**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：

方式一，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：

1. /\*\*
2. \* struct subsys\_interface - interfaces to device functions
3. \* @name: name of the device function
4. \* @subsys: subsytem of the devices to attach to
5. \* @node: the list of functions registered at the subsystem
6. \* @add\_dev: device hookup to device function handler
7. \* @remove\_dev: device hookup to device function handler
8. \*
9. \* Simple interfaces attached to a subsystem. Multiple interfaces can
10. \* attach to a subsystem and its devices. Unlike drivers, they do not
11. \* exclusively claim or control devices. Interfaces usually represent
12. \* a specific functionality of a subsystem/class of devices.
13. \*/
14. **struct** subsys\_interface {
15. **const** **char** \*name;
16. **struct** bus\_type \*subsys;
17. **struct** list\_head node;
18. int (\*add\_dev)(**struct** device \*dev, **struct** subsys\_interface \*sif);
19. int (\*remove\_dev)(**struct** device \*dev, **struct** subsys\_interface \*sif);
20. };

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。