Linux设备驱动开发

# 1. platform相关概念

在Linux3.3.7的设备驱动模型中，关心总线、设备和驱动三个概念，总线将设备和驱动绑定。在系统每注册一个设备的时候，会寻找与之匹配的驱动；相反，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

一个现实的Linux设备和驱动通常都需要挂在一种总线上，对于本身依附于PCI、USB、I2C、SPI等的设备而言，这自然不是问题。但是，在嵌入式系统里面，SoC系统中集成的独立的外设控制器、挂载在SoC内存空间的外设等确不依附于此类总线。基于这一背景，Linux发明了一种虚拟的总线，称为platform总线，相应的设备称为platform\_device，而驱动则称为platform\_driver。

注意，所谓的platform\_device并不是与字符设备、块设备和网络设备并列的概念，而是Linux系统提供的一种附加手段。例如，在S3C6410处理器中，把内部集成的I2C、RTC、SPI、LCD、看门狗等控制器都归纳为platform\_device，而它们本身就是字符设备。

基于platform总线的驱动开发流程如下：

* A 定义初始化platform bus
* b 定义各种platform devices
* c 注册各种platform devices
* d 定义相关platform driver
* e 注册相关platform driver
* f 操作相关设备

# 2. 相关结构体定义

## 2.1 平台相关结构-platform\_device

|  |
| --- |
| struct platform\_device {  const char \* name; /\* 设备名称 \*/  int id;/\*设备id，用于给插入给该总线并且具有相同name的设备编号，如果只有一个设备的话填-1。\*/  struct device dev;/\* 结构体中内嵌的device结构体 \*/  u32 num\_resources; /\* 设备所使用的各类资源数量 \*/  struct resource \* resource; /\*定义平台设备的资源\*/  const struct platform\_device\_id \*id\_entry;  struct mfd\_cell \*mfd\_cell; /\* 多功能设备单元指针 \*/  struct pdev\_archdata archdata; /\* 特定于架构的补充 \*/  }; |

## 2.2 设备的驱动-platform\_driver

这个结构体包含probe()、remove()、shutdown()、suspend()、 resume()函数，通常也需要由驱动实现。

|  |
| --- |
| struct platform\_driver {  int (\*probe)(struct platform\_device \*);  int (\*remove)(struct platform\_device \*);  void (\*shutdown)(struct platform\_device \*);  int (\*suspend)(struct platform\_device \*, pm\_message\_t state);  int (\*resume)(struct platform\_device \*);  struct device\_driver driver;  const struct platform\_device\_id \*id\_table;  }; |

直接填充platform\_driver的suspend()、resume()做电源管理回调的方法目前已经过时，较好的做法是实现platform\_driver的device\_driver中的dev\_pm\_ops结构体成员（后续的Linux电源管理章节会对此进行更为细致的介绍），第2.3节给出了device\_driver的定义。

## 2.3 device\_driver

|  |
| --- |
| /\*\*  \* struct device\_driver – 最基本的设备驱动结构体  \* @name: 设备驱动的名称  \* @bus: 该驱动的设备所属的总线  \* @owner: 模块拥有者  \* @mod\_name: 内嵌模块使用  \* @suppress\_bind\_attrs: 禁止通过sysfs绑定/解绑定  \* @of\_match\_table: 开放固件表  \* @probe: 调用以查询特定设备的存在，该驱动是否可以使用该设备，  \* 然后绑定驱动和该设备  \* @remove: 当设备从系统中移除时被调用，用来解除驱动和设备的绑定  \* @shutdown: 在关机时调用，用来静默设备  \* @suspend: 让设备进入睡眠模式时调用。通常进入低功耗模式  \* @resume: 从睡眠模式唤醒设备时调用该函数  \* @groups: 驱动核心程序创建的默认属性  \* @pm: 匹配该驱动的设备的电源管理  \* @p: 驱动核心程序的私有数据，除它之外，其它不能使用  \*  \* 设备驱动模型跟踪系统已知的所有驱动程序。这样做的主要目的就是保证驱动核心程序  \* 能够匹配新设备和驱动。驱动被系统里的对象所知，其它的事情才会成为可能。设备驱动  \* 程序可以导出独立于任何特定设备的信息和配置变量。  \*/  struct device\_driver {  const char \*name;  struct bus\_type \*bus;  struct module \*owner;  const char \*mod\_name; /\* 内嵌模块使用 \*/  bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/  const struct of\_device\_id \*of\_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;  }; |

与platform\_driver地位对等的i2c\_driver，spi\_driver，usb\_driver，pci\_driver中都包含了device\_driver结构体实例成员。它其实描述了各种xxx\_driver（xxx是总线名称）在驱动意义上的一些共性。

## 2.4 platform总线类型实例platform\_bus\_type

系统为platform总线定义了一个bus\_type的实例platform\_bus\_type，其定义位于  
drivers/base/platform.c文件中，如下代码所示：

|  |
| --- |
| struct bus\_type platform\_bus\_type = {  .name = "platform",  .dev\_attrs = platform\_dev\_attrs,  .match = platform\_match,  .uevent = platform\_uevent,  .pm = &platform\_dev\_pm\_ops,  };  EXPORT\_SYMBOL\_GPL(platform\_bus\_type); |

这里要重点关注match()成员函数，正式此成员表明了platform\_device和platform\_driver之间如何匹配。

|  |
| --- |
| static int platform\_match(struct device \*dev, struct device\_driver \*drv)  {  struct platform\_device \*pdev = to\_platform\_device(dev);  struct platform\_driver \*pdrv = to\_platform\_driver(drv);  /\* Attempt an OF style match first \*/  if (of\_driver\_match\_device(dev, drv))  return 1;  /\* Then try to match against the id table \*/  if (pdrv->id\_table)  return platform\_match\_id(pdrv->id\_table, pdev) != NULL;  /\* fall-back to driver name match \*/  return (strcmp(pdev->name, drv->name) == 0);  } |

platform device IDs假定按照如下格式编码：

"<name><instance>"，

在这里，<name>是设备类型的一个简要描述，比如“pci”或“floppy”； <instance>是设备的编号，比如“0”或“42”。而driver IDs就仅有<name>。

所以，比较platform\_device结构体中的name和driver中的name是否一致就可以了。

# 3. platform初始化

系统启动时初始化时创建了platform\_bus总线设备和platform\_bus\_type总线,platform总线是在内核初始化的时候就注册进了内核。

内核初始化函数kernel\_init()中调用了do\_basic\_setup()，该函数中调用driver\_init()，该函数中调用platform\_bus\_init()，我们看看platform\_bus\_init()函数：

|  |
| --- |
| int \_\_init platform\_bus\_init(void)  {  int error;  early\_platform\_cleanup();  error = device\_register(&platform\_bus);  if (error)  return error;  error = bus\_register(&platform\_bus\_type);  if (error)  device\_unregister(&platform\_bus);  return error;  } |

early\_platform\_cleanup()函数的作用是，清除platform设备链表。该函数把设备名为platform的设备platform\_bus注册到系统中，其它的platform的设备都会以它为parent。它在sysfs中的目录下，即/sys/devices/platform。platform\_bus 总线也是设备，所以要进行设备的注册。

|  |
| --- |
| struct device platform\_bus = {  .init\_name = "platform",  };  EXPORT\_SYMBOL\_GPL(platform\_bus); |

device\_register(&platform\_bus)将platform\_bus作为一个设备注册，出现在sys文件系统的device目录下。

bus\_register(&platform\_bus\_type)注册了platform\_bus\_type总线。将出现在sys文件系统bus目录下，创建一个platform的目录以及相关属性文件。默认platform\_bus\_type中没有定义probe函数。

platform\_bus\_type 中的match函数是在设备匹配驱动时调用，uevent函数在产生事件时使用。

platform\_match函数在当属于platform的设备或者驱动注册到内核时就会调用，完成设备与驱动的匹配工作。关于platform\_match函数请参考第2.3小节。

|  |
| --- |
| static int platform\_match(struct device \*dev, struct device\_driver \*drv)  {  struct platform\_device \*pdev = to\_platform\_device(dev);  struct platform\_driver \*pdrv = to\_platform\_driver(drv);  /\* Attempt an OF style match first \*/  if (of\_driver\_match\_device(dev, drv))  return 1;  /\* Then try to match against the id table \*/  if (pdrv->id\_table)  return platform\_match\_id(pdrv->id\_table, pdev) != NULL;  /\* fall-back to driver name match \*/  return (strcmp(pdev->name, drv->name) == 0);  }  static const struct platform\_device\_id \*platform\_match\_id(  const struct platform\_device\_id \*id,  struct platform\_device \*pdev)  {  while (id->name[0]) {  if (strcmp(pdev->name, id->name) == 0) {  pdev->id\_entry = id;  return id;  }  id++;  }  return NULL;  } |

不能看出，如果pdrv的id\_table数组中包含了pdev->name，或者drv->name和pdev->name名字相同，都会认为是匹配成功。id\_table数组是为了应对那些对应设备和驱动的drv->name和pdev->name名字不同的情况。

那么我们再看看platform\_uevent()函数：platform\_uevent热插拔操作函数。

|  |
| --- |
| static int platform\_uevent(struct device \*dev, struct kobj\_uevent\_env \*env)  {  struct platform\_device \*pdev = to\_platform\_device(dev);  int rc;  /\* Some devices have extra OF data and an OF-style MODALIAS \*/  rc = of\_device\_uevent(dev,env);  if (rc != -ENODEV)  return rc;  add\_uevent\_var(env, "MODALIAS=%s%s", PLATFORM\_MODULE\_PREFIX,  pdev->name);  return 0;  } |

添加了MODALIAS环境变量，我们回顾一下：platform\_bus. parent->kobj->kset->uevent\_ops为device\_uevent\_ops，bus\_uevent\_ops的定义如下

|  |
| --- |
| static struct kset\_uevent\_ops device\_uevent\_ops = {  .filter = dev\_uevent\_filter,  .name = dev\_uevent\_name,  .uevent = dev\_uevent,  }; |

当调用device\_add()时会调用kobject\_uevent(&dev->kobj, KOBJ\_ADD)产生一个事件，这个函数中会调用相应的kset\_uevent\_ops的uevent函数。

# 4. platform设备的注册

## 4.1 第一种平台设备注册方式

platform\_device是静态的全局设备，即platform\_device结构的成员已经初始化完成。直接将platform\_device注册到platform总线上就可以了。platform\_device\_register和device\_register的区别:

1. 主要是有没有resource的区别，前者的结构体包含后面，并且增加了struct resource结构体成员，后者没有。platform\_device\_register在device\_register的基础上增加了struct resource部分的注册。由此，可以看出，platform\_device\_register机制与device\_driver的主要区别就在于resource。前者适合于具有独立资源设备的描述，后者则不是。
2. 其实linux的各种其他驱动机制的基础都是device\_driver。只不过是增加了部分功能，适合于不同的应用场合。

|  |
| --- |
| /\*\*  \* platform\_device\_register – 添加一个platform设备  \* @pdev: 正要添加的设备  \*/  int platform\_device\_register(struct platform\_device \*pdev)  {  device\_initialize(&pdev->dev);  arch\_setup\_pdev\_archdata(pdev);  return platform\_device\_add(pdev);  }  EXPORT\_SYMBOL\_GPL(platform\_device\_register);  /\*\*  \* platform\_device\_add - 添加platform设备到设备树中。  \* @pdev: 正要添加的设备  \*  \* 这是platform\_device\_register()函数的第2部分，尽管如果pdev是由  \* platform\_device\_alloc()函数分配的，可以单独调用。  \*/  int platform\_device\_add(struct platform\_device \*pdev)  {  int i, ret = 0;  if (!pdev)  return -EINVAL;  if (!pdev->dev.parent)  pdev->dev.parent = &platform\_bus;// 设置父节点，即platform\_bus作为总线设备的父节点，其余的platform设备都是它的子节点。  // platform\_bus是一个设备，platform\_bus\_type才是真正的总线  // 设置platform总线，指定bus类型为platform\_bus\_type  pdev->dev.bus = &platform\_bus\_type;  // 设置pdev->dev中的kobj的name字段，将platform下的名字传到内部device，  // 最终会传到kobj  if (pdev->id != -1)  dev\_set\_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);  else  dev\_set\_name(&pdev->dev, "%s", pdev->name);  // 初始化资源并将其分配给它，每个资源的它的parent不存在则根据flags域  // 设置parent，flags为IORESOURCE\_MEM，则表示的资源为I/O映射内存，flags  // flags为IORESOURCE\_IO，则所表示的资源为I/O端口。  for (i = 0; i < pdev->num\_resources; i++) {  struct resource \*p, \*r = &pdev->resource[i];  // 资源名称为NULL，则把设备名称设置给它  if (r->name == NULL)  r->name = dev\_name(&pdev->dev);    // 取得资源的父节点，资源在内核中也是层次安排的。  p = r->parent;  if (!p) {  // 如果父节点为NULL，并且资源类型为IORESOURCE\_MEM，则把  // 父节点设置为iomem\_resource；否则设置为ioport\_resource  if (resource\_type(r) == IORESOURCE\_MEM)  p = &iomem\_resource;  else if (resource\_type(r) == IORESOURCE\_IO)  p = &ioport\_resource;  }    // 从父节点申请资源，也就是出现在父节点目录层次下  if (p && insert\_resource(p, r)) {  printk(KERN\_ERR  "%s: failed to claim resource %d\n",  dev\_name(&pdev->dev), i);  ret = -EBUSY;  goto failed;  }  }  pr\_debug("Registering platform device '%s'. Parent at %s\n",  dev\_name(&pdev->dev), dev\_name(pdev->dev.parent));  // device\_creat()创建一个设备并注册到内核驱动架构  // device\_add()注册一个设备到内核，少了一个创建设备；  // 就在这里把设备注册到总线设备上，标准设备注册，即在sys文件系统中添加  // 目录和各种属性文件  ret = device\_add(&pdev->dev);  if (ret == 0)  return ret;  failed:  while (--i >= 0) {  struct resource \*r = &pdev->resource[i];  unsigned long type = resource\_type(r);  if (type == IORESOURCE\_MEM || type == IORESOURCE\_IO)  release\_resource(r);  }  return ret;  }  EXPORT\_SYMBOL\_GPL(platform\_device\_add); |

## 4.2 第二种平台设备的注册方式

先分配一个platform\_device结构，对其进行资源等的初始化；之后再对其进行注册，再调用platform\_device\_register()函数。

|  |
| --- |
| struct platform\_device \* platform\_device\_alloc(const char \*name, int id)  {  struct platform\_object \*pa;  /\*  struct platform\_object {  struct platform\_device pdev;  char name[1];  };  \*/  pa = kzalloc(sizeof(struct platform\_object) + strlen(name), GFP\_KERNEL);//该函数首先为platform设备分配内存空间  if (pa) {  strcpy(pa->name, name);  pa->pdev.name = pa->name;//初始化platform\_device设备的名称  pa->pdev.id = id;//初始化platform\_device设备的id  device\_initialize(&pa->pdev.dev);//初始化platform\_device内嵌的device  pa->pdev.dev.release = platform\_device\_release;  }  return pa ? &pa->pdev : NULL;  } |

一个更好的方法是，通过下面的函数platform\_device\_register\_simple()动态创建一个设备，并把这个设备注册到系统中：

|  |
| --- |
| /\*\*  \* platform\_device\_register\_simple – 添加一个platform设备及其资源文件  \* @name: 我们添加设备的基本名称  \* @id: 实例化id  \* @res: 分配给设备的资源集合  \* @num: 资源数量  \*  \* 这个函数是创建一个简单的platform设备，需要最少的资源和内存管理。分配给设备的  \* 释放内存的释放函数允许驱动使用的设备不用等待最后一个引用被放下就能够被卸载。  \*  \* 该接口主要用于直接探测硬件的传统驱动程序。因为这些驱动程序本身自己就创建sysfs  \* 设备节点，而不是让系统基础部分处理这类设备枚举任务，所以它们不完全符合Linux  \* 驱动程序模型。特别是，当这样的驱动程序被构建为模块时，它们不能被“热插拔”。  \*  \* 成功返回 &struct platform\_device 指针，或者是被返回ERR\_PTR()  \*/  static inline struct platform\_device \*platform\_device\_register\_simple(  const char \*name, int id,  const struct resource \*res, unsigned int num)  {  return platform\_device\_register\_resndata(NULL, name, id,  res, num, NULL, 0);  } |

platform\_device\_register\_resndatah() 函数的内容如下，

|  |
| --- |
| /\*\*  \* platform\_device\_register\_resndata – 使用resource和特定于platform的数据添加一  \* 个platform设备  \*  \* @parent: 添加设备的父设备  \* @name: 添加设备的基本名称  \* @id: 添加设备的id  \* @res: 需要为添加设备分配的资源集合  \* @num: 资源数量  \* @data: platform设备的特定于platform的数据  \* @size: 数据大小  \*  \* 成功返回&struct platform\_device 指针，失败返回ERR\_PTR()。  \*/  static inline struct platform\_device \*platform\_device\_register\_resndata(  struct device \*parent, const char \*name, int id,  const struct resource \*res, unsigned int num,  const void \*data, size\_t size) {  struct platform\_device\_info pdevinfo = {  .parent = parent,  .name = name,  .id = id,  .res = res,  .num\_res = num,  .data = data,  .size\_data = size,  .dma\_mask = 0,  };  return platform\_device\_register\_full(&pdevinfo);  } |

|  |
| --- |
| /\*\*  \* platform\_device\_register\_full – 使用资源和数据添加设备  \*  \* @pdevinfo: 用来创建设备的数据  \*  \* 成功返回&struct platform\_device 指针，失败返回ERR\_PTR()。  \*/  struct platform\_device \*platform\_device\_register\_full(  const struct platform\_device\_info \*pdevinfo)  {  int ret = -ENOMEM;  struct platform\_device \*pdev;  pdev = platform\_device\_alloc(pdevinfo->name, pdevinfo->id);  if (!pdev)  goto err\_alloc;  pdev->dev.parent = pdevinfo->parent;  if (pdevinfo->dma\_mask) {  /\*  \* This memory isn't freed when the device is put,  \* I don't have a nice idea for that though. Conceptually  \* dma\_mask in struct device should not be a pointer.  \* See http://thread.gmane.org/gmane.linux.kernel.pci/9081  \*/  pdev->dev.dma\_mask =  kmalloc(sizeof(\*pdev->dev.dma\_mask), GFP\_KERNEL);  if (!pdev->dev.dma\_mask)  goto err;  \*pdev->dev.dma\_mask = pdevinfo->dma\_mask;  pdev->dev.coherent\_dma\_mask = pdevinfo->dma\_mask;  }  ret = platform\_device\_add\_resources(pdev,  pdevinfo->res, pdevinfo->num\_res);  if (ret)  goto err;  ret = platform\_device\_add\_data(pdev,  pdevinfo->data, pdevinfo->size\_data);  if (ret)  goto err;  ret = platform\_device\_add(pdev);  if (ret) {  err:  kfree(pdev->dev.dma\_mask);  err\_alloc:  platform\_device\_put(pdev);  return ERR\_PTR(ret);  }  return pdev;  }  EXPORT\_SYMBOL\_GPL(platform\_device\_register\_full); |

该函数就是调用了platform\_device\_alloc()和platform\_device\_add()函数来创建注册platform设备，并根据资源和数据分别调用platform\_device\_add\_resources和platform\_device\_add\_data。看一下platform\_device\_add\_resources()函数：

|  |
| --- |
| /\*\*  \* platform\_device\_add\_resources - add resources to a platform device  \* @pdev: platform device allocated by platform\_device\_alloc to add resources to  \* @res: set of resources that needs to be allocated for the device  \* @num: number of resources  \*  \* 为platform设备添加资源的拷贝。当设备被释放时，资源所占用的内存也会被释放。  \*/  int platform\_device\_add\_resources(struct platform\_device \*pdev,  const struct resource \*res, unsigned int num)  {  struct resource \*r = NULL;  if (res) {  r = kmemdup(res, sizeof(struct resource) \* num, GFP\_KERNEL);  if (!r)  return -ENOMEM;  }  kfree(pdev->resource);  pdev->resource = r;  pdev->num\_resources = num;  return 0;  }  EXPORT\_SYMBOL\_GPL(platform\_device\_add\_resources); |

5. platform设备驱动的注册

我们在设备驱动模型的分析中已经知道驱动在注册时要调用driver\_register()，platform驱动的注册函数platform\_driver\_register()同样也是进行其它的一些初始化后调用  
driver\_register()将驱动注册到platform\_bus\_type总线上。

|  |
| --- |
| /\*\*  \* platform\_driver\_register – 为platform设备注册驱动  \* @drv: platform 驱动结构  \*/  int platform\_driver\_register(struct platform\_driver \*drv)  {  // 它将要注册到总线设置成为platform\_bus\_type，这个很重要，因为driver和  // device是通过bus联系在一起的，具体在本例中是通过platform\_bus\_type中  // 注册的回调函数和属性来实现的，driver与device的匹配就是通过  // platform\_bus\_type注册的回调函数platform\_match()来完成的。  drv->driver.bus = &platform\_bus\_type;  if (drv->probe)  drv->driver.probe = platform\_drv\_probe;  if (drv->remove)  drv->driver.remove = platform\_drv\_remove;  if (drv->shutdown)  drv->driver.shutdown = platform\_drv\_shutdown;  return driver\_register(&drv->driver);  }  EXPORT\_SYMBOL\_GPL(platform\_driver\_register); |

然后设定了platform\_driver中的driver成员的probe()，remove()和shutdown()函数。

|  |
| --- |
| static int platform\_drv\_probe(struct device \*\_dev)  {  struct platform\_driver \*drv = to\_platform\_driver(\_dev->driver);  struct platform\_device \*dev = to\_platform\_device(\_dev);  return drv->probe(dev);  } |

调用platform\_driver的probe()函数，这个函数一般由用户自己实现。例如，下面的结构体，回调的是serial8250\_prove()函数：

|  |
| --- |
| static struct platform\_driver serial8250\_isa\_driver = {  .probe = serial8250\_probe,  .remove = \_\_devexit\_p(serial8250\_remove),  .suspend = serial8250\_suspend,  .resume = serial8250\_resume,  .driver = {  .name = "serial8250",  .owner = THIS\_MODULE,  },  }; |

另外，

|  |
| --- |
| static int platform\_drv\_remove(struct device \*\_dev)  {  struct platform\_driver \*drv = to\_platform\_driver(\_dev->driver);  struct platform\_device \*dev = to\_platform\_device(\_dev);  return drv->remove(dev);  }  static void platform\_drv\_shutdown(struct device \*\_dev)  {  struct platform\_driver \*drv = to\_platform\_driver(\_dev->driver);  struct platform\_device \*dev = to\_platform\_device(\_dev);  drv->shutdown(dev);  } |

我们再来看driver\_register(&drv->driver)这个函数的作用：

|  |
| --- |
| /\*\*  \* driver\_register – 使用bus注册驱动  \* @drv: 要注册的驱动  \*  \* We pass off most of the work to the bus\_add\_driver() call,  \* since most of the things we have to do deal with the bus  \* structures.  \*  \*/  int driver\_register(struct device\_driver \*drv)  {  int ret;  struct device\_driver \*other;  BUG\_ON(!drv->bus->p);  if ((drv->bus->probe && drv->probe) ||  (drv->bus->remove && drv->remove) ||  (drv->bus->shutdown && drv->shutdown))  printk(KERN\_WARNING "Driver '%s' needs updating - please use "  "bus\_type methods\n", drv->name);  other = driver\_find(drv->name, drv->bus);  if (other) {  put\_driver(other);  printk(KERN\_ERR "Error: Driver '%s' is already registered, "  "aborting...\n", drv->name);  return -EBUSY;  }  ret = bus\_add\_driver(drv);  if (ret)  return ret;  ret = driver\_add\_groups(drv, drv->groups);  if (ret)  bus\_remove\_driver(drv);  return ret;  }  EXPORT\_SYMBOL\_GPL(driver\_register); |

总结：

1、从这三个函数的代码可以看到，又找到了相应的platform\_driver和platform\_device，然后调用platform\_driver的probe、remove、shutdown函数。这是一种高明的做法：

在不针对某个驱动具体的probe、remove、shutdown指向的函数，而通过上三个过度函数来找到platform\_driver，然后调用probe、remove、shutdown接口。

如果设备和驱动都注册了，就可以通过bus ->match、bus->probe或driver->probe进行设备驱动匹配了。

2、驱动注册的时候platform\_driver\_register() -> driver\_register() -> bus\_add\_driver() -> driver\_attach() -> bus\_for\_each\_dev()，

对每个挂在虚拟的platform bus的设备作\_\_driver\_attach() -> driver\_probe\_device() -> drv -> bus -> match()==platform\_match()->比较strncmp(pdev->name, drv->name, BUS\_ID\_SIZE)，如果相符就调用platform\_drv\_probe()->driver->probe()，如果probe成功则绑定该设备到该驱动。