Linux 内核中创建 cache 节点由函数 kmem_cache_create()实现。

该函数的执行流程：

1，从全局 cache_cache 中获得 cache 结构，因为全局 cache_cache 初始化对象的大小就是 kmem_cache 结构的大小，所以返回的指针正好可以转换为 cache 结构；调用 kmem_cache_zalloc(&cache_cache, gfp);

2，获得 slab 中碎片大小，由函数 calculate_slab_order()实现；

3，计算并初始化 cache 的各种属性，如果是外置式，需要用 kmem_find_general_cachep(slab_size, 0u)指定 cachep->slabp_cache，用于存放 slab 对象和 kmem_bufctl_t[]数组；

4，设置每个 CPU 上得本地 cache，setup_cpu_cache();

5，cache 创建完毕，将其加入到全局 slab cache 链表中；

一、主实现

[cpp] view plaincopyprint?

```
/**
 * kmem_cache_create - Create a cache.
 * @name: A string which is used in /proc/slabinfo to identify this cache.
 * @size: The size of objects to be created in this cache.
 * @align: The required alignment for the objects.
 * @flags: SLAB flags
 * @ctor: A constructor for the objects.
 *
 * Returns a ptr to the cache on success, NULL on failure.
 * Cannot be called within a int, but can be interrup .
 * The @ctor is run when new pages are allocated by the cache.
 *
 * @name must be valid until the cache is destroyed. This implies that
 * the module calling this has to destroy the cache before getting unloaded.
 * Note that kmem_cache_name() is not guaranteed to return the same pointer,
 * therefore applications must manage it themselves.
 *
 * The flags are
 *
 * %SLAB_POISON - Poison the slab with a known test pattern (a5a5a5a5)
 * to catch references to uninitialised memory.
 *
 * %SLAB_RED_ZONE - Insert `Red' zones around the allocated memory to check
```

```
 * for buffer overruns.
 *
 * %SLAB_HWCACHE_ALIGN - Align the objects in this cache to a hardware
 * cacheline.  This can be beneficial if you're counting cycles as closely
 * as davem.
 */
/*创建 slab 系统顶层的 cache 节点。创建完成后，cache
里并没有任何 slab 以及对象，只有当分配对象
，并且 cache 中没有空闲对象时，才会创建新的 slab。*/
struct kmem_cache *
kmem_cache_create (const char *name, size_t size, size_t align,
    unsigned long flags, void (*ctor)(void *))
{
    size_t left_over, slab_size, ralign;
    struct kmem_cache *cachep = NULL, *pc;
    gfp_t gfp;

    /*
     * Sanity checks... these are all serious usage bugs.
     *//* 安全性检查 */
    if (!name || in_interrupt() || (size < BYTES_PER_WORD) ||
        size > KMALLOC_MAX_SIZE) {
        printk(KERN_ERR "%s: Early error in slab %s\n", __func__,
                name);
        BUG();
    }

    /*
     * We use cache_chain_mutex to ensure a consistent view of
     * cpu_online_mask as well.  Please see cpuup_callback
     */
     /* slab 分配器是否已经初始化好，如果是内核启动阶段
      ，则只有一个 cpu 执行 slab 分配器的初始化动作，无需加锁，否则需要加锁 */
    if (slab_is_available()) {
        get_online_cpus();
        mutex_lock(&cache_chain_mutex);
    }
    /* 遍历 cache 链，做些校验工作 */
    list_for_each_entry(pc, &cache_chain, next) {
        char tmp;
        int res;

        /*
```

```
	 * This happens when the module gets unloaded and doesn't
	 * destroy its slab cache and no-one else reuses the vmalloc
	 * area of the module.  Print a warning.
	 */
	/* 检查 cache 链表中的 cache 是否都有名字 */
	res = probe_kernel_address(pc->name, tmp);
	if (res) {/*没有名字，报错*/
	    printk(KERN_ERR
	           "SLAB: cache with size %d has lost its name\n",
	           pc->buffer_size);
	    continue;
	}
	 /* 检查 cache 链表中是否已经存在相同名字的 cache */
	if (!strcmp(pc->name, name)) {
	    printk(KERN_ERR
	           "kmem_cache_create: duplicate cache %s\n", name);
	    dump_stack();
	    goto oops;
	}
    }

#if DEBUG
    WARN_ON(strchr(name, ' ')); /* It confuses parsers */
#if FORCED_DEBUG
    /*
     * Enable redzoning and last user accounting, except for caches with
     * large objects, if the increased size would increase the object size
     * above the next power of two: caches with object sizes just above a
     * power of two have a significant amount of internal fragmentation.
     */
    if (size < 4096 || fls(size - 1) == fls(size-1 + REDZONE_ALIGN +
                2 * sizeof(unsigned long long)))
        flags |= SLAB_RED_ZONE | SLAB_STORE_USER;
    if (!(flags & SLAB_DESTROY_BY_RCU))
        flags |= SLAB_POISON;
#endif
    if (flags & SLAB_DESTROY_BY_RCU)
        BUG_ON(flags & SLAB_POISON);
#endif
    /*
     * Always checks flags, a caller might be expecting debug support which
     * isn't available.
     */
```

```
BUG_ON(flags & ~CREATE_MASK);

/*
 * Check that size is in terms of words.  This is needed to avoid
 * unaligned accesses for some archs when redzoning is used, and makes
 * sure any on-slab bufctl's are also correctly aligned.
 */
if (size & (BYTES_PER_WORD - 1)) {
    size += (BYTES_PER_WORD - 1);
    size &= ~(BYTES_PER_WORD - 1);
}

/* calculate the final buffer alignment: */

/* 1) arch recommendation: can be overridden for debug */
if (flags & SLAB_HWCACHE_ALIGN) {
    /*
     * Default alignment: as specified by the arch code.  Except if
     * an object is really small, then squeeze multiple objects into
     * one cacheline.
     */
    ralign = cache_line_size();
    while (size <= ralign / 2)
        ralign /= 2;
} else {
    ralign = BYTES_PER_WORD;
}

/*
 * Redzoning and user store require word alignment or possibly larger.
 * Note this will be overridden by architecture or caller mandated
 * alignment if either is greater than BYTES_PER_WORD.
 */
if (flags & SLAB_STORE_USER)
    ralign = BYTES_PER_WORD;

if (flags & SLAB_RED_ZONE) {
    ralign = REDZONE_ALIGN;
    /* If redzoning, ensure that the second redzone is suitably
     * aligned, by adjusting the object size accordingly. */
    size += REDZONE_ALIGN - 1;
    size &= ~(REDZONE_ALIGN - 1);
}
```

```

/* 2) arch mandated alignment */
if (ralign < ARCH_SLAB_MINALIGN) {
    ralign = ARCH_SLAB_MINALIGN;
}
/* 3) caller mandated alignment */
if (ralign < align) {
    ralign = align;
}
/* disable debug if necessary */
if (ralign > __alignof__(unsigned long long))
    flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);
/*
 * 4) Store it.
 */
align = ralign;
/* slab 分配器是否已经可用 */
if (slab_is_available())
    gfp = GFP_KERNEL;
else
    /* slab 初始化好之前，不允许阻塞，且只能在低端内存区分配 */
    gfp = GFP_NOWAIT;

/* Get cache's description obj. */
/* 获得 struct kmem_cache 对象 ,为什么能从 cache 中获得的对象是
kmem_cache 结构呢，因为这里的全局变量 cache_cache 的对象大小
就是 kmem_cache 结构大小*/
cachep = kmem_cache_zalloc(&cache_cache, gfp);
if (!cachep)
    goto oops;

#if DEBUG
cachep->obj_size = size;

/*
 * Both debugging options require word-alignment which is calculated
 * into align above.
 */
if (flags & SLAB_RED_ZONE) {
    /* add space for red zone words */
    cachep->obj_offset += sizeof(unsigned long long);
    size += 2 * sizeof(unsigned long long);
}
```

```
	if (flags & SLAB_STORE_USER) {
		/* user store requires one word storage behind the end of
		 * the real object. But if the second red zone needs to be
		 * aligned to 64 bits, we must allow that much space.
		 */
		if (flags & SLAB_RED_ZONE)
			size += REDZONE_ALIGN;
		else
			size += BYTES_PER_WORD;
	}
#if FORCED_DEBUG && defined(CONFIG_DEBUG_PAGEALLOC)
	if (size >= malloc_sizes[INDEX_L3 + 1].cs_size
		&& cachep->obj_size > cache_line_size() && size < PAGE_SIZE) {
		cachep->obj_offset += PAGE_SIZE - size;
		size = PAGE_SIZE;
	}
#endif
#endif

	/*
	 * Determine if the slab management is 'on' or 'off' slab.
	 * (bootstrapping cannot cope with offslab caches so don't do
	 * it too early on.)
	 */
	/* 确定 slab 管理对象的存储方式：内置还是外置
	。通常，当对象大于等于 512 时，使用外置方式
	。初始化阶段采用内置式。
	slab_early_init 参见 kmem_cache_init 函数 */
	if ((size >= (PAGE_SIZE >> 3)) && !slab_early_init)
		/*
		 * Size is large, assume best to place the slab management obj
		 * off-slab (should allow better packing of objs).
		 */
		flags |= CFLGS_OFF_SLAB;

	size = ALIGN(size, align);
	/* 获得 slab 中碎片的大小 */
	left_over = calculate_slab_order(cachep, size, align, flags);
	/* cachep->num 为该 cache 中每个 slab 的对象数，为 0，表示为该对象创建 cache 失败 */
	if (!cachep->num) {
		printk(KERN_ERR
		       "kmem_cache_create: couldn't create cache %s.\n", name);
		kmem_cache_free(&cache_cache, cachep);
```

```
		cachep = NULL;
		goto oops;
	}
	/* 计算 slab 管理对象的大小，包括 struct slab 对象和 kmem_bufctl_t 数组 */
	slab_size = ALIGN(cachep->num * sizeof(kmem_bufctl_t)
			  + sizeof(struct slab), align);

	/*
	 * If the slab has been placed off-slab, and we have enough space then
	 * move it on-slab. This is at the expense of any extra colouring.
	 */

	/* 如果这是一个外置式 slab，并且碎片大小大于 slab 管理对象的大小
	，则可将 slab 管理对象移到 slab 中，改造成一个内置式 slab */
	if (flags & CFLGS_OFF_SLAB && left_over >= slab_size) {
		/* 除去 off-slab 标志位 */
		flags &= ~CFLGS_OFF_SLAB;
		/* 更新碎片大小 */
		left_over -= slab_size;
	}

	if (flags & CFLGS_OFF_SLAB) {
		/* really off slab. No need for manual alignment */
		/* align 是针对 slab 对象的，如果 slab 管理对象是外置存储
		 ，自然不会像内置那样影响到后面 slab 对象的存储位置
		 ，也就不需要对齐了 */
		slab_size =
		    cachep->num * sizeof(kmem_bufctl_t) + sizeof(struct slab);

#ifdef CONFIG_PAGE_POISONING
		/* If we're going to use the generic kernel_map_pages()
		 * poisoning, then it's going to smash the contents of
		 * the redzone and userword anyhow, so switch them off.
		 */
		if (size % PAGE_SIZE == 0 && flags & SLAB_POISON)
			flags &= ~(SLAB_RED_ZONE | SLAB_STORE_USER);
#endif
	}
	/* cache 的着色块的单位大小 */
	cachep->colour_off = cache_line_size();
	/* Offset must be a multiple of the alignment. */
	/* 着色块大小必须是对象要求对齐方式的倍数 */
	if (cachep->colour_off < align)
```

```
    cachep->colour_off = align;
 /* 计算碎片区需要多少个着色快 */
cachep->colour = left_over / cachep->colour_off;
 /* slab 管理对象的大小 */
cachep->slab_size = slab_size;
cachep->flags = flags;
cachep->gfpflags = 0;
if (CONFIG_ZONE_DMA_FLAG && (flags & SLAB_CACHE_DMA))
    cachep->gfpflags |= GFP_DMA;
/* slab 对象的大小 */
cachep->buffer_size = size;
 /* 计算对象在 slab 中索引时用，参见 obj_to_index 函数 */
cachep->reciprocal_buffer_size = reciprocal_value(size);

if (flags & CFLGS_OFF_SLAB) {
    /* 分配一个 slab 管理区域对象，保存在 slabp_cache 中，
    这个函数传入的大小为 slab_size,也就是分配 slab_size 大小的 cache
    ,在 slab 创建的时候如果是外置式，那么需要从分配的这里面
    分配出 slab 对象，剩下的空间放 kmem_bufctl_t[]数组，
    如果是内置式的 slab，此指针为空 */
    cachep->slabp_cache = kmem_find_general_cachep(slab_size, 0u);
    /*
     * This is a possibility for one of the malloc_sizes caches.
     * But since we go off slab only for object size greater than
     * PAGE_SIZE/8, and malloc_sizes gets created in ascending order,
     * this should not happen at all.
     * But leave a BUG_ON for some lucky dude.
     */
    BUG_ON(ZERO_OR_NULL_PTR(cachep->slabp_cache));
}
cachep->ctor = ctor;
cachep->name = name;
/* 设置每个 cpu 上的 local cache */
if (setup_cpu_cache(cachep, gfp)) {
    __kmem_cache_destroy(cachep);
    cachep = NULL;
    goto oops;
}

/* cache setup completed, link it into the list */
/* cache 创建完毕，将其加入到全局 slab cache 链表中 */
list_add(&cachep->next, &cache_chain);
oops:
```

```
    if (!cachep && (flags & SLAB_PANIC))
        panic("kmem_cache_create(): failed to create slab `%s'\n",
              name);
    if (slab_is_available()) {
        mutex_unlock(&cache_chain_mutex);
        put_online_cpus();
    }
    return cachep;
}
```

其中，cache_cache

[cpp] view plaincopyprint?

```
/* internal cache of cache description objs */
static struct kmem_cache cache_cache = {
    .batchcount = 1,
    .limit = BOOT_CPUCACHE_ENTRIES,
    .shared = 1,
    .buffer_size = sizeof(struct kmem_cache),/*大小为 cache 结构，难怪名称为 cache_cache*/
    .name = "kmem_cache",
};
```

二、计算 slab 碎片大小

[cpp] view plaincopyprint?

```
/**
 * calculate_slab_order - calculate size (page order) of slabs
 * @cachep: pointer to the cache that is being created
 * @size: size of objects to be created in this cache.
 * @align: required alignment for the objects.
 * @flags: slab allocation flags
 *
 * Also calculates the number of objects per slab.
 *
 * This could be made much more intelligent.  For now, try to avoid using
 * high order pages for slabs.  When the gfp() functions are more friendly
 * towards high-order requests, this should be changed.
 */
/*计算 slab 由几个页面组成，同时计算每个 slab 中有多少对象*/
static size_t calculate_slab_order(struct kmem_cache *cachep,
            size_t size, size_t align, unsigned long flags)
{
    unsigned long offslab_limit;
    size_t left_over = 0;
```

```
    int gfporder;

    for (gfporder = 0; gfporder <= KMALLOC_MAX_ORDER; gfporder++) {
        unsigned int num;
        size_t remainder;
        /* 计算 slab 中对象数 */
        cache_estimate(gfporder, size, align, flags, &remainder, &num);
        /* 对象数为 0，表示此 order 下，一个对象都放不下，检查下一 order */
        if (!num)
            continue;

        if (flags & CFLGS_OFF_SLAB) {
            /*
             * Max number of objs-per-slab for caches which
             * use off-slab slabs. Needed to avoid a possible
             * looping condition in cache_grow().
             */
             /* 创建一个外置式 slab 时，要相应分配该 slab 的管理对象
             ，包含 struct slab 对象和 kmem_bufctl_t 数组，分配管理对象的流程就是分配普通对象的流程
             ，再来看一下分配对象的流程：
            kmem_cache_alloc->__cache_alloc-> __do_cache_alloc-> ____cache_alloc-> cache_alloc_refill-> cache_grow-> alloc_slabmgmt-> kmem_cache_alloc_node-> kmem_cache_alloc
            可以看出这里可能存在一个循环，循环的关键在于 alloc_slabmgmt 函数
            ，当 slab 管理对象是 off-slab 方式时，就形成了循环
            。那么什么时候 slab 管理对象会采用外置式 slab 呢？显然当其管理的 slab 中对象很多
            ，从而 kmem_bufctl_t 数组很大，致使整个管理对象也很大，此时才会形成循环
            。故需要对 kmem_bufctl_t 的数目做限制，下面的算法是很粗略的，既然对象大小为 size 时
            ，是外置式 slab，那么我们假设管理对象的大小也是 size，计算出 kmem_bufctl_t 数组的大小
            ，即此大小的 kmem_bufctl_t 数组一定会造成管理对象是外置式 slab。之所以说粗略
            ，是指数组大小小于这个限制时，也不能确保管理对象一定是内置式 slab。但这也不会引发错误
            ，因为还有一个 slab_break_gfp_order 阀门来控制每个 slab 所占页面数，通常其值为 1，即每个 slab 最多两个页面
            ，外置式 slab 存放的都是大于 512 的大对象，所以
            slab 中不会有太多的大对象，kmem_bufctl_t 数组也不会很大，粗略判断一下就足够了。
            */
            offslab_limit = size - sizeof(struct slab);
            offslab_limit /= sizeof(kmem_bufctl_t);
            /* 对象数目大于限制，跳出循环，不再尝试更大的 order
```

```
            ，避免 slab 中对象数目过多
            ，此时计算的对象数也是有效的，循环一次没什么 */
        if (num > offslab_limit)
            break;
    }

    /* Found something acceptable - save it away */
    /* 每个 slab 中的对象数 */
    cachep->num = num;
     /* slab 的 order，即由几个页面组成 */
    cachep->gfporder = gfporder;
     /* slab 中剩余空间（碎片）的大小 */
    left_over = remainder;

    /*
     * A VFS-reclaimable slab tends to have most allocations
     * as GFP_NOFS and we really don't want to have to be allocating
     * higher-order pages when we are unable to shrink dcache.
     */
     /* SLAB_RECLAIM_ACCOUNT 表示此 slab 所占页面为可回收的
      ，当内核检测是否有足够的页面满足用户态的需求时
      ，此类页面将被计算在内，通过调用
     kmem_freepages()函数可以释放分配给 slab 的页框。由于是可回收的
      ，所以不需要做后面的碎片检测了 */
    if (flags & SLAB_RECLAIM_ACCOUNT)
        break;

    /*
     * Large number of objects is good, but very large slabs are
     * currently bad for the gfp()s.
     */
     /* slab_break_gfp_order 为 slab 所占页面的阀门，超过这个阀门时
      ，无论碎片大小，都不再检测更高的 order 了 */
    if (gfporder >= slab_break_gfp_order)
        break;

    /*
     * Acceptable internal fragmentation?
     */
     /* slab 所占页面的大小是碎片大小的 8 倍以上
      ，页面利用率较高，可以接受这样的 order */
    if (left_over * 8 <= (PAGE_SIZE << gfporder))
        break;
```

```
    }
    /* 返回碎片大小 */
    return left_over;
}
```

三、查找指定大小 cache

[cpp] view plaincopyprint?

```
/*在 general cache 中分配一个 struct kmem_cache 对象。直接调用__find_general_cachep。*/

static struct kmem_cache *kmem_find_general_cachep(size_t size, gfp_t gfpflags)
{
    return __find_general_cachep(size, gfpflags);
}
```

[cpp] view plaincopyprint?

```
static inline struct kmem_cache *__find_general_cachep(size_t size,
                            gfp_t gfpflags)
{
    struct cache_sizes *csizep = malloc_sizes;

#if DEBUG
    /* This happens if someone tries to call
     * kmem_cache_create(), or __kmalloc(), before
     * the generic caches are initialized.
     */
    BUG_ON(malloc_sizes[INDEX_AC].cs_cachep == NULL);
#endif
    if (!size)
        return ZERO_SIZE_PTR;
    /* 找到合适的 malloc size */
    while (size > csizep->cs_size)
        csizep++;

    /*
     * Really subtle: The last entry with cs->cs_size==ULONG_MAX
     * has cs_{dma,}cachep==NULL. Thus no special case
     * for large kmalloc calls required.
     */
#ifdef CONFIG_ZONE_DMA
    if (unlikely(gfpflags & GFP_DMA))
        return csizep->cs_dmacachep;
#endif
    /* 返回该大小级别的 cache */
```

```
    return csizep->cs_cachep;
}
```

四、设置 CPU 本地 cache

[cpp] view plaincopyprint?

```
/*配置 local cache 和 slab 三链。*/
static int __init_refok setup_cpu_cache(struct kmem_cache *cachep, gfp_t gfp)
{
    /* general cache 初始化完毕，配置每个 cpu 的 local cache */
    if (g_cpucache_up == FULL)
        return enable_cpucache(cachep, gfp);
    /* 此时处于系统初始化阶段，g_cpucache_up 记录 general cache 初始化的进度
    ，比如 PARTIAL_AC 表示 struct array_cache 所在的 cache 已经创建，
    PARTIAL_L3 表示 struct kmem_list3 所在的 cache 已经创建
    ，注意创建这两个 cache 的先后顺序
    。在初始化阶段只需配置主 cpu 的 local cache 和 slab 三链 */
    if (g_cpucache_up == NONE) {
        /*
         * Note: the first kmem_cache_create must create the cache
         * that's used by kmalloc(24), otherwise the creation of
         * further caches will BUG().
         */
         /* 初始化阶段创建 struct array_cache 所在 cache 时进入这个流程
          ，此时 struct array_cache 所在的 general cache 还未创建
          ，只能使用静态分配的全局变量 initarray_generic 表示的 local cache */
        cachep->array[smp_processor_id()] = &initarray_generic.cache;

        /*
         * If the cache that's used by kmalloc(sizeof(kmem_list3)) is
         * the first cache, then we need to set up all its list3s,
         * otherwise the creation of further caches will BUG().
         */
         /* 创建 struct kmem_list3 所在的 cache 是在 struct array_cache 所在 cache 之后
          ，所以此时 struct kmem_list3 所在的
         cache 也一定没有创建，也需要使用全局变量 */
        set_up_list3s(cachep, SIZE_AC);
        /* 执行到这 struct array_cache 所在的 cache 创建完毕
         ，如果 struct kmem_list3 和 struct array_cache 位于同一个 general cache 中
         ，不会再重复创建了
         ，g_cpucache_up 表示的进度更进一步 */
        if (INDEX_AC == INDEX_L3)
            g_cpucache_up = PARTIAL_L3;
        else
```

```
            g_cpucache_up = PARTIAL_AC;
    } else {
        /* g_cpucache_up 至少为 PARTIAL_AC 时进入这个流程，struct array_cache 所在的
        general cache 已经建立起来，可以通过 kmalloc 分配了 */
        cachep->array[smp_processor_id()] =
            kmalloc(sizeof(struct arraycache_init), gfp);

        if (g_cpucache_up == PARTIAL_AC) {
            /* struct kmem_list3 所在 cache 仍未创建完毕，还需使用全局的 slab 三链 */
            set_up_list3s(cachep, SIZE_L3);
            /* 后面将会分析 kmem_cache_init 函数，只有创建 struct kmem_list3 所在
            cache 时才会进入此流程，上面的代码执行完，struct kmem_list3 所在
            cache 也就创建完毕可以使用了，更新 g_cpucache_up */
            g_cpucache_up = PARTIAL_L3;
        } else {
            int node;
            for_each_online_node(node) {
                cachep->nodelists[node] =/* 通过 kmalloc 分配 struct kmem_list3 对象 */
                    kmalloc_node(sizeof(struct kmem_list3),
                        gfp, node);
                BUG_ON(!cachep->nodelists[node]);
                /* 初始化 slab 三链 */
                kmem_list3_init(cachep->nodelists[node]);
            }
        }
    }
    /* 设置回收时间 */
    cachep->nodelists[numa_node_id()]->next_reap =
            jiffies + REAPTIMEOUT_LIST3 +
            ((unsigned long)cachep) % REAPTIMEOUT_LIST3;

    cpu_cache_get(cachep)->avail = 0;
    cpu_cache_get(cachep)->limit = BOOT_CPUCACHE_ENTRIES;
    cpu_cache_get(cachep)->batchcount = 1;
    cpu_cache_get(cachep)->touched = 0;
    cachep->batchcount = 1;
    cachep->limit = BOOT_CPUCACHE_ENTRIES;
    return 0;
}
```