此文档的全部内部来自于wowo的网站的整理，地址[www.wowotech.net](http://www.wowotech.net/linux_kenrel/13.html)。作者是嵌入式Linux深耕多年的大佬，我的学习和理解嵌入式Linux也参考此文档的资料。

# Linux设备模型(1) 基本概念

## **前言**

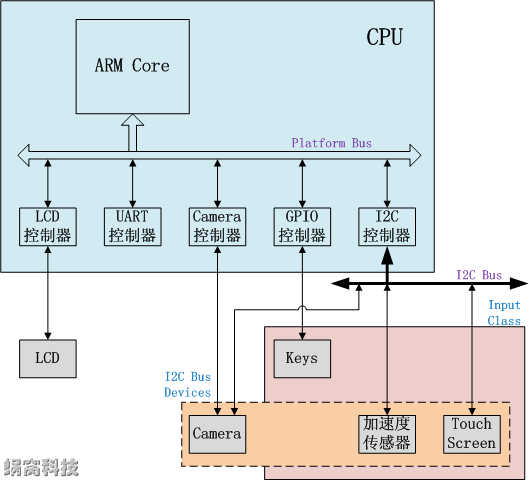
在“[Linux内核的整体架构](http://www.wowotech.net/linux_kenrel/11.html)”中，蜗蜗有提到，由于Linux支持世界上几乎所有的、不同功能的硬件设备（这是Linux的优点），导致Linux内核中有一半的代码是设备驱动，而且随着硬件的快速升级换代，设备驱动的代码量也在快速增长。个人意见，这种现象打破了“简洁就是美”的理念，是丑陋的。它导致Linux内核看上去非常臃肿、杂乱、不易维护。但蜗蜗也知道，这不是Linux的错，Linux是一个宏内核，它必须面对设备的多样性，并实现对应的驱动。

为了降低设备多样性带来的Linux驱动开发的复杂度，以及设备热拔插处理、电源管理等，Linux内核提出了设备模型（也称作Driver Model）的概念。设备模型将硬件设备归纳、分类，然后抽象出一套标准的数据结构和接口。驱动的开发，就简化为对内核所规定的数据结构的填充和实现。

本文将会从设备模型的基本概念开始，通过分析内核相应的代码，一步一步解析Linux设备模型的实现及使用方法。

## 2. Linux设备模型的基本概念

### 2.1 Bus, Class, Device和Device Driver的概念

下图是嵌入式系统常见的硬件拓扑的一个示例：  
[](http://www.wowotech.net/content/uploadfile/201402/861308c496428d7590deccd7507b61f720140227080145.gif)

硬件拓扑描述Linux设备模型中四个重要概念中三个：Bus，Class和Device（第四个为Device Driver，后面会说）。

Bus（总线）：Linux认为（可以参考include/linux/device.h中struct bus\_type的注释），总线是CPU和一个或多个设备之间信息交互的通道。而为了方便设备模型的抽象，所有的设备都应连接到总线上（无论是CPU内部总线、虚拟的总线还是“platform Bus”）。

Class（分类）：在Linux设备模型中，Class的概念非常类似面向对象程序设计中的Class（类），它主要是集合具有相似功能或属性的设备，这样就可以抽象出一套可以在多个设备之间共用的数据结构和接口函数。因而从属于相同Class的设备的驱动程序，就不再需要重复定义这些公共资源，直接从Class中继承即可。

Device（设备）：抽象系统中所有的硬件设备，描述它的名字、属性、从属的Bus、从属的Class等信息。

Device Driver（驱动）：Linux设备模型用Driver抽象硬件设备的驱动程序，它包含设备初始化、电源管理相关的接口实现。而Linux内核中的驱动开发，基本都围绕该抽象进行（实现所规定的接口函数）。

注：什么是Platform Bus？  
在计算机中有这样一类设备，它们通过各自的设备控制器，直接和CPU连接，CPU可以通过常规的寻址操作访问它们（或者说访问它们的控制器）。这种连接方式，并不属于传统意义上的总线连接。但设备模型应该具备普适性，因此Linux就虚构了一条Platform Bus，供这些设备挂靠。

### 2.2 设备模型的核心思想

Linux设备模型的核心思想是（通过xxx手段，实现xxx目的）：

1. 用Device（struct device）和Device Driver（struct device\_driver）两个数据结构，分别从“有什么用”和“怎么用”两个角度描述硬件设备。这样就统一了编写设备驱动的格式，使驱动开发从论述题变为填空体，从而简化了设备驱动的开发。

2. 同样使用Device和Device Driver两个数据结构，实现硬件设备的即插即用（热拔插）。

在Linux内核中，只要任何Device和Device Driver具有相同的名字，内核就会执行Device Driver结构中的初始化函数（probe），该函数会初始化设备，使其为可用状态。

而对大多数热拔插设备而言，它们的Device Driver一直存在内核中。当设备没有插入时，其Device结构不存在，因而其Driver也就不执行初始化操作。当设备插入时，内核会创建一个Device结构（名称和Driver相同），此时就会触发Driver的执行。这就是即插即用的概念。

3. 通过”Bus-->Device”类型的树状结构（见2.1章节的图例）解决设备之间的依赖，而这种依赖在开关机、电源管理等过程中尤为重要。

试想，一个设备挂载在一条总线上，要启动这个设备，必须先启动它所挂载的总线。很显然，如果系统中设备非常多、依赖关系非常复杂的时候，无论是内核还是驱动的开发人员，都无力维护这种关系。

而设备模型中的这种树状结构，可以自动处理这种依赖关系。启动某一个设备前，内核会检查该设备是否依赖其它设备或者总线，如果依赖，则检查所依赖的对象是否已经启动，如果没有，则会先启动它们，直到启动该设备的条件具备为止。而驱动开发人员需要做的，就是在编写设备驱动时，告知内核该设备的依赖关系即可。

4. 使用Class结构，在设备模型中引入面向对象的概念，这样可以最大限度地抽象共性，减少驱动开发过程中的重复劳动，降低工作量。

# 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，并实现其中的回调函数
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。

# Linux设备模型(3) uevent

## 1. uevent的功能

uevent是kobject的一部分，用于在kobject状态发生改变时，例如增加、移除等，通知用户空间程序。用户空间程序收到这样的事件后，会做相应的处理。

该机制通常是用来支持热拔插设备的，例如U盘插入后，USB相关的驱动软件会动态创建用于表示该U盘的device结构（相应的也包括其中的kobject），并告知用户空间程序，为该U盘动态的创建/dev/目录下的设备节点，更进一步，可以通知其它的应用程序，将该U盘设备mount到系统中，从而动态的支持该设备。

## 2. uevent在kernel中的位置

下面图片描述了uevent模块在内核中的位置：

[](http://www.wowotech.net/content/uploadfile/201403/272229a8f5b5912c352e857e119e516320140311063951.gif)

由此可知，uevent的机制是比较简单的，设备模型中任何设备有事件需要上报时，会触发uevent提供的接口。uevent模块准备好上报事件的格式后，可以通过两个途径把事件上报到用户空间：一种是通过kmod模块，直接调用用户空间的可执行文件；另一种是通过netlink通信机制，将事件从内核空间传递给用户空间。

注1：有关kmod和netlink，会在其它文章中描述，因此本文就不再详细说明了。

## 3. uevent的内部逻辑解析

### 3.1 Source Code位置

uevent的代码比较简单，主要涉及kobject.h和kobject\_uevent.c两个文件，如下：

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

### 3.2 数据结构描述

kobject.h定义了uevent相关的常量和数据结构，如下：

* kobject\_action

1: /\* include/linux/kobject.h, line 50 \*/

2: enum kobject\_action {

3:     KOBJ\_ADD,

4:     KOBJ\_REMOVE,

5:     KOBJ\_CHANGE,

6:     KOBJ\_MOVE,

7:     KOBJ\_ONLINE,

8:     KOBJ\_OFFLINE,

9:     KOBJ\_MAX

10: };

kobject\_action定义了event的类型，包括：

ADD/REMOVE，kobject（或上层数据结构）的添加/移除事件。

ONLINE/OFFLINE，kobject（或上层数据结构）的上线/下线事件，其实是是否使能。

CHANGE，kobject（或上层数据结构）的状态或者内容发生改变。

MOVE，kobject（或上层数据结构）更改名称或者更改Parent（意味着在sysfs中更改了目录结构）。

CHANGE，如果设备驱动需要上报的事件不再上面事件的范围内，或者是自定义的事件，可以使用该event，并携带相应的参数。

* kobj\_uevent\_env

1: /\* include/linux/kobject.h, line 31 \*/

2: #define UEVENT\_NUM\_ENVP 32 /\* number of env pointers \*/

3: #define UEVENT\_BUFFER\_SIZE 2048 /\* buffer for the variables \*/

4:

5: /\* include/linux/kobject.h, line 116 \*/

6: struct kobj\_uevent\_env {

7:     char \*envp[UEVENT\_NUM\_ENVP];

8:     int envp\_idx;

9:     char buf[UEVENT\_BUFFER\_SIZE];

10:    int buflen;

11: };

前面有提到过，在利用Kmod向用户空间上报event事件时，会直接执行用户空间的可执行文件。而在Linux系统，可执行文件的执行，依赖于环境变量，因此kobj\_uevent\_env用于组织此次事件上报时的环境变量。

envp，指针数组，用于保存每个环境变量的地址，最多可支持的环境变量数量为UEVENT\_NUM\_ENVP。

envp\_idx，用于访问环境变量指针数组的index。

buf，保存环境变量的buffer，最大为UEVENT\_BUFFER\_SIZE。

buflen，访问buf的变量。

* kset\_uevent\_ops

1: /\* include/linux/kobject.h, line 123 \*/

2: struct kset\_uevent\_ops {

3:     int (\* const filter)(struct kset \*kset, struct kobject \*kobj);

4:     const char \*(\* const name)(struct kset \*kset, struct kobject \*kobj);

5:     int (\* const uevent)(struct kset \*kset, struct kobject \*kobj,

6:                         struct kobj\_uevent\_env \*env);

7: };

kset\_uevent\_ops是为kset量身订做的一个数据结构，里面包含filter和uevent两个回调函数，用处如下：

filter，当任何kobject需要上报uevent时，它所属的kset可以通过该接口过滤，阻止不希望上报的event，从而达到从整体上管理的目的。

name，该接口可以返回kset的名称。如果一个kset没有合法的名称，则其下的所有kobject将不允许上报uvent

uevent，当任何kobject需要上报uevent时，它所属的kset可以通过该接口统一为这些event添加环境变量。因为很多时候上报uevent时的环境变量都是相同的，因此可以由kset统一处理，就不需要让每个kobject独自添加了。

### 3.3 内部动作

通过kobject.h，uevent模块提供了如下的API（这些API的实现是在"lib/kobject\_uevent.c”文件中）：

1: /\* include/linux/kobject.h, line 206 \*/

2: int kobject\_uevent(struct kobject \*kobj, enum kobject\_action action);

3: int kobject\_uevent\_env(struct kobject \*kobj, enum kobject\_action action,

4:                         char \*envp[]);

5:

6: \_\_printf(2, 3)

7: int add\_uevent\_var(struct kobj\_uevent\_env \*env, const char \*format, ...);

8:

9: int kobject\_action\_type(const char \*buf, size\_t count,

10:                         enum kobject\_action \*type);

kobject\_uevent\_env，以envp为环境变量，上报一个指定action的uevent。环境变量的作用是为执行用户空间程序指定运行环境。具体动作如下：

* 查找kobj本身或者其parent是否从属于某个kset，如果不是，则报错返回（注2：由此可以说明，如果一个kobject没有加入kset，是不允许上报uevent的）
* 查看kobj->uevent\_suppress是否设置，如果设置，则忽略所有的uevent上报并返回（注3：由此可知，可以通过kobject的uevent\_suppress标志，管控kobject的uevent的上报）
* 如果所属的kset有uevent\_ops->filter函数，则调用该函数，过滤此次上报（注4：这佐证了3.2小节有关filter接口的说明，kset可以通过filter接口过滤不希望上报的event，从而达到整体的管理效果）
* 判断所属的kset是否有合法的名称（称作subsystem，和前期的内核版本有区别），否则不允许上报uevent
* 分配一个用于此次上报的、存储环境变量的buffer（结果保存在env指针中），并获得该kobject在sysfs中路径信息（用户空间软件需要依据该路径信息在sysfs中访问它）
* 调用add\_uevent\_var接口（下面会介绍），将Action、路径信息、subsystem等信息，添加到env指针中
* 如果传入的envp不空，则解析传入的环境变量中，同样调用add\_uevent\_var接口，添加到env指针中
* 如果所属的kset存在uevent\_ops->uevent接口，调用该接口，添加kset统一的环境变量到env指针
* 根据ACTION的类型，设置kobj->state\_add\_uevent\_sent和kobj->state\_remove\_uevent\_sent变量，以记录正确的状态
* 调用add\_uevent\_var接口，添加格式为"SEQNUM=%llu”的序列号
* 如果定义了"CONFIG\_NET”，则使用netlink发送该uevent
* 以uevent\_helper、subsystem以及添加了标准环境变量（HOME=/，PATH=/sbin:/bin:/usr/sbin:/usr/bin）的env指针为参数，调用kmod模块提供的call\_usermodehelper函数，上报uevent。  
  其中uevent\_helper的内容是由内核配置项CONFIG\_UEVENT\_HELPER\_PATH(位于./drivers/base/Kconfig)决定的(可参考lib/kobject\_uevent.c, line 32)，该配置项指定了一个用户空间程序（或者脚本），用于解析上报的uevent，例如"/sbin/hotplug”。  
  call\_usermodehelper的作用，就是fork一个进程，以uevent为参数，执行uevent\_helper。

kobject\_uevent，和kobject\_uevent\_env功能一样，只是没有指定任何的环境变量。

add\_uevent\_var，以格式化字符的形式（类似printf、printk等），将环境变量copy到env指针中。

kobject\_action\_type，将enum kobject\_action类型的Action，转换为字符串。

|  |
| --- |
| **说明：怎么指定处理uevent的用户空间程序(简称uevent helper)？ 上面介绍kobject\_uevent\_env的内部动作时，有提到，uevent模块通过Kmod上报uevent时，会通过call\_usermodehelper函数，调用用户空间的可执行文件（或者脚本，简称uevent helper ）处理该event。而该uevent helper的路径保存在uevent\_helper数组中。  可以在编译内核时，通过CONFIG\_UEVENT\_HELPER\_PATH配置项，静态指定uevent helper。但这种方式会为每个event fork一个进程，随着内核支持的设备数量的增多，这种方式在系统启动时将会是致命的（可以导致内存溢出等）。因此只有在早期的内核版本中会使用这种方式，现在内核不再推荐使用该方式。因此内核编译时，需要把该配置项留空。  在系统启动后，大部分的设备已经ready，可以根据需要，重新指定一个uevent helper，以便检测系统运行过程中的热拔插事件。这可以通过把helper的路径写入到"/sys/kernel/uevent\_helper“文件中实现。实际上，内核通过sysfs文件系统的形式，将uevent\_helper数组开放到用户空间，供用户空间程序修改访问，具体可参考"./kernel/ksysfs.c“中相应的代码，这里不再详细描述。** |

# Linux设备模型(4) sysfs

## 1. 前言

sysfs是一个基于RAM的文件系统，它和Kobject一起，可以将Kernel的数据结构导出到用户空间，以文件目录结构的形式，提供对这些数据结构（以及数据结构的属性）的访问支持。

sysfs具备文件系统的所有属性，而本文主要侧重其设备模型的特性,因此不会涉及过多的文件系统实现细节，而只介绍sysfs在Linux设备模型中的作用和使用方法。具体包括：

* sysfs和Kobject的关系
* attribute的概念
* sysfs的文件系统操作接口

## 2. sysfs和Kobject的关系

在"[Linux设备模型\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”文章中，有提到过，每一个Kobject，都会对应sysfs中的一个目录。因此在将Kobject添加到Kernel时，create\_dir接口会调用sysfs文件系统的创建目录接口，创建和Kobject对应的目录，相关的代码如下：

1: /\* lib/kobject.c, line 47 \*/

2: static int create\_dir(struct kobject \*kobj)

3: {

4:     int error = 0;

5:     error = sysfs\_create\_dir(kobj);

6:     if (!error) {

7:         error = populate\_dir(kobj);

8:     if (error)

9:         sysfs\_remove\_dir(kobj);

10:     }

11:     return error;

12: }

13:

14: /\* fs/sysfs/dir.c, line 736 \*/

15: \*\*

16: \* sysfs\_create\_dir - create a directory for an object.

17: \* @kobj: object we're creating directory for.

18: \*/

19: int sysfs\_create\_dir(struct kobject \* kobj)

20: {

21:     enum kobj\_ns\_type type;

22:     struct sysfs\_dirent \*parent\_sd, \*sd;

23:     const void \*ns = NULL;

24:     int error = 0;

25:     ...

26: }

## 3. attribute

### 3.1 attribute的功能概述

在sysfs中，为什么会有attribute的概念呢？其实它是对应kobject而言的，指的是kobject的“属性”。我们知道，

sysfs中的目录描述了kobject，而kobject是特定数据类型变量（如struct device）的体现。因此kobject的属性，就是这些变量的属性。它可以是任何东西，名称、一个内部变量、一个字符串等等。而attribute，在sysfs文件系统中是以文件的形式提供的，即：kobject的所有属性，都在它对应的sysfs目录下以文件的形式呈现。这些文件一般是可读、写的，而kernel中定义了这些属性的模块，会根据用户空间的读写操作，记录和返回这些attribute的值。

总结一下：所谓的attibute，就是内核空间和用户空间进行信息交互的一种方法。例如某个driver定义了一个变量，却希望用户空间程序可以修改该变量，以控制driver的运行行为，那么就可以将该变量以sysfs attribute的形式开放出来。

Linux内核中，attribute分为普通的attribute和二进制attribute，如下：

1: /\* include/linux/sysfs.h, line 26 \*/

2: struct attribute {

3:     const char \*name;

4:     umode\_t mode;

5: #ifdef CONFIG\_DEBUG\_LOCK\_ALLOC

6:     bool ignore\_lockdep:1;

7:     struct lock\_class\_key \*key;

8:     struct lock\_class\_key skey;

9: #endif

10: };

11:

12: /\* include/linux/sysfs.h, line 100 \*/

13: struct bin\_attribute {

14:     struct attribute attr;

15:     size\_t size;

16:     void \*private;

17:     ssize\_t (\*read)(struct file \*, struct kobject \*, struct bin\_attribute \*,

18:                     char \*, loff\_t, size\_t);

19:     ssize\_t (\*write)(struct file \*,struct kobject \*, struct bin\_attribute \*,

20:                     char \*, loff\_t, size\_t);

21:     int (\*mmap)(struct file \*, struct kobject \*, struct bin\_attribute \*attr,

22:                     struct vm\_area\_struct \*vma);

23: };

struct attribute为普通的attribute，使用该attribute生成的sysfs文件，只能用字符串的形式读写（后面会说为什么）。而struct bin\_attribute在struct attribute的基础上，增加了read、write等函数，因此它所生成的sysfs文件可以用任何方式读写。

说完基本概念，我们要问两个问题：

Kernel怎么把attribute变成sysfs中的文件呢？

用户空间对sysfs的文件进行的读写操作，怎么传递给Kernel呢？

下面来看看这个过程。

### 3.2 attibute文件的创建

在linux内核中，attibute文件的创建是由fs/sysfs/file.c中sysfs\_create\_file接口完成的，该接口的实现没有什么特殊之处，大多是文件系统相关的操作，和设备模型没有太多的关系，这里先略过不提。

### 3.3 attibute文件的read和write

看到3.1章节struct attribute的原型时，也许我们会犯嘀咕，该结构很简单啊，name表示文件名称，mode表示文件模式，其它的字段都是内核用于debug Kernel Lock的，那文件操作的接口在哪里呢？

不着急，我们去fs/sysfs目录下看看sysfs相关的代码逻辑。

所有的文件系统，都会定义一个struct file\_operations变量，用于描述本文件系统的操作接口，sysfs也不例外：

1: /\* fs/sysfs/file.c, line 472 \*/

2: const struct file\_operations sysfs\_file\_operations = {

3:     .read = sysfs\_read\_file,

4:     .write = sysfs\_write\_file,

5:     .llseek = generic\_file\_llseek,

6:     .open = sysfs\_open\_file,

7:     .release = sysfs\_release,

8:     .poll = sysfs\_poll,

9: };

attribute文件的read操作，会由VFS转到sysfs\_file\_operations的read（也就是sysfs\_read\_file）接口上，让我们大概看一下该接口的处理逻辑。

1: /\* fs/sysfs/file.c, line 127 \*/

2: static ssize\_t

3: sysfs\_read\_file(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

4: {

5:     struct sysfs\_buffer \* buffer = file->private\_data;

6:     ssize\_t retval = 0;

7:

8:     mutex\_lock(&buffer->mutex);

9:     if (buffer->needs\_read\_fill || \*ppos == 0) {

10:        retval = fill\_read\_buffer(file->f\_path.dentry,buffer);

11:        if (retval)

12:            goto out;

13:    }

14: ...

15: }

16: /\* fs/sysfs/file.c, line 67 \*/

17: static int fill\_read\_buffer(struct dentry \* dentry, struct sysfs\_buffer \* buffer)

18: {

19:    struct sysfs\_dirent \*attr\_sd = dentry->d\_fsdata;

20:    struct kobject \*kobj = attr\_sd->s\_parent->s\_dir.kobj;

21:    const struct sysfs\_ops \* ops = buffer->ops;

22:    ...

23:    count = ops->show(kobj, attr\_sd->s\_attr.attr, buffer->page);

24:    ...

25: }

read处理看着很简单，sysfs\_read\_file从file指针中取一个私有指针（注：大家可以稍微留一下心，私有数据的概念，在VFS中使用是非常普遍的），转换为一个struct sysfs\_buffer类型的指针，以此为参数（buffer），转身就调用fill\_read\_buffer接口。

而fill\_read\_buffer接口，直接从buffer指针中取出一个struct sysfs\_ops指针，调用该指针的show函数，即完成了文件的read操作。

那么后续呢？当然是由ops->show接口接着处理。而具体怎么处理，就是其它模块（例如某个driver）的事了，sysfs不再关心（其实，Linux大多的核心代码，都是只提供架构和机制，具体的实现，也就是苦力，留给那些码农吧！这就是设计的魅力）。

不过还没完，这个struct sysfs\_ops指针哪来的？好吧，我们再看看open(sysfs\_open\_file)接口吧。

1: /\* fs/sysfs/file.c, line 326 \*/

2: static int sysfs\_open\_file(struct inode \*inode, struct file \*file)

3: {

4:     struct sysfs\_dirent \*attr\_sd = file->f\_path.dentry->d\_fsdata;

5:     struct kobject \*kobj = attr\_sd->s\_parent->s\_dir.kobj;

6:     struct sysfs\_buffer \*buffer;

7:     const struct sysfs\_ops \*ops;

8:     int error = -EACCES;

9:

10:    /\* need attr\_sd for attr and ops, its parent for kobj \*/

11:    if (!sysfs\_get\_active(attr\_sd))

12:    return -ENODEV;

13:

14:    /\* every kobject with an attribute needs a ktype assigned \*/

15:    if (kobj->ktype && kobj->ktype->sysfs\_ops)

16:        ops = kobj->ktype->sysfs\_ops;

17:    else {

18:        WARN(1, KERN\_ERR "missing sysfs attribute operations for "

19:            "kobject: %s\n", kobject\_name(kobj));

20:        goto err\_out;

21:    }

22:

23:    ...

24:

25:    buffer = kzalloc(sizeof(struct sysfs\_buffer), GFP\_KERNEL);

26:    if (!buffer)

27:        goto err\_out;

28:

29:    mutex\_init(&buffer->mutex);

30:    buffer->needs\_read\_fill = 1;

31:    buffer->ops = ops;

32:    file->private\_data = buffer;

33:    ...

34: }

哦，原来和ktype有关系。这个指针是从该attribute所从属的kobject中拿的。再去看一下"[Linux设备模型\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”中ktype的定义，还真有一个struct sysfs\_ops的指针。

我们注意一下14行的注释以及其后代码逻辑，如果从属的kobject（就是attribute文件所在的目录）没有ktype，或者没有ktype->sysfs\_ops指针，是不允许它注册任何attribute的！

经过确认后，sysfs\_open\_file从ktype中取出struct sysfs\_ops指针，并在随后的代码逻辑中，分配一个struct sysfs\_buffer类型的指针（buffer），并把struct sysfs\_ops指针保存在其中，随后（注意哦），把buffer指针交给file的private\_data，随后read/write等接口便可以取出使用。嗯！惯用伎俩！

顺便看一下struct sysfs\_ops吧，我想你已经能够猜到了。

1: /\* include/linux/sysfs.h, line 124 \*/

2: struct sysfs\_ops {

3:     ssize\_t (\*show)(struct kobject \*, struct attribute \*,char \*);

4:     ssize\_t (\*store)(struct kobject \*,struct attribute \*,const char \*, size\_t);

5:     const void \*(\*namespace)(struct kobject \*, const struct attribute \*);

6: };

attribute文件的write过程和read类似，这里就不再多说。另外，上面只分析了普通attribute的逻辑，而二进制类型的呢？也类似，去看看fs/sysfs/bin.c吧，这里也不说了。

讲到这里，应该已经结束了，事实却不是如此。上面read/write的数据流，只到kobject（也就是目录）级别哦，而真正需要操作的是attribute（文件）啊！这中间一定还有一层转换！确实，不过又交给其它模块了。 下面我们通过一个例子，来说明如何转换的。

## 4. sysfs在设备模型中的应用总结

让我们通过设备模型class.c中有关sysfs的实现，来总结一下sysfs的应用方式。

首先，在class.c中，定义了Class所需的ktype以及sysfs\_ops类型的变量，如下：

1: /\* drivers/base/class.c, line 86 \*/

2: static const struct sysfs\_ops class\_sysfs\_ops = {

3:     .show = class\_attr\_show,

4:     .store = class\_attr\_store,

5:     .namespace = class\_attr\_namespace,

6: };

7:

8: static struct kobj\_type class\_ktype = {

9:     .sysfs\_ops = &class\_sysfs\_ops,

10:    .release = class\_release,

11:    .child\_ns\_type = class\_child\_ns\_type,

12: };

由前面章节的描述可知，所有class\_type的Kobject下面的attribute文件的读写操作，都会交给class\_attr\_show和class\_attr\_store两个接口处理。以class\_attr\_show为例：

1: /\* drivers/base/class.c, line 24 \*/

2: #define to\_class\_attr(\_attr) container\_of(\_attr, struct class\_attribute, attr)

3:

4: static ssize\_t class\_attr\_show(struct kobject \*kobj, struct attribute \*attr,

5: char \*buf)

6: {

7:     struct class\_attribute \*class\_attr = to\_class\_attr(attr);

8:     struct subsys\_private \*cp = to\_subsys\_private(kobj);

9:     ssize\_t ret = -EIO;

10:

11:    if (class\_attr->show)

12:    ret = class\_attr->show(cp->class, class\_attr, buf);

13:    return ret;

14: }

该接口使用container\_of从struct attribute类型的指针中取得一个class模块的自定义指针：struct class\_attribute，该指针中包含了class模块自身的show和store接口。下面是struct class\_attribute的声明：

1: /\* include/linux/device.h, line 399 \*/

2: struct class\_attribute {

3:     struct attribute attr;

4:     ssize\_t (\*show)(struct class \*class, struct class\_attribute \*attr,

5:                     char \*buf);

6:     ssize\_t (\*store)(struct class \*class, struct class\_attribute \*attr,

7:                     const char \*buf, size\_t count);

8:     const void \*(\*namespace)(struct class \*class,

9:                                 const struct class\_attribute \*attr);

10: };

因此，所有需要使用attribute的模块，都不会直接定义struct attribute变量，而是通过一个自定义的数据结构，该数据结构的一个成员是struct attribute类型的变量，并提供show和store回调函数。然后在该模块ktype所对应的struct sysfs\_ops变量中，实现该本模块整体的show和store函数，并在被调用时，转接到自定义数据结构（struct class\_attribute）中的show和store函数中。这样，每个atrribute文件，实际上对应到一个自定义数据结构变量中了。

# Linux设备模型(5) device和device driver

## 1. 前言

device和device driver是Linux驱动开发的基本概念。Linux kernel的思路很简单：驱动开发，就是要开发指定的软件（driver）以驱动指定的设备，所以kernel就为设备和驱动它的driver定义了两个数据结构，分别是device和device\_driver。因此本文将会围绕这两个数据结构，介绍Linux设备模型的核心逻辑，包括：

* 设备及设备驱动在kernel中的抽象、使用和维护；
* 设备及设备驱动的注册、加载、初始化原理；
* 设备模型在实际驱动开发过程中的使用方法。

注：在介绍device和device\_driver的过程中，会遇到很多额外的知识点，如Class、Bus、DMA、电源管理等等，这些知识点都很复杂，任何一个都可以作为一个单独的专题区阐述，因此本文不会深入解析它们，而会在后续的文章中专门描述。

## 2. struct device和struct device\_driver

在阅读Linux内核源代码时，通过核心数据结构，即可理解某个模块60%以上的逻辑，设备模型部分尤为明显。

在include/linux/device.h中，Linux内核定义了设备模型中最重要的两个数据结构，struct device和struct device\_driver。

* struct device

1: /\* include/linux/device.h, line 660 \*/

2: struct device {

3:     struct device \*parent;

4:

5:     struct device\_private \*p;

6:

7:     struct kobject kobj;

8:     const char \*init\_name; /\* initial name of the device \*/

9:     const struct device\_type \*type;

10:

11:    struct mutex mutex; /\* mutex to synchronize calls to its driver \*/

15:    struct bus\_type \*bus; /\* type of bus device is on \*/

16:    struct device\_driver \*driver; /\* which driver has allocated this device \*/

18:    void \*platform\_data; /\* Platform specific data, device

core doesn't touch it \*/

20:    struct dev\_pm\_info power;

21:    struct dev\_pm\_domain \*pm\_doamain;

22:

23: #ifdef CONFIG\_PINCTRL

24:    struct dev\_pin\_info \*pins;

25: #endif

26:

27: #ifdef CONFIG\_NUMA

28:    int numa\_node; /\* NUMA node this device is close to \*/

29: #endif

30:    u64 \*dma\_mask; /\* dma mask (if dma'able device) \*/

31:    u64 coherent\_dma\_mask;/\* Like dma\_mask, but for alloac\_coherent mappings as not all hardware supports 64 bit addresses for consistent allocations such descriptors. \*/

36:

37:    struct device\_dma\_parameters \*dma\_parms;

38:

39:    struct list\_head dma\_pools; /\* dma pools (if dma'able) \*/

40:

41:    struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent mem override \*/

43: #ifdef CONFIG\_CMA

44:    struct cma \*cma\_area; /\* contiguous memory area for dma allocations \*/

46: #endif

47:    /\* arch specific additions \*/

48:    struct dev\_archdata archdata;

49:

50:    struct device\_node \*of\_node; /\* associated device tree node \*/

51:    struct acpi\_dev\_node acpi\_node; /\* associated ACPI device node \*/

52:

53:    dev\_t devt; /\* dev\_t, creates the sysfs "dev" \*/

54:    u32 id; /\* device instance \*/

55:

56:    spinlock\_t devres\_lock;

57:    struct list\_head devres\_head;

58:

59:    struct klist\_node knode\_class;

60:    struct class \*class;

61:    const struct attribute\_group \*\*groups; /\* optional groups \*/

63:    void (\*release)(struct device \*dev);

64:    struct iommu\_group \*iommu\_group;

65: };

device结构很复杂（不过linux内核的开发人员素质是很高的，该接口的注释写的非常详细，感兴趣的同学可以参考内核源代码），这里将会选一些对理解设备模型非常关键的字段进行说明。

parent，该设备的父设备，一般是该设备所从属的bus、controller等设备。

p，一个用于struct device的私有数据结构指针，该指针中会保存子设备链表、用于添加到bus/driver/prent等设备中的链表头等，具体可查看源代码。

kobj，该数据结构对应的struct kobject。

init\_name，该设备的名称。

       注1：在设备模型中，名称是一个非常重要的变量，任何注册到内核中的设备，都必须有一个合法的名称，可以在初始化时给出，也可以由内核根据“bus name + device ID”的方式创造。

type，struct device\_type结构是新版本内核新引入的一个结构，它和struct device关系，非常类似[stuct kobj\_type和struct kobject](http://www.wowotech.net/linux_kenrel/kobject.html)之间的关系，后续会再详细说明。

bus，该device属于哪个总线（后续会详细描述）。

driver，该device对应的device driver。

platform\_data，一个指针，用于保存具体的平台相关的数据。具体的driver模块，可以将一些私有的数据，暂存在这里，需要使用的时候，再拿出来，因此设备模型并不关心该指针得实际含义。

power、pm\_domain，电源管理相关的逻辑，后续会由电源管理专题讲解。

pins，"PINCTRL”功能，暂不描述。

numa\_node，"NUMA”功能，暂不描述。

dma\_mask~archdata，DMA相关的功能，暂不描述。

devt，dev\_t是一个32位的整数，它由两个部分（Major和Minor）组成，在需要以设备节点的形式（字符设备和块设备）向用户空间提供接口的设备中，当作设备号使用。在这里，该变量主要用于在sys文件系统中，为每个具有设备号的device，创建/sys/dev/\* 下的对应目录，如下：

1|root@android:/storage/sdcard0 #ls /sys/dev/char/1\:                                                                       
1:1/  1:11/ 1:13/ 1:14/ 1:2/  1:3/  1:5/  1:7/  1:8/  1:9/   
1|root@android:/storage/sdcard0 #ls /sys/dev/char/1:1                                                                      
1:1/  1:11/ 1:13/ 1:14/  
1|root@android:/storage/sdcard0 # ls /sys/dev/char/1\:1   
/sys/dev/char/1:1

class，该设备属于哪个class。

groups，该设备的默认attribute集合。将会在设备注册时自动在sysfs中创建对应的文件。

* struct device\_driver

1: /\* include/linux/device.h, line 213 \*/

2: struct device\_driver {

3:     const char \*name;

4:     struct bus\_type \*bus;

5:

6:     struct module \*owner;

7:     const char \*mod\_name; /\* used for built-in modules \*/

8:

9:     bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

10:

11:    const struct of\_device\_id \*of\_match\_table;

12:    const struct acpi\_device\_id \*acpi\_match\_table;

13:

14:    int (\*probe) (struct device \*dev);

15:    int (\*remove) (struct device \*dev);

16:    void (\*shutdown) (struct device \*dev);

17:    int (\*suspend) (struct device \*dev, pm\_message\_t state);

18:    int (\*resume) (struct device \*dev);

19:    const struct attribute\_group \*\*groups;

20:

21:    const struct dev\_pm\_ops \*pm;

22:

23:    struct driver\_private \*p;

24: };

device\_driver就简单多了（在早期的内核版本中driver的数据结构为"struct driver”，不知道从哪个版本开始，就改成device\_driver了）：

name，该driver的名称。和device结构一样，该名称非常重要，后面会再详细说明。

bus，该driver所驱动设备的总线设备。为什么driver需要记录总线设备的指针呢？因为内核要保证在driver运行前，设备所依赖的总线能够正确初始化。

owner、mod\_name，內核module相关的变量，暂不描述。

suppress\_bind\_attrs，是不在sysfs中启用bind和unbind attribute，如下：root@android:/storage/sdcard0 # ls /sys/bus/platform/drivers/switch-gpio/                                                    
bind   uevent unbind  
在kernel中，bind/unbind是从用户空间手动的为driver绑定/解绑定指定的设备的机制。这种机制是在bus.c中完成的，后面会详细解释。

probe、remove，这两个接口函数用于实现driver逻辑的开始和结束。Driver是一段软件code，因此会有开始和结束两个代码逻辑，就像PC程序，会有一个main函数，main函数的开始就是开始，return的地方就是结束。而内核driver却有其特殊性：在设备模型的结构下，只有driver和device同时存在时，才需要开始执行driver的代码逻辑。这也是probe和remove两个接口名称的由来：检测到了设备和移除了设备（就是为热拔插起的！）。

shutdown、suspend、resume、pm，电源管理相关的内容，会在电源管理专题中详细说明。

groups，和struct device结构中的同名变量类似，driver也可以定义一些默认attribute，这样在将driver注册到内核中时，内核设备模型部分的代码（driver/base/driver.c）会自动将这些attribute添加到sysfs中。

p，driver core的私有数据指针，其它模块不能访问。

## 3. 设备模型框架下驱动开发的基本步骤

在设备模型框架下，设备驱动的开发是一件很简单的事情，主要包括2个步骤：

步骤1：分配一个struct device类型的变量，填充必要的信息后，把它注册到内核中。

步骤2：分配一个struct device\_driver类型的变量，填充必要的信息后，把它注册到内核中。

这两步完成后，内核会在合适的时机（后面会讲），调用struct device\_driver变量中的probe、remove、suspend、resume等回调函数，从而触发或者终结设备驱动的执行。而所有的驱动程序逻辑，都会由这些回调函数实现，此时，驱动开发者眼中便不再有“设备模型”，转而只关心驱动本身的实现。

以上两个步骤的补充说明：

1. 一般情况下，Linux驱动开发很少直接使用device和device\_driver，因为内核在它们之上又封装了一层，如soc device、platform device等等，而这些层次提供的接口更为简单、易用（也正是因为这个原因，本文并不会过多涉及device、device\_driver等模块的实现细节）。

2. 内核提供很多struct device结构的操作接口（具体可以参考include/linux/device.h和drivers/base/core.c的代码），主要包括初始化（device\_initialize）、注册到内核（device\_register）、分配存储空间+初始化+注册到内核（device\_create）等等，可以根据需要使用。

3. device和device\_driver必须具备相同的名称，内核才能完成匹配操作，进而调用device\_driver中的相应接口。这里的同名，作用范围是同一个bus下的所有device和device\_driver。

4. device和device\_driver必须挂载在一个bus之下，该bus可以是实际存在的，也可以是虚拟的。

5. driver开发者可以在struct device变量中，保存描述设备特征的信息，如寻址空间、依赖的GPIOs等，因为device指针会在执行probe等接口时传入，这时driver就可以根据这些信息，执行相应的逻辑操作了。

## 4. 设备驱动probe的时机

所谓的"probe”，是指在Linux内核中，如果存在相同名称的device和device\_driver（注：还存在其它方式，我们先不关注了），内核就会执行device\_driver中的probe回调函数，而该函数就是所有driver的入口，可以执行诸如硬件设备初始化、字符设备注册、设备文件操作ops注册等动作（"remove”是它的反操作，发生在device或者device\_driver任何一方从内核注销时，其原理类似，就不再单独说明了）。

设备驱动prove的时机有如下几种（分为自动触发和手动触发）：

* 将struct device类型的变量注册到内核中时自动触发（device\_register，device\_add，device\_create\_vargs，device\_create）
* 将struct device\_driver类型的变量注册到内核中时自动触发（driver\_register）
* 手动查找同一bus下的所有device\_driver，如果有和指定device同名的driver，执行probe操作（device\_attach）
* 手动查找同一bus下的所有device，如果有和指定driver同名的device，执行probe操作（driver\_attach）
* 自行调用driver的probe接口，并在该接口中将该driver绑定到某个device结构中----即设置dev->driver（device\_bind\_driver）

注2：probe动作实际是由bus模块（会在下一篇文章讲解）实现的，这不难理解：device和device\_driver都是挂载在bus这根线上，因此只有bus最清楚应该为哪些device、哪些driver配对。

注3：每个bus都有一个drivers\_autoprobe变量，用于控制是否在device或者driver注册时，自动probe。该变量默认为1（即自动probe），bus模块将它开放到sysfs中了，因而可在用户空间修改，进而控制probe行为。

## 5. 其它杂项

### 5.1 device\_attribute和driver\_attribute

在"[Linux设备模型(4)\_sysfs](http://www.wowotech.net/linux_kenrel/dm_sysfs.html)”中，我们有讲到，大多数时候，attribute文件的读写数据流为：vfs---->sysfs---->kobject---->attibute---->kobj\_type---->sysfs\_ops---->xxx\_attribute，其中kobj\_type、sysfs\_ops和xxx\_attribute都是由包含kobject的上层数据结构实现。

Linux内核中关于该内容的例证到处都是，device也不无例外的提供了这种例子，如下：

1: /\* driver/base/core.c, line 118 \*/

2: static ssize\_t dev\_attr\_show(struct kobject \*kobj, struct attribute \*attr,

3: char \*buf)

4: {

5:     struct device\_attribute \*dev\_attr = to\_dev\_attr(attr);

6:     struct device \*dev = kobj\_to\_dev(kobj);

7:     ssize\_t ret = -EIO;

8:

9:     if (dev\_attr->show)

10:        ret = dev\_attr->show(dev, dev\_attr, buf);

11:        if (ret >= (ssize\_t)PAGE\_SIZE) {

12:            print\_symbol("dev\_attr\_show: %s returned bad count\n",

13:                        (unsigned long)dev\_attr->show);

14:    }

15:    return ret;

16: }

17:

18: static ssize\_t dev\_attr\_store(struct kobject \*kobj, struct attribute \*attr,

19: const char \*buf, size\_t count)

20: {

21:    struct device\_attribute \*dev\_attr = to\_dev\_attr(attr);

22:    struct device \*dev = kobj\_to\_dev(kobj);

23:    ssize\_t ret = -EIO;

24:

25:    if (dev\_attr->store)

26:        ret = dev\_attr->store(dev, dev\_attr, buf, count);

27:    return ret;

28: }

29:

30: static const struct sysfs\_ops dev\_sysfs\_ops = {

31:    .show = dev\_attr\_show,

32:    .store = dev\_attr\_store,

33: };

34:

35: /\* driver/base/core.c, line 243 \*/

36: static struct kobj\_type device\_ktype = {

37:    .release = device\_release,

38:    .sysfs\_ops = &dev\_sysfs\_ops,

39:    .namespace = device\_namespace,

40: };

41:

42: /\* include/linux/device.h, line 478 \*/

43: /\* interface for exporting device attributes \*/

44: struct device\_attribute {

45:    struct attribute attr;

46:    ssize\_t (\*show)(struct device \*dev, struct device\_attribute \*attr,

47:                    char \*buf);

48:    ssize\_t (\*store)(struct device \*dev, struct device\_attribute \*attr,

49:                    const char \*buf, size\_t count);

50: };

至于driver的attribute，则要简单的多，其数据流为：vfs---->sysfs---->kobject---->attribute---->driver\_attribute，如下：

1: /\* include/linux/device.h, line 247 \*/

2: /\* sysfs interface for exporting driver attributes \*/

3:

4: struct driver\_attribute {

5:     struct attribute attr;

6:     ssize\_t (\*show)(struct device\_driver \*driver, char \*buf);

7:     ssize\_t (\*store)(struct device\_driver \*driver, const char \*buf,

8:                     size\_t count);

9: };

10:

11: #define DRIVER\_ATTR(\_name, \_mode, \_show, \_store) \

12: struct driver\_attribute driver\_attr\_##\_name = \

13:    \_\_ATTR(\_name, \_mode, \_show, \_store)

### 5.2 device\_type

device\_type是内嵌在struct device结构中的一个数据结构，用于指明设备的类型，并提供一些额外的辅助功能。它的形式如下：

1: /\* include/linux/device.h, line 467 \*/

2: struct device\_type {

3:     const char \*name;

4:     const struct attribute\_group \*\*groups;

5:     int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

6:     char \*(\*devnode)(struct device \*dev, umode\_t \*mode,

7:                     kuid\_t \*uid, kgid\_t \*gid);

8:     void (\*release)(struct device \*dev);

9:

10:    const struct dev\_pm\_ops \*pm;

11: };

device\_type的功能包括：

* name表示该类型的名称，当该类型的设备添加到内核时，内核会发出"DEVTYPE=‘name’”类型的uevent，告知用户空间某个类型的设备available了
* groups，该类型设备的公共attribute集合。设备注册时，会同时注册这些attribute。这就是面向对象中“继承”的概念
* uevent，同理，所有相同类型的设备，会有一些共有的uevent需要发送，由该接口实现
* devnode，devtmpfs有关的内容，暂不说明
* release，如果device结构没有提供release接口，就要查询它所属的type是否提供。用于释放device变量所占的空间

### 5.3 root device

在sysfs中有这样一个目录：/sys/devices，系统中所有的设备，都归集在该目录下。有些设备，是通过device\_register注册到Kernel并体现在/sys/devices/xxx/下。但有时候我们仅仅需要在/sys/devices/下注册一个目录，该目录不代表任何的实体设备，这时可以使用下面的接口：

1: /\* include/linux/device.h, line 859 \*/

2: /\*

3: \* Root device objects for grouping under /sys/devices

4: \*/

5: extern struct device \*\_\_root\_device\_register(const char \*name,

6: struct module \*owner);

7:

8: /\*

9: \* This is a macro to avoid include problems with THIS\_MODULE,

10: \* just as per what is done for device\_schedule\_callback() above.

11: \*/

12: #define root\_device\_register(name) \

13: \_\_root\_device\_register(name, THIS\_MODULE)

14:

15: extern void root\_device\_unregister(struct device \*root);

该接口会调用device\_register函数，向内核中注册一个设备，但是（你也想到了），没必要注册与之对应的driver（顺便提一下，内核中有很多不需要driver的设备，这是之一）。

# Linux设备模型(6) bus

## 1. 概述

在Linux设备模型中，Bus（总线）是一类特殊的设备，它是连接处理器和其它设备之间的通道（channel）。为了方便设备模型的实现，内核规定，系统中的每个设备都要连接在一个Bus上，这个Bus可以是一个内部Bus、虚拟Bus或者Platform Bus。

内核通过struct bus\_type结构，抽象Bus，它是在include/linux/device.h中定义的。本文会围绕该结构，描述Linux内核中Bus的功能，以及相关的实现逻辑。最后，会简单的介绍一些标准的Bus（如Platform），介绍它们的用途、它们的使用场景。

## 2. 功能说明

按照老传统，描述功能前，先介绍一下该模块的一些核心数据结构，对bus模块而言，核心数据结构就是struct bus\_type，另外，还有一个sub system相关的结构，会一并说明。

### 2.1 struct bus\_type

1: /\* inlcude/linux/device.h, line 93 \*/

2: struct bus\_type {

3:     const char \*name;

4:     const char \*dev\_name;

5:     struct device \*dev\_root;

6:     struct bus\_attribute \*bus\_attrs;

7:     struct device\_attribute \*dev\_attrs;

8:     struct driver\_attribute \*drv\_attrs;

9:

10:    int (\*match)(struct device \*dev, struct device\_driver \*drv);

11:    int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

12:    int (\*probe)(struct device \*dev);

13:    int (\*remove)(struct device \*dev);

14:    void (\*shutdown)(struct device \*dev);

15:

16:    int (\*suspend)(struct device \*dev, pm\_message\_t state);

17:    int (\*resume)(struct device \*dev);

18:

19:    const struct dev\_pm\_ops \*pm;

20:

21:    struct iommu\_ops \*iommu\_ops;

22:

23:    struct subsys\_private \*p;

24:    struct lock\_class\_key lock\_key;

25: };

name，该bus的名称，会在sysfs中以目录的形式存在，如platform bus在sysfs中表现为"/sys/bus/platform”。

dev\_name，该名称和"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”所讲述的struct device结构中的init\_name有关。对有些设备而言（例如批量化的USB设备），设计者根本就懒得为它起名字的，而内核也支持这种懒惰，允许将设备的名字留空。这样当设备注册到内核后，设备模型的核心逻辑就会用"bus->dev\_name+device ID”的形式，为这样的设备生成一个名称。

bus\_attrs、dev\_attrs、drv\_attrs，一些默认的attribute，可以在bus、device或者device\_driver添加到内核时，自动为它们添加相应的attribute。

dev\_root，根据内核的注释，dev\_root设备为bus的默认父设备（Default device to use as the parent），但在内核实际实现中，只和一个叫sub system的功能有关，随后会介绍。

match，一个由具体的bus driver实现的回调函数。当任何属于该Bus的device或者device\_driver添加到内核时，内核都会调用该接口，如果新加的device或device\_driver匹配上了自己的另一半的话，该接口要返回非零值，此时Bus模块的核心逻辑就会执行后续的处理。

uevent，一个由具体的bus driver实现的回调函数。当任何属于该Bus的device，发生添加、移除或者其它动作时，Bus模块的核心逻辑就会调用该接口，以便bus driver能够修改环境变量。

probe、remove，这两个回调函数，和[device\_driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)中的非常类似，但它们的存在是非常有意义的。可以想象一下，如果需要probe（其实就是初始化）指定的device话，需要保证该device所在的bus是被初始化过、确保能正确工作的。这就要就在执行device\_driver的probe前，先执行它的bus的probe。remove的过程相反。  
注1：并不是所有的bus都需要probe和remove接口的，因为对有些bus来说（例如platform bus），它本身就是一个虚拟的总线，无所谓初始化，直接就能使用，因此这些bus的driver就可以将这两个回调函数留空。

shutdown、suspend、resume，和probe、remove的原理类似，电源管理相关的实现，暂不说明。

pm，电源管理相关的逻辑，暂不说明。

iommu\_ops，暂不说明。

p，一个struct subsys\_private类型的指针，后面我们会用一个小节说明。

### 2.2 struct subsys\_private

该结构和[device\_driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)中的struct driver\_private类似，在"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”章节中有提到它，但没有详细说明。

要说明subsys\_private的功能，让我们先看一下该结构的定义：

1: /\* drivers/base/base.h, line 28 \*/

2: struct subsys\_private {

3:     struct kset subsys;

4:     struct kset \*devices\_kset;

5:     struct list\_head interfaces;

6:     struct mutex mutex;

7:

8:     struct kset \*drivers\_kset;

9:     struct klist klist\_devices;

10:    struct klist klist\_drivers;

11:    struct blocking\_notifier\_head bus\_notifier;

12:    unsigned int drivers\_autoprobe:1;

13:    struct bus\_type \*bus;

14:

15:    struct kset glue\_dirs;

16:    struct class \*class;

17: };

看到结构内部的字段，就清晰多了，没事不要乱起名字嘛！什么subsys啊，看的晕晕的！不过还是试着先理解一下为什么起名为subsys吧：

按理说，这个结构就是集合了一些bus模块需要使用的私有数据，例如kset啦、klist啦等等，命名为bus\_private会好点（就像device\_driver模块一样）。不过为什么内核没这么做呢？看看include/linux/device.h中的struct class结构（我们会在下一篇文章中介绍class）就知道了，因为class结构中也包含了一个一模一样的struct subsys\_private指针，看来class和bus很相似啊。

想到这里，就好理解了，无论是bus，还是class，还是我们会在后面看到的一些虚拟的子系统，它都构成了一个“子系统（sub-system）”，该子系统会包含形形色色的device或device\_driver，就像一个独立的王国一样，存在于内核中。而这些子系统的表现形式，就是/sys/bus（或/sys/class，或其它）目录下面的子目录，每一个子目录，都是一个子系统（如/sys/bus/spi/）。

好了，我们回过头来看一下struct subsys\_private中各个字段的解释：

subsys、devices\_kset、drivers\_kset是三个kset，由"[Linux设备模型(2)\_Kobject](http://www.wowotech.net/linux_kenrel/kobject.html)”中对kset的描述可知，kset是一个特殊的kobject，用来集合相似的kobject，它在sysfs中也会以目录的形式体现。其中subsys，代表了本bus（如/sys/bus/spi），它下面可以包含其它的kset或者其它的kobject；devices\_kset和drivers\_kset则是bus下面的两个kset（如/sys/bus/spi/devices和/sys/bus/spi/drivers），分别包括本bus下所有的device和device\_driver。

interface是一个list head，用于保存该bus下所有的interface。有关interface的概念后面会详细介绍。

klist\_devices和klist\_drivers是两个链表，分别保存了本bus下所有的device和device\_driver的指针，以方便查找。

drivers\_autoprobe，用于控制该bus下的drivers或者device是否自动probe，"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”中有提到。

bus和class指针，分别保存上层的bus或者class指针。

### 2.3 功能总结

根据上面的核心数据结构，可以总结出bus模块的功能包括：

* bus的注册和注销
* 本bus下有device或者device\_driver注册到内核时的处理
* 本bus下有device或者device\_driver从内核注销时的处理
* device\_drivers的probe处理
* 管理bus下的所有device和device\_driver

## 3. 内部执行逻辑分析

### 3.1 bus的注册

bus的注册是由bus\_register接口实现的，该接口的原型是在include/linux/device.h中声明的，并在drivers/base/bus.c中实现，其原型如下：

1: /\* include/linux/device.h, line 118 \*/

2: extern int \_\_must\_check bus\_register(struct bus\_type \*bus);

该功能的执行逻辑如下：

* 为bus\_type中struct subsys\_private类型的指针分配空间，并更新priv->bus和bus->p两个指针为正确的值
* 初始化priv->subsys.kobj的name、kset、ktype等字段，启动name就是该bus的name（它会体现在sysfs中），kset和ktype由bus模块实现，分别为bus\_kset和bus\_ktype
* 调用kset\_register将priv->subsys注册到内核中，该接口同时会向sysfs中添加对应的目录（如/sys/bus/spi）
* 调用bus\_create\_file向bus目录下添加一个uevent attribute（如/sys/bus/spi/uevent）
* 调用kset\_create\_and\_add分别向内核添加devices和device\_drivers kset，同时会体现在sysfs中
* 初始化priv指针中的mutex、klist\_devices和klist\_drivers等变量
* 调用add\_probe\_files接口，在bus下添加drivers\_probe和drivers\_autoprobe两个attribute（如/sys/bus/spi/drivers\_probe和/sys/bus/spi/drivers\_autoprobe），其中drivers\_probe允许用户空间程序主动出发指定bus下的device\_driver的probe动作，而drivers\_autoprobe控制是否在device或device\_driver添加到内核时，自动执行probe
* 调用bus\_add\_attrs，添加由bus\_attrs指针定义的bus的默认attribute，这些attributes最终会体现在/sys/bus/xxx目录下

### 3.2 device和device\_driver的添加

我们有在"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”中讲过，内核提供了device\_register和driver\_register两个接口，供各个driver模块使用。而这两个接口的核心逻辑，是通过bus模块的bus\_add\_device和bus\_add\_driver实现的，下面我们看看这两个接口的处理逻辑。

这两个接口都是在drivers/base/base.h中声明，在drivers/base/bus.c中实现，其原型为：

1: /\* drivers/base/base.h, line 106 \*/

2: extern int bus\_add\_device(struct device \*dev);

3:

4: /\* drivers/base/base.h, line 110 \*/

5: extern int bus\_add\_driver(struct device\_driver \*drv);

bus\_add\_device的处理逻辑：

* 调用内部的device\_add\_attrs接口，将由bus->dev\_attrs指针定义的默认attribute添加到内核中，它们会体现在/sys/devices/xxx/xxx\_device/目录中
* 调用sysfs\_create\_link接口，将该device在sysfs中的目录，链接到该bus的devices目录下，例如：

xxx# ls /sys/bus/spi/devices/spi1.0 -l                                                          
lrwxrwxrwx root     root              2014-04-11 10:46 spi1.0 -> ../../../devices/platform/s3c64xx-spi.1/spi\_master/spi1/spi1.0  
其中/sys/devices/…/spi1.0，为该device在sysfs中真正的位置，而为了方便管理，内核在该设备所在的bus的xxx\_bus/devices目录中，创建了一个符号链接

* 调用sysfs\_create\_link接口，在该设备的sysfs目录中（如/sys/devices/platform/alarm/）中，创建一个指向该设备所在bus目录的链接，取名为subsystem，例如：

xxx # ls /sys/devices/platform/alarm/subsystem -l                                                  
lrwxrwxrwx root     root              2014-04-11 10:28 subsystem -> ../../../bus/platform

* 最后，毫无疑问，要把该设备指针保存在bus->priv->klist\_devices中

bus\_add\_driver的处理逻辑：

* 为该driver的struct driver\_private指针（priv）分配空间，并初始化其中的priv->klist\_devices、priv->driver、priv->kobj.kset等变量，同时将该指针保存在device\_driver的p处
* 将driver的kset（priv->kobj.kset）设置为bus的drivers kset（bus->p->drivers\_kset），这就意味着所有driver的kobject都位于bus->p->drivers\_kset之下（寄/sys/bus/xxx/drivers目录下）
* 以driver的名字为参数，调用kobject\_init\_and\_add接口，在sysfs中注册driver的kobject，体现在/sys/bus/xxx/drivers/目录下，如/sys/bus/spi/drivers/spidev
* 将该driver保存在bus的klist\_drivers链表中，并根据drivers\_autoprobe的值，选择是否调用driver\_attach进行probe
* 调用driver\_create\_file接口，在sysfs的该driver的目录下，创建uevent attribute
* 调用driver\_add\_attrs接口，在sysfs的该driver的目录下，创建由bus->drv\_attrs指针定义的默认attribute
* 同时根据suppress\_bind\_attrs标志，决定是否在sysfs的该driver的目录下，创建bind和unbind attribute（具体可参考"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”中的介绍）

### 3.3 driver的probe

我们在"[Linux设备模型(5)\_device和device driver](http://www.wowotech.net/linux_kenrel/device_and_driver.html)”中，我们已经介绍过driver的probe时机及过程，其中大部分的逻辑会依赖bus模块的实现，主要为bus\_probe\_device和driver\_attach接口。同样，这两个接口都是在drivers/base/base.h中声明，在drivers/base/bus.c中实现。

这两个结构的行为类似，逻辑也很简单，既：搜索所在的bus，比对是否有同名的device\_driver（或device），如果有并且该设备没有绑定Driver（注：这一点很重要，通过它，可以使同一个Driver，驱动相同名称的多个设备，后续在Platform设备的描述中会提及）则调用device\_driver的probe接口。

## 4. 杂项

### 4.1 再谈Subsystem

在旧的Linux内核版本中（以蜗蜗使用的linux2.6.23版本的内核为例），sysfs下所有的顶层目录（包括一些二级目录）都是以调用subsystem\_register接口，以sub-system的形式注册到内核的，如：

/sys/bus/

/sys/devices/

/sys/devices/system/

/sys/block

/sys/kernel/

/sys/slab/

…

那时的subsystem\_register的实现很简单，就是调用kset\_register，创建一个kset。我们知道，kset就是一堆kobject的集合，并会在sysfs中以目录的形式呈现出来。

在新版本的内核中（如[“Linux内核分析”系列文章](http://www.wowotech.net/linux_kenrel/11.html)所参考的linux3.10.29），subsystem的实现有了很大变化，例如：去掉了subsystem\_register接口（但为了兼容/sys/device/system子系统，在drivers/base/bus.c中，增加了一个subsys\_register的内部接口，用于实现相应的功能）。根据这些变化，现在注册subsystem有两种方式：

方式1，在各自的初始化函数中，调用kset\_create\_and\_add接口，创建对应的子系统，包括：

* bus子系统，/sys/bus/，buses\_init（drivers/base/bus.c）
* class子系统，/sys/class
* kernel子系统，/sys/kernel
* firmware子系统，/sys/firmware
* 等等

其中bus子系统就是本文所讲的Bus模块，而其它的，我们会在后续的文章中陆续讲述。这个方式和旧版本内核使用kset\_register接口的方式基本一样。

方式二，在bus模块中，利用subsys\_register接口，封装出两个API：subsys\_system\_register和subsys\_virtual\_register，分别用于注册system设备(/sys/devices/system/\*)和virtual设备(/sys/devices/virtual/\*)。 而该方式和方式一的区别是：它不仅仅创建了sysfs中的目录，同时会注册同名的bus和device。

### 4.2 system/virtual/platform

在Linux内核中，有三种比较特殊的bus（或者是子系统），分别是system bus、virtual bus和platform bus。它们并不是一个实际存在的bus（像USB、I2C等），而是为了方便设备模型的抽象，而虚构的。

system bus是旧版内核提出的概念，用于抽象系统设备（如CPU、Timer等等）。而新版内核认为它是个坏点子，因为任何设备都应归属于一个普通的子系统（New subsystems should use plain subsystems, drivers/base/bus.c, line 1264），所以就把它抛弃了（不建议再使用，它的存在只为兼容旧有的实现）。

virtaul bus是一个比较新的bus，主要用来抽象那些虚拟设备，所谓的虚拟设备，是指不是真实的硬件设备，而是用软件模拟出来的设备，例如虚拟机中使用的虚拟的网络设备（有关该bus的描述，可参考该链接处的解释：<https://lwn.net/Articles/326540/>）。

platform bus就比较普通，它主要抽象集成在CPU（SOC）中的各种设备。这些设备直接和CPU连接，通过总线寻址和中断的方式，和CPU交互信息。

我们会在后续的文章中，进一步分析这些特殊的bus，这里就暂时不详细描述了。

### ****4.3 subsys interface****

subsys interface是一个很奇怪的东西，除非有一个例子，否则很难理解。代码中是这样注释的：

Interfaces usually represent a specific functionality of a subsystem/class of devices.

字面上理解，它抽象了bus下所有设备的一些特定功能。

kernel使用struct subsys\_interface结构抽象subsys interface，并提供了subsys\_interface\_register/subsys\_interface\_unregister用于注册/注销subsys interface，bus下所有的interface都挂载在struct subsys\_private变量的“interface”链表上（具体可参考2.2小节的描述）。struct subsys\_interface的定义如下：

/\*\*

\* struct subsys\_interface - interfaces to device functions

\* @name: name of the device function

\* @subsys: subsytem of the devices to attach to

\* @node: the list of functions registered at the subsystem

\* @add\_dev: device hookup to device function handler

\* @remove\_dev: device hookup to device function handler

\*

\* Simple interfaces attached to a subsystem. Multiple interfaces can

\* attach to a subsystem and its devices. Unlike drivers, they do not

\* exclusively claim or control devices. Interfaces usually represent

d

\*/

**struct** subsys\_interface {

**const** **char** \*name;

**struct** bus\_type \*subsys;

**struct** list\_head node;

int (\*add\_dev)(**struct** device \*dev, **struct** subsys\_interface \*sif);

int (\*remove\_dev)(**struct** device \*dev, **d** subsys\_interface \*sif);

};

name，interface的名称。

subsys，interface所属的bus。

node，用于将interface挂到bus中。

add\_dev/remove\_dev，两个回调函数，subsys interface的核心功能。当bus下有设备增加或者删除的时候，bus core会调用它下面所有subsys interface的add\_dev或者remove\_dev回调。设计者可以在这两个回调函数中实现所需功能，例如绑定该“specific functionality”所对应的driver，等等。

subsys interface的实现逻辑比较简单，这里不再详细描述了，具体可参考“drivers/base/bus.c”中相应的代码。另外，后续分析cpufreq framework的时候，会遇到使用subsys interface的例子，到时候我们再进一步理解它的现实意义。