**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

12:                             \* its driver.

13:                             \*/

14:

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

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

17:                                 device \*/

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

19:                         core doesn't touch it \*/

20:    struct dev\_pm\_info power;

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

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

32:                             alloc\_coherent mappings as

33:                             not all hardware supports

34:                             64 bit addresses for consistent

35:                             allocations such descriptors. \*/

36:

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

38:

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

40:

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

42:                            override \*/

43: #ifdef CONFIG\_CMA

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

45:                            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 \*/

62:

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，私有数据的指针，具体的driver代码可以把任何需要的内容放在这里，反正设备模型代码不关心。

**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）。