**Linux设备模型(2)\_Kobject**

**1. 前言**

Kobject是Linux设备模型的基础，也是设备模型中最难理解的一部分（可参考Documentation/kobject.txt的表述）。因此有必要先把它分析清楚。

**2. 基本概念**

由“[Linux设备模型(1)\_基本概念](http://www.wowotech.net/linux_kenrel/13.html)”可知，Linux设备模型的核心是使用Bus、Class、Device、Driver四个核心数据结构，将大量的、不同功能的硬件设备（以及驱动该硬件设备的方法），以树状结构的形式，进行归纳、抽象，从而方便Kernel的统一管理。

而硬件设备的数量、种类是非常多的，这就决定了Kernel中将会有大量的有关设备模型的数据结构。这些数据结构一定有一些共同的功能，需要抽象出来统一实现，否则就会不可避免的产生冗余代码。这就是Kobject诞生的背景。

目前为止，Kobject主要提供如下功能：

1. 通过parent指针，可以将所有Kobject以层次结构的形式组合起来。
2. 使用一个引用计数（reference count），来记录Kobject被引用的次数，并在引用次数变为0时把它释放（这是Kobject诞生时的唯一功能）。
3. 和sysfs虚拟文件系统配合，将每一个Kobject及其特性，以文件的形式，开放到用户空间（有关sysfs，会在其它文章中专门描述，本文不会涉及太多内容）。

注1：在Linux中，Kobject几乎不会单独存在。它的主要功能，就是内嵌在一个大型的数据结构中，为这个数据结构提供一些底层的功能实现。   
注2：Linux driver开发者，很少会直接使用Kobject以及它提供的接口，而是使用构建在Kobject之上的设备模型接口。

**3. 代码解析**

**3.1 在Linux Kernel source code中的位置**

在Kernel源代码中，Kobject由如下两个文件实现：

* include/linux/kobject.h
* lib/kobject.c

其中kobject.h为Kobject的头文件，包含所有的数据结构定义和接口声明。kobject.c为核心功能的实现。

**3.2 主要的数据结构**

在描述数据结构之前，有必要说明一下Kobject, Kset和Ktype这三个概念。

Kobject是基本数据类型，每个Kobject都会在"/sys/“文件系统中以目录的形式出现。

Ktype代表Kobject（严格地讲，是包含了Kobject的数据结构）的属性操作集合（由于通用性，多个Kobject可能共用同一个属性操作集，因此把Ktype独立出来了）。   
注3：在设备模型中，ktype的命名和解释，都非常抽象，理解起来非常困难，后面会详细说明。

Kset是一个特殊的Kobject（因此它也会在"/sys/“文件系统中以目录的形式出现），它用来集合相似的Kobject（这些Kobject可以是相同属性的，也可以不同属性的）。

* 首先看一下Kobject的原型

1: /\* Kobject: include/linux/kobject.h line 60 \*/

2: struct kobject {

3:     const char \*name;

4:     struct list\_head entry;

5:     struct kobject \*parent;

6:     struct kset \*kset;

7:     struct kobj\_type \*ktype;

8:     struct sysfs\_dirent \*sd;

9:     struct kref kref;

10:    unsigned int state\_initialized:1;

11:    unsigned int state\_in\_sysfs:1;

12:    unsigned int state\_add\_uevent\_sent:1;

13:    unsigned int state\_remove\_uevent\_sent:1;

14:    unsigned int uevent\_suppress:1;

15: };

name，该Kobject的名称，同时也是sysfs中的目录名称。由于Kobject添加到Kernel时，需要根据名字注册到sysfs中，之后就不能再直接修改该字段。如果需要修改Kobject的名字，需要调用kobject\_rename接口，该接口会主动处理sysfs的相关事宜。

entry，用于将Kobject加入到Kset中的list\_head。

parent，指向parent kobject，以此形成层次结构（在sysfs就表现为目录结构）。

kset，该kobject属于的Kset。可以为NULL。如果存在，且没有指定parent，则会把Kset作为parent（别忘了Kset是一个特殊的Kobject）。

ktype，该Kobject属于的kobj\_type。每个Kobject必须有一个ktype，或者Kernel会提示错误。

sd，该Kobject在sysfs中的表示。

kref，"struct kref”类型（在include/linux/kref.h中定义）的变量，为一个可用于原子操作的引用计数。

state\_initialized，指示该Kobject是否已经初始化，以在Kobject的Init，Put，Add等操作时进行异常校验。

state\_in\_sysfs，指示该Kobject是否已在sysfs中呈现，以便在自动注销时从sysfs中移除。

state\_add\_uevent\_sent/state\_remove\_uevent\_sent，记录是否已经向用户空间发送ADD uevent，如果有，且没有发送remove uevent，则在自动注销时，补发REMOVE uevent，以便让用户空间正确处理。

uevent\_suppress，如果该字段为1，则表示忽略所有上报的uevent事件。

注4：Uevent提供了“用户空间通知”的功能实现，通过该功能，当内核中有Kobject的增加、删除、修改等动作时，会通知用户空间。有关该功能的具体内容，会在其它文章详细描述。

* Kset的原型为

1: /\* include/linux/kobject.h, line 159 \*/

2: struct kset {

3:     struct list\_head list;

4:     spinlock\_t list\_lock;

5:     struct kobject kobj;

6:     const struct kset\_uevent\_ops \*uevent\_ops;

7: };

list/list\_lock，用于保存该kset下所有的kobject的链表。

kobj，该kset自己的kobject（kset是一个特殊的kobject，也会在sysfs中以目录的形式体现）。

uevent\_ops，该kset的uevent操作函数集。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。

* Ktype的原型为

1: /\* include/linux/kobject.h, line 108 \*/

2: struct kobj\_type {

3:     void (\*release)(struct kobject \*kobj);

4:     const struct sysfs\_ops \*sysfs\_ops;

5:     struct attribute \*\*default\_attrs;

6:     const struct kobj\_ns\_type\_operations \*(\*child\_ns\_type)(struct kobject \*kobj);

7:     const void \*(\*namespace)(struct kobject \*kobj);

8: };

release，通过该回调函数，可以将包含该种类型kobject的数据结构的内存空间释放掉。

sysfs\_ops，该种类型的Kobject的sysfs文件系统接口。

default\_attrs，该种类型的Kobject的atrribute列表（所谓attribute，就是sysfs文件系统中的一个文件）。将会在Kobject添加到内核时，一并注册到sysfs中。

child\_ns\_type/namespace，和文件系统（sysfs）的命名空间有关，这里不再详细说明。

|  |
| --- |
| **总结，Ktype以及整个Kobject机制的理解。** Kobject的核心功能是：保持一个引用计数，当该计数减为0时，自动释放（由本文所讲的kobject模块负责）Kobject所占用的meomry空间。这就决定了Kobject必须是动态分配的（只有这样才能动态释放）。   而Kobject大多数的使用场景，是内嵌在大型的数据结构中（如Kset、device\_driver等），因此这些大型的数据结构，也必须是动态分配、动态释放的。那么释放的时机是什么呢？是内嵌的Kobject释放时。但是Kobject的释放是由Kobject模块自动完成的（在引用计数为0时），那么怎么一并释放包含自己的大型数据结构呢？   这时Ktype就派上用场了。我们知道，Ktype中的release回调函数负责释放Kobject（甚至是包含Kobject的数据结构）的内存空间，那么Ktype及其内部函数，是由谁实现呢？是由上层数据结构所在的模块！因为只有它，才清楚Kobject嵌在哪个数据结构中，并通过Kobject指针以及自身的数据结构类型，找到需要释放的上层数据结构的指针，然后释放它。   讲到这里，就清晰多了。所以，每一个内嵌Kobject的数据结构，例如kset、device、device\_driver等等，都要实现一个Ktype，并定义其中的回调函数。同理，sysfs相关的操作也一样，必须经过ktype的中转，因为sysfs看到的是Kobject，而真正的文件操作的主体，是内嵌Kobject的上层数据结构！    顺便提一下，Kobject是面向对象的思想在Linux kernel中的极致体现，但C语言的优势却不在这里，所以Linux kernel需要用比较巧妙（也很啰嗦）的手段去实现， |

**3.3 功能分析**

**3.3.1 Kobject使用流程**

Kobject大多数情况下（有一种例外，下面会讲）会嵌在其它数据结构中使用，其使用流程如下：

1. 定义一个struct kset类型的指针，并在初始化时为它分配空间，添加到内核中
2. 根据实际情况，定义自己所需的数据结构原型，该数据结构中包含有Kobject
3. 定义一个适合自己的ktype，并实现其中回调函数
4. 在需要使用到包含Kobject的数据结构时，动态分配该数据结构，并分配Kobject空间，添加到内核中
5. 每一次引用数据结构时，调用kobject\_get接口增加引用计数；引用结束时，调用kobject\_put接口，减少引用计数
6. 当引用计数减少为0时，Kobject模块调用ktype所提供的release接口，释放上层数据结构以及Kobject的内存空间

上面有提过，有一种例外，Kobject不再嵌在其它数据结构中，可以单独使用，这个例外就是：开发者只需要在sysfs中创建一个目录，而不需要其它的kset、ktype的操作。这时可以直接调用kobject\_create\_and\_add接口，分配一个kobject结构并把它添加到kernel中。

**3.3.2 Kobject的分配和释放**

前面讲过，Kobject必须动态分配，而不能静态定义或者位于堆栈之上，它的分配方法有两种。

1. 通过kmalloc自行分配（一般是跟随上层数据结构分配），并在初始化后添加到kernel。这种方法涉及如下接口：

1: /\* include/linux/kobject.h, line 85 \*/

2: extern void kobject\_init(struct kobject \*kobj, struct kobj\_type \*ktype);

3: extern \_\_printf(3, 4) \_\_must\_check

4: int kobject\_add(struct kobject \*kobj, struct kobject \*parent,

5:                 const char \*fmt, ...);

6: extern \_\_printf(4, 5) \_\_must\_check

7: int kobject\_init\_and\_add(struct kobject \*kobj,

8:             struct kobj\_type \*ktype, struct kobject \*parent,

9:             const char \*fmt, ...);

kobject\_init，初始化通过kmalloc等内存分配函数获得的struct kobject指针。主要执行逻辑为：

* 确认kobj和ktype不为空
* 如果该指针已经初始化过（判断kobj->state\_initialized），打印错误提示及堆栈信息（但不是致命错误，所以还可以继续）
* 初始化kobj内部的参数，包括引用计数、list、各种标志等
* 根据输入参数，将ktype指针赋予kobj->ktype

kobject\_add，将初始化完成的kobject添加到kernel中，参数包括需要添加的kobject、该kobject的parent（用于形成层次结构，可以为空）、用于提供kobject name的格式化字符串。主要执行逻辑为：

* 确认kobj不为空，确认kobj已经初始化，否则错误退出
* 调用内部接口kobject\_add\_varg，完成添加操作

kobject\_init\_and\_add，是上面两个接口的组合，不再说明。

==========================内部接口======================================

kobject\_add\_varg，解析格式化字符串，将结果赋予kobj->name，之后调用kobject\_add\_internal接口，完成真正的添加操作。

kobject\_add\_internal，将kobject添加到kernel。主要执行逻辑为：

* 校验kobj以及kobj->name的合法性，若不合法打印错误信息并退出
* 调用kobject\_get增加该kobject的parent的引用计数（如果存在parent的话）
* 如果存在kset（即kobj->kset不为空），则调用kobj\_kset\_join接口加入kset。同时，如果该kobject没有parent，却存在kset，则将它的parent设为kset（kset是一个特殊的kobject），并增加kset的引用计数
* 通过create\_dir接口，调用sysfs的相关接口，在sysfs下创建该kobject对应的目录
* 如果创建失败，执行后续的回滚操作，否则将kobj->state\_in\_sysfs置为1

kobj\_kset\_join，负责将kobj加入到对应kset的链表中。

这种方式分配的kobject，会在引用计数变为0时，由kobject\_put调用其ktype的release接口，释放内存空间，具体可参考后面有关kobject\_put的讲解。

2. 使用kobject\_create创建

Kobject模块可以使用kobject\_create自行分配空间，并内置了一个ktype（dynamic\_kobj\_ktype），用于在计数为0是释放空间。代码如下：

1: /\* include/linux/kobject.h, line 96 \*/

2: extern struct kobject \* \_\_must\_check kobject\_create(void);

3: extern struct kobject \* \_\_must\_check kobject\_create\_and\_add(const char \*name,

4:             struct kobject \*parent);

1: /\* lib/kobject.c, line 605 \*/

2: static void dynamic\_kobj\_release(struct kobject \*kobj)

3: {

4:     pr\_debug("kobject: (%p): %s\n", kobj, \_\_func\_\_);

5:     kfree(kobj);

6: }

7:

8: static struct kobj\_type dynamic\_kobj\_ktype = {

9:     .release = dynamic\_kobj\_release,

10:    .sysfs\_ops = &kobj\_sysfs\_ops,

11: };

kobject\_create，该接口为kobj分配内存空间，并以dynamic\_kobj\_ktype为参数，调用kobject\_init接口，完成后续的初始化操作。

kobject\_create\_and\_add，是kobject\_create和kobject\_add的组合，不再说明。

dynamic\_kobj\_release，直接调用kfree释放kobj的空间。

**3.3.3 Kobject引用计数的修改**

通过kobject\_get和kobject\_put可以修改kobject的引用计数，并在计数为0时，调用ktype的release接口，释放占用空间。

1: /\* include/linux/kobject.h, line 103 \*/

2: extern struct kobject \*kobject\_get(struct kobject \*kobj);

3: extern void kobject\_put(struct kobject \*kobj);

kobject\_get，调用kref\_get，增加引用计数。

kobject\_put，以内部接口kobject\_release为参数，调用kref\_put。kref模块会在引用计数为零时，调用kobject\_release。

==========================内部接口======================================

kobject\_release，通过kref结构，获取kobject指针，并调用kobject\_cleanup接口继续。

kobject\_cleanup，负责释放kobject占用的空间，主要执行逻辑如下：

* 检查该kobject是否有ktype，如果没有，打印警告信息
* 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
* 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
* 调用该kobject的ktype的release接口，释放内存空间
* 释放该kobject的name所占用的内存空间

**3.3.4 Kset的初始化、注册**

Kset是一个特殊的kobject，因此其初始化、注册等操作也会调用kobject的相关接口，除此之外，会有它特有的部分。另外，和Kobject一样，kset的内存分配，可以由上层软件通过kmalloc自行分配，也可以由Kobject模块负责分配，具体如下。

1: /\* include/linux/kobject.h, line 166 \*/

2: extern void kset\_init(struct kset \*kset);

3: extern int \_\_must\_check kset\_register(struct kset \*kset);

4: extern void kset\_unregister(struct kset \*kset);

5: extern struct kset \* \_\_must\_check kset\_create\_and\_add(const char \*name,

6:             const struct kset\_uevent\_ops \*u,

7:             struct kobject \*parent\_kobj);

kset\_init，该接口用于初始化已分配的kset，主要包括调用kobject\_init\_internal初始化其kobject，然后初始化kset的链表。需要注意的时，如果使用此接口，上层软件必须提供该kset中的kobject的ktype。

kset\_register，先调用kset\_init，然后调用kobject\_add\_internal将其kobject添加到kernel。

kset\_unregister，直接调用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。

kset\_create\_and\_add，会调用内部接口kset\_create动态创建一个kset，并调用kset\_register将其注册到kernel。

==========================内部接口======================================

kset\_create，该接口使用kzalloc分配一个kset空间，并定义一个kset\_ktype类型的ktype，用于释放所有由它分配的kset空间。