Linux设备模型基础之kobject

Linux内核之旅 2021-10-22 20:57

以下文章来源于CodeTrip ，作者张翔哲

CodeTrip

今天也要加油鸭！

最近看书看到网络设备相关知识，对设备模型不是很了解，这部分网上内容也很多，就相当于做一个知识点整理。本文重点介绍 `kobject` 、 `ktype` 、 `kset` 三个结构。

相关结构体

struct kobject

`kobject`： 组成设备模型基本结构，使所有设备在底层都有同一接口。kobject一般嵌套别的数据结构中，实现了一系列方法，对自身无用但对其他对象非常有效。为一些大的数据结构和子系统提供了基本对象管理，主要功能有：

- 对象引用计数： 跟踪对象存活时间
- 维护对象链表： 主要通过parent字段和kset实现
- sysfs表述： sysfs中显示的每一个对象都对应一个kobject，用来与内核交互,
- 热插拔事件处理： 将产生事件通知用户空间

> include/linux/kobject.h

```
struct kobject {
 const char  *name; /* kobject名称 kobject添加带kernel时该名字为sysfs中名称 */
 struct list_head entry; /* 将Kobject加入到Kset中的list_head */
 struct kobject  *parent;
 struct kset  *kset; /* 该kobject属于的Kset */
```

```
 struct kobj_type *ktype; /* 该Kobject属于的kobj_type。每个Kobject必须有一个ktype */
 struct kernfs_node *sd; /* sysfs directory entry该Kobject在sysfs中的表示 VFS文件系统的目录项，
 struct kref  kref; /* kobject的引用计数，当计数为0时，回调之前注册的release方法释放该对象 */
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
 struct delayed_work release;
#endif
 unsigned int state_initialized:1; /* 初始化标志位，初始化时被置位 */
 unsigned int state_in_sysfs:1;  /* kobject在sysfs中的状态，在目录中创建则为1，否则为0 */
 unsigned int state_add_uevent_sent:1; /* 添加设备的uevent事件是否发送标志，添加设备时向用户空间发
 unsigned int state_remove_uevent_sent:1; /* 删除设备的uevent事件是否发送标志，删除设备时向用户空
 unsigned int uevent_suppress:1; /* 是否忽略上报（不上报uevent）Uevent提供了“用户空间通知”的功能实
 当内核中有Kobject的增加、删除、修改等动作时，会通知用户空间。 */
};
```

struct kobj_type

Kobj_type 代表 Kobject 的属性操作集合,多个 Kobject 可能共用同一个属性操作集，因此把 Kobj_type 独立出来， kobj_type 关心的是对象的类型。

同时 kobj_type 中指定了删除 kobject 时要调用的 release 函数， kobject 结构体中有 struct kref 字段用于对 kobject 进行引用计数，当计数值为0时，就会调用 kobj_type 中的 release 函数对 kobject 进行释放。

> include/linux/kobject.h

```
struct kobj_type {
 void (*release)(struct kobject *kobj);
 const struct sysfs_ops *sysfs_ops;//Kobject的sysfs文件系统接口
 struct attribute **default_attrs;//Kobject的atrribute列表（所谓attribute，就是sysfs文件系统中的
 const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
 const void *(*namespace)(struct kobject *kobj);
};
```

struct sysfs_ops

sysfs 操作表包括 store 和 show 两个函数，在用户态读取数据时，调用 show 函数将指定属性值存入 buffer 中返回给用户态，与之相反 store 存储用户态传入的属性值。

在 /sys 文件系统中，通过 echo/cat 的操作，最终会调用到 show/store 函数，而这两个函数的具体实现可以放置到驱动程序中；

> include/linux/sysfs.h

```
struct sysfs_ops {
 ssize_t (*show)(struct kobject *, struct attribute *, char *);
 ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

struct attribute

创建 kobject 时会给每个 kobject 西医猎魔人属性，保存在 ktype 中的 attribute 结构中。

> include/linux/sysfs.h

```
struct attribute {
 const char  *name;//文件名
 umode_t   mode;// 文件模式
#ifdef CONFIG_DEBUG_LOCK_ALLOC
 bool   ignore_lockdep:1;
 struct lock_class_key *key;
 struct lock_class_key skey;
#endif
};
```

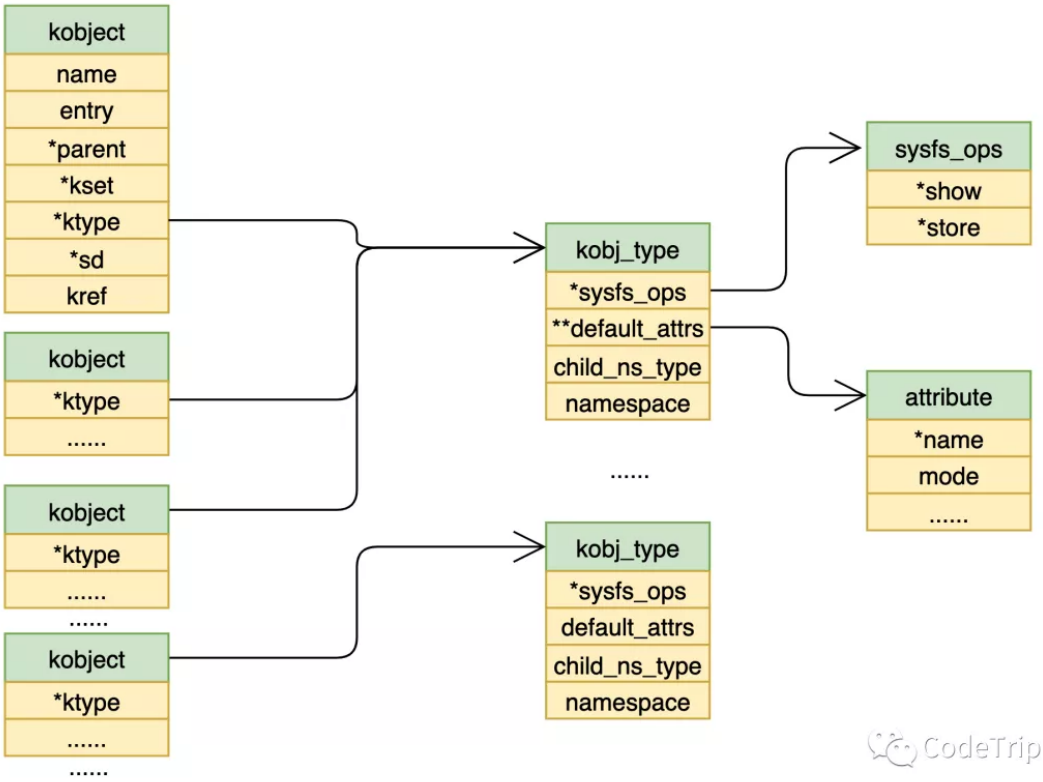

图1 kobjet_type

struct kset

kset 是嵌入 kobj_type 的 kobject 的集合， kset 关心对象的聚集与集合。

Kset 是一个特殊的 Kobject ， kset 总在 /sysfs 中出现，设置并向系统添加了 kset ，将在/sysfs中创建一个目录,比如 /sys/bus 就是一个 kset 对象。

创建添加一个对象时，通常将 kobject 添加到 kset 中，之后 kobject_add_internal 函数会讲解。

> include/linux/kobject.h

```
struct kset {
 struct list_head list;/* 包含在kset内的所有kobject构成一个双向链表 */
 spinlock_t list_lock;
 struct kobject kobj;/* 该kset自己的kobject 归属于该kset的所有的kobject的共有parent */
 /*当任何Kobject需要上报uevent时，都要调用它所从属的kset的uevent_ops，添加环境变量，
 或者过滤event（kset可以决定哪些event可以上报）。因此，如果一个kobject不属于任何kset时，
 是不允许发送uevent的。*/
 const struct kset_uevent_ops *uevent_ops;// 该kset的uevent操作函数集
} __randomize_layout;
```

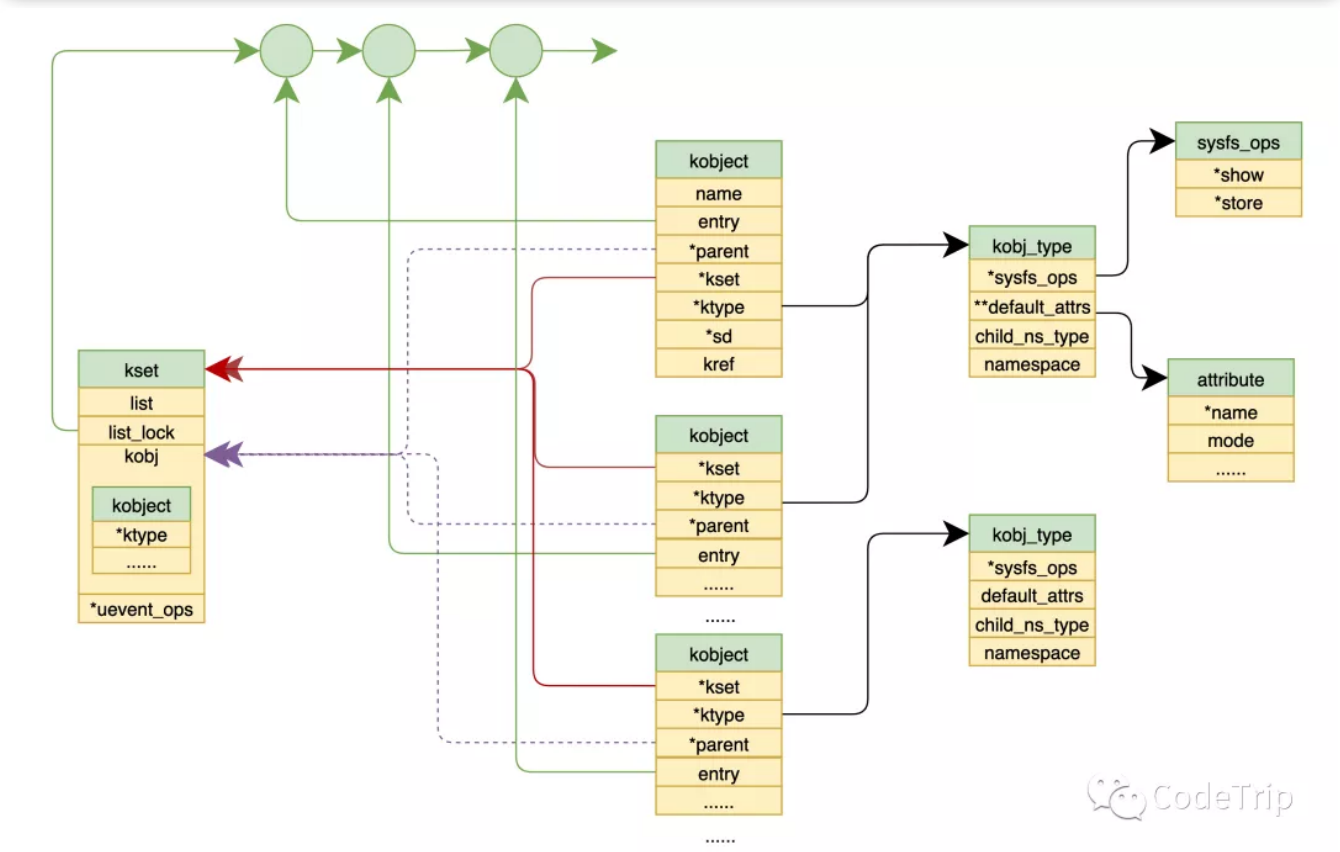

图2 kset

`kset` 即是一个 `kobject` 集合，其本身又可作为一个 `kobject` 加入到别的 `kset` 中，如下图所示。下图所显示目录结构为：

```
/A
  /D
  /E
/B
/C
```

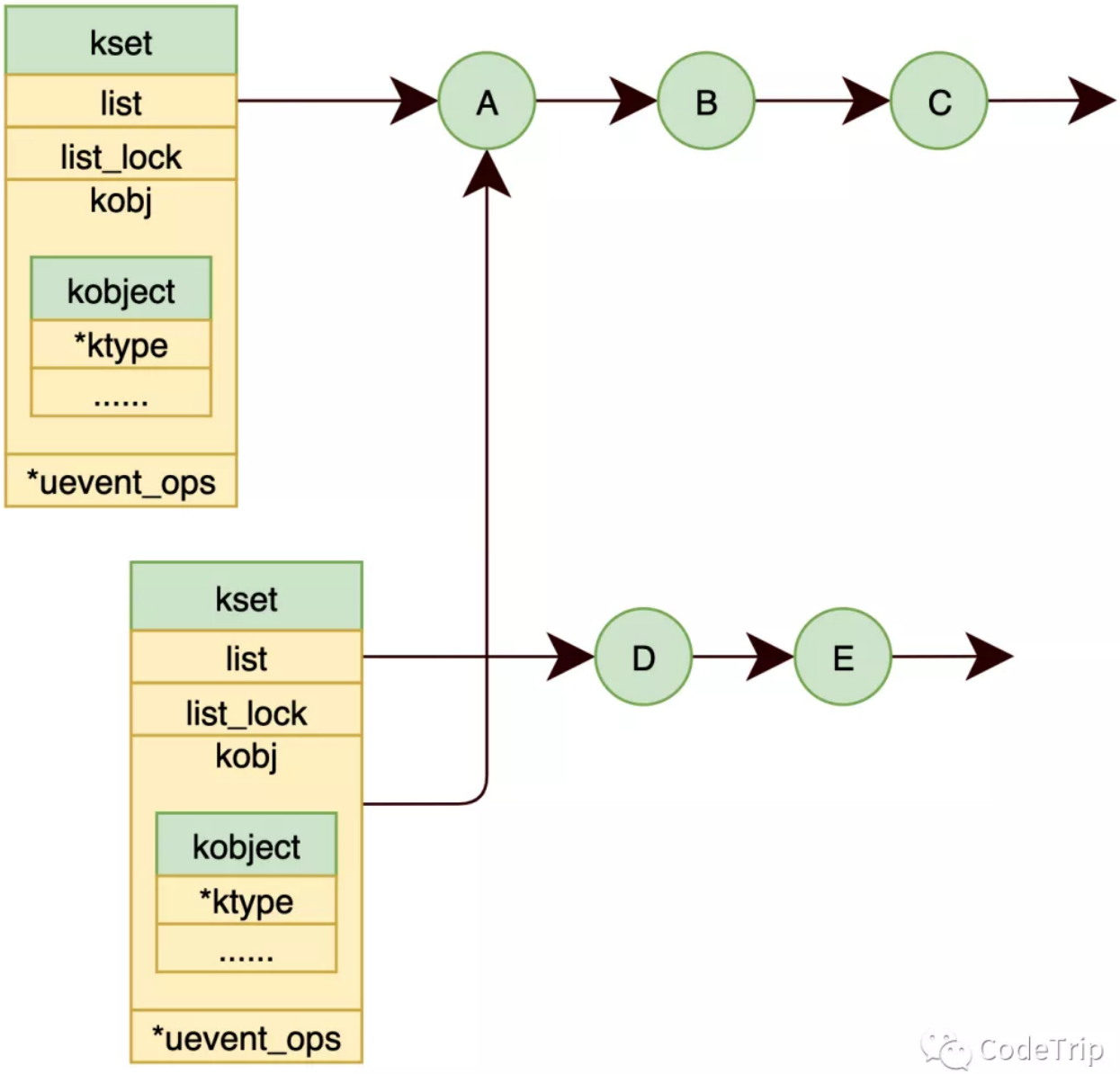

图3 kset层次结构

kobject操作

kobject创建

创建 `kobject` 结构时需要将整个 `kobject` 清0，否则会发生一些奇怪的错误。

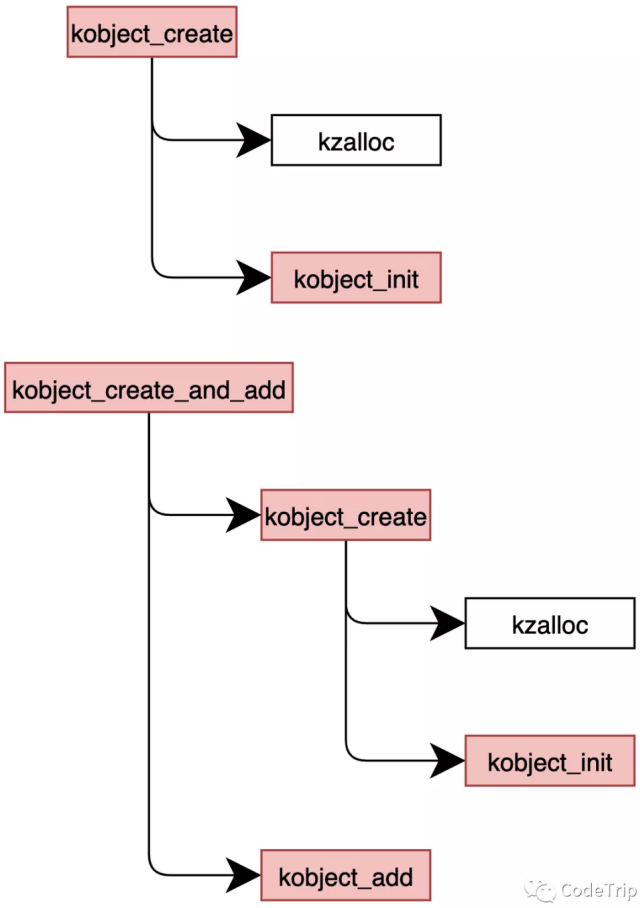

图4 kobject创建函数结构

kobject_create

使用 `kobject_create` 创建：首先调用 `kzalloc` 创建一个 `kobject` 对象并清零，接着调用 `kobject_init` (后面会介绍)对该 `kobject` 对象进行初始化。

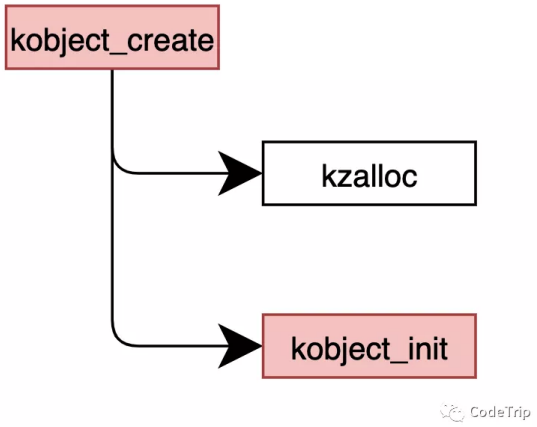

图5 kobject_create函数结构

> lib/kobject.c

```
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;
}
```

kobject_create_and_add

使用 kobject_create_and_add 创建：封装了 kobject_create 函数，完成创建初始化后调用 kobject_add 将初始化完成的 kobject 添加到内核中。

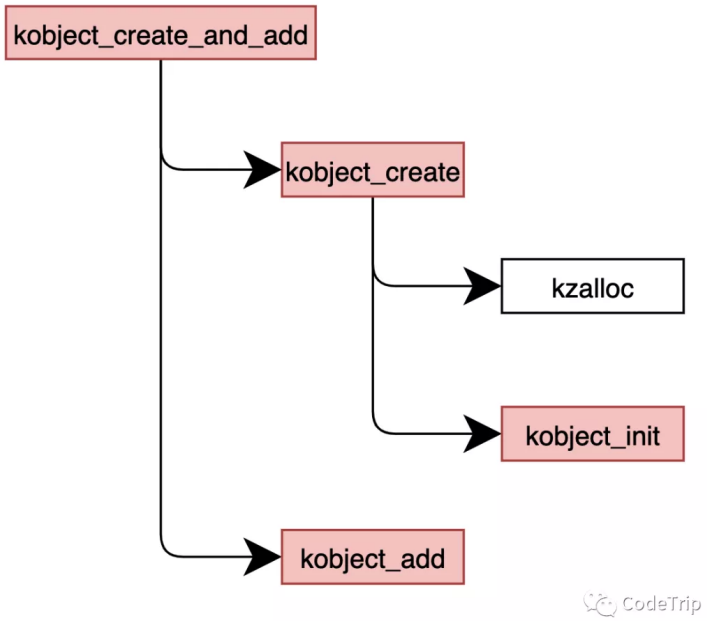

图6 kobject_create_and_add函数结构

> lib/kobject.c

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent)
{
 struct kobject *kobj;
 int retval;

 kobj = kobject_create();
 if (!kobj)
  return NULL;

 retval = kobject_add(kobj, parent, "%s", name);
 if (retval) {
  printk(KERN_WARNING "%s: kobject_add error: %d\n",
         __func__, retval);
  kobject_put(kobj);
  kobj = NULL;
 }
 return kobj;
}
EXPORT_SYMBOL_GPL(kobject_create_and_add);
```

kobject初始化

kobject_init

使用 `kobject_init` 初始化 `kobject` ： 首先确认 `kobj` 和 `ktype` 都不为空，然后通过 `kobj->state_initialized` 判断是否被初始化过，最后调用 `kobject_init_internal` 函数对 `kobject` 参数进行初始化并将 `ktype` 加入到 `kobject` 中。

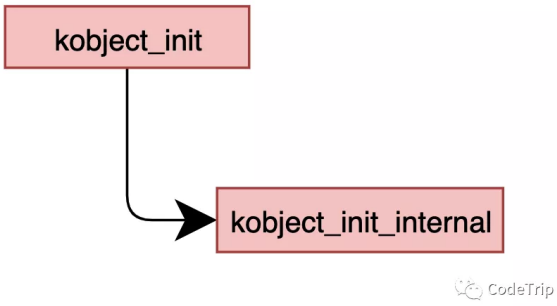

图6 kobject_init函数结构

> lib/kobject.c

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype)
{
     char *err_str;
     // 确认kobj、ktype都不为空
     if (!kobj) {
      err_str = "invalid kobject pointer!";
      goto error;
     }
     if (!ktype) {
      err_str = "must have a ktype to be initialized properly!\n";
      goto error;
     }
      // 若该指针已经初始化过，打印错误信息及堆栈信息
     if (kobj->state_initialized) {
      /* do not error out as sometimes we can recover */
      printk(KERN_ERR "kobject (%p): tried to init an initialized "
             "object, something is seriously wrong.\n", kobj);
      dump_stack();
     }
     // 初始化kobj内部参数，包括引用计数、list、各种指标等
     kobject_init_internal(kobj);
     // 将ktype指针赋予kobj
     kobj->ktype = ktype;
     return;

    error:
     printk(KERN_ERR "kobject (%p): %s\n", kobj, err_str);
     dump_stack();
}
EXPORT_SYMBOL(kobject_init);
```

kobject_init_internal

`kobject_init_internal` 初始化 `kobject` 内部参数:

> lib/kobject.c

```
static void kobject_init_internal(struct kobject *kobj)
{
     if (!kobj)
```

```
        return;
    kref_init(&kobj->kref);
    INIT_LIST_HEAD(&kobj->entry);
    kobj->state_in_sysfs = 0;
    kobj->state_add_uevent_sent = 0;
    kobj->state_remove_uevent_sent = 0;
    kobj->state_initialized = 1;
}
```

kobject添加

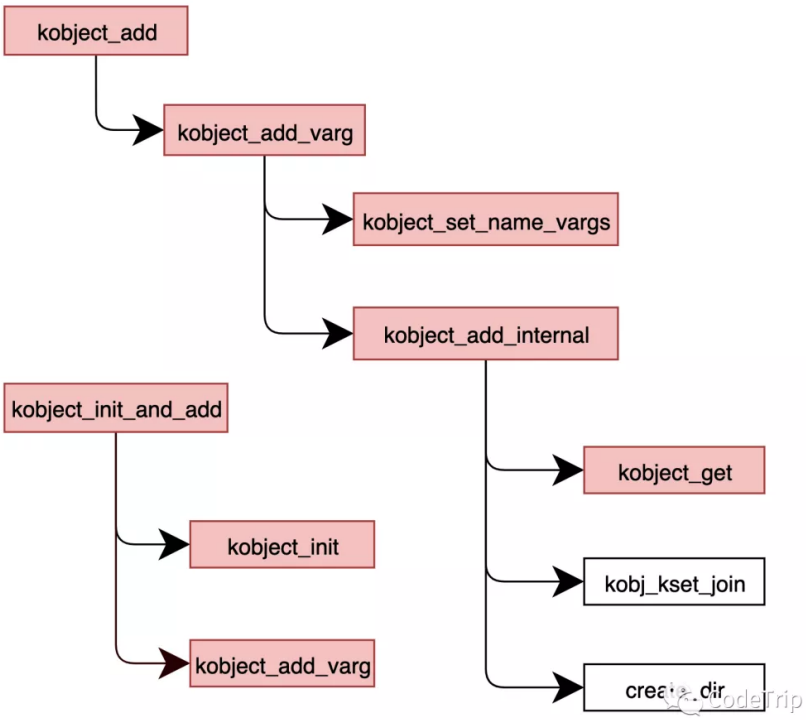

图7 kobject添加函数结构

kobject_add

`kobject_add` 将初始化完成的kobject添加到kernel中:该函数主要做一些kobject的判断，真正的操作在 `kobject_add_varg` 中。

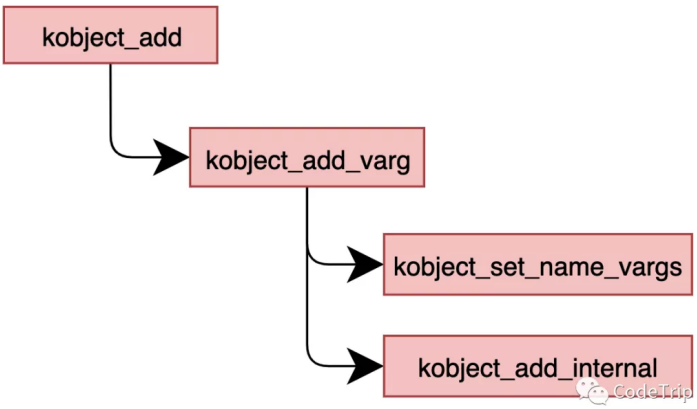

图8 kobjec_add函数结构

> lib/kobject.c

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
  const char *fmt, ...)
{
 va_list args;
 int retval;
 // 确认kboj不为空
 if (!kobj)
  return -EINVAL;
 if (!kobj->state_initialized) {
  printk(KERN_ERR "kobject '%s' (%p): tried to add an "
         "uninitialized object, something is seriously wrong.\n",
         kobject_name(kobj), kobj);
  dump_stack();
  return -EINVAL;
 }
 va_start(args, fmt);
 retval = kobject_add_varg(kobj, parent, fmt, args);
 va_end(args);

 return retval;
}
EXPORT_SYMBOL(kobject_add);
```

kobject_add_varg

`kobject_add_varg` 函数：这还不是真正的操作函数。首先调用 `kobject_set_name_vargs` 函数将字符串赋予 `kobject->name` ,再调用 `kobject_add_internal` 将 `kobject` 加入到 `kernel` 中

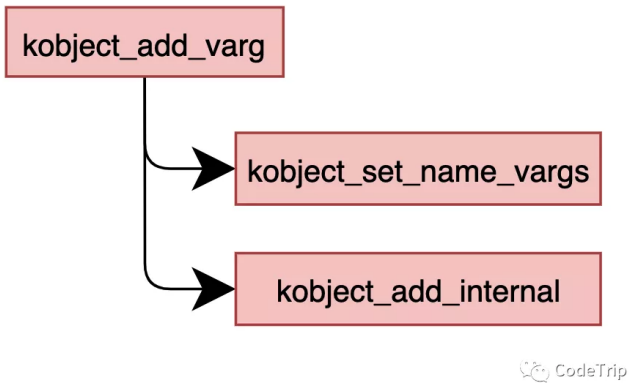

图9 kobjec_add_varg函数结构

> lib/kobject.c

```
static __printf(3, 0) int kobject_add_varg(struct kobject *kobj,
        struct kobject *parent,
        const char *fmt, va_list vargs)
{
 int retval;
 retval = kobject_set_name_vargs(kobj, fmt, vargs);
 if (retval) {
  printk(KERN_ERR "kobject: can not set name properly!\n");
  return retval;
 }
 kobj->parent = parent;
 return kobject_add_internal(kobj);
}
```

kobject_set_name_vargs

`kobject_set_name_vargs` 函数：将字符串赋予 `kobject->name`

> lib/kobject.c

```
int kobject_set_name_vargs(struct kobject *kobj, const char *fmt,
      va_list vargs)
{
 const char *s;

 if (kobj->name && !fmt)
  return 0;

 s = kvasprintf_const(GFP_KERNEL, fmt, vargs);
 if (!s)
  return -ENOMEM;

 /*
  * ewww... some of these buggers have '/' in the name ... If
  * that's the case, we need to make sure we have an actual
  * allocated copy to modify, since kvasprintf_const may have
  * returned something from .rodata.
  */
 if (strchr(s, '/')) {
  char *t;

  t = kstrdup(s, GFP_KERNEL);
  kfree_const(s);
  if (!t)
   return -ENOMEM;
  strreplace(t, '/', '!');
  s = t;
 }
 kfree_const(kobj->name);
 kobj->name = s;

 return 0;
}
```

kobject_add_internal

kobject_add_internal :将 kobject 添加到 kernel 。首先判断 kobject 的合法性，检验通过后通过调用 kobject_get 函数增加 kobject 父 kobj 的引用计数，若存在kset，则通过 kobj_kset_join 函数将 kobject 加入 kset 。注意这里做了一个判断，若 kobject 没有 父kobject ，则将 kobj->parant 指向 kset ，增加 kset 引用计数。最后调用 create_dir 函数调用 sysfs 相关接口函数，在 /sysfs 下创建该 kobject 对应目录。

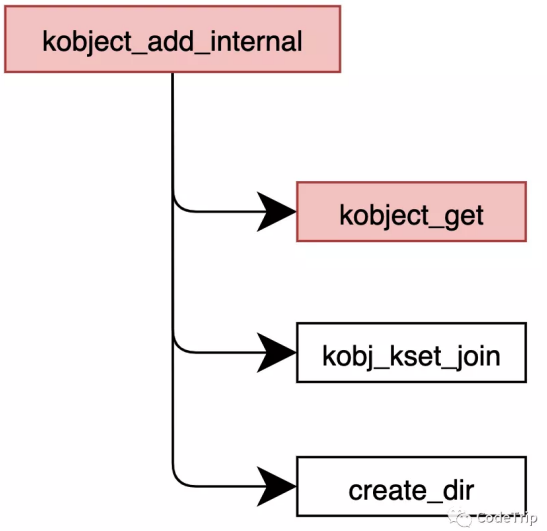

图10 kobjec_add_internal函数结构

> lib/kobject.c

```
static int kobject_add_internal(struct kobject *kobj)
{
 int error = 0;
 struct kobject *parent;
 // 检验kobj是否为空
 if (!kobj)
  return -ENOENT;
 // 检验kobj->name的有效性
 if (!kobj->name || !kobj->name[0]) {
  WARN(1, "kobject: (%p): attempted to be registered with empty "
     "name!\n", kobj);
  return -EINVAL;
 }
 // 增加该kobject的parent引用计数
 parent = kobject_get(kobj->parent);

 /* join kset if set, use it as parent if we do not already have one */
 // 如果kset不为空
 if (kobj->kset) {
  if (!parent)
  // 该kobject没有父亲kobj，但是存在kset，将它的parant设置为kset
   parent = kobject_get(&kobj->kset->kobj);
```

```
    // 将kobject加入到kset链表中
    kobj_kset_join(kobj);
    kobj->parent = parent;
  }

  pr_debug("kobject: '%s' (%p): %s: parent: '%s', set: '%s'\n",
    kobject_name(kobj), kobj, __func__,
    parent ? kobject_name(parent) : "<NULL>",
    kobj->kset ? kobject_name(&kobj->kset->kobj) : "<NULL>");

  error = create_dir(kobj);
  if (error) {
    kobj_kset_leave(kobj);
    kobject_put(parent);
    kobj->parent = NULL;

    /* be noisy on error issues */
    if (error == -EEXIST)
      pr_err("%s failed for %s with -EEXIST, don't try to register things with the same name
             __func__, kobject_name(kobj));
    else
      pr_err("%s failed for %s (error: %d parent: %s)\n",
             __func__, kobject_name(kobj), error,
             parent ? kobject_name(parent) : "'none'");
  } else
    kobj->state_in_sysfs = 1;

  return error;
}
```

kobject_init_and_add

`kobject_init_and_add` 函数集成了 `kobject_init` 和 `kobject_add_varg` ，顾名思义先初始化再添加进内核。

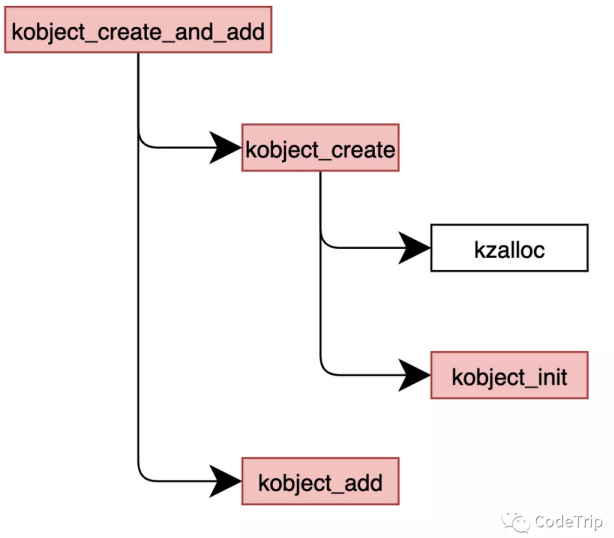

图11 kobjec_init_and_add函数结构

> lib/kobject.c

```
int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,
    struct kobject *parent, const char *fmt, ...)
{
 va_list args;
 int retval;

 kobject_init(kobj, ktype);

 va_start(args, fmt);
 retval = kobject_add_varg(kobj, parent, fmt, args);
 va_end(args);

 return retval;
}
EXPORT_SYMBOL_GPL(kobject_init_and_add);
```

kobject引用计数操作

对象的引用计数存在，对象就必须存在，底层控制kobject引用计数的函数有： `kobject_get` 和 `kobject_put`

kobject_get

`kobject_get` 将增加kobject的引用计数，并返回指向kobject的指针。

函数首先对kobj的初始化标志位进行检查确保kobj对象已经被初始化，调用 `kref_get` 函数增加引用计数。

kref是一个引用计数器，通常被嵌套在其他结构中，记录所嵌套结构的引用计数。

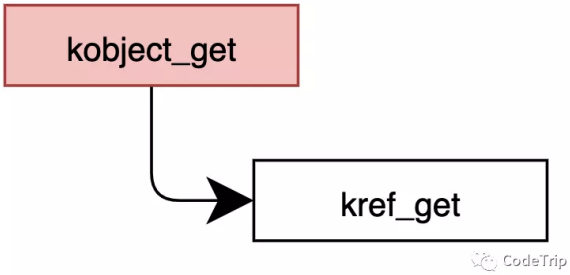

图12 kobjec_get函数结构

> lib/kobject.c

```
struct kobject *kobject_get(struct kobject *kobj)
{
 if (kobj) {
  if (!kobj->state_initialized)
   WARN(1, KERN_WARNING "kobject: '%s' (%p): is not "
          "initialized, yet kobject_get() is being "
          "called.\n", kobject_name(kobj), kobj);
  kref_get(&kobj->kref);
 }
 return kobj;
}
EXPORT_SYMBOL(kobject_get);
```

kobject_put

引用被释放时调用 `kobject_put` 减少引用计数，并在可能情况下释放对象。调用kref_put函数减少引用计数，注意与kref_get的区别，此时参数二为kobject_release，表示在引用计数为0时调用该函数。

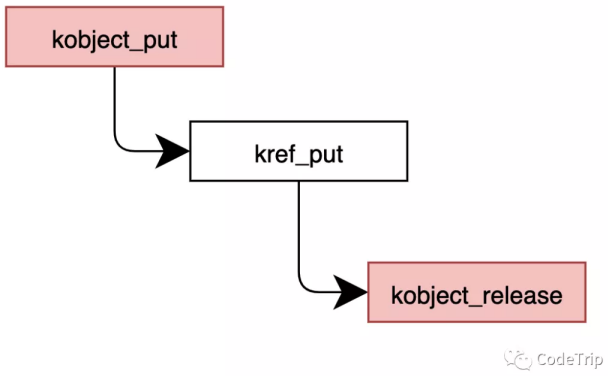

图13 kobjec_put函数结构

> lib/kobject.c

```
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);
 }
}
EXPORT_SYMBOL(kobject_put);
```

kobject_release

kobject_release函数首先通过 `container_of` 与kref得到 `kobject` 地址，其次通过 `kobject_cleanup` 继续执行释放操作。

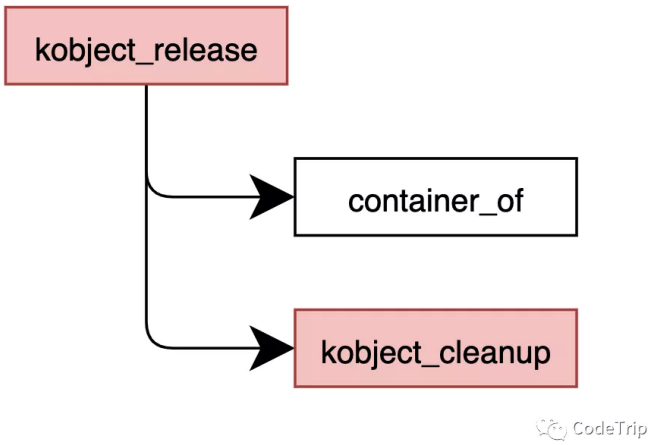

图14 kobjec_release函数结构

> lib/kobject.c

```
static void kobject_release(struct kref *kref)
{
 struct kobject *kobj = container_of(kref, struct kobject, kref);
#ifdef CONFIG_DEBUG_KOBJECT_RELEASE
 unsigned long delay = HZ + HZ * (get_random_int() & 0x3);
 pr_info("kobject: '%s' (%p): %s, parent %p (delayed %ld)\n",
   kobject_name(kobj), kobj, __func__, kobj->parent, delay);
 INIT_DELAYED_WORK(&kobj->release, kobject_delayed_cleanup);

 schedule_delayed_work(&kobj->release, delay);
#else
 kobject_cleanup(kobj);
#endif
}
```

kobject_cleanup

`kobject_cleanup` 函数主要释放操作。

> lib/kobject.c

```
static void kobject_cleanup(struct kobject *kobj)
{
```

```
{
 // 检查该kobject是否有ktype，若没有打印警告信息
 struct kobj_type *t = get_ktype(kobj);
 const char *name = kobj->name;

 pr_debug("kobject: '%s' (%p): %s, parent %p\n",
   kobject_name(kobj), kobj, __func__, kobj->parent);

 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" */
 // 如果该kobject向用户空间发送了ADD uevent但没有发送REMOVE uevent，补发REMOVE uevent
 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);
 }
 // 如果该kobject有在sysfs文件系统注册，调用kobject_del接口，删除它在sysfs中的注册
 /* 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);
 }
 // 调用该kobject的ktype的release接口，释放内存空间
 if (t && t->release) {
  pr_debug("kobject: '%s' (%p): calling ktype release\n",
    kobject_name(kobj), kobj);
  t->release(kobj);
 }
 // 释放该kobject的name所占用的内存空间
 /* free name if we allocated it */
 if (name) {
  pr_debug("kobject: '%s': free name\n", name);
  kfree_const(name);
 }
}
```

kset操作

kset初始化

kset_init

`kset_init` 用于已经分配好的kset，调用 `kobject_init_internal` 初始化 `kobject` ，然后初始化 `kset` 链表。

> lib/kobject.c

```
void kset_init(struct kset *k)
{
 // 初始化其kobject
 kobject_init_internal(&k->kobj);
 // 然后初始化kset的链表
 INIT_LIST_HEAD(&k->list);
 spin_lock_init(&k->list_lock);
}
```

kset_register

`kset_register` 先初始化kset再调用 `kobject_add_internal` 将将其kobject添加到kernel

> lib/kobject.c

```
int kset_register(struct kset *k)
{
 int err;

 if (!k)
  return -EINVAL;
 kset_init(k);

 err = kobject_add_internal(&k->kobj);
 if (err)
  return err;
 kobject_uevent(&k->kobj, KOBJ_ADD);
```

```
 return 0;
}
EXPORT_SYMBOL(kset_register);
```

kset_unregister

`kset_unregister` 调用kobject_del从层次结构中取消 kobject 的链接，然后调用 `kobject_put` 释放其kobject，当其kobject的引用计数为0时，即调用ktype的release接口释放kset占用的空间。

> lib/kobject.c

```
void kset_unregister(struct kset *k)
{
 if (!k)
  return;
 kobject_del(&k->kobj);
 kobject_put(&k->kobj);
}
EXPORT_SYMBOL(kset_unregister);
```

参考文献

[1] http://www.wowotech.net/device_model/kobject.html

[2] **https://mp.weixin.qq.com/s/Z7VPNgB0-goJ98PubPRdjg**

[3] 《Linux设备驱动程序(第二版)》

[4] 《linux那些事系列丛书之我是sysfs》

Python到底是强类型语言，还是弱类型语言?

Python Web与Django开发

20个提高生产力的 Linux 命令与技巧，用完带你起飞

高效运维