第十四章 linux设备模型

在2.5开发周期中，一个明确的目标是创建一个统一的内核设备模型。以前的内核没有一个单一的数据结构可以用来获取关于系统如何组装的信息。尽管缺乏这些信息，但事情在一段时间内还是运行得很好。然而，新系统的需求，它们更复杂的拓扑结构和对诸如电源管理等功能的支持，明确表明需要一个描述系统结构的一般抽象。

2.6设备模型提供了这种抽象。它现在在内核中用于支持各种任务，包括：

电源管理和系统关机 这需要理解系统的结构。例如，USB主机适配器在处理所有连接到该适配器的设备之前不能被关闭。设备模型使得可以按照正确的顺序遍历系统的硬件。

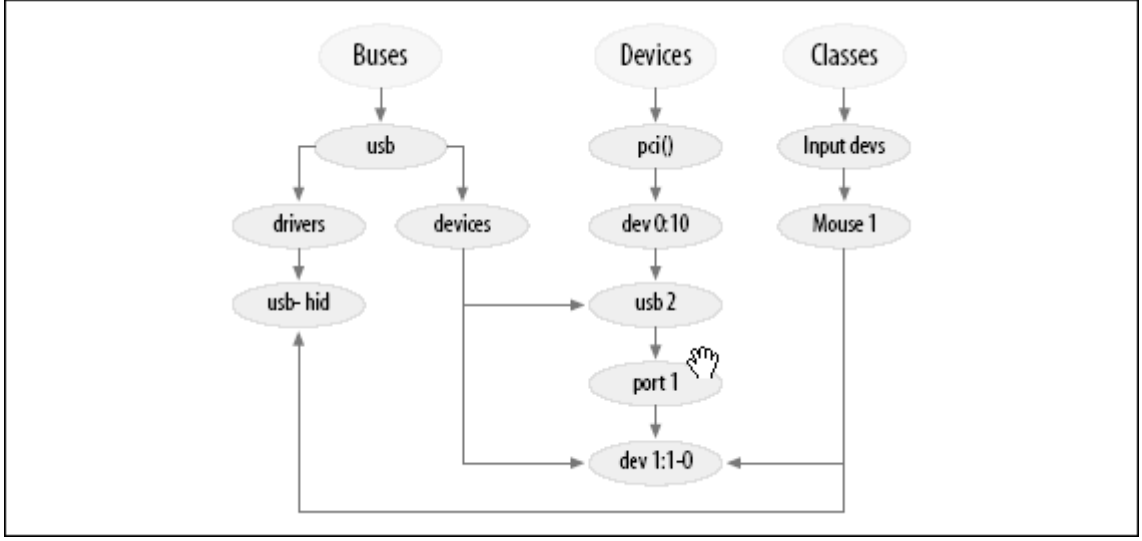
与用户空间的通信 sysfs虚拟文件系统的实现与设备模型紧密相连，并暴露出由它表示的结构。通过sysfs，以及因此通过设备模型，向用户空间提供关于系统的信息和改变操作参数的旋钮的提供越来越多。

热插拔设备 计算机硬件越来越动态；外设可以随用户的意愿来去。内核中用于处理和（尤其是）与用户空间通信关于设备插入和拔出的热插拔机制是通过设备模型管理的。

设备类 系统的许多部分对设备如何连接不感兴趣，但它们需要知道有哪些类型的设备可用。设备模型包括一种将设备分配给类的机制，这些类在更高的、功能级别上描述了这些设备，并允许它们从用户空间被发现。

对象生命周期 上面描述的许多功能，包括热插拔支持和sysfs，复杂化了内核中创建和操作对象的过程。设备模型的实现需要创建一套处理对象生命周期、它们之间的关系以及它们在用户空间中的表示的机制。

Linux设备模型是一个复杂的数据结构。例如，考虑图14-1，它显示了与USB鼠标相关的设备模型结构的一小部分（以简化的形式）。在图的中心，我们看到了核心“设备”树的一部分，显示了鼠标如何连接到系统。“总线”树跟踪每个总线上连接了什么，而“类”下的子树关心的是设备提供的功能，而不管它们如何连接。即使是一个简单系统的设备模型树也包含了像图中显示的那样的数百个节点；它是一个很难整体可视化的数据结构。



大部分情况下，Linux设备模型代码会处理所有这些考虑因素，而不会对驱动程序作者造成影响。它主要在后台运行；与设备模型的直接交互通常由总线级逻辑和各种其他内核子系统处理。因此，许多驱动程序作者可以完全忽略设备模型，并相信它会自我处理。

然而，有时候，理解设备模型是一件好事。有时设备模型会从其他层“泄露出来”；例如，通用DMA代码（我们在第15章中遇到）与struct device一起工作。你可能想要使用设备模型提供的一些功能，如kobjects提供的引用计数和相关特性。通过sysfs与用户空间的通信也是设备模型的功能；本章将解释这种通信是如何工作的。

然而，我们从底层开始介绍设备模型。设备模型的复杂性使得从高级视图开始理解它变得困难。我们希望，通过展示低级设备组件是如何工作的，我们可以为你准备好理解如何使用这些组件来构建更大结构的挑战。

对于许多读者来说，这一章可以被视为高级材料，第一次阅读时不需要阅读。然而，对于那些对Linux设备模型如何工作感兴趣的人，我们鼓励他们继续前进，因为我们将深入到低级细节。

14.1. Kobjects, Ksets 和 Subsystems

kobject是将设备模型组合在一起的基本结构。它最初被设想为一个简单的引用计数器，但随着时间的推移，它的责任和字段都增长了。现在由struct kobject及其支持代码处理的任务包括：

- 对象的引用计数 通常，当创建一个内核对象时，无法知道它将存在多长时间。跟踪这些对象的生命周期的一种方式是通过引用计数。当内核中没有代码持有对给定对象的引用时，该对象已经完成了其有用的生命周期，可以被删除。

- Sysfs表示 每个出现在sysfs中的对象下面都有一个kobject，它与内核交互以创建其可见的表示。

- 数据结构胶水 设备模型完全是一个极其复杂的数据结构，由多个层次结构组成，它们之间有许多链接。kobject实现了这个结构并将其组合在一起。

- 热插拔事件处理 kobject子系统处理生成的事件，通知用户空间系统上硬件的来去。

从前面的列表中，人们可能会得出kobject是一个复杂的结构的结论。这个结论是正确的。然而，通过一次查看一部分，可以理解这个结构以及它是如何工作的。

14.1.1. Kobject 基础

kobject具有类型struct kobject；它在<linux/kobject.h>中定义。该文件还包括与kobjects相关的许多其他结构的声明，当然，还有一个用于操作它们的函数的长列表。

14.1.1.1. 嵌入的 kobjects

在我们深入细节之前，值得花一点时间理解kobjects是如何使用的。如果你回顾一下kobjects处理的函数列表，你会看到它们都是代表其他对象执行的服务。换句话说，kobject本身的兴趣不大；它只存在于将更高级别的对象绑定到设备模型中。

因此，内核代码创建一个独立的kobject是很少见的（甚至是未知的）；相反，kobjects被用来控制对更大的，特定领域的对象的访问。为此，kobjects被发现嵌入在其他结构中。如果你习惯于以面向对象的方式思考事物，kobjects可以被看作是一个顶级的，抽象的类，其他类从中派生。kobject实现了一组功能，这些功能本身并不特别有用，但在其他对象中很好用。C语言不允许直接表达继承，所以必须使用其他技术——比如在另一个结构中嵌入一个结构。

作为一个例子，让我们回顾一下我们在第3章中遇到的struct cdev。在2.6.10内核中找到的那个结构看起来是这样的：

```c
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

如我们所见，cdev结构内嵌有一个kobject。如果你有一个这样的结构，找到其内嵌的kobject只是使用kobj字段的问题。然而，处理kobjects的代码通常有相反的问题：给定一个struct kobject指针，包含结构的指针是什么？你应该避免使用技巧（比如假设kobject在结构的开始处），而应该使用container_of宏（在第3章的“The open Method”部分中介绍）。所以，将一个名为kp的struct kobject的指针转换为嵌入在struct cdev中的方式是：

```c
struct cdev *device = container_of(kp, struct cdev,
kobj);
```

程序员通常定义一个简单的宏，用于将kobject指针“反向转换”为包含类型。

14.1.1.2. kobject 初始化

本书已经介绍了许多类型，它们具有在编译或运行时初始化的简单机制。然而，kobject的初始化稍微复杂一些，尤其是当它的所有功能都被使用时。无论kobject如何被使用，都必须执行一些步骤。

首先，只需将整个kobject设置为0，通常通过调用memset来实现。这种初始化通常作为将kobject嵌入的结构归零的一部分进行。如果不将kobject归零，通常会导致后续出现非常奇怪的崩溃；这不是你想跳过的步骤。

下一步是通过调用kobject_init()来设置一些内部字段：

```c
void kobject_init(struct kobject *kobj);
```

在其他事情中，kobject_init将kobject的引用计数设置为一。然而，调用kobject_init是不够的。Kobject的用户必须至少设置kobject的名称；这是在sysfs条目中使用的名称。如果你深入研究内核源码，你可以找到将字符串直接复制到kobject的name字段的代码，但应避免这种方法。相反，使用：

```c
int kobject_set_name(struct kobject *kobj, const char
*format, ...);
```

这个函数接受一个printk-style的可变参数列表。信不信由你，这个操作实际上可能会失败（它可能会尝试分配内存）；有责任心的代码应该检查返回值并做出相应的反应。

创建者应该直接或间接设置的其他kobject字段是ktype，kset和parent。我们将在本章后面讨论这些。

14.1.1.3. 引用计数的操作

kobject的一个关键功能是作为其嵌入对象的引用计数器。只要对象的引用存在，对象（和支持它的代码）就必须继续存在。操作kobject的引用计数的低级函数是：

```c
struct kobject *kobject_get(struct kobject *kobj);

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

调用kobject_get会增加kobject的引用计数，并返回一个指向kobject的指针。

当一个引用被释放时，调用kobject_put会减少引用计数，并可能释放对象。请记住，kobject_init将引用计数设置为一；所以当你创建一个kobject时，你应该确保当初始引用不再需要时，做出相应的kobject_put调用。

请注意，在许多情况下，kobject本身的引用计数可能不足以防止竞态条件。例如，kobject（及其包含的结构）的存在可能需要模块的持续存在，该模块创建了该kobject。当kobject仍在传递时，卸载该模块是不行的。这就是为什么我们上面看到的cdev结构包含一个struct module指针。struct cdev的引用计数实现如下：

```c
struct kobject *cdev_get(struct cdev *p)
{
    struct module *owner = p->owner;
    struct kobject *kobj;
    if (owner && !try_module_get(owner))
        return NULL;
    kobj = kobject_get(&p->kobj);
    if (!kobj)
        module_put(owner);
    return kobj;
}
```

创建对cdev结构的引用需要同时创建对拥有它的模块的引用。所以cdev_get使用try_module_get尝试增加该模块的使用计数。如果该操作成功，kobject_get被用来增加kobject的引用计数。当然，这个操作可能会失败，所以代码检查kobject_get的返回值，并在事情不顺利时释放对模块的引用。

14.1.1.4. 释放Release函数和 kobject 类型

讨论中还缺少一个重要的事情，那就是当kobject的引用计数达到0时会发生什么。创建kobject的代码通常不知道这将何时发生；如果它知道，那么首先使用引用计数就没有什么意义。即使是可预测的对象生命周期，在引入sysfs时也会变得更复杂；用户空间程序可以保留对kobject的引用（通过保持其关联的sysfs文件打开）一个任意的时间段。

最终的结果是，由kobject保护的结构不能在驱动程序生命周期的任何单一、可预测的点被释放，而是在必须准备在kobject的引用计数变为0的任何时刻运行的代码中。引用计数不受创建kobject的代码的直接控制。所以，当最后一个对其kobjects的引用消失时，该代码必须异步地得到通知。

这个通知是通过kobject的release方法完成的。通常，这个方法的形式如下：

```c
void my_object_release(struct kobject *kobj)

{

    struct my_object *mine = container_of(kobj, struct
my_object, kobj);

    /* Perform any additional cleanup on this object,
then... */

    kfree(mine);

}
```

一个重要的点不能过分强调：每个kobject必须有一个release方法，kobject必须持续存在（在一个一致的状态）直到该方法被调用。如果没有满足这些约束，代码就有缺陷。它冒着在对象仍在使用时释放对象的风险，或者在最后一个引用返回后未能释放对象。

有趣的是，release方法并不存储在kobject本身；相反，它与包含kobject的结构的类型相关联。这种类型是用类型为struct kobj_type的结构跟踪的，通常简单地称为“ktype”。这个结构看起来像下面这样：

```c
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

struct kobj_type中的release字段当然是指向这种类型的kobject的release方法的指针。我们将在本章后面回到其他两个字段（sysfs_ops和default_attrs）。

每个kobject都需要有一个关联的kobj_type结构。令人困惑的是，这个结构的指针可以在两个不同的地方找到。kobject结构本身包含一个字段（称为ktype），可以包含这个指针。然而，如果这个kobject是一个kset的成员，那么kobj_type指针是由那个kset提供的。（我们将在下一节查看ksets。）同时，宏：

```c
struct kobj_type *get_ktype(struct kobject *kobj);
```

找到给定kobject的kobj_type指针。

14.1.2. kobject 层次, kset, 和子系统

kobject结构通常用于将对象链接成一个层次结构，这个结构匹配被建模的子系统的结构。有两种独立的机制用于这种链接：父指针和ksets。

struct kobject中的parent字段是指向另一个kobject的指针——代表层次结构中的下一级。例如，如果一个kobject代表一个USB设备，它的父指针可能指示代表设备插入的集线器的对象。

父指针的主要用途是在sysfs层次结构中定位对象。我们将在“低级Sysfs操作”一节中看到这是如何工作的。

14.1.2.1. Ksets 对象

在许多方面，kset看起来像是kobj_type结构的扩展；kset是一个集合，其中嵌入了同一类型的结构中的kobjects。然而，虽然struct kobj_type关心的是对象的类型，但struct kset关心的是聚合和收集。这两个概念已经被分开，以便相同类型的对象可以出现在不同的集合中。

因此，kset的主要功能是包含；它可以被认为是kobjects的顶级容器类。实际上，每个kset内部都包含自己的kobject，它在许多方面可以被视为与kobject相同。值得注意的是，ksets总是在sysfs中表示；一旦一个kset被设置并添加到系统中，就会有一个sysfs目录为它。kobjects不一定会出现在sysfs中，但是每个是kset成员的kobject都在那里表示。

通常在创建对象时将kobject添加到kset中；这是一个两步过程。kobject的kset字段必须指向感兴趣的kset；然后应将kobject传递给：

```c
int kobject_add(struct kobject *kobj);
```

像往常一样，程序员应该意识到这个函数可能会失败（在这种情况下，它返回一个负的错误代码）并相应地做出反应。内核提供了一个便利函数：

```c
extern int kobject_register(struct kobject *kobj);
```

这个函数只是kobject_init和kobject_add的组合。

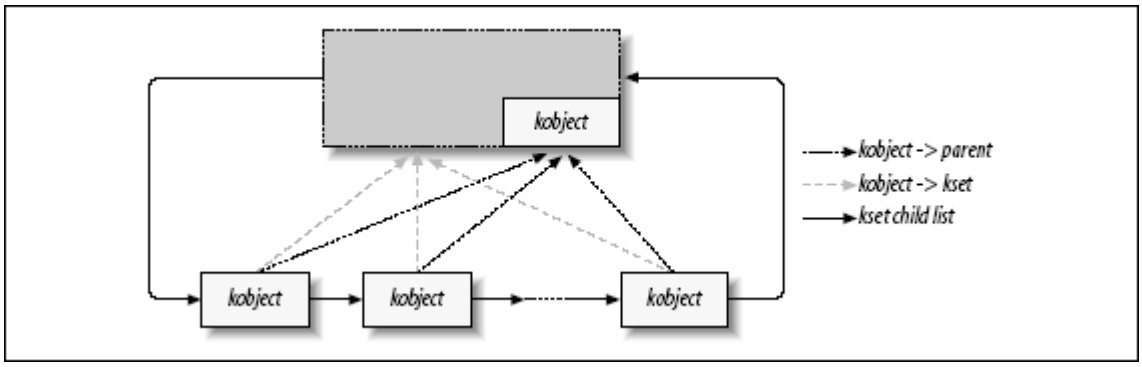
当一个kobject被传递给kobject_add时，它的引用计数会增加。毕竟，包含在kset中是对对象的引用。在某个时候，可能需要从kset中移除kobject以清除该引用；这是通过以下方式完成的：

```c
void kobject_del(struct kobject *kobj);
```

还有一个kobject_unregister函数，它是kobject_del和kobject_put的组合。

kset将其子对象保持在一个标准的内核链接列表中。在几乎所有的情况下，包含的kobjects也有指向kset（或者严格地说，它的嵌入式kobject）的指针在他们的父字段中。所以，通常，一个kset和它的kobjects看起来像你在图14-2中看到的那样。请记住：

- 图中的所有包含的kobjects实际上都嵌入在一些其他类型中，可能甚至是其他ksets。
- 不要求kobject的父对象是包含它的kset（尽管任何其他组织都会很奇怪和罕见）。



14.1.2.2. ksets 之上的操作

对于初始化和设置，ksets有一个与kobjects非常相似的接口。存在以下函数：

```c
void kset_init(struct kset *kset);

int kset_add(struct kset *kset);

int kset_register(struct kset *kset);

void kset_unregister(struct kset *kset);
```

在大多数情况下，这些函数只是在kset的嵌入式kobject上调用类似的kobject_函数。

要管理ksets的引用计数，情况大致相同：

```c
struct kset *kset_get(struct kset *kset);

void kset_put(struct kset *kset);
```

kset也有一个名字，它存储在嵌入的kobject中。所以，如果你有一个叫做my_set的kset，你可以用以下方式设置它的名字：

```c
kobject_set_name(&my_set→kobj, "The name");
```

ksets还有一个指针（在ktype字段中）指向描述它包含的kobjects的kobj_type结构。这种类型优先于kobject本身的ktype字段使用。因此，在典型的使用中，struct kobject中的ktype字段被留空，因为kset内部的同一字段是实际使用的。

最后，kset包含一个子系统指针（称为subsys）。所以现在是时候谈谈子系统了。

14.1.2.3. 子系统Subsystems

子系统是内核整体的高级部分的表示。子系统通常（但不总是）出现在sysfs层次结构的顶部。内核中的一些示例子系统包括block_subsys（/sys/block，用于块设备），devices_subsys（/sys/devices，核心设备层次结构），以及内核已知的每种总线类型的特定子系统。驱动程序作者几乎从不需要创建新的子系统；如果你感到想这样做，再想想。你可能最终想要的是添加一个新的类，如“类”一节中所讨论的。

子系统由一个简单的结构表示：

```c
struct subsystem {
    struct kset kset;
    struct rw_semaphore rwsem;
};
```

因此，子系统实际上只是一个围绕kset的包装器，其中还包含一个信号量。每个kset必须属于一个子系统。子系统成员资格有助于确定kset在层次结构中的位置，但更重要的是，子系统的rwsem信号量用于序列化对kset的内部链接列表的访问。这种成员资格由struct kset中的subsys指针表示。因此，可以从kset的结构中找到每个kset的包含子系统，但不能直接从子系统结构中找到包含在子系统中的多个ksets。

子系统通常用一个特殊的宏声明：

```c
decl_subsys(name, struct kobj_type *type,
            struct kset_hotplug_ops *hotplug_ops);
```

这个宏创建一个struct subsystem，其名称由给定的宏的名称和_subsys组成。宏还使用给定的类型和hotplug_ops初始化内部kset。（我们在本章后面讨论热插拔操作。）

子系统有通常的设置和拆卸函数列表：

```c
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

这些操作大部分只是对子系统的kset进行操作。

14.2. 低级 sysfs 操作

kobjects是sysfs虚拟文件系统背后的机制。对于在sysfs中找到的每个目录，内核中都有一个kobject潜伏在某处。每个感兴趣的kobject也导出一个或多个属性，这些属性在

kobject的sysfs目录中以包含内核生成信息的文件形式出现。本节将研究kobjects和sysfs在低级别如何交互。

与sysfs一起工作的代码应包含<linux/sysfs.h>。

让kobject在sysfs中显示只是调用kobject_add的问题。我们已经看到该函数作为将kobject添加到kset的方式；在sysfs中创建条目也是它的工作的一部分。关于如何创建sysfs条目，有几件事值得了解：

- kobjects的sysfs条目总是目录，所以调用kobject_add会在sysfs中创建一个目录。通常，该目录包含一个或多个属性；我们马上就会看到如何指定属性。
- 分配给kobject的名称（使用kobject_set_name）是用于sysfs目录的名称。因此，在sysfs层次结构的同一部分出现的kobjects必须有唯一的名称。分配给kobjects的名称也应该是合理的文件名：它们不能包含斜杠字符，强烈建议不使用空格。
- sysfs条目位于与kobject的父指针对应的目录中。如果在调用kobject_add时parent为NULL，那么它将被设置为新kobject的kset中的kobject；因此，sysfs层次结构通常与使用ksets创建的内部层次结构匹配。如果parent和kset都为NULL，那么sysfs目录将在顶级创建，这几乎肯定不是你想要的。

使用我们到目前为止描述的机制，我们可以使用kobject在sysfs中创建一个空目录。通常，你想做一些比这更有趣的事情，所以现在是时候看看属性的实现了。

14.2.1. 缺省属性

在创建时，每个kobject都被赋予一组默认属性。这些属性是通过kobj_type结构体指定的。记住，该结构体看起来像这样：

```c
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

default_attrs字段列出了每个此类型的kobject要创建的属性，sysfs_ops提供了实现这些属性的方法。我们从default_attrs开始，它指向属性结构的指针数组：

```c
struct attribute {
    char *name;
    struct module *owner;
    mode_t mode;
};
```

在这个结构中，name是属性的名称（如它在kobject的sysfs目录中出现），owner是指向负责实现此属性的模块（如果有）的指针，mode是要应用于此属性的保护位。对于只读属性，mode通常为S_IRUGO；如果属性是可写的，你可以加入S_IWUSR以仅给root写入权限（模式的宏在<linux/stat.h>中定义）。default_attrs列表中的最后一个条目必须填充零。

default_attrs数组说明了属性是什么，但并没有告诉sysfs如何实际实现这些属性。这个任务落在kobj_type→sysfs_ops字段上，它指向一个定义为以下结构的结构：

```c
struct sysfs_ops {

    ssize_t (*show)(struct kobject *kobj, struct
attribute *attr,

                    char *buffer);

    ssize_t (*store)(struct kobject *kobj, struct
attribute *attr,

                     const char *buffer, size_t size);

};
```

每当从用户空间读取属性时，都会调用show方法，并传入指向kobject和适当的属性结构的指针。该方法应将给定属性的值编码到buffer中，确保不会溢出（它是PAGE_SIZE字节），并返回返回数据的实际长度。sysfs的约定是每个属性应包含一个单一的、人类可读的值；如果你有很多信息要返回，你可能想考虑将其分割为多个属性。

同一个show方法用于与给定kobject关联的所有属性。传入函数的attr指针可以用来确定正在请求哪个属性。一些show方法包括对属性名称的一系列测试。其他实现将属性结构嵌入到另一个结构中，该结构包含返回属性值所需的信息；在这种情况下，可以在show方法中使用container_of来获取指向嵌入结构的指针。

store方法类似；它应解码存储在buffer中的数据（size包含该数据的长度，不超过PAGE_SIZE），存储并以合理的方式响应新值，并返回实际解码的字节数。只有当属性的权限允许写入时，才能调用store方法。在编写store方法时，永远不要忘记你正在从用户空间接收任意信息；在响应之前，你应该非常仔细地验证它。如果传入的数据不符合预期，返回一个负的错误值，而不是可能做一些不希望的和无法恢复的事情。

如果你的设备导出了一个self_destruct属性，你应该要求写入特定的字符串来调用该功能；一个意外的、随机的写入应该只产生一个错误。

14.2.2. 非缺省属性

在许多情况下，kobject类型的default_attrs字段描述了kobject将永远拥有的所有属性。但这并不是设计中的限制；可以随意向kobjects添加和删除属性。如果你希望在

kobject的sysfs目录中添加一个新属性，只需填写一个属性结构，并将其传递给：

```c
int sysfs_create_file(struct kobject *kobj, struct
attribute *attr);
```

如果一切顺利，将以属性结构中给定的名称创建文件，并返回值为0；否则，返回通常的负错误代码。

注意，同样的show()和store()函数被调用来实现对新属性的操作。在你向kobject添加一个新的、非默认属性之前，你应该采取必要的步骤来确保这些函数知道如何实现该属性。

要删除一个属性，调用：

```c
int sysfs_remove_file(struct kobject *kobj, struct
attribute *attr);
```

调用后，属性将不再出现在kobject的sysfs条目中。但是，请注意，用户空间进程可能对该属性有一个打开的文件描述符，而且在属性被删除后，仍然可能调用show和store。

14.2.3. 二进制属性

sysfs的约定要求所有属性都包含一个以人类可读的文本格式表示的单一值。也就是说，偶尔会有创建可以处理较大块二进制数据的属性的需求。这种需求真正的出现只是在数据必须在用户空间和设备之间保持不变时。例如，上传固件到设备需要这个特性。当在系统中遇到这样的设备时，可以启动一个用户空间程序（通过热插拔机制）；然后该程序通过一个二进制的sysfs属性将固件代码传递给内核，如在“内核固件接口”部分所示。

二进制属性用bin_attribute结构描述：

```c
struct bin_attribute {
    struct attribute attr;
    size_t size;
    ssize_t (*read)(struct kobject *kobj, char *buffer,
                    loff_t pos, size_t size);
    ssize_t (*write)(struct kobject *kobj, char *buffer,
                    loff_t pos, size_t size);
};
```

这里，attr是一个属性结构，给出了二进制属性的名称、所有者和权限，size是二进制属性的最大大小（如果没有最大值，则为0）。read和write方法的工作方式类似于普通的字符驱动程序等效项；它们可以在单个加载中被多次调用，每次调用最多可以处理一页的数据。sysfs没有办法信号一组写操作的最后一个，所以实现一个二进制属性的代码必须能够以其他方式确定数据的结束。

二进制属性必须显式创建；它们不能被设置为默认属性。要创建一个二进制属性，调用：

```c
int sysfs_create_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

二进制属性可以用以下方法删除：

```c
int sysfs_remove_bin_file(struct kobject *kobj,
                          struct bin_attribute *attr);
```

14.2.4. 符号连接

sysfs文件系统具有通常的树形结构，反映了它所代表的kobjects的层次组织。然而，内核中对象之间的关系通常比这更复杂。例如，一个sysfs子树（/sys/devices）代表系统已知的所有设备，而其他子树（在/sys/bus下）代表设备驱动程序。然而，这些树并没有代表驱动程序和它们管理的设备之间的关系。显示这些额外的关系需要额外的指针，这在sysfs中是通过符号链接实现的。

在sysfs内创建一个符号链接很简单：

```c
int sysfs_create_link(struct kobject *kobj, struct
kobject *target,
                      char *name);
```

此函数创建一个链接（称为name），指向target的sysfs条目作为kobj的属性。它是一个相对链接，所以无论sysfs在任何特定系统上挂载在哪里，它都能工作。

即使target从系统中移除，链接也会持续存在。如果你正在创建到其他kobjects的符号链接，你可能应该有一种了解那些kobjects变化的方式，或者有一种保证目标kobjects不会消失的保证。后果（sysfs内的死符号链接）并不特别严重，但它们并不代表最好的编程风格，可能会在用户空间引起混淆。

符号链接可以用以下方法删除：

```c
void sysfs_remove_link(struct kobject *kobj, char *name);
```

14.3. 热插拔事件产生

热插拔事件是内核向用户空间的通知，表明系统的配置发生了变化。每当创建或销毁kobject时，都会生成这样的事件。例如，当用USB线插入数码相机，用户切换控制台模式，或者重新分区磁盘时，都会生成这样的事件。热插拔事件会转变为对/sbin/hotplug的调用，它可以通过加载驱动程序、创建设备节点、挂载分区或采取任何其他适当的行动来响应每个事件。

我们看的最后一个主要的kobject函数是这些事件的生成。实际的事件生成发生在kobject被传递给kobject_add或kobject_del时。在事件被交给用户空间之前，与kobject相关的代码（或者更具体地说，它所属的kset）有机会添加用户空间的信息或完全禁用事件生成。

14.3.1. 热插拔操作

通过kset_hotplug_ops结构中存储的一组方法来实实际控制热插拔事件：

```c
struct kset_hotplug_ops {

    int (*filter)(struct kset *kset, struct kobject
*kobj);

    char *(*name)(struct kset *kset, struct kobject
*kobj);

    int (*hotplug)(struct kset *kset, struct kobject
*kobj,

                   char **envp, int num_envp, char
*buffer,

                   int buffer_size);

};
```

这个结构的指针在kset结构的hotplug_ops字段中找到。如果给定的kobject不包含在kset中，内核会通过层次结构（通过parent指针）向上搜索，直到找到一个有kset的kobject；然后使用该kset的热插拔操作。

每当内核考虑为给定的kobject生成事件时，都会调用filter热插拔操作。如果filter返回0，事件就不会被创建。因此，这个方法给了kset代码一个机会来决定哪些事件应该传递给用户空间，哪些不应该。

作为这个方法可能如何使用的一个例子，考虑一下块子系统。那里至少使用了三种类型的kobjects，代表磁盘、分区和请求队列。用户空间可能想对添加磁盘或分区做出反应，但通常不关心请求队列。所以filter方法只允许为代表磁盘和分区的kobjects生成事件。它看起来像这样：

```c
static int block_hotplug_filter(struct kset *kset, struct
kobject *kobj)

{

    struct kobj_type *ktype = get_ktype(kobj);

    return ((ktype == &ktype_block) || (ktype ==
&ktype_part));

}
```

这里，对kobject类型的快速测试就足以决定是否应该生成事件。

当用户空间的热插拔程序被调用时，它被传递给相关子系统的名称作为它的唯一参数。name热插拔方法负责提供那个名称。它应该返回一个适合传递给用户空间的简单字符串。

热插拔脚本可能想知道的其他所有事情都在环境中传递。最后的热插拔方法（hotplug）在调用该脚本之前提供了添加有用环境变量的机会。再次，这个方法的原型是：

```c
int (*hotplug)(struct kset *kset, struct kobject *kobj,
               char **envp, int num_envp, char *buffer,
               int buffer_size);
```

通常，kset和kobject描述了正在生成事件的对象。envp数组是存储额外环境变量定义的地方（以通常的NAME=value格式）；它有num_envp个可用条目。变量本身应该被编码到buffer中，buffer的长度是buffer_size字节。如果你向envp添加了任何变量，一定要在你的最后一个添加后添加一个NULL条目，这样内核就知道哪里是结束。返回值通常应该是0；任何非零返回都会中止热插拔事件的生成。

热插拔事件的生成（像设备模型中的许多工作一样）通常由总线驱动程序级别的逻辑处理。

14.4. 总线, 设备, 和驱动

到目前为止，我们已经看到了大量的底层基础设施和相对缺乏的例子。在本章的剩余部分，我们尝试弥补这个缺陷，因为我们将进入Linux设备模型的更高层次。为此，我们引入了一个新的虚拟总线，我们称之为lddbus，并修改了scullp驱动程序以“连接”到该总线。

再次强调，这里涵盖的许多材料许多驱动程序作者永远不会需要。这个级别的细节通常在总线级别处理，很少有作者需要添加新的总线类型。然而，对于任何想知道在PCI、USB等层内部发生了什么，或者需要在那个级别进行更改的人来说，这些信息是有用的。

14.4.1. 总线

总线是处理器和一个或多个设备之间的通道。对于设备模型来说，所有的设备都通过总线连接，即使它是一个内部的、虚拟的、“平台”总线。总线可以插入到其他总线中——例如，USB控制器通常是PCI设备。设备模型表示总线和它们控制的设备之间的实际连接。

在Linux设备模型中，总线由bus_type结构表示，定义在<linux/device.h>中。这个结构看起来像这样：

```c
struct bus_type {

    char *name;

    struct subsystem subsys;

    struct kset drivers;

    struct kset devices;

    int (*match)(struct device *dev, struct device_driver
*drv);

    struct device *(*add)(struct device * parent, char *
bus_id);

    int (*hotplug) (struct device *dev, char **envp,

                    int num_envp, char *buffer, int
buffer_size);

    /* Some fields omitted */

};
```

name字段是总线的名称，比如pci。你可以从结构中看出，每个总线都是它自己的子系统；然而，这些子系统并不在sysfs的顶层。相反，它们位于总线子系统下面。一个总线包含两个ksets，代表该总线的已知驱动程序和所有插入到总线的设备。然后，有一组我们将在短时间内讨论的方法。

14.4.1.1. 总线注册

如我们所提到的，示例源代码包括一个名为lddbus的虚拟总线实现。这个总线设置它的bus_type结构如下：

```c
struct bus_type ldd_bus_type = {
    .name = "ldd",
    .match = ldd_match,
    .hotplug  = ldd_hotplug,
};
```

注意，bus_type字段很少需要初始化；大部分都由设备模型核心处理。然而，我们必须指定总线的名称，以及任何与之相关的方法。

不可避免地，新的总线必须通过调用bus_register与系统注册。lddbus代码这样做：

```c
ret = bus_register(&ldd_bus_type);
if (ret)
    return ret;
```

这个调用当然可能会失败，所以必须始终检查返回值。如果成功，新的总线子系统已经添加到系统中；它在sysfs下的/sys/bus中可见，可以开始添加设备。

如果需要从系统中移除一个总线（例如，当关联的模块被移除时），应该调用bus_unregister：

```c
void bus_unregister(struct bus_type *bus);
```

14.4.1.2. 总线方法

bus_type结构定义了几个方法；它们允许总线代码在设备核心和单个驱动程序之间充当中介。在2.6.10内核中定义的方法有：

```c
int (*match)(struct device *device, struct device_driver
*driver);
```

每当为这个总线添加新的设备或驱动程序时，可能会多次调用这个方法。如果给定的设备可以由给定的驱动程序处理，它应该返回一个非零值。（我们将在稍后讨论设备和device_driver结构的细节）。这个函数必须在总线级别处理，因为那里存在适当的逻辑；核心内核无法知道如何匹配每种可能的总线类型的设备和驱动程序。

```c
int (*hotplug) (struct device *device, char **envp, int
num_envp, char *buffer, int buffer_size);
```

这个方法允许总线在用户空间生成热插拔事件之前向环境添加变量。参数与kset热插拔方法（在早期的“热插拔事件生成”部分中描述）相同。

lddbus驱动程序有一个非常简单的匹配函数，它只是比较驱动程序和设备的名称：

```c
static int ldd_match(struct device *dev, struct
device_driver *driver)

{

    return !strncmp(dev->bus_id, driver->name,
strlen(driver->name));

}
```

当涉及到真实的硬件时，匹配函数通常会在设备本身提供的硬件ID和驱动程序支持的ID之间进行某种比较。

lddbus的热插拔方法看起来像这样：

```c
static int ldd_hotplug(struct device *dev, char **envp,
int num_envp,

        char *buffer, int buffer_size)

{

    envp[0] = buffer;

    if (snprintf(buffer, buffer_size,
"LDDBUS_VERSION=%s",

                Version) ≥ buffer_size)

        return -ENOMEM;

    envp[1] = NULL;

    return 0;

}
```

在这里，我们添加了lddbus源的当前修订号，以防万一有人感到好奇。

14.4.1.3. 列举设备和驱动

如果你正在编写总线级别的代码，你可能会发现自己需要对所有已经注册到你的总线的设备或驱动程序执行一些操作。直接深入到bus_type结构中的结构可能很诱人，但最好使用已经提供的辅助函数。

要对总线已知的每个设备进行操作，使用：

```c
int bus_for_each_dev(struct bus_type *bus, struct device
*start,

                     void *data, int (*fn)(struct device
*, void *));
```

这个函数遍历总线上的每个设备，将关联的设备结构和作为数据传入的值传递给fn。如果start为NULL，迭代从总线上的第一个设备开始；否则迭代从start后的第一个设备开始。如果fn返回一个非零值，迭代停止，并从bus_for_each_dev返回该值。

有一个类似的函数用于遍历驱动程序：

```c
int bus_for_each_drv(struct bus_type *bus, struct
device_driver *start,

                     void *data, int (*fn)(struct
device_driver *, void *));
```

这个函数的工作方式就像bus_for_each_dev一样，当然，它是与驱动程序一起工作的。

应该注意的是，这两个函数在工作期间都会持有总线子系统的读/写信号量。所以试图一起使用它们会导致死锁——每个都会试图获取同一个信号量。修改总线的操作（如注销设备）也会锁定。所以，要小心使用bus_for_each函数。

14.4.1.4. 总线属性

Linux设备模型中的几乎每一层都提供了添加属性的接口，总线层也不例外。bus_attribute类型在<linux/device.h>中定义如下：

```c
struct bus_attribute {
    struct attribute attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char
*buf,
                     size_t count);
};
```

我们已经在“默认属性”部分看到了struct attribute。bus_attribute类型还包括两个用于显示和设置属性值的方法。大多数在kobject级别以上的设备模型层都是这样工作的。

提供了一个方便的宏用于编译时创建和初始化bus_attribute结构：

```c
BUS_ATTR(name, mode, show, store);
```

这个宏声明了一个结构，通过在给定的名称前面添加字符串bus*attr*来生成它的名称。

属于总线的任何属性都应该通过bus_create_file明确创建：

```c
int bus_create_file(struct bus_type *bus, struct
bus_attribute *attr);
```

属性也可以通过以下方式移除：

```c
void bus_remove_file(struct bus_type *bus, struct
bus_attribute *attr);
```

lddbus驱动程序创建了一个简单的属性文件，再次包含源版本号。show方法和bus_attribute结构如下设置：

```c
static ssize_t show_bus_version(struct bus_type *bus,
char *buf)

{

    return snprintf(buf, PAGE_SIZE, "%s\n", Version);

}

static BUS_ATTR(version, S_IRUGO, show_bus_version,
NULL);
```

在模块加载时创建属性文件：

```c
if (bus_create_file(&ldd_bus_type, &bus_attr_version))

    printk(KERN_NOTICE "Unable to create version
attribute\n");
```

这个调用创建了一个属性文件(/sys/bus/ldd/version)，包含了lddbus代码的修订号。

14.4.2. 设备

在最低级别，Linux系统中的每个设备都由一个struct device实例表示：

```c
struct device {
    struct device *parent;
    struct kobject kobj;
    char bus_id[BUS_ID_SIZE];
    struct bus_type *bus;
    struct device_driver *driver;
    void *driver_data;
    void (*release)(struct device *dev);
    /* Several fields omitted */
};
```

还有许多其他只对设备核心代码有兴趣的struct device字段。然而，这些字段值得了解：

```c
struct device *parent
```

设备的“父”设备——设备所连接的设备。在大多数情况下，父设备是某种类型的总线或主机控制器。如果parent为NULL，设备是顶级设备，这通常不是你想要的。

```c
struct kobject kobj;
```

代表这个设备并将其链接到层次结构中的kobject。注意，一般规则是，device→kobj->parent等于&device→parent→kobj。

```c
char bus_id[BUS_ID_SIZE];
```

一个字符串，唯一标识总线上的这个设备。例如，PCI设备使用包含域、总线、设备和函数号的标准PCI ID格式。

```c
struct bus_type *bus;
```

标识设备所在的总线类型。

```c
struct device_driver *driver;
```

管理这个设备的驱动程序；我们在下一节中检查struct device_driver。

```c
void *driver_data;
```

设备驱动程序可以使用的私有数据字段。

```c
void (*release)(struct device *dev);
```

当最后一个对设备的引用被移除时调用的方法；它是从嵌入的kobject的release方法中调用的。所有注册到核心的设备结构都必须有一个release方法，否则内核会打印出可怕的投诉。

至少，parent、bus_id、bus和release字段必须在设备结构可以注册之前设置。

14.4.2.1. 设备注册

存在常规的注册和注销函数集：

```c
int device_register(struct device *dev);

void device_unregister(struct device *dev);
```

我们已经看到了lddbus代码如何注册其总线类型。然而，一个实际的总线是一个设备，必须单独注册。为了简单起见，lddbus模块只支持一个虚拟总线，所以驱动程序在编译时设置其设备：

```c
static void ldd_bus_release(struct device *dev)
{
    printk(KERN_DEBUG "lddbus release\n");
}
struct device ldd_bus = {
    .bus_id   = "ldd0",
    .release  = ldd_bus_release
};
```

这是一个顶级总线，所以parent和bus字段被留空。我们有一个简单的、无操作的release方法，作为第一个（也是唯一的）总线，它的名字是ldd0。这个总线设备是通过以下方式注册的：

```c
ret = device_register(&ldd_bus);

if (ret)

    printk(KERN_NOTICE "Unable to register ldd0\n");
```

一旦那个调用完成，新的总线就可以在sysfs的/sys/devices下看到。然后添加到这个总线的任何设备都会出现在/sys/devices/ldd0/下。

14.4.2.2. 设备属性

sysfs中的设备条目可以有属性。相关结构是：

```c
struct device_attribute {

    struct attribute attr;

    ssize_t (*show)(struct device *dev, char *buf);

    ssize_t (*store)(struct device *dev, const char *buf,

                     size_t count);

};
```

这些属性结构可以在编译时用这个宏设置：

```c
DEVICE_ATTR(name, mode, show, store);
```

生成的结构通过在给定的名称前面添加dev*attr*来命名。实际的属性文件管理是用通常的一对函数处理的：

```c
int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

struct bus_type的dev_attrs字段指向为添加到该总线的每个设备创建的默认属性列表。

14.4.2.3. 设备结构嵌入embedding

设备结构包含设备模型核心需要模拟系统的信息。然而，大多数子系统还跟踪它们托管的设备的额外信息。因此，设备很少由裸设备结构表示；相反，该结构，像kobject结构一样，通常嵌入在设备的高级表示中。如果你看一下struct pci_dev或struct usb_device的定义，你会发现一个struct device被埋在里面。通常，底层驱动程序甚至不知道那个struct device，但可能会有例外。

lddbus驱动程序创建了它自己的设备类型（struct ldd_device）并期望单个设备驱动程序使用该类型注册他们的设备。它是一个简单的结构：

```c
struct ldd_device {
    char *name;
    struct ldd_driver *driver;
    struct device dev;
};
#define to_ldd_device(_dev) container_of(_dev, struct
ldd_device, dev);
```

这个结构允许驱动程序为设备提供一个实际的名称（可以与存储在设备结构中的总线ID不同）和一个指向驱动程序信息的指针。真实设备的结构通常也包含关于供应商、设备模型、设备配置、使用的资源等信息。在struct pci_dev（<linux/pci.h>）或struct usb_device（<linux/usb.h>）中可以找到好的例子。还为struct ldd_device定义了一个方便的宏（to_ldd_device），使得将指向嵌入的设备结构的指针转换为ldd_device指针变得容易。

lddbus导出的注册接口如下：

```c
int register_ldd_device(struct ldd_device *lddev)
{
    lddev->dev.bus = &ldd_bus_type;
    lddev->dev.parent = &ldd_bus;
    lddev->dev.release = ldd_dev_release;
    strncpy(lddev->dev.bus_id, lddev->name,
BUS_ID_SIZE);
    return device_register(&lddev->dev);
}
EXPORT_SYMBOL(register_ldd_device);
```

在这里，我们只是填充一些嵌入的设备结构字段（单个驱动程序不应该需要知道），并将设备注册到驱动程序核心。如果我们想要添加总线特定的属性到设备，我们可以在这里这样做。

为了展示如何使用这个接口，让我们介绍另一个示例驱动程序，我们称之为sculld。它是第8章首次介绍的scullp驱动程序的又一个变体。它实现了通常的内存区域设备，但sculld也通过lddbus接口与Linux设备模型一起工作。

sculld驱动程序为其设备条目添加了一个自己的属性；这个属性，称为dev，只包含关联的设备号。这个属性可以被一个加载脚本的模块或热插拔子系统用来在设备被添加到系统时自动创建设备节点。这个属性的设置遵循通常的模式：

```c
static ssize_t sculld_show_dev(struct device *ddev, char
*buf)

{

    struct sculld_dev *dev = ddev->driver_data;

    return print_dev_t(buf, dev->cdev.dev);

}

static DEVICE_ATTR(dev, S_IRUGO, sculld_show_dev, NULL);
```

然后，在初始化时，设备被注册，dev属性通过以下函数创建：

```c
static void sculld_register_dev(struct sculld_dev *dev,
int index)

{

    sprintf(dev->devname, "sculld%d", index);

    dev->ldev.name = dev->devname;

    dev->ldev.driver = &sculld_driver;

    dev->ldev.dev.driver_data = dev;

    register_ldd_device(&dev->ldev);

    device_create_file(&dev->ldev.dev, &dev_attr_dev);

}
```

注意，我们使用driver_data字段来存储指向我们自己的内部设备结构的指针。

14.4.3. 设备驱动

设备模型跟踪系统已知的所有驱动程序。进行此跟踪的主要原因是使驱动程序核心能够将驱动程序与新设备匹配。然而，一旦驱动程序成为系统内的已知对象，就可能实现许多其他事情。例如，设备驱动程序可以导出与任何特定设备无关的信息和配置变量。

驱动程序由以下结构定义：

```c
struct device_driver {
    char *name;
    struct bus_type *bus;
    struct kobject kobj;
    struct list_head devices;
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown) (struct device *dev);
};
```

再次，省略了结构的几个字段（请参阅<linux/device.h>以获取完整的故事）。这里，name是驱动程序的名称（它出现在sysfs中），bus是这个驱动程序工作的总线类型，kobj是不可避免的kobject，devices是当前绑定到这个驱动程序的所有设备的列表，probe是一个函数，被调用来查询一个特定设备的存在（以及这个驱动程序是否可以与它一起工作），remove在设备从系统中移除时被调用，shutdown在关闭时间被调用以使设备静止。

处理device_driver结构的函数的形式现在应该看起来很熟悉（所以我们很快就会介绍它们）。注册函数是：

```c
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

通常的属性结构存在：

```c
struct driver_attribute {

    struct attribute attr;

    ssize_t (*show)(struct device_driver *drv, char
*buf);

    ssize_t (*store)(struct device_driver *drv, const
char *buf,

                     size_t count);

};

DRIVER_ATTR(name, mode, show, store);
```

并且属性文件以通常的方式创建：

```c
int driver_create_file(struct device_driver *drv,

                       struct driver_attribute *attr);

void driver_remove_file(struct device_driver *drv,

                        struct driver_attribute *attr);
```

bus_type结构包含一个字段（drv_attrs），该字段指向一组默认属性，这些属性为与该总线关联的所有驱动程序创建。

14.4.3.1. 驱动结构嵌入

像大多数驱动程序核心结构一样，device_driver结构通常嵌入在更高级别的，特定于总线的结构中。lddbus子系统绝不会违反这样的趋势，所以它定义了自己的ldd_driver结

构：

```c
struct ldd_driver {
    char *version;
    struct module *module;
    struct device_driver driver;
    struct driver_attribute version_attr;
};

#define to_ldd_driver(drv) container_of(drv, struct
ldd_driver, driver);
```

这里，我们要求每个驱动程序提供其当前的软件版本，lddbus为它知道的每个驱动程序导出该版本字符串。特定于总线的驱动程序注册函数是：

```c
int register_ldd_driver(struct ldd_driver *driver)
{
    int ret;
    driver->driver.bus = &ldd_bus_type;
    ret = driver_register(&driver->driver);
    if (ret)
        return ret;
    driver->version_attr.attr.name = "version";
    driver->version_attr.attr.owner = driver->module;
    driver->version_attr.attr.mode = S_IRUGO;
    driver->version_attr.show = show_version;
    driver->version_attr.store = NULL;
    return driver_create_file(&driver->driver, &driver->version_attr);
}
```

函数的前半部分简单地将低级device_driver结构注册到核心；其余部分设置版本属性。由于此属性是在运行时创建的，我们不能使用DRIVER_ATTR宏；相反，必须手动填充driver_attribute结构。注意，我们将属性的所有者设置为驱动程序模块，而不是lddbus模块；这个原因可以在这个属性的show函数的实现中看到：

```c
static ssize_t show_version(struct device_driver *driver,
char *buf)

{

    struct ldd_driver *ldriver = to_ldd_driver(driver);

    sprintf(buf, "%s\n", ldriver->version);

    return strlen(buf);

}
```

人们可能会认为属性所有者应该是lddbus模块，因为实现属性的函数在那里定义。然而，这个函数正在处理驱动程序自身创建（并拥有）的ldd_driver结构。如果在用户空间进程试图读取版本号时，该结构消失，事情可能会变得混乱。将驱动程序模块指定为属性的所有者，可以防止在用户空间保持属性文件打开时卸载模块。由于每个驱动程序模块都创建了对lddbus模块的引用，我们可以确保lddbus不会在不适当的时候被卸载。

为了完整性，sculld如下创建其ldd_driver结构：

```c
static struct ldd_driver sculld_driver = {
    .version = "$Revision$",
    .module = THIS_MODULE,
    .driver = {
        .name = "sculld",
    },
};
```

简单地调用register_ldd_driver将其添加到系统。一旦初始化完成，就可以在sysfs中看到驱动程序信息：

```c
$ tree /sys/bus/ldd/drivers
/sys/bus/ldd/drivers
`-- sculld
    |-- sculld0 -> ../../../../devices/ldd0/sculld0
    |-- sculld1 -> ../../../../devices/ldd0/sculld1
    |-- sculld2 -> ../../../../devices/ldd0/sculld2
    |-- sculld3 -> ../../../../devices/ldd0/sculld3
    `-- version
```

14.5. 类

我们在本章中研究的最后一个设备模型概念是类。类是对设备的更高级别的视图，它抽象出了低级实现细节。驱动程序可能看到SCSI磁盘或ATA磁盘，但在类级别，它们都只是磁盘。类允许用户空间基于设备的功能而不是它们如何连接或如何工作来处理设备。

几乎所有的类都出现在/sys/class下的sysfs中。因此，例如，所有的网络接口都可以在/sys/class/net下找到，无论接口的类型如何。输入设备可以在/sys/class/input中找到，串行设备在/sys/class/tty中。唯一的例外是块设备，由于历史原因，它们可以在/sys/block下找到。

类成员资格通常由高级代码处理，无需驱动程序的明确支持。当sbull驱动程序（参见第16章）创建一个虚拟磁盘设备时，它会自动出现在/sys/block中。snull网络驱动程序（参见第17章）不需要做任何特殊的事情，它的接口就可以在/sys/class/net中表示。然而，有时候，驱动程序最终会直接处理类。

在许多情况下，类子系统是向用户空间导出信息的最好方式。当一个子系统创建一个类时，它完全拥有这个类，所以不需要担心哪个模块拥有那里找到的属性。在sysfs的更多硬件导向部分中漫步也需要很少的时间，以意识到它可以是直接浏览的不友好的地方。用户更愿意在/sys/class/some-widget下找到信息，而不是在/sys/devices/pci0000:00/0000:00:10.0/usb2/2-0:1.0下。

驱动程序核心为管理类导出了两个不同的接口。class_simple例程旨在尽可能简单地向系统添加新的类；它们的主要目的，通常是暴露包含设备号的属性，以便自动创建设备节点。常规类接口更复杂，但也提供了更多的功能。我们从简单版本开始。

14.5.1. class_simple 接口

class_simple接口的设计目的是使其易于使用，以至于没有人有任何借口不导出至少包含设备分配号的属性。使用这个接口只是简单地调用几个函数，与Linux设备模型相关的通常样板文件很少。

第一步是创建类本身。这是通过调用class_simple_create来完成的：

```c
struct class_simple *class_simple_create(struct module
*owner, char *name);
```

此函数创建具有给定名称的类。操作当然可能失败，所以在继续之前应该始终检查返回值（使用IS_ERR，在第11章的“指针和错误值”一节中描述）。

可以用以下方法销毁一个简单的类：

```c
void class_simple_destroy(struct class_simple *cs);
```

创建一个简单类的真正目的是向其中添加设备；这个任务是通过以下方式实现的：

```c
struct class_device *class_simple_device_add(struct
class_simple *cs,
                                             dev_t
devnum,
                                             struct
device *device,
                                             const char
*fmt, ...);
```

这里，cs是之前创建的简单类，devnum是分配的设备号，device是表示此设备的struct device，剩下的参数是一个printk风格的格式字符串和参数，用于创建设备名称。这个调用向类中添加一个包含一个属性dev的条目，该属性保存设备号。如果device参数不为NULL，一个符号链接（称为device）指向/sys/devices下的设备条目。

可以向设备条目添加其他属性。只是使用class_device_create_file的问题，我们将在下一节中与完整的类子系统一起讨论。

当设备来来去去时，类会生成热插拔事件。如果你的驱动程序需要为用户空间事件处理器添加环境变量，它可以设置一个热插拔回调：

```c
int class_simple_set_hotplug(struct class_simple *cs,
                             int (*hotplug)(struct
class_device *dev,
                                            char **envp,
int num_envp,
                                            char *buffer,
int buffer_size));
```

当你的设备消失时，应该删除类条目：

```c
void class_simple_device_remove(dev_t dev);
```

注意，class_simple_device_add返回的class_device结构在这里不需要；设备号（肯定应该是唯一的）就足够了。

14.5.2. 完整的类接口

class_simple接口满足了许多需求，但有时需要更大的灵活性。以下讨论描述了如何使用完整的类机制，class_simple就是基于这个机制的。它很简短：类函数和结构遵循与设备模型其余部分相同的模式，所以这里真正新的东西很少。

14.5.2.1. 管理类

一个类由struct class的一个实例定义：

```c
struct class {
    char *name;
    struct class_attribute *class_attrs;
    struct class_device_attribute *class_dev_attrs;
    int (*hotplug)(struct class_device *dev, char **envp,
                   int num_envp, char *buffer, int
buffer_size);
    void (*release)(struct class_device *dev);
    void (*class_release)(struct class *class);
    /* Some fields omitted */
};
```

每个类都需要一个唯一的名称，这是这个类在/sys/class下的显示方式。当类被注册时，由class_attrs指向的（以NULL结尾的）数组中列出的所有属性都被创建。每个添加到类的设备也有一组默认属性；class_dev_attrs指向这些属性。有一个常规的热插拔函数，用于在生成事件时向环境添加变量。还有两个释放方法：当设备从类中移除时，调用release，当类本身被释放时，调用class_release。

注册函数是：

```c
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

此时，处理属性的接口不应该让任何人感到惊讶：

```c
struct class_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class *cls, char *buf);
    ssize_t (*store)(struct class *cls, const char *buf,
size_t count);
};
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls,
                      const struct class_attribute
*attr);
void class_remove_file(struct class *cls,
                       const struct class_attribute
*attr);
```

14.5.2.2. 类设备

类的真正目的是作为其成员设备的容器。成员由struct class_device表示：

```c
struct class_device {

    struct kobject kobj;

    struct class *class;

    struct device *dev;

    void *class_data;

    char class_id[BUS_ID_SIZE];

};
```

class_id字段保存了此设备在sysfs中的名称。class指针应指向持有此设备的类，dev应指向关联的设备结构。设置dev是可选的；如果它非NULL，它用于从类条目创建一个符

号链接到/sys/devices下的相应条目，使得在用户空间中容易找到设备条目。类可以使用class_data来保存一个私有指针。

提供了常规的注册函数：

```c
int class_device_register(struct class_device *cd);

void class_device_unregister(struct class_device *cd);
```

类设备接口还允许重命名已注册的条目：

```c
int class_device_rename(struct class_device *cd, char
*new_name);
```

类设备条目有属性：

```c
struct class_device_attribute {

   struct attribute attr;

   ssize_t (*show)(struct class_device *cls, char *buf);

   ssize_t (*store)(struct class_device *cls, const char
*buf,

                    size_t count);

};

CLASS_DEVICE_ATTR(name, mode, show, store);

int class_device_create_file(struct class_device *cls,

                             const struct
class_device_attribute *attr);

void class_device_remove_file(struct class_device *cls,

                              const struct
class_device_attribute *attr);
```

当注册类设备时，会创建类的class_dev_attrs字段中的默认属性集；class_device_create_file可用于创建额外的属性。也可以向使用class_simple接口创建的类设备添加属性。

14.5.2.3. 类接口

类子系统有一个在Linux设备模型的其他部分中找不到的额外概念。这个机制被称为接口，但它可能最好被认为是一种触发机制，可以用来在设备进入或离开类时获得通知。

接口由以下结构表示：

```c
struct class_interface {

    struct class *class;

    int (*add) (struct class_device *cd);

    void (*remove) (struct class_device *cd);

};
```

接口可以通过以下方式注册和注销：

```c
int class_interface_register(struct class_interface
*intf);

void class_interface_unregister(struct class_interface
*intf);
```

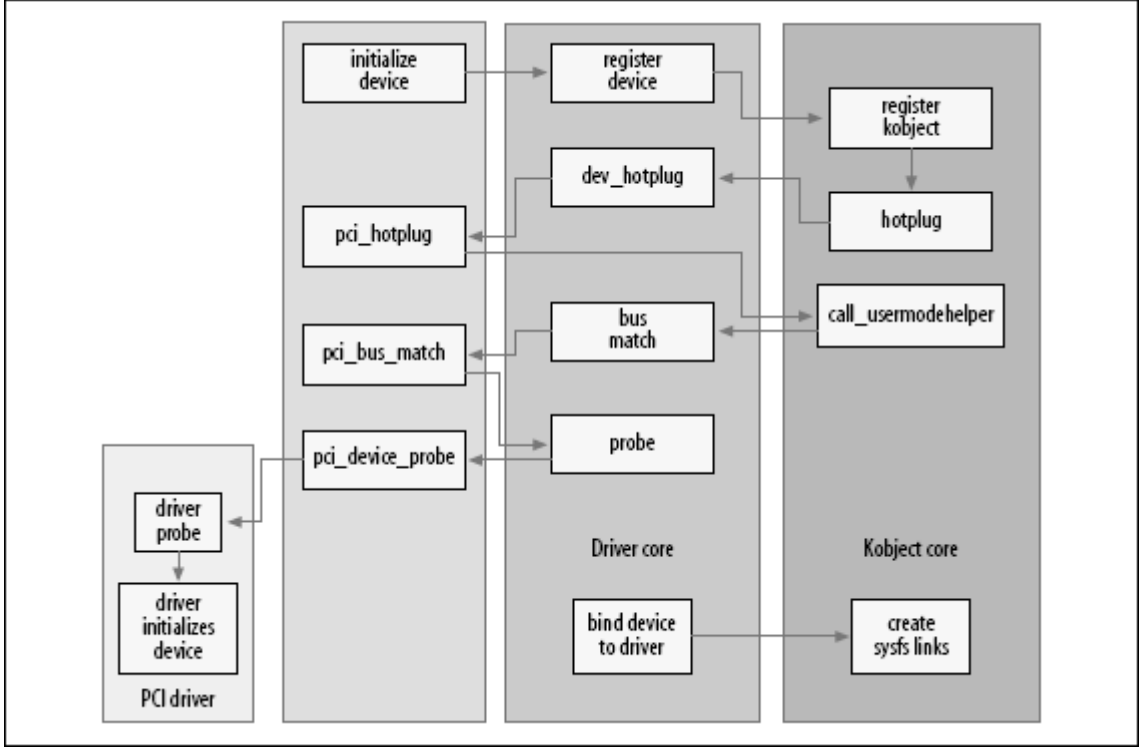
接口的功能很直接。每当一个类设备被添加到class_interface结构中指定的类中，就会调用接口的add函数。该函数可以执行对该设备所需的任何额外设置；这种设置通常采取添加更多属性的形式，但也可能有其他应用。当设备从类中移除时，调用remove方法来执行任何必需的清理。

一个类可以注册多个接口。

14.6. 集成起来Putting It All Together

为了更好地理解驱动模型的作用，让我们走一遍内核中设备的生命周期。我们描述了PCI子系统如何与驱动模型交互，驱动程序如何添加和删除的基本概念，以及如何从系统中添加和删除设备。这些细节，虽然具体描述了PCI内核代码，但适用于所有使用驱动核心来管理其驱动程序和设备的其他子系统。

PCI核心、驱动核心和各个PCI驱动程序之间的交互非常复杂，如图14-3所示。



14.6.1. 添加一个设备

PCI子系统声明了一个名为pci_bus_type的单一struct bus_type，它被初始化为以下值：

```c
struct bus_type pci_bus_type = {
    .name       = "pci",
    .match      = pci_bus_match,
    .hotplug    = pci_hotplug,
    .suspend    = pci_device_suspend,
    .resume     = pci_device_resume,
    .dev_attrs = pci_dev_attrs,
};
```

当PCI子系统在内核中加载时，这个pci_bus_type变量通过调用bus_register注册到驱动核心。当这发生时，驱动核心在/sys/bus/pci中创建一个sysfs目录，该目录包含两个

目录：devices和drivers。

所有的PCI驱动程序都必须定义一个struct pci_driver变量，该变量定义了这个PCI驱动程序可以做的不同功能（关于PCI子系统以及如何编写PCI驱动程序的更多信息，请参见第12章）。该结构包含一个struct device_driver，当PCI驱动程序注册时，PCI核心会初始化这个结构：

```c
/* initialize common driver fields */

drv->driver.name = drv->name;

drv->driver.bus = &pci_bus_type;

drv->driver.probe = pci_device_probe;

drv->driver.remove = pci_device_remove;

drv->driver.kobj.ktype = &pci_driver_kobj_type;
```

这段代码设置了驱动程序的总线指向pci_bus_type，并将probe和remove函数指向PCI核心内的函数。驱动程序的kobject的ktype被设置为变量pci_driver_kobj_type，以使PCI驱动程序的属性文件能够正常工作。然后PCI核心将PCI驱动程序注册到驱动核心：

```c
/* register with core */

error = driver_register(&drv->driver);
```

驱动程序现在已经准备好绑定到它支持的任何PCI设备。

PCI核心，在与实际与PCI总线通信的架构特定代码的帮助下，开始探测PCI地址空间，寻找所有的PCI设备。当找到一个PCI设备时，PCI核心在内存中创建一个新的类型为struct pci_dev的变量。struct pci_dev结构的一部分如下所示：

```c
struct pci_dev {
    /* ... */
    unsigned int    devfn;
    unsigned short vendor;
    unsigned short device;
    unsigned short subsystem_vendor;
    unsigned short subsystem_device;
    unsigned int    class;
    /* ... */
    struct pci_driver *driver;
    /* ... */
    struct device dev;
    /* ... */
};
```

这个PCI设备的总线特定字段由PCI核心初始化（devfn、vendor、device和其他字段），并且struct device变量的parent变量被设置为这个PCI设备所在的PCI总线设备。bus变量被设置为指向pci_bus_type结构。然后设置name和bus_id变量，取决于从PCI设备读取的名称和ID。

在初始化PCI设备结构后，通过调用以下函数将设备注册到驱动核心：

```c
device_register(&dev->dev);
```

在device_register函数中，驱动核心初始化了设备的一些字段，将设备的kobject注册到kobject核心（这会生成一个热插拔事件，但我们将在本章后面讨论），然后将设备添加到设备父项持有的设备列表中。这样做是为了让所有设备都能按正确的顺序遍历，始终知道每个设备在设备层次结构中的位置。

然后将设备添加到特定总线的所有设备列表中，在这个例子中，是pci_bus_type列表。然后遍历所有已经注册到总线的驱动程序列表，并为每个驱动程序调用总线的match函数，指定这个设备。对于pci_bus_type总线，match函数被设置为指向pci_bus_match函数，这是在设备提交给驱动核心之前由PCI核心设置的。

pci_bus_match函数将驱动核心传递给它的struct device转换回struct pci_dev。它还将struct device_driver转换回struct pci_driver，然后查看设备和驱动程序的PCI设备特定信息，看驱动程序是否声明它可以支持这种设备。如果匹配不成功，函数返回0给驱动核心，驱动核心移动到其列表中的下一个驱动程序。

如果匹配成功，函数返回1给驱动核心。这导致驱动核心设置struct device中的驱动程序指针指向这个驱动程序，然后它调用在struct device_driver中指定的probe函数。早些时候，在PCI驱动程序注册到驱动核心之前，probe变量被设置为指向pci_device_probe函数。这个函数再次将struct device转换回struct pci_dev，将设备中设置的struct driver转换回struct pci_driver。它再次验证这个驱动程序声明它可以支持这个设备（这似乎是出于某种未知原因的多余的额外检查），增加设备的引用计数，然后用一个指向它应该绑定的struct pci_dev结构的指针调用PCI驱动程序的probe函数。

如果PCI驱动程序的probe函数确定它由于某种原因不能处理这个设备，它返回一个负的错误值，这个值被传播回驱动核心，导致它继续查看驱动程序列表，以匹配一个与这个设备相匹配的驱动程序。如果probe函数可以声明设备，它做所有需要做的初始化，以正确处理设备，然后它返回0给驱动核心。这导致驱动核心将设备添加到所有当前由这个特定驱动程序绑定的设备的列表中，并在sysfs中的驱动程序目录中为它现在控制的设备创建一个符号链接。这个符号链接允许用户准确地看到哪些设备绑定到哪些设备。这可以看作是:

```
$ tree /sys/bus/pci
/sys/bus/pci/
|-- devices
|  |-- 0000:00:00.0 →
../../../devices/pci0000:00/0000:00:00.0
|  |-- 0000:00:00.1 →
../../../devices/pci0000:00/0000:00:00.1
|  |-- 0000:00:00.2 →
../../../devices/pci0000:00/0000:00:00.2
|  |-- 0000:00:02.0 →
../../../devices/pci0000:00/0000:00:02.0
|  |-- 0000:00:04.0 →
../../../devices/pci0000:00/0000:00:04.0
|  |-- 0000:00:06.0 →
../../../devices/pci0000:00/0000:00:06.0
|  |-- 0000:00:07.0 →
../../../devices/pci0000:00/0000:00:07.0
|  |-- 0000:00:09.0 →
../../../devices/pci0000:00/0000:00:09.0
|  |-- 0000:00:09.1 →
../../../devices/pci0000:00/0000:00:09.1
|  |-- 0000:00:09.2 →
../../../devices/pci0000:00/0000:00:09.2
|  |-- 0000:00:0c.0 →
../../../devices/pci0000:00/0000:00:0c.0
|  |-- 0000:00:0f.0 →
../../../devices/pci0000:00/0000:00:0f.0
|  |-- 0000:00:10.0 →
../../../devices/pci0000:00/0000:00:10.0
|  |-- 0000:00:12.0 →
../../../devices/pci0000:00/0000:00:12.0
|  |-- 0000:00:13.0 →
../../../devices/pci0000:00/0000:00:13.0
|  `-- 0000:00:14.0 →
../../../devices/pci0000:00/0000:00:14.0
`-- drivers
 |-- ALI15x3_IDE
 | `-- 0000:00:0f.0 →
```

```
../../../../devices/pci0000:00/0000:00:0f.0
 |-- ehci_hcd
 |  `-- 0000:00:09.2 ->
../../../../devices/pci0000:00/0000:00:09.2
 |-- ohci_hcd
 |  |-- 0000:00:02.0 ->
../../../../devices/pci0000:00/0000:00:02.0
 |  |-- 0000:00:09.0 ->
../../../../devices/pci0000:00/0000:00:09.0
 |  `-- 0000:00:09.1 ->
../../../../devices/pci0000:00/0000:00:09.1
 |-- orinoco_pci
 |  `-- 0000:00:12.0 ->
../../../../devices/pci0000:00/0000:00:12.0
 |-- radeonfb
 |  `-- 0000:00:14.0 ->
../../../../devices/pci0000:00/0000:00:14.0
 |-- serial
 `-- trident
 `-- 0000:00:04.0 ->
../../../../devices/pci0000:00/0000:00:04.0
```

14.6.2. 去除一个设备

PCI设备可以通过多种不同的方式从系统中移除。所有的CardBus设备实际上都是以不同的物理形态的PCI设备，内核的PCI核心并不区分它们。允许在机器运行时添加或移除PCI设备的系统越来越受欢迎，Linux也支持它们。还有一个假的PCI热插拔驱动程序，允许开发者测试他们的PCI驱动程序是否能正确处理在系统运行时移除设备的情况。这个模块叫做fakephp，它让内核认为PCI设备已经消失，但它不允许用户从没有适当硬件的系统中物理移除PCI设备。查看这个驱动程序的文档以获取更多关于如何使用它来测试你的PCI驱动程序的信息。

PCI核心在移除设备时的工作量要比添加设备时少得多。当要移除一个PCI设备时，调用pci_remove_bus_device函数。这个函数做一些PCI特定的清理和维护工作，然后用一个指向struct pci_dev的struct device成员的指针调用device_unregister函数。

在device_unregister函数中，驱动核心只是从绑定到设备的驱动程序（如果有的话）中取消链接sysfs文件，从其内部设备列表中移除设备，并用一个指向struct device结构中包含的struct kobject的指针调用kobject_del函数。那个函数向用户空间发出一个热

插拔调用，声明kobject现在已经从系统中移除，然后它删除所有与kobject相关的sysfs文件和kobject最初创建的sysfs目录本身。

`kobject_del`函数也移除了设备本身的kobject引用。如果那个引用是最后一个（意味着没有用户空间文件打开设备的sysfs条目），那么就调用PCI设备本身的release函数，pci_release_dev。那个函数只是释放struct pci_dev占用的内存。

在此之后，所有与设备相关的sysfs条目都被移除，与设备相关的内存被释放。PCI设备现在完全从系统中移除。

14.6.3. 添加一个驱动

当PCI驱动程序调用pci_register_driver函数时，它被添加到PCI核心。这个函数只是初始化包含在struct pci_driver结构中的struct device_driver结构，就像前面在添加设备的部分提到的那样。然后PCI核心用一个指向struct pci_driver结构中包含的struct device_driver结构的指针调用驱动核心的driver_register函数。

driver_register函数在struct device_driver结构中初始化了一些锁，然后调用bus_add_driver函数。这个函数执行以下步骤：

- 查找驱动程序要关联的总线。如果找不到这个总线，函数立即返回。
- 根据驱动程序的名称和它关联的总线创建驱动程序的sysfs目录。
- 抓取总线的内部锁，然后遍历所有已经注册到总线的设备，并为它们调用match函数，就像添加新设备时一样。如果match函数成功，那么就会发生剩下的绑定过程，就像前面的部分描述的那样。

14.6.4. 去除一个驱动

移除驱动程序是一个非常简单的操作。对于PCI驱动程序，驱动程序调用pci_unregister_driver函数。这个函数只是用一个指向传递给它的struct pci_driver结构的struct device_driver部分的指针调用驱动核心函数driver_unregister。

driver_unregister函数通过清理一些附加到sysfs树中驱动程序条目的sysfs属性来处理一些基本的清理工作。然后它遍历所有附加到这个驱动程序的设备，并为它调用release函数。这就像前面提到的当设备从系统中移除时的release函数一样。

在所有设备都从驱动程序中解绑之后，驱动程序代码做了这个独特的逻辑：

```c
down(&drv->unload_sem);

up(&drv->unload_sem);
```

这是在返回给函数调用者之前做的。这个锁被抓取是因为代码需要等待所有对这个驱动程序的引用计数降到0才能安全返回。这是必要的，因为driver_unregister函数最常被作为正在卸载的模块的退出路径调用。只要设备引用驱动程序，模块就需要保持在内存中，通过等待这个锁被释放，这让内核知道何时可以安全地从内存中移除驱动程序。

14.7. 热插拔

热插拔有两种不同的视角。内核将热插拔视为硬件、内核和内核驱动程序之间的交互。用户将热插拔视为内核和用户空间通过名为/sbin/hotplug的程序之间的交互。当内核想要通知用户空间内核中刚刚发生了某种类型的热插拔事件时，它会调用这个程序。

14.7.1. 动态设备

“热插拔”这个术语最常用的含义是，现在几乎所有的计算机系统都可以在系统通电的情况下处理设备的出现或消失。这与几年前的计算机系统非常不同，那时的程序员知道他们只需要在启动时扫描所有设备，他们从不需要担心他们的设备会消失，直到整个机器的电源被切断。现在，随着USB、CardBus、PCMCIA、IEEE1394和PCI热插拔控制器的出现，Linux内核需要能够在任何硬件被添加或从系统中移除的情况下可靠地运行。这给设备驱动程序作者增加了负担，因为他们现在必须始终处理设备突然在他们下面被拔出，而没有任何通知。

每种不同的总线类型都以不同的方式处理设备的丢失。例如，当一个PCI、CardBus或PCMCIA设备从系统中移除时，通常在驱动程序通过其remove函数被通知这个动作之前会有一段时间。在那之前，所有从PCI总线的读取都返回所有位设置。这意味着驱动程序需要始终检查他们从PCI总线读取的数据的值，并能够正确处理0xff值。

这个可以在drivers/usb/host/ehci-hcd.c驱动程序中看到，这是一个USB 2.0（高速）控制器卡的PCI驱动程序。它在其主握手循环中有以下代码来检测控制器卡是否已经从系统中移除：

```c
result = readl(ptr);

if (result = = ~(u32)0)     /* card removed */

    return -ENODEV;
```

对于USB驱动程序，当一个USB驱动程序绑定的设备从系统中移除时，提交给设备的任何待处理的urb都会开始返回错误-ENODEV。驱动程序需要识别这个错误，并在发生时正确清理任何待处理的I/O。

热插拔设备不仅限于传统设备，如鼠标、键盘和网络卡。现在有许多系统支持移除和添加整个CPU和内存条。幸运的是，Linux内核正确处理了这些核心“系统”设备的添加和移除，所以单个设备驱动程序不需要关注这些事情。

14.7.2. /sbin/hotplug 工具

如本章前面所提到的，每当一个设备被添加到系统中或从系统中移除时，都会生成一个“热插拔事件”。这意味着内核调用用户空间程序/sbin/hotplug。这个程序通常是一个非常小的bash脚本，它只是将执行权传递给放在/etc/hotplug.d/目录树中的其他程序列表。对于大多数Linux发行版，这个脚本看起来像下面这样：

```c
DIR="/etc/hotplug.d"

for I in "${DIR}/$1/"*.hotplug "${DIR}/"default/*.hotplug
; do

    if [ -f $I ]; then

        test -x $I && $I $1 ;

    fi

done

exit 1
```

换句话说，这个脚本搜索所有可能对这个事件感兴趣的带有.hotplug后缀的程序，并调用它们，传递给它们一些由内核设置的不同的环境变量。关于/sbin/hotplug脚本如何工作的更多细节可以在程序的注释和hotplug(8) manpage中找到。

如前所述，每当创建或销毁一个kobject时，都会调用/sbin/hotplug。hotplug程序被调用时，会有一个单独的命令行参数提供事件的名称。核心内核和特定子系统也设置了一系列的环境变量（下面描述）来提供刚刚发生了什么的信息。这些变量被hotplug程序用来确定内核中刚刚发生了什么，以及是否应该进行任何特定的操作。

传递给/sbin/hotplug的命令行参数是与这个热插拔事件相关的名称，由分配给kobject的kset确定。这个名称可以通过调用本章前面描述的kset的hotplug_ops结构中的name函数来设置；如果该函数不存在或从未被调用，那么名称就是kset本身的名称。

总是为/sbin/hotplug程序设置的默认环境变量是：

- ACTION：根据问题对象是刚刚创建还是被销毁，字符串为add或remove。
- DEVPATH：在sysfs文件系统中的一个目录路径，指向正在被创建或销毁的kobject。注意，sysfs文件系统的挂载点没有添加到这个路径中，所以用户空间程序需要确定这个。
- SEQNUM：这个热插拔事件的序列号。序列号是一个64位的数字，每生成一个热插拔事件就增加一次。这允许用户空间按照内核生成它们的顺序对热插拔事件进行排序，因为用户空间程序可能会乱序运行。

- SUBSYSTEM：与上述命令行参数相同的字符串。

当与总线相关的设备被添加到系统中或从系统中移除时，不同的总线子系统都会在他们的hotplug回调中向/sbin/hotplug调用添加他们自己的环境变量，这个回调在他们的总线的struct kset_hotplug_ops中指定（如“Hotplug Operations”部分所述）。这允许用户空间能够自动加载可能需要的任何模块来控制总线找到的设备。以下是不同总线类型和他们添加到/sbin/hotplug调用的环境变量的列表。

14.7.2.1. IEEE1394(火线)

在IEEE1394总线上的任何设备，也被称为Firewire，都将/sbin/hotplug参数名称和SUBSYSTEM环境变量设置为值ieee1394。ieee1394子系统也总是添加以下四个环境变量：

- VENDOR_ID：IEEE1394设备的24位供应商ID
- MODEL_ID：IEEE1394设备的24位型号ID
- GUID：设备的64位GUID
- SPECIFIER_ID：指定此设备的协议规范所有者的24位值
- VERSION：指定此设备的协议规范版本的值。

14.7.2.2. 网络

所有网络设备在设备在内核中注册或注销时都会创建一个热插拔事件。/sbin/hotplug调用将参数名称和SUBSYSTEM环境变量设置为值net，并添加以下环境变量：

- INTERFACE：已经在内核中注册或注销的接口的名称。例如lo和eth0。

14.7.2.3. PCI 总线

在PCI总线上的任何设备都将参数名称和SUBSYSTEM环境变量设置为值pci。PCI子系统也总是添加以下四个环境变量：

- PCI_CLASS：设备的PCI类号，以十六进制表示。
- PCI_ID：设备的PCI供应商和设备ID，以十六进制表示，格式为vendor:device。
- PCI_SUBSYS_ID：PCI子系统供应商和子系统设备ID，格式为subsys_vendor:subsys_device。
- PCI_SLOT_NAME：内核给设备的PCI插槽“名称”。它的格式为domain:bus:slot:function。一个例子可能是0000:00:0d.0。

14.7.2.4. 输入

对于所有输入设备（鼠标、键盘、操纵杆等），当设备被添加到内核和从内核中移除时，都会生成一个热插拔事件。/sbin/hotplug参数和SUBSYSTEM环境变量被设置为值input。输入子系统也总是添加以下环境变量：

- PRODUCT：一个多值字符串，列出的值为十六进制，没有前导零。它的格式为bustype:vendor:product:version。

如果设备支持，以下环境变量可能存在：

- NAME：设备给出的输入设备的名称。
- PHYS：输入子系统给这个设备的设备的物理地址。它应该是稳定的，取决于设备插入的总线位置。
- EV
- KEY
- REL
- ABS
- MSC
- LED
- SND
- FF

这些都来自输入设备描述符，如果特定的输入设备支持，它们被设置为适当的值。

14.7.2.5. USB 总线

在USB总线上的任何设备都将参数名称和SUBSYSTEM环境变量设置为值usb。USB子系统也总是添加以下环境变量：

- PRODUCT：一个格式为idVendor/idProduct/bcdDevice的字符串，指定那些USB设备特定的字段
- TYPE：一个格式为bDeviceClass/bDeviceSubClass/bDeviceProtocol的字符串，指定那些USB设备特定的字段

如果bDeviceClass字段设置为0，还会设置以下环境变量：

- INTERFACE：一个格式为bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol的字符串，指定那些USB设备特定的字段。

如果选择了内核构建选项CONFIG_USB_DEVICEFS，该选项选择在内核中构建usbfs文件系统，还会设置以下环境变量：

- DEVICE：一个字符串，显示设备在usbfs文件系统中的位置。这个字符串的格式为/proc/bus/usb/USB_BUS_NUMBER/USB_DEVICE_NUMBER，其中USB_BUS_NUMBER是设备所在的USB总线的三位数，USB_DEVICE_NUMBER是内核为该USB设备分配的三位数。

14.7.2.6. SCSI 总线

所有SCSI设备在SCSI设备被创建或从内核中移除时都会创建一个热插拔事件。对于每一个被添加或从系统中移除的SCSI设备，/sbin/hotplug调用将参数名称和SUBSYSTEM环境变量设置为值scsi。SCSI系统不会添加任何额外的环境变量，但在这里提到它是因为有一个特定于SCSI的用户空间脚本可以确定应该为指定的SCSI设备加载哪些SCSI驱动程序（磁盘、磁带、通用等）。

14.7.3. 使用 /sbin/hotplug

现在，Linux内核在每个设备被添加到内核和从内核中移除时都会调用/sbin/hotplug，因此在用户空间中创建了许多非常有用的工具来利用这一点。最受欢迎的两个工具是Linux Hotplug脚本和udev。

14.7.3.1. Linux 热插拔脚本

Linux热插拔脚本最初是/sbin/hotplug调用的第一个用户。这些脚本查看内核设置的不同环境变量，以描述刚刚发现的设备，然后试图找到与该设备匹配的内核模块。

如前所述，当驱动程序使用MODULE_DEVICE_TABLE宏时，depmod程序会取得该信息并创建位于`/lib/module/KERNEL_VERSION/modules._map`的文件。取决于驱动程序支持的总线类型。目前，为支持PCI、USB、IEEE1394、INPUT、ISAPNP和CCW子系统的设备工作的驱动程序生成了模块映射文件。

热插拔脚本使用这些模块映射文本文件来确定尝试加载哪个模块以支持内核最近发现的设备。它们加载所有模块，并不在第一个匹配项处停止，以便让内核找出哪个模块最适合。当设备被移除时，这些脚本不会卸载任何模块。如果它们试图这样做，它们可能会意外地关闭也由被移除设备的同一驱动程序控制的设备。

注意，现在modprobe程序可以直接从模块中读取MODULE_DEVICE_TABLE信息，而不需要模块映射文件，热插拔脚本可能会被减少到围绕modprobe程序的小包装器。

14.7.3.2. udev

在内核中创建统一驱动模型的主要原因之一是允许用户空间以动态方式管理/dev树。这之前已经在用户空间通过实现devfs来完成，但由于缺乏活跃的维护者和一些无法修复的核心bug，这个代码库已经慢慢腐烂。许多内核开发者意识到，如果所有设备信息都导出到用户空间，它就可以执行所有必要的/dev树管理。

devfs在其设计中有一些非常基本的缺陷。它要求每个设备驱动程序都被修改以支持它，并且要求设备驱动程序指定其在/dev树中的名称和位置。它也不能正确处理动态的主要和次要号码，也不允许用户空间以简单的方式覆盖设备的命名，这迫使设备命名策略驻留在内核而不是用户空间。Linux内核开发者真的很讨厌在内核中有策略，而且由于devfs的命名策略不遵循Linux标准基础规范，所以它真的让他们很困扰。

当Linux内核开始被安装在大型服务器上时，许多用户遇到了如何管理大量设备的问题。超过10,000个独特设备的磁盘驱动器阵列提出了一个非常困难的任务，即确保特定的磁盘总是用完全相同的名称命名，无论它在磁盘阵列中的位置如何，或者它何时被内核发现。这个问题也困扰着那些试图在他们的系统中插入两个USB打印机的桌面用户，然后他们意识到他们没有办法确保被称为/dev/lpt0的打印机不会改变并被分配给其他打印机，如果系统被重启的话。

所以，udev被创建了。它依赖于所有设备信息通过sysfs导出到用户空间，并通过/sbin/hotplug通知设备被添加或移除。策略决策，如给设备命名，可以在用户空间中指定，不在内核中。这确保了命名策略从内核中移除，并允许对每个设备的名称有大量的灵活性。

有关如何使用udev和如何配置它的更多信息，请参阅您的发行版中的udev包中包含的文档。

设备驱动程序需要做的所有事情，以便udev能够正确地与它一起工作，就是确保任何分配给驱动程序控制的设备的主要和次要号码都通过sysfs导出到用户空间。对于任何使用子系统来分配主要和次要号码的驱动程序，这已经由子系统完成，驱动程序不需要做任何工作。做这个的子系统的例子有tty、misc、usb、input、scsi、block、i2c、network和frame buffer子系统。如果你的驱动程序通过调用cdev_init函数或者旧的register_chrdev函数自己处理获取主要和次要号码，那么驱动程序需要被修改以便udev能够正确地与它一起工作。

udev在sysfs的/class/树中寻找一个叫做dev的文件，以确定当它通过/sbin/hotplug接口被内核调用时，特定设备被分配的主要和次要号码是什么。设备驱动程序只需要为它控制的每个设备创建那个文件。class_simple接口通常是做这个的最简单的方式。

如“class_simple接口”部分所述，使用class_simple接口的第一步是通过调用class_simple_create函数创建一个struct class_simple：

```c
static struct class_simple *foo_class;

...

foo_class = class_simple_create(THIS_MODULE, "foo");

if (IS_ERR(foo_class)) {

    printk(KERN_ERR "Error creating foo class.\n");

    goto error;

}
```

这段代码在/sys/class/foo中创建了一个sysfs目录。

每当你的驱动程序发现一个新设备，并且你按照第3章的描述为它分配一个次要号码时，驱动程序应该调用class_simple_device_add函数：

class_simple_device_add(foo_class, MKDEV(FOO_MAJOR, minor), NULL, "foo%d", minor);

这段代码会在/sys/class/foo下创建一个名为fooN的子目录，其中N是这个设备的次要号码。在这个目录中创建了一个文件dev，这正是udev为你的设备创建设备节点所需要的。

当你的驱动程序从设备解绑，并放弃它附加的次要号码时，需要调用class_simple_device_remove来删除这个设备的sysfs条目：

```c
class_simple_device_remove(MKDEV(FOO_MAJOR, minor));
```

稍后，当你的整个驱动程序正在关闭时，需要调用class_simple_destroy来删除你最初通过调用class_simple_create创建的类：

```c
class_simple_destroy(foo_class);
```

通过调用class_simple_device_add创建的dev文件由主要和次要号码组成，中间用:字符分隔。如果你的驱动程序不想使用class_simple接口，因为你想在子系统的类目录中提供其他文件，使用print_dev_t函数来正确格式化特定设备的主要和次要号码。

14.8. 处理固件Eirmware

作为驱动程序作者，你可能会发现自己面临一个必须在其正常工作之前下载固件的设备。硬件市场的许多部分的竞争如此激烈，以至于即使是设备控制固件的一点EEPROM的成本也超过了制造商愿意花费的金额。因此，固件是与硬件一起在CD上分发的，操作系统负责将固件传送到设备本身。

你可能会试图用这样的声明来解决固件问题：

```c
static char my_firmware[ ] = { 0x34, 0x78, 0xa4, ... };
```

然而，这种方法几乎肯定是一个错误。将固件编码到驱动程序中会使驱动程序代码膨胀，使固件升级变得困难，并且很可能会遇到许可问题。供应商很可能没有在GPL下发布固件映像，所以将它与GPL许可的代码混合通常是一个错误。因此，包含有线固件的驱动程序不太可能被接受到主线内核或被Linux发行商包含。

14.8.1. 内核固件接口

正确的解决方案是在需要时从用户空间获取固件。但是，请抵制直接从内核空间打开包含固件的文件的诱惑；这是一个容易出错的操作，而且它将策略（以文件名的形式）放入内核。相反，正确的方法是使用固件接口，这个接口就是为此目的而创建的：

```c
#include <linux/firmware.h>

int request_firmware(const struct firmware **fw, char
*name,

                     struct device *device);
```

调用request_firmware请求用户空间定位并提供一个固件映像给内核；我们稍后会看到它是如何工作的。name应该标识所需的固件；正常的用法是供应商提供的固件文件的名称。像my_firmware.bin这样的名称是典型的。如果固件成功加载，返回值为0（否则返回通常的错误代码），并且fw参数指向以下结构之一：

```c
struct firmware {

        size_t size;

        u8 *data;

};
```

该结构包含实际的固件，现在可以下载到设备。请注意，这个固件是来自用户空间的未经检查的数据；在将其发送到硬件之前，你应该应用你能想到的所有测试来确信它是一个正确的固件映像。设备固件通常包含标识字符串、校验和等；在信任数据之前，检查它们所有。

在你将固件发送到设备后，你应该用以下方法释放内核结构：

```c
void release_firmware(struct firmware *fw);
```

由于request_firmware请求用户空间的帮助，所以在返回之前它保证会睡眠。如果你的驱动程序在必须请求固件时不能睡眠，可以使用异步替代方法：

```c
int request_firmware_nowait(struct module *module,
                            char *name, struct device
*device, void *context,
                            void (*cont)(const struct
firmware *fw, void *context));
```

这里的额外参数是module（几乎总是THIS_MODULE）、context（一个私有数据指针，固件子系统不使用）和cont。如果一切顺利，request_firmware_nowait开始固件加载过程并返回0。在未来的某个时间，cont将被调用，加载的结果将作为参数传入。如果固件加载由于某种原因失败，fw为NULL。

14.8.2. 它如何工作

固件子系统通过sysfs和热插拔机制工作。当调用request_firmware时，会在/sys/class/firmware下创建一个新目录，使用你的设备的名称。该目录包含三个属性：

loading 用户空间进程加载固件时，应将此属性设置为1。当加载过程完成时，应将其设置为0。向loading写入-1的值会中止固件加载过程。

data data是一个二进制属性，接收固件数据本身。设置loading后，用户空间进程应将固件写入此属性。

device 此属性是指向/sys/devices下相关条目的符号链接。

一旦创建了sysfs条目，内核就会为你的设备生成一个热插拔事件。传递给热插拔处理程序的环境包括一个变量FIRMWARE，它被设置为提供给request_firmware的名称。处理程序应定位固件文件，并使用提供的属性将其复制到内核。如果找不到文件，处理程序应将loading属性设置为-1。

如果固件请求在10秒内没有得到服务，内核就会放弃并向驱动程序返回失败状态。可以通过sysfs属性/sys/class/firmware/timeout更改该超时期。

使用request_firmware接口允许你将设备固件与你的驱动程序一起分发。当正确地集成到热插拔机制中时，固件加载子系统允许设备“开箱即用”。这显然是处理问题的最佳方

式。

然而，请允许我们再次警告：设备固件不应在没有制造商许可的情况下分发。许多制造商在礼貌地询问时会同意以合理的条款许可他们的固件；有些人可能不那么合作。无论哪种方式，未经许可复制和分发他们的固件都是侵犯版权法的行为，也是招惹麻烦的行为。

14.9. 快速参考

许多函数在本章中已经被介绍过; 这是它们全部的一个快速总结.

14.9.1. Kobjects结构

```
#include <linux/kobject.h>
```

包含文件, 包含 kobject 的定义, 相关结构, 和函数.

```
void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj, const char
*format, ...);
```

用作 kobject 初始化的函数

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

为 kobjects 管理引用计数的函数.

```
struct kobj_type;
struct kobj_type *get_ktype(struct kobject *kobj);
```

表示一个kobjct 被嵌入的结构类型. 使用 get_ktype 来获得关联到一个给定 kobject 的 kobj_type.

```
int kobject_add(struct kobject *kobj);
extern int kobject_register(struct kobject *kobj);
void kobject_del(struct kobject *kobj);
void kobject_unregister(struct kobject *kobj);
```

kobject_add 添加一个 kobject 到系统, 处理 kset 成员关系, sysfs 表示, 以及热插拔事件产生. kobject_register 是一个方便函数, 它结合 kobject_init 和 kobject_add. 使用 kobject_del 来去除一个 kobject 或者 kobject_unregister, 它结合了 kobject_del 和 kobject_put.

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

为 ksets 初始化和注册的函数.

```
decl_subsys(name, type, hotplug_ops);
```

易于声明子系统的一个宏.

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys);
void subsys_put(struct subsystem *subsys);
```

对子系统的操作.

14.9.2. sysfs 操作

`#include` <linux/sysfs.h>
包含 sysfs 声明的包含文件.

```
int sysfs_create_file(struct kobject *kobj, struct
attribute *attr);
int sysfs_remove_file(struct kobject *kobj, struct
attribute *attr);
int sysfs_create_bin_file(struct kobject *kobj, struct
bin_attribute *attr);
int sysfs_remove_bin_file(struct kobject *kobj, struct
bin_attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject
*target, char *name);
void sysfs_remove_link(struct kobject *kobj, char *name);
```

创建和去除和一个 kobject 关联的属性文件的函数.

14.9.3. 总线, 设备, 和驱动

```
int bus_register(struct bus_type *bus);
void bus_unregister(struct bus_type *bus);
```

在设备模型中进行注册和注销总线的函数.

```
int bus_for_each_dev(struct bus_type *bus, struct device
*start, void *data, int (*fn)(struct device *, void *));
int bus_for_each_drv(struct bus_type *bus, struct
device_driver *start, void *data, int (*fn)(struct
device_driver *, void *));
```

列举每个设备和驱动的函数, 特别地, 绑定到给定总线的设备.

```

BUS_ATTR(name, mode, show, store);
int bus_create_file(struct bus_type *bus, struct
bus_attribute *attr);
void bus_remove_file(struct bus_type *bus, struct
bus_attribute *attr);
```

BUS_ATTR 宏可能用来声明一个 bus_attribute 结构, 它可能接着被添加和去除, 使用上面 2 个函数.

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

处理设备注册的函数.

```
DEVICE_ATTR(name, mode, show, store);
int device_create_file(struct device *device, struct
device_attribute *entry);
void device_remove_file(struct device *dev, struct
device_attribute *attr);
```

处理设备属性的宏和函数.

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

注册和注销一个设备驱动的函数.

```
DRIVER_ATTR(name, mode, show, store);
int driver_create_file(struct device_driver *drv, struct
driver_attribute *attr);
void driver_remove_file(struct device_driver *drv, struct
driver_attribute *attr);
```

关联驱动属性的宏和函数.

14.9.4. 类

```
struct class_simple *class_simple_create(struct module
*owner, char *name);
void class_simple_destroy(struct class_simple *cs);
struct class_device *class_simple_device_add(struct
class_simple *cs, dev_t devnum, struct device *device,
const char *fmt, ...);
void class_simple_device_remove(dev_t dev);
int class_simple_set_hotplug(struct class_simple *cs, int
(*hotplug)(struct class_device *dev, char **envp, int
num_envp, char *buffer, int buffer_size));
```

实现 class_simple 接口的函数; 它们管理包含一个 dev 属性和很少其他属性的简单的类入口

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

注册和注销类.

```
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls, const struct
class_attribute *attr);
void class_remove_file(struct class *cls, const struct
class_attribute *attr);
```

处理类属性的常用宏和函数.

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
int class_device_rename(struct class_device *cd, char
*new_name);
CLASS_DEVICE_ATTR(name, mode, show, store);
int class_device_create_file(struct class_device *cls,
const struct class_device_attribute *attr);
```

属性类设备接口的函数和宏.

```
int class_interface_register(struct class_interface
*intf);
void class_interface_unregister(struct class_interface
*intf);
```

添加一个接口到一个类(或去除它)的函数.

14.9.5. 固件

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char
*name, struct device *device);
int request_firmware_nowait(struct module *module, char
*name, struct device *device, void *context, void (*cont)
(const struct firmware *fw, void *context));
void release_firmware(struct firmware *fw);
```

属性内核固件加载接口的函数.