Linux内核部件分析

设备驱动模型的基石kobject

[日期：2011-10-06] 来源：Linux社区 作者：qb_2008 [字体：大 中 小]

之前我们分析了引用计数kref，总结了sysfs提供的API，并翻译了介绍kobject原理及用法的文档。应该说准备工作做得足够多，kobject的实现怎么都可以看懂了，甚至只需要总结下API就行了。可我还是决定把kobject的实现代码从头分析一遍。一是因为kobject的代码很重要，会在设备驱动模型代码中无数次被用到，如果不熟悉的话可以说是举步维艰。二是为了熟悉linux的编码风格，为以后分析更大规模的代码奠定基础。

kobject的头文件在include/linux/kobject.h，实现在lib/kobject.c。闲话少说，上代码。

```
1. struct kobject {
2.     const char      *name;
3.     struct list_head    entry;
4.     struct kobject      *parent;
5.     struct kset     *kset;
6.     struct kobj_type    *ktype;
7.     struct sysfs_dirent *sd;
8.     struct kref     kref;
9.     unsigned int state_initialized:1;
10.    unsigned int state_in_sysfs:1;
11.    unsigned int state_add_uevent_sent:1;
12.    unsigned int state_remove_uevent_sent:1;
13.    unsigned int uevent_suppress:1;
14. };
```

在struct kobject中，name是名字，entry是用于kobject所属kset下的子kobject链表，parent指向kobject的父节点，kset指向kobject所属的kset，ktype定义了kobject所属的类型，sd指向kobject对应的sysfs目录，kref记录kobject的引用计数，之后是一系列标志。

```
1. struct kobj_type {
2.     void (*release)(struct kobject *kobj);
3.     struct sysfs_ops *sysfs_ops;
4.     struct attribute **default_attrs;
5. };
```

struct kobj_type就是定义了kobject的公共类型，其中既有操作的函数，也有公共的属性。其中release()是在kobject释放时调用的，sysfs_ops中定义了读写属性文件时调用的函数。default_attrs中定义了这类kobject公共的属性。

```
1. struct kset {
2.     struct list_head list;
3.     spinlock_t list_lock;
4.     struct kobject kobj;
5.     struct kset_uevent_ops *uevent_ops;
6. };
```

struct kset可以看成在kobject上的扩展，它包含一个kobject的链表，可以方便地表示sysfs中目录与子目录的关系。其中，list是所属kobject的链表头，list_lock用于在访问链表时加锁，kobj是kset的内部kobject，要表现为sysfs中的目录就必须拥有kobject的功能，最后的kset_uevent_ops定义了对发往用户空间的uevent的处理。我对uevent不了解，会尽量忽略。

```
1. struct kobj_attribute {
2.     struct attribute attr;
3.     ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
4.             char *buf);
5.     ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
6.              const char *buf, size_t count);
7. };
```

struct kobj_attribute是kobject在attribute上做出的扩展，添加了两个专门读写kobject属性的函数。无论是kobject，还是kset（说到底是kset内部的kobject），都提供了使用kobj_attribute的快速创建方法。

结构差不多介绍完了，下面看看实现。我所知道的代码分析风格，喜欢自顶向下的方式，从一个函数开始，介绍出一个函数调用树。在代码量很大，涉及调用层次很深的时候，确实要采用这种打洞的方式来寻找突破口。但这种自顶向下的方式有两个问题：一是很容易迷失，二是代码分析的难度会逐渐增大而不是减小。在茫茫的代码中，你一头下去，周围都是你不认识的函数，一个函数里调用了三个陌生的函数，其中一个陌生的函数又调用了五个更陌生的函数...不久你就会产生很强的挫败感。这就像走在沙漠上，你不知道终点在哪，也许翻过一个沙丘就到了，也许还有无数个沙丘。而且在这种分析时，人是逐渐走向细节，容易被细

节所困扰，忽略了整体的印象与代码的层次感。所以，我觉得在分析代码时，也可以采用自底向上的方式，从细小的、内部使用的函数，到比较宏观的、供外部调用的函数。而且按照这种顺序来看代码，基本就是文件从头读到尾的顺序，也比较符合写代码的流程。linux代码喜欢在文件开始处攒内部静态函数，攒到一定程度爆发，突然实现几个外部API，然后再攒，再实现。而且之前的内部静态函数会反复调用到。linux代码写得很有层次感，除了内外有别，还把意思相近的，或者功能刚好相反的，或者使用时顺序调用的函数放在一起，很便于阅读。闲话少说，等你看完kobject的实现自然就清楚了。

```
1. static int populate_dir(struct kobject *kobj)
2. {
3.     struct kobj_type *t = get_ktype(kobj);
4.     struct attribute *attr;
5.     int error = 0;
6.     int i;
7.
8.     if (t && t->default_attrs) {
9.         for (i = 0; (attr = t->default_attrs[i]) != NULL; i++) {
10.            error = sysfs_create_file(kobj, attr);
11.            if (error)
12.                break;
13.        }
14.    }
15.    return error;
16. }
17.
18. static int create_dir(struct kobject *kobj)
19. {
20.    int error = 0;
21.    if (kobject_name(kobj)) {
22.        error = sysfs_create_dir(kobj);
23.        if (!error) {
24.            error = populate_dir(kobj);
25.            if (error)
26.                sysfs_remove_dir(kobj);
27.        }
28.    }
29.    return error;
30. }
```

create_dir()在sysfs中创建kobj对应的目录，populate_dir()创建kobj中默认属性对应的文件。create_dir()正是调用populate_dir()实现的。

```
1. static int get_kobj_path_length(struct kobject *kobj)
2. {
3.     int length = 1;
4.     struct kobject *parent = kobj;
5.
6.     /* walk up the ancestors until we hit the one pointing to the
7.      * root.
8.      * Add 1 to strlen for leading '/' of each level.
9.      */
10.    do {
11.        if (kobject_name(parent) == NULL)
12.            return 0;
13.        length += strlen(kobject_name(parent)) + 1;
14.        parent = parent->parent;
15.    } while (parent);
16.    return length;
17. }
18.
19. static void fill_kobj_path(struct kobject *kobj, char *path, int length)
20. {
21.    struct kobject *parent;
22.
23.    --length;
24.    for (parent = kobj; parent; parent = parent->parent) {
25.        int cur = strlen(kobject_name(parent));
26.        /* back up enough to print this name with '/' */
27.        length -= cur;
28.        strncpy(path + length, kobject_name(parent), cur);
29.        *(path + --length) = '/';
30.    }
31.
32.    pr_debug("kobject: '%s' (%p): %s: path = '%s'\n", kobject_name(kobj),
33.         kobj, __func__, path);
```

```
34. }
35.
36. /**
37.  * kobject_get_path - generate and return the path associated with a given kobj and kset pair.
38.  *
39.  * @kobj:   kobject in question, with which to build the path
40.  * @gfp_mask:   the allocation type used to allocate the path
41.  *
42.  * The result must be freed by the caller with kfree().
43.  */
44. char *kobject_get_path(struct kobject *kobj, gfp_t gfp_mask)
45. {
46.     char *path;
47.     int len;
48.
49.     len = get_kobj_path_length(kobj);
50.     if (len == 0)
51.         return NULL;
52.     path = kzalloc(len, gfp_mask);
53.     if (!path)
54.         return NULL;
55.     fill_kobj_path(kobj, path, len);
56.
57.     return path;
58. }
```

前面两个是内部函数，get_kobj_path_length()获得kobj路径名的长度，fill_kobj_path()把kobj路径名填充到path缓冲区中。

kobject_get_path()靠两个函数获得kobj的路径名，从攒函数到爆发一气呵成。

```
1. static void kobj_kset_join(struct kobject *kobj)
2. {
3.     if (!kobj->kset)
4.         return;
5.
6.     kset_get(kobj->kset);
7.     spin_lock(&kobj->kset->list_lock);
8.     list_add_tail(&kobj->entry, &kobj->kset->list);
9.     spin_unlock(&kobj->kset->list_lock);
10. }
11.
12. /* remove the kobject from its kset's list */
13. static void kobj_kset_leave(struct kobject *kobj)
14. {
15.     if (!kobj->kset)
16.         return;
17.
18.     spin_lock(&kobj->kset->list_lock);
19.     list_del_init(&kobj->entry);
20.     spin_unlock(&kobj->kset->list_lock);
21.     kset_put(kobj->kset);
22. }
```

kobj_kset_join()把kobj加入kobj->kset的链表中，kobj_kset_leave()把kobj从kobj->kset的链表中去除，两者功能相对。

```
1. static void kobject_init_internal(struct kobject *kobj)
2. {
3.     if (!kobj)
4.         return;
5.     kref_init(&kobj->kref);
6.     INIT_LIST_HEAD(&kobj->entry);
7.     kobj->state_in_sysfs = 0;
8.     kobj->state_add_uevent_sent = 0;
9.     kobj->state_remove_uevent_sent = 0;
10.     kobj->state_initialized = 1;
11. }
12.
13.
14. static int kobject_add_internal(struct kobject *kobj)
15. {
16.     int error = 0;
```

```
17.    struct kobject *parent;
18.
19.    if (!kobj)
20.        return -ENOENT;
21.
22.    if (!kobj->name || !kobj->name[0]) {
23.        WARN(1, "kobject: (%p): attempted to be registered with empty "
24.             "name!\n", kobj);
25.        return -EINVAL;
26.    }
27.
28.    parent = kobject_get(kobj->parent);
29.
30.    /* join kset if set, use it as parent if we do not already have one */
31.    if (kobj->kset) {
32.        if (!parent)
33.            parent = kobject_get(&kobj->kset->kobj);
34.        kobj_kset_join(kobj);
35.        kobj->parent = parent;
36.    }
37.
38.    pr_debug("kobject: '%s' (%p): %s: parent: '%s', set: '%s'\n",
39.         kobject_name(kobj), kobj, __func__,
40.         parent ? kobject_name(parent) : "<NULL>",
41.         kobj->kset ? kobject_name(&kobj->kset->kobj) : "<NULL>");
42.
43.    error = create_dir(kobj);
44.    if (error) {
45.        kobj_kset_leave(kobj);
46.        kobject_put(parent);
47.        kobj->parent = NULL;
48.
49.        /* be noisy on error issues */
50.        if (error == -EEXIST)
51.            printk(KERN_ERR "%s failed for %s with "
52.                   "-EEXIST, don't try to register things with "
53.                   "the same name in the same directory.\n",
54.                   __func__, kobject_name(kobj));
55.        else
56.            printk(KERN_ERR "%s failed for %s (%d)\n",
57.                   __func__, kobject_name(kobj), error);
58.        dump_stack();
59.    } else
60.        kobj->state_in_sysfs = 1;
61.
62.    return error;
63. }
```

kobject_init_internal()初始化kobj。

kobject_add_internal()把kobj加入已有的结构。

这两个函数看似无关，实际很有关系。在kobject中有好几个结构变量，但重要的只有两个，一个是kset，一个是parent。这两个都是表示当前kobject在整个体系中的位置，决不能自行决定，需要外部参与设置。那把kobject创建的过程分为init和add两个阶段也就很好理解了。kobject_init_internal()把一些能自动初始化的结构变量初始化掉，等外界设置了parent和kset，再调用kobject_add_internal()把kobject安在适当的位置，并创建相应的sysfs目录及文件。

```
1. int kobject_set_name_vargs(struct kobject *kobj, const char *fmt,
2.                   va_list vargs)
3. {
4.    const char *old_name = kobj->name;
5.    char *s;
6.
7.    if (kobj->name && !fmt)
8.        return 0;
9.
10.    kobj->name = kvasprintf(GFP_KERNEL, fmt, vargs);
11.    if (!kobj->name)
12.        return -ENOMEM;
13.
14.    /* ewww... some of these buggers have '/' in the name ... */
15.    while ((s = strchr(kobj->name, '/')))
```

```
16.         s[0] = '!';
17.
18.     kfree(old_name);
19.     return 0;
20. }
21.
22. /**
23.  * kobject_set_name - Set the name of a kobject
24.  * @kobj: struct kobject to set the name of
25.  * @fmt: format string used to build the name
26.  *
27.  * This sets the name of the kobject.  If you have already added the
28.  * kobject to the system, you must call kobject_rename() in order to
29.  * change the name of the kobject.
30.  */
31. int kobject_set_name(struct kobject *kobj, const char *fmt, ...)
32. {
33.     va_list vargs;
34.     int retval;
35.
36.     va_start(vargs, fmt);
37.     retval = kobject_set_name_vargs(kobj, fmt, vargs);
38.     va_end(vargs);
39.
40.     return retval;
41. }
```

kobject_set_name()是设置kobj名称的，它又调用kobject_set_name_vargs()实现。但要注意，这个kobject_set_name()仅限于kobject添加到体系之前，因为它只是修改了名字，并未通知用户空间。

```
1. void kobject_init(struct kobject *kobj, struct kobj_type *ktype)
2. {
3.     char *err_str;
4.
5.     if (!kobj) {
6.         err_str = "invalid kobject pointer!";
7.         goto error;
8.     }
9.     if (!ktype) {
10.         err_str = "must have a ktype to be initialized properly!\n";
11.         goto error;
12.     }
13.     if (kobj->state_initialized) {
14.         /* do not error out as sometimes we can recover */
15.         printk(KERN_ERR "kobject (%p): tried to init an initialized "
16.                "object, something is seriously wrong.\n", kobj);
17.         dump_stack();
18.     }
19.
20.     kobject_init_internal(kobj);
21.     kobj->ktype = ktype;
22.     return;
23.
24. error:
25.     printk(KERN_ERR "kobject (%p): %s\n", kobj, err_str);
26.     dump_stack();
27. }
```

kobject_init()就是调用kobject_init_internal()自动初始化了一些结构变量，然后又设置了ktype。其实这个ktype主要是管理一些默认属性什么的，只要在kobject_add_internal()调用create_dir()之前设置就行，之所以会出现在kobject_init()中，完全是为了与后面的kobject_create()相对比。

```
1. static int kobject_add_varg(struct kobject *kobj, struct kobject *parent,
2.                 const char *fmt, va_list vargs)
3. {
4.     int retval;
5.
6.     retval = kobject_set_name_vargs(kobj, fmt, vargs);
7.     if (retval) {
8.         printk(KERN_ERR "kobject: can not set name properly!\n");
9.         return retval;
10.     }
```

```
11.     kobj->parent = parent;
12.     return kobject_add_internal(kobj);
13. }
14.
15. /**
16.  * kobject_add - the main kobject add function
17.  * @kobj: the kobject to add
18.  * @parent: pointer to the parent of the kobject.
19.  * @fmt: format to name the kobject with.
20.  *
21.  * The kobject name is set and added to the kobject hierarchy in this
22.  * function.
23.  *
24.  * If @parent is set, then the parent of the @kobj will be set to it.
25.  * If @parent is NULL, then the parent of the @kobj will be set to the
26.  * kobject associted with the kset assigned to this kobject.  If no kset
27.  * is assigned to the kobject, then the kobject will be located in the
28.  * root of the sysfs tree.
29.  *
30.  * If this function returns an error, kobject_put() must be called to
31.  * properly clean up the memory associated with the object.
32.  * Under no instance should the kobject that is passed to this function
33.  * be directly freed with a call to kfree(), that can leak memory.
34.  *
35.  * Note, no "add" uevent will be created with this call, the caller should set
36.  * up all of the necessary sysfs files for the object and then call
37.  * kobject_uevent() with the UEVENT_ADD parameter to ensure that
38.  * userspace is properly notified of this kobject's creation.
39.  */
40. int kobject_add(struct kobject *kobj, struct kobject *parent,
41.         const char *fmt, ...)
42. {
43.     va_list args;
44.     int retval;
45.
46.     if (!kobj)
47.         return -EINVAL;
48.
49.     if (!kobj->state_initialized) {
50.         printk(KERN_ERR "kobject '%s' (%p): tried to add an "
51.                "uninitialized object, something is seriously wrong.\n",
52.                kobject_name(kobj), kobj);
53.         dump_stack();
54.         return -EINVAL;
55.     }
56.     va_start(args, fmt);
57.     retval = kobject_add_varg(kobj, parent, fmt, args);
58.     va_end(args);
59.
60.     return retval;
61. }
```

kobject_add()把kobj添加到体系中。但它还有一个附加功能，设置kobj的名字。parent也是作为参数传进来的，至于为什么kset没有同样传进来，或许是历史遗留原因吧。

```
1. int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,
2.             struct kobject *parent, const char *fmt, ...)
3. {
4.     va_list args;
5.     int retval;
6.
7.     kobject_init(kobj, ktype);
8.
9.     va_start(args, fmt);
10.     retval = kobject_add_varg(kobj, parent, fmt, args);
11.     va_end(args);
12.
13.     return retval;
14. }
```

kobject_init_and_add()虽然是kobject_init()和kobject_add()的合并，但并不常用，因为其中根本没留下设置kset的空挡，这无疑不太合适。

```
1. int kobject_rename(struct kobject *kobj, const char *new_name)
2. {
3.     int error = 0;
4.     const char *devpath = NULL;
5.     const char *dup_name = NULL, *name;
6.     char *devpath_string = NULL;
7.     char *envp[2];
8.
9.     kobj = kobject_get(kobj);
10.    if (!kobj)
11.        return -EINVAL;
12.    if (!kobj->parent)
13.        return -EINVAL;
14.
15.    devpath = kobject_get_path(kobj, GFP_KERNEL);
16.    if (!devpath) {
17.        error = -ENOMEM;
18.        goto out;
19.    }
20.    devpath_string = kmalloc(strlen(devpath) + 15, GFP_KERNEL);
21.    if (!devpath_string) {
22.        error = -ENOMEM;
23.        goto out;
24.    }
25.    sprintf(devpath_string, "DEVPATH_OLD=%s", devpath);
26.    envp[0] = devpath_string;
27.    envp[1] = NULL;
28.
29.    name = dup_name = kstrdup(new_name, GFP_KERNEL);
30.    if (!name) {
31.        error = -ENOMEM;
32.        goto out;
33.    }
34.
35.    error = sysfs_rename_dir(kobj, new_name);
36.    if (error)
37.        goto out;
38.
39.    /* Install the new kobject name */
40.    dup_name = kobj->name;
41.    kobj->name = name;
42.
43.    /* This function is mostly/only used for network interface.
44.     * Some hotplug package track interfaces by their name and
45.     * therefore want to know when the name is changed by the user. */
46.    kobject_uevent_env(kobj, KOBJ_MOVE, envp);
47.
48. out:
49.    kfree(dup_name);
50.    kfree(devpath_string);
51.    kfree(devpath);
52.    kobject_put(kobj);
53.
54.    return error;
55. }
```

kobject_rename()就是在kobj已经添加到系统之后，要改名字时调用的函数。它除了完成kobject_set_name()的功能，还向用户空间通知这一消息。

```
1. int kobject_move(struct kobject *kobj, struct kobject *new_parent)
2. {
3.     int error;
4.     struct kobject *old_parent;
5.     const char *devpath = NULL;
6.     char *devpath_string = NULL;
7.     char *envp[2];
8.
9.     kobj = kobject_get(kobj);
10.    if (!kobj)
11.        return -EINVAL;
12.    new_parent = kobject_get(new_parent);
```

```
13.     if (!new_parent) {
14.         if (kobj->kset)
15.             new_parent = kobject_get(&kobj->kset->kobj);
16.     }
17.     /* old object path */
18.     devpath = kobject_get_path(kobj, GFP_KERNEL);
19.     if (!devpath) {
20.         error = -ENOMEM;
21.         goto out;
22.     }
23.     devpath_string = kmalloc(strlen(devpath) + 15, GFP_KERNEL);
24.     if (!devpath_string) {
25.         error = -ENOMEM;
26.         goto out;
27.     }
28.     sprintf(devpath_string, "DEVPATH_OLD=%s", devpath);
29.     envp[0] = devpath_string;
30.     envp[1] = NULL;
31.     error = sysfs_move_dir(kobj, new_parent);
32.     if (error)
33.         goto out;
34.     old_parent = kobj->parent;
35.     kobj->parent = new_parent;
36.     new_parent = NULL;
37.     kobject_put(old_parent);
38.     kobject_uevent_env(kobj, KOBJ_MOVE, envp);
39. out:
40.     kobject_put(new_parent);
41.     kobject_put(kobj);
42.     kfree(devpath_string);
43.     kfree(devpath);
44.     return error;
45. }
```

kobject_move()则是在kobj添加到系统后，想移动到新的parent kobject下所调用的函数。在通知用户空间上，与kobject_rename()调用的是同一操作。

```
1. void kobject_del(struct kobject *kobj)
2. {
3.     if (!kobj)
4.         return;
5.
6.     sysfs_remove_dir(kobj);
7.     kobj->state_in_sysfs = 0;
8.     kobj_kset_leave(kobj);
9.     kobject_put(kobj->parent);
10.     kobj->parent = NULL;
11. }
```

kobject_del()仅仅是把kobj从系统中退出，相对于kobject_add()操作。

```
1. /**
2.  * kobject_get - increment refcount for object.
3.  * @kobj: object.
4.  */
5. struct kobject *kobject_get(struct kobject *kobj)
6. {
7.     if (kobj)
8.         kref_get(&kobj->kref);
9.     return kobj;
10. }
11.
12. /*
13.  * kobject_cleanup - free kobject resources.
14.  * @kobj: object to cleanup
15.  */
16. static void kobject_cleanup(struct kobject *kobj)
17. {
18.     struct kobj_type *t = get_ktype(kobj);
19.     const char *name = kobj->name;
20.
21.     pr_debug("kobject: '%s' (%p): %s\n",
```

```
        kobject_name(kobj), kobj, __func__);

    if (t && !t->release)
        pr_debug("kobject: '%s' (%p): does not have a release() "
             "function, it is broken and must be fixed.\n",
             kobject_name(kobj), kobj);

    /* send "remove" if the caller did not do it but sent "add" */
    if (kobj->state_add_uevent_sent && !kobj->state_remove_uevent_sent) {
        pr_debug("kobject: '%s' (%p): auto cleanup 'remove' event\n",
             kobject_name(kobj), kobj);
        kobject_uevent(kobj, KOBJ_REMOVE);
    }

    /* remove from sysfs if the caller did not do it */
    if (kobj->state_in_sysfs) {
        pr_debug("kobject: '%s' (%p): auto cleanup kobject_del\n",
             kobject_name(kobj), kobj);
        kobject_del(kobj);
    }

    if (t && t->release) {
        pr_debug("kobject: '%s' (%p): calling ktype release\n",
             kobject_name(kobj), kobj);
        t->release(kobj);
    }

    /* free name if we allocated it */
    if (name) {
        pr_debug("kobject: '%s': free name\n", name);
        kfree(name);
    }
}

static void kobject_release(struct kref *kref)
{
    kobject_cleanup(container_of(kref, struct kobject, kref));
}

/**
 * kobject_put - decrement refcount for object.
 * @kobj: object.
 *
 * Decrement the refcount, and if 0, call kobject_cleanup().
 */
void kobject_put(struct kobject *kobj)
{
    if (kobj) {
        if (!kobj->state_initialized)
            WARN(1, KERN_WARNING "kobject: '%s' (%p): is not "
                 "initialized, yet kobject_put() is being "
                 "called.\n", kobject_name(kobj), kobj);
        kref_put(&kobj->kref, kobject_release);
    }
}
```

kobject_get()和kobject_put()走的完全是引用计数的路线。kobject_put()会在引用计数降为零时撤销整个kobject的存在：向用户空间发生REMOVE消息，从sysfs中删除相应目录，调用kobj_type中定义的release函数，释放name所占的空间。

看看前面介绍的API。

```
int kobject_set_name(struct kobject *kobj, const char *name, ...)
                __attribute__((format(printf, 2, 3)));
int kobject_set_name_vargs(struct kobject *kobj, const char *fmt,
                  va_list vargs);
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
int __must_check kobject_add(struct kobject *kobj,
                    struct kobject *parent,
                    const char *fmt, ...);
int __must_check kobject_init_and_add(struct kobject *kobj,
                         struct kobj_type *ktype,
```

```
                    struct kobject *parent,
                    const char *fmt, ...);
void kobject_del(struct kobject *kobj);

int __must_check kobject_rename(struct kobject *, const char *new_name);
int __must_check kobject_move(struct kobject *, struct kobject *);

struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);

char *kobject_get_path(struct kobject *kobj, gfp_t flag);
```

基本上概扩了kobject从创建到删除，包括中间改名字，改位置，以及引用计数的变动。

当然，kobject创建仍比较麻烦，因为ktype需要自己写。下面就是kobject提供的一种快速创建方法。

```
static ssize_t kobj_attr_show(struct kobject *kobj, struct attribute *attr,
                              char *buf)
{
    struct kobj_attribute *kattr;
    ssize_t ret = -EIO;

    kattr = container_of(attr, struct kobj_attribute, attr);
    if (kattr->show)
        ret = kattr->show(kobj, kattr, buf);
    return ret;
}

static ssize_t kobj_attr_store(struct kobject *kobj, struct attribute *attr,
                               const char *buf, size_t count)
{
    struct kobj_attribute *kattr;
    ssize_t ret = -EIO;

    kattr = container_of(attr, struct kobj_attribute, attr);
    if (kattr->store)
        ret = kattr->store(kobj, kattr, buf, count);
    return ret;
}

struct sysfs_ops kobj_sysfs_ops = {
    .show   = kobj_attr_show,
    .store  = kobj_attr_store,
};

static void dynamic_kobj_release(struct kobject *kobj)
{
    pr_debug("kobject: (%p): %s\n", kobj, __func__);
    kfree(kobj);
}

static struct kobj_type dynamic_kobj_ktype = {
    .release    = dynamic_kobj_release,
    .sysfs_ops  = &kobj_sysfs_ops,
};
```

这个就是kobject自身提供的一种kobj_type，叫做dynamic_kobj_ktype。它没有提供默认的属性，但提供了release函数及访问属性的方法。

```
struct kobject *kobject_create(void)
{
    struct kobject *kobj;

    kobj = kzalloc(sizeof(*kobj), GFP_KERNEL);
    if (!kobj)
        return NULL;

    kobject_init(kobj, &dynamic_kobj_ktype);
    return kobj;
}

struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
```

```
14. {
15.     struct kobject *kobj;
16.     int retval;
17.
18.     kobj = kobject_create();
19.     if (!kobj)
20.         return NULL;
21.
22.     retval = kobject_add(kobj, parent, "%s", name);
23.     if (retval) {
24.         printk(KERN_WARNING "%s: kobject_add error: %d\n",
25.                __func__, retval);
26.         kobject_put(kobj);
27.         kobj = NULL;
28.     }
29.     return kobj;
30. }
```

在kobject_create()及kobject_create_add()中，使用了这种dynamic_kobj_ktype。这是一种很好的偷懒方法。因为release()函数会释放kobj，所以这里的kobj必须是kobject_create()动态创建的。这里的kobject_create()和kobject_init()相对，kobject_create_and_add()和kobject_init_and_add()相对。值得一提的是，这里用kobject_create()和kobject_create_and_add()创建的kobject无法嵌入其它结构，是独立的存在，所以用到的地方很少。

```
1. void kset_init(struct kset *k)
2. {
3.     kobject_init_internal(&k->kobj);
4.     INIT_LIST_HEAD(&k->list);
5.     spin_lock_init(&k->list_lock);
6. }
```

kset_init()对kset进行初始化。不过它的界限同kobject差不多。

```
1. int kset_register(struct kset *k)
2. {
3.     int err;
4.
5.     if (!k)
6.         return -EINVAL;
7.
8.     kset_init(k);
9.     err = kobject_add_internal(&k->kobj);
10.     if (err)
11.         return err;
12.     kobject_uevent(&k->kobj, KOBJ_ADD);
13.     return 0;
14. }
```

kset_register()最大的特点是简单，它只负责把kset中的kobject连入系统，并发布KOBJ_ADD消息。所以在调用它之前，你要先设置好k->kobj.name、k->kobj.parent、k->kobj.kset。

```
1. void kset_unregister(struct kset *k)
2. {
3.     if (!k)
4.         return;
5.     kobject_put(&k->kobj);
6. }
```

kset_unregister()只是简单地释放创建时获得的引用计数。使用引用计数就是这么简单。

```
1. struct kobject *kset_find_obj(struct kset *kset, const char *name)
2. {
3.     struct kobject *k;
4.     struct kobject *ret = NULL;
5.
6.     spin_lock(&kset->list_lock);
7.     list_for_each_entry(k, &kset->list, entry) {
8.         if (kobject_name(k) && !strcmp(kobject_name(k), name)) {
9.             ret = kobject_get(k);
10.             break;
11.         }
12.     }
13.     spin_unlock(&kset->list_lock);
14.     return ret;
15. }
```

kset_find_obj()从kset的链表中找到名为name的kobject。这纯粹是一个对外的API。

```
1. static void kset_release(struct kobject *kobj)
2. {
3.     struct kset *kset = container_of(kobj, struct kset, kobj);
4.     pr_debug("kobject: '%s' (%p): %s\n",
5.          kobject_name(kobj), kobj, __func__);
6.     kfree(kset);
7. }
8.
9. static struct kobj_type kset_ktype = {
10.     .sysfs_ops  = &kobj_sysfs_ops,
11.     .release = kset_release,
12. };
```

与kobject相对的，kset也提供了一种kobj_type，叫做kset_ktype。

```
1. static struct kset *kset_create(const char *name,
2.                 struct kset_uevent_ops *uevent_ops,
3.                 struct kobject *parent_kobj)
4. {
5.     struct kset *kset;
6.     int retval;
7.
8.     kset = kzalloc(sizeof(*kset), GFP_KERNEL);
9.     if (!kset)
10.         return NULL;
11.     retval = kobject_set_name(&kset->kobj, name);
12.     if (retval) {
13.         kfree(kset);
14.         return NULL;
15.     }
16.     kset->uevent_ops = uevent_ops;
17.     kset->kobj.parent = parent_kobj;
18.
19.     /*
20.      * The kobject of this kset will have a type of kset_ktype and belong to
21.      * no kset itself.  That way we can properly free it when it is
22.      * finished being used.
23.      */
24.     kset->kobj.ktype = &kset_ktype;
25.     kset->kobj.kset = NULL;
26.
27.     return kset;
28. }
29.
30. /**
31.  * kset_create_and_add - create a struct kset dynamically and add it to sysfs
32.  *
33.  * @name: the name for the kset
34.  * @uevent_ops: a struct kset_uevent_ops for the kset
35.  * @parent_kobj: the parent kobject of this kset, if any.
36.  *
37.  * This function creates a kset structure dynamically and registers it
38.  * with sysfs.  When you are finished with this structure, call
39.  * kset_unregister() and the structure will be dynamically freed when it
40.  * is no longer being used.
41.  *
42.  * If the kset was not able to be created, NULL will be returned.
43.  */
44. struct kset *kset_create_and_add(const char *name,
45.                 struct kset_uevent_ops *uevent_ops,
46.                 struct kobject *parent_kobj)
47. {
48.     struct kset *kset;
49.     int error;
50.
51.     kset = kset_create(name, uevent_ops, parent_kobj);
52.     if (!kset)
53.         return NULL;
54.     error = kset_register(kset);
```

```
55.     if (error) {
56.         kfree(kset);
57.         return NULL;
58.     }
59.     return kset;
60. }
```

kset_create()和kset_create_and_add()就是使用kset_type的快速创建函数。

说实话，使用kobject_create_and_add()的比较少见，但使用 kset_create_and_add()的情形还是见过一些的。比如sysfs中那些顶层的目录，就是单纯的目录，不需要嵌入什么很复杂的结构，用简单的kset_create_and_add()创建就好了。

```
1. static inline const char *kobject_name(const struct kobject *kobj)
2. {
3.     return kobj->name;
4. }
5.
6. static inline struct kset *to_kset(struct kobject *kobj)
7. {
8.     return kobj ? container_of(kobj, struct kset, kobj) : NULL;
9. }
10.
11. static inline struct kset *kset_get(struct kset *k)
12. {
13.     return k ? to_kset(kobject_get(&k->kobj)) : NULL;
14. }
15.
16. static inline void kset_put(struct kset *k)
17. {
18.     kobject_put(&k->kobj);
19. }
20.
21. static inline struct kobj_type *get_ktype(struct kobject *kobj)
22. {
23.     return kobj->ktype;
24. }
```

这些是在kobject.h中的内联函数。这里内联函数更多的意思是方便，易于屏蔽内部实现。

以上就是kobject共800余行的代码实现，当然我们忽略了uevent的那部分。

事实证明，自底向上或者顺序的代码分析方法，还是很适合千行左右的代码分析。而且这样分析很全面，容易我们洞察整个模块的意图，从而在理解代码时从较高的抽象角度去看。

上一页 1 2 3 4 5 6 7 8 9 下一页 6

【内容导航】

Linux内核的学习方法 Linux根目录下主要目录功能说明及常用分区方案

相关资讯 Linux内核

Linux内核Git源码树中的代码已达 (今 20:48)
Linux 5.4.7 / 4.19.92 / 4.14.161 (01月01日)
Linux内核将用Rust编程语言编写？ (09/03/2019 12:06:17)
Linux内核将很快默认情况启用"- (05/11/2019 13:43:07)
Linux内核正在努力实现快速高效的I (02/15/2019 14:51:33)
Linux内核的冷热缓存 (01/27/2019 19:10:52)

本文评论 查看全部评论 (5)

表情： 姓名： 匿名 匿名 字数 0