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(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 {  struct device \*parent;  struct device\_private \*p;  struct kobject kobj;  const char \*init\_name; /\* initial name of the device \*/  const struct device\_type \*type;  struct mutex mutex; /\* mutex to synchronize calls to its driver. \*/  struct bus\_type \*bus; /\* type of bus device is on \*/  struct device\_driver \*driver; /\* which driver has allocated this device \*/  void \*platform\_data; /\* Platform specific data, device core doesn't touch it \*/  void \*driver\_data; /\* Driver data, set and get with dev\_set/get\_drvdata \*/  struct dev\_pm\_info power;  struct dev\_pm\_domain \*pm\_domain;  #ifdef CONFIG\_GENERIC\_MSI\_IRQ\_DOMAIN  struct irq\_domain \*msi\_domain;  #endif  #ifdef CONFIG\_PINCTRL  struct dev\_pin\_info \*pins;  #endif  #ifdef CONFIG\_GENERIC\_MSI\_IRQ  struct list\_head msi\_list;  #endif  #ifdef CONFIG\_NUMA  int numa\_node; /\* NUMA node this device is close to \*/  #endif  u64 \*dma\_mask; /\* dma mask (if dma'able device) \*/  u64 coherent\_dma\_mask;  unsigned long dma\_pfn\_offset;  struct device\_dma\_parameters \*dma\_parms;  struct list\_head dma\_pools; /\* dma pools (if dma'ble) \*/  struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent mem override \*/  #ifdef CONFIG\_DMA\_CMA  struct cma \*cma\_area; /\* contiguous memory area for dma allocations \*/  #endif  struct removed\_region \*removed\_mem;  /\* arch specific additions \*/  struct dev\_archdata archdata;  struct device\_node \*of\_node; /\* associated device tree node \*/  struct fwnode\_handle \*fwnode; /\* firmware device node \*/  dev\_t **devt; /\* dev\_t, creates the sysfs "dev" \*/**  u32  **id; /\* device instance \*/**  spinlock\_t devres\_lock;  struct list\_head devres\_head;  struct klist\_node knode\_class;  struct class \*class;  const struct attribute\_group \*\*groups; /\* optional groups \*/  void (\*release)(struct device \*dev);  struct iommu\_group \*iommu\_group;  }; |

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。

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

**(2)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等，具体可查看源代码。

(3) kobj，该数据结构对应的struct kobject。

(4) 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。

(5) 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之间的关系，后续会再详细说明。

(7)bus，该device属于哪个总线（后续会详细描述）。

(8)driver，该device对应的device driver。

(9)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。

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

(11) pins，"PINCTRL”功能，暂不描述。

(12)**numa\_node，"NUMA”功能**，暂不描述。

(13)dma\_mask~ archdata，DMA相关的功能，暂不描述。

(14)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：

(15)class，该设备属于哪个class。

(16)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 {  const char \*name;  struct bus\_type \*bus;  struct module \*owner;  const char \*mod\_name; /\* used for built-in modules \*/  bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/  enum probe\_type probe\_type;  const struct of\_device\_id \*of\_match\_table;  const struct acpi\_device\_id \*acpi\_match\_table;  int (\*probe) (struct device \*dev);  int (\*remove) (struct device \*dev);  void (\*shutdown) (struct device \*dev);  int (\*suspend) (struct device \*dev, pm\_message\_t state);  int (\*resume) (struct device \*dev);  const struct attribute\_group \*\*groups;  const struct dev\_pm\_ops \*pm;  struct driver\_private \*p;  }; |

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

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。

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。

**Linux设备模型(6)\_Bus**

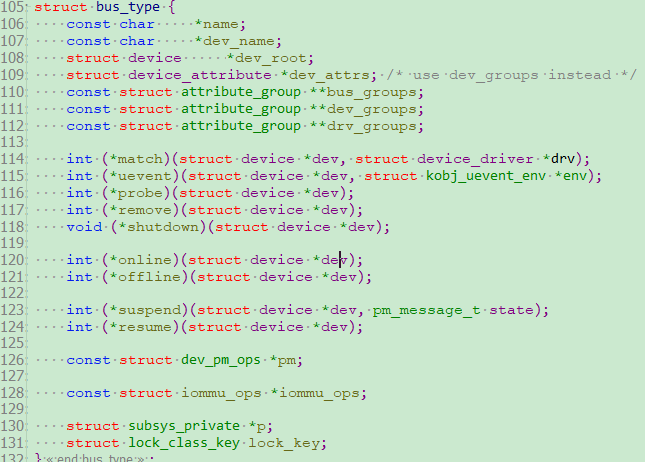
**1. 概述**

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。



(1)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”。

(2)dev\_name，该名称和"Linux设备模型(5)\_device和device driver”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。

(3)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。

(4)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。

(5)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。

(6)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。

(7)probe、remove，这两个回调函数，和device\_driver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。

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

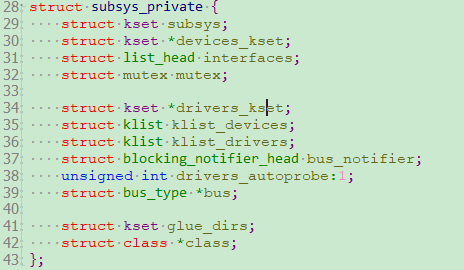
(9)pm，电源管理相关的逻辑，暂不说明。

(10)iommu\_ops，暂不说明。

(11)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中的struct driver\_private类似，在"Linux设备模型(5)\_device和device driver”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：

subsys、devices\_kset、drivers\_kset是三个kset，由"Linux设备模型(2)\_Kobject”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”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

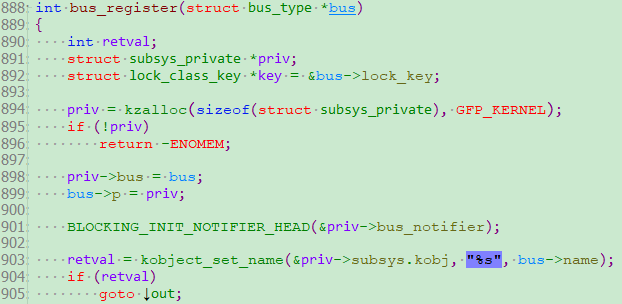
**3. 内部执行逻辑分析**

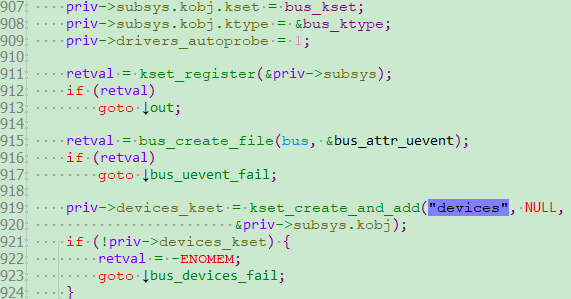
3.1 bus的注册

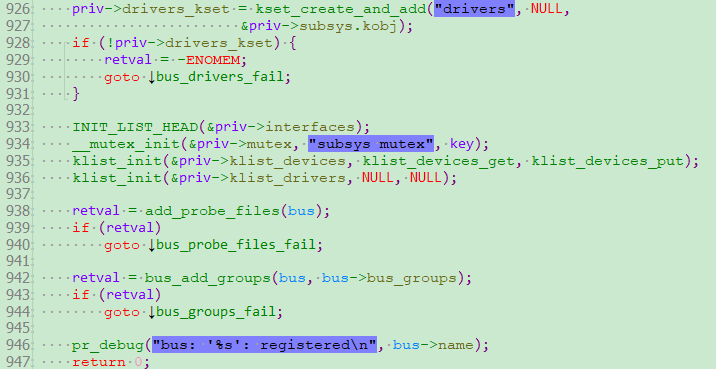
bus的注册是由bus\_register接口实现的，该接口的原型是在include/linux/device.h中声明的，并在drivers/base/bus.c中实现，其原型如下：

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

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

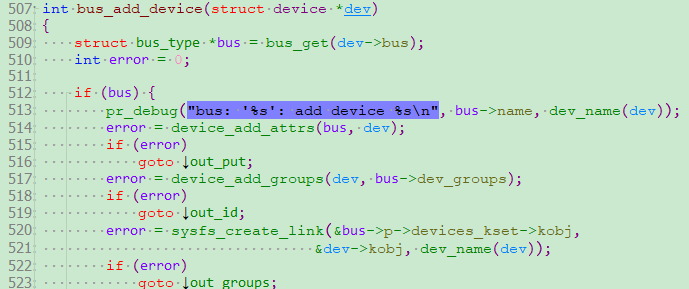
调用add\_probe\_files接口，在bus下添加drivers\_probe和drivers\_autoprobe两个attribute（如/sys/bus/spi/drivers\_probe和/sys/bus/spi/drivers\_autoprobe），其中drivers\_probe允许用户空间程序主动出发指定bus下的device\_driver的probe动作，而drivers\_autoprobe控制是否在device或device\_driver添加到内核时，自动执行probe

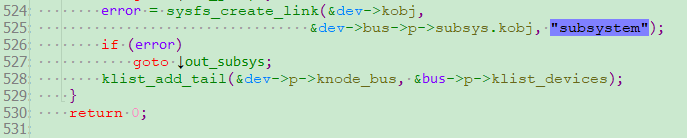
调用bus\_add\_attrs，添加由bus\_attrs指针定义的bus的默认attribute，这些attributes最终会体现在/sys/bus/xxx目录下

3.2 device和device\_driver的添加

我们有在"Linux设备模型(5)\_device和device driver”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：





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

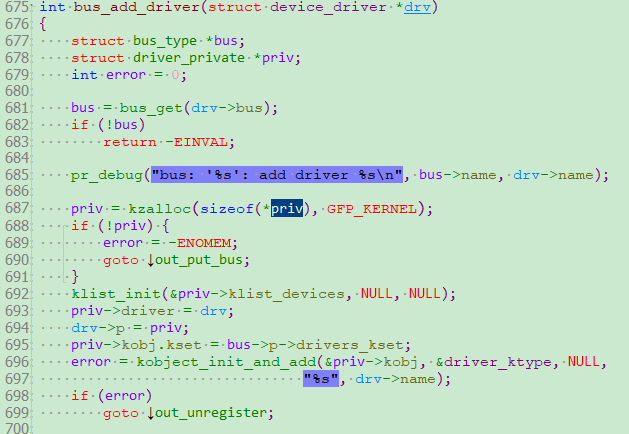
调用sysfs\_create\_link接口，在该设备的sysfs目录中（如/sys/devices/platform/alarm/）中，创建一个指向该设备所在bus目录的链接，取名为subsystem，例如：

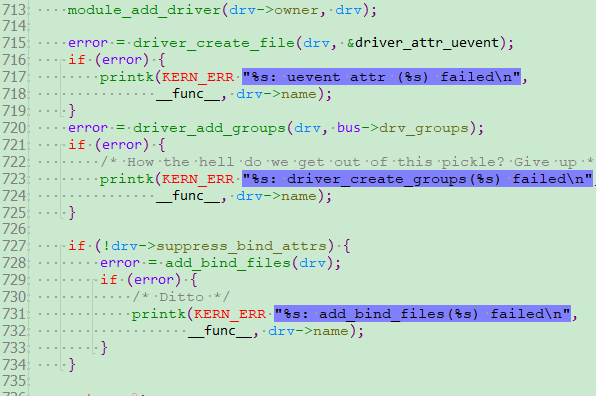
xxx # ls /sys/devices/platform/alarm/subsystem -l

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”中的介绍）

**Linux设备模型(7)\_Class**

1. 概述

在设备模型中，Bus、Device、Device driver等等，都比较好理解，因为它们对应了实实在在的东西，所有的逻辑都是围绕着这些实体展开的。而本文所要描述的Class就有些不同了，因为它是虚拟出来的，只是为了抽象设备的共性。

举个例子，一些年龄相仿、需要获取的知识相似的人，聚在一起学习，就构成了一个班级（Class）。这个班级可以有自己的名称（如295），但如果离开构成它的学生（device），它就没有任何存在意义。另外，班级存在的最大意义是什么呢？是由老师讲授的每一个课程！因为老师只需要讲一遍，一个班的学生都可以听到。不然的话（例如每个学生都在家学习），就要为每人请一个老师，讲授一遍。而讲的内容，大多是一样的，这就是极大的浪费。

设备模型中的Class所提供的功能也一样了，例如一些相似的device（学生），需要向用户空间提供相似的接口（课程），如果每个设备的驱动都实现一遍的话，就会导致内核有大量的冗余代码，这就是极大的浪费。所以，Class说了，我帮你们实现吧，你们会用就行了。

这就是设备模型中Class的功能，再结合内核的注释：A class is a higher-level view of a device that abstracts out low-level implementation details(include/linux/device.h line326)，就容易理解了。

2. 数据结构描述

2.1 struct class

struct class是class的抽象，它的定义如下：

|  |
| --- |
| struct class {  const char \*name;  struct module \*owner;  struct class\_attribute \*class\_attrs;  const struct attribute\_group \*\*dev\_groups;  struct kobject \*dev\_kobj;  int (\*dev\_uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);  char \*(\*devnode)(struct device \*dev, umode\_t \*mode);  void (\*class\_release)(struct class \*class);  void (\*dev\_release)(struct device \*dev);  int (\*suspend)(struct device \*dev, pm\_message\_t state);  int (\*resume)(struct device \*dev);  int (\*shutdown)(struct device \*dev);  const struct kobj\_ns\_type\_operations \*ns\_type;  const void \*(\*namespace)(struct device \*dev);  const struct dev\_pm\_ops \*pm;  struct subsys\_private \*p;  }; |

其实struct class和struct bus很类似，解释如下：

name，class的名称，会在“/sys/class/”目录下体现。

class\_atrrs，该class的默认attribute，会在class注册到内核时，自动在“/sys/class/xxx\_class”下创建对应的attribute文件。

dev\_attrs，该class下每个设备的attribute，会在设备注册到内核时，自动在该设备的sysfs目录下创建对应的attribute文件。

dev\_bin\_attrs，类似dev\_attrs，只不过是二进制类型attribute。

dev\_kobj，表示该class下的设备在/sys/dev/下的目录，现在一般有char和block两个，如果dev\_kobj为NULL，则默认选择char。

dev\_uevent，当该class下有设备发生变化时，会调用class的uevent回调函数。

class\_release，用于release自身的回调函数。

dev\_release，用于release class内设备的回调函数。在device\_release接口中，会依次检查Device、Device Type以及Device所在的class，是否注册release接口，如果有则调用相应的release接口release设备指针。

p，和“Linux设备模型(6)\_Bus”中struct bus结构一样，不再说明。

2.2 struct class\_interface

struct class\_interface是这样的一个结构：它允许class driver在class下有设备添加或移除的时候，调用预先设置好的回调函数（add\_dev和remove\_dev）。那调用它们做什么呢？想做什么都行（例如修改设备的名称），由具体的class driver实现。

该结构的定义如下：

struct class\_interface {

struct list\_head node;

struct class \*class;

int (\*add\_dev) (struct device \*, struct class\_interface \*);

void (\*remove\_dev) (struct device \*, struct class\_interface \*);

};

3. 功能及内部逻辑解析

3.1 class的功能

看完上面的东西，蜗蜗依旧糊里糊涂的，class到底提供了什么功能？怎么使用呢？让我们先看一下现有Linux系统中有关class的状况（这里以input class为例）：

看上面的例子，发现input class也没做什么实实在在的事儿，它（input class）的功能，仅仅是：

在/sys/class/目录下，创建一个本class的目录（input）

在本目录下，创建每一个属于该class的设备的符号链接（如，把“sys/devices/platform/s3c2440-i2c.3/i2c-3/3-0048/input/input2/event2”设备链接到”/sys/class/input/event2”），这样就可以在本class目录下，访问该设备的所有特性（即attribute）

另外，device在sysfs的目录下，也会创建一个subsystem的符号链接，链接到本class的目录

算了，我们还是先分析一下Class的核心逻辑都做了哪些事情，至于class到底有什么用处，可以在后续具体的子系统里面（如input子系统），更为细致的探讨。