如下图



其中，kmalloc 使用的对象按照大小分属不同的 cache，32、64、128、.....，每种大小对应两个 cache 节点，一个用于 DMA，一个用于普通分配。通过 kmalloc 分配的对象叫作通用数据对象。

可见通用数据 cache 是按照大小进行划分的，结构不同的对象，只要大小在同一个级别内，它们就会在同一个 general cache 中。专用 cache 指系统为特定结构创建的对象，比如 struct file，此类 cache 中的对象来源于同一个结构。

2，slab 对象管理器

Slab 结构如下

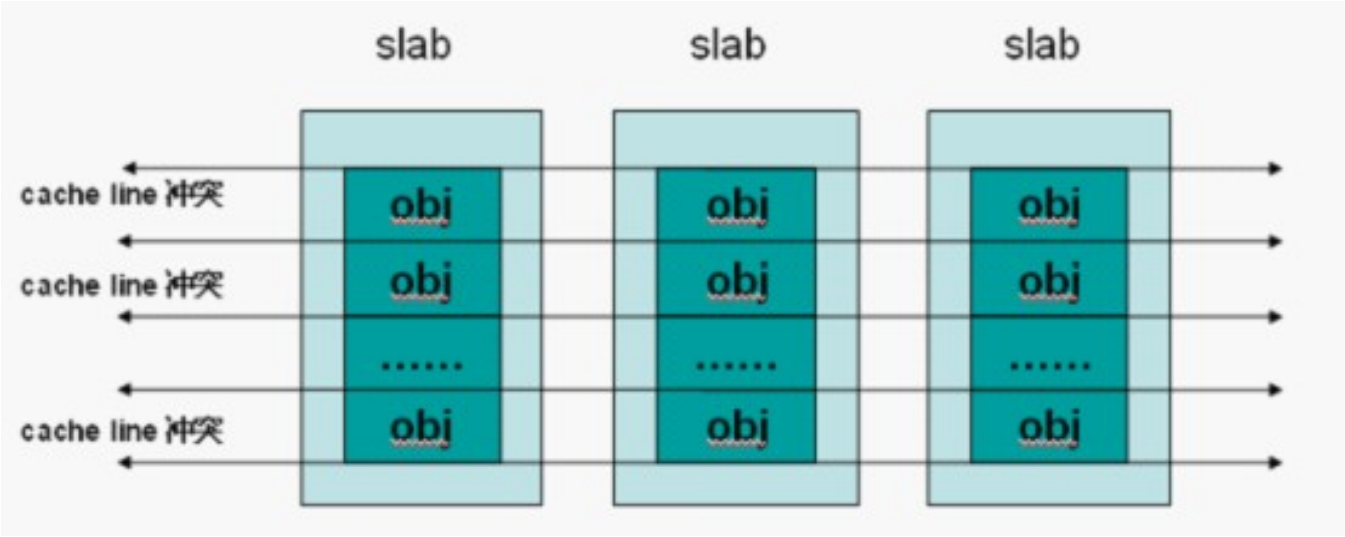
[cpp] view plaincopyprint?

```
/*
 * struct slab
 *
 * Manages the objs in a slab. Placed either at the beginning of mem allocated
 * for a slab, or allocated from an general cache.
 * Slabs are chained into three list: fully used, partial, fully free slabs.
 */
struct slab {
   struct list_head list;
   /* 第一个对象的页内偏移，对于内置式 slab，colouroff 成员不仅包括着色区
    ，还包括管理对象占用的空间
    ，外置式 slab，colouroff 成员只包括着色区。*/
   unsigned long colouroff;
   void *s_mem;/* 第一个对象的虚拟地址 *//* including colour offset */
   unsigned int inuse;/*已分配的对象个数*/ /* num of objs active in slab */
   kmem_bufctl_t free;/* 第一个空闲对象索引*/
   unsigned short nodeid;
};
```

关于 slab 管理对象的整体框架以及 slab 管理对象与对象、页面之间的联系在前面的 slab 创建一文中已经总结的很清楚了。

3，slab 着色

CPU 访问内存时使用哪个 cache line 是通过低地址的若干位确定的，比如 cache line 大小为 32，那么是从 bit5 开始的若干位。因此相距很远的内存地址，如果这些位的地址相同，还是会被映射到同一个 cache line。Slab cache 中存放的是相同大小的对象，如果没有着色区，那么同一个 cache 内，不同 slab 中具有相同 slab 内部偏 移的对象，其低地址的若干位是相同的，映射到同一个 cache line。如图所示。



如此一来，访问 cache line 冲突的对象时，就会出现 cache miss， 不停的在 cache line 和内存之间来回切换，与此同时，其他的 cache line 可能无所事事，严重影响了 cache 的效率。解决这一问题的方法 是通过着色区使对象的 slab 内偏移各不相同，从而避免 cache line 冲突。

着色貌似很好的解决了问题，实质不然，当 slab 数目不多时，着色工作的很好，当 slab 数目很多时，着色发生了循环，仍然存在 cache line 冲突的问题。