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，内核 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。

```
struct bus_type {
    const char      *name;
    const char      *dev_name;
    struct device       *dev_root;
    struct device_attribute *dev_attrs; /* use dev_groups instead */
    const struct attribute_group **bus_groups;
    const struct attribute_group **dev_groups;
    const struct attribute_group **drv_groups;

    int (*match)(struct device *dev, struct device_driver *drv);
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown)(struct device *dev);

    int (*online)(struct device *dev);
    int (*offline)(struct device *dev);

    int (*suspend)(struct device *dev, pm_message_t state);
    int (*resume)(struct device *dev);

    const struct dev_pm_ops *pm;

    const struct iommu_ops *iommu_ops;

    struct subsys_private *p;
    struct lock_class_key lock_key;
};«end bus_type»
```

(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。

```
struct subsys_private {
    struct kset subsys;
    struct kset *devices_kset;
    struct list_head interfaces;
    struct mutex mutex;

    struct kset *drivers_kset;
    struct klist klist_devices;
    struct klist klist_drivers;
    struct blocking_notifier_head bus_notifier;
    unsigned int drivers_autoprobe:1;
    struct bus_type *bus;

    struct kset glue_dirs;
    struct class *class;
};
```

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);

```
888: int bus_register(struct bus_type *bus)
889: {
890:     int retval;
891:     struct subsys_private *priv;
892:     struct lock_class_key *key = &bus->lock_key;
893:
894:     priv = kzalloc(sizeof(struct subsys_private), GFP_KERNEL);
895:     if (!priv)
896:         return -ENOMEM;
897:
898:     priv->bus = bus;
899:     bus->p = priv;
900:
901:     BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);
902:
903:     retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
904:     if (retval)
905:         goto out;
```

```
907:     priv->subsys.kobj.kset = bus_kset;
908:     priv->subsys.kobj.ktype = &bus_ktype;
909:     priv->drivers_autoprobe = 1;
910:
911:     retval = kset_register(&priv->subsys);
912:     if (retval)
913:         goto out;
914:
915:     retval = bus_create_file(bus, &bus_attr_uevent);
916:     if (retval)
917:         goto bus_uevent_fail;
918:
919:     priv->devices_kset = kset_create_and_add("devices", NULL,
920:                                              &priv->subsys.kobj);
921:     if (!priv->devices_kset) {
922:         retval = -ENOMEM;
923:         goto bus_devices_fail;
924:     }
```

```
926:     priv->drivers_kset = kset_create_and_add("drivers", NULL,
927:                                              &priv->subsys.kobj);
928:     if (!priv->drivers_kset) {
929:         retval = -ENOMEM;
930:         goto bus_drivers_fail;
931:     }
932:
933:     INIT_LIST_HEAD(&priv->interfaces);
934:     __mutex_init(&priv->mutex, "subsys mutex", key);
935:     klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
936:     klist_init(&priv->klist_drivers, NULL, NULL);
937:
938:     retval = add_probe_files(bus);
939:     if (retval)
940:         goto bus_probe_files_fail;
941:
942:     retval = bus_add_groups(bus, bus->bus_groups);
943:     if (retval)
944:         goto bus_groups_fail;
945:
946:     pr_debug("bus: '%s': registered\n", bus->name);
947:     return 0;
```

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：

```
int bus_add_device(struct device *dev)
{
    struct bus_type *bus = bus_get(dev->bus);
    int error = 0;

    if (bus) {
        pr_debug("bus: '%s': add device %s\n", bus->name, dev_name(dev));
        error = device_add_attrs(bus, dev);
        if (error)
            goto out_put;
        error = device_add_groups(dev, bus->dev_groups);
        if (error)
            goto out_id;
        error = sysfs_create_link(&bus->p->devices_kset->kobj,
                                  &dev->kobj, dev_name(dev));
        if (error)
            goto out_groups;
```

```
        error = sysfs_create_link(&dev->kobj,
                                  &dev->bus->p->subsys.kobj, "subsystem");
        if (error)
            goto out_subsys;
        klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices);
    }
    return 0;
```

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：

```
675 int bus_add_driver(struct device_driver *drv)
676 {
677     struct bus_type *bus;
678     struct driver_private *priv;
679     int error = 0;
680
681     bus = bus_get(drv->bus);
682     if (!bus)
683         return -EINVAL;
684
685     pr_debug("bus: '%s': add driver %s\n", bus->name, drv->name);
686
687     priv = kzalloc(sizeof(*priv), GFP_KERNEL);
688     if (!priv) {
689         error = -ENOMEM;
690         goto out_put_bus;
691     }
692     klist_init(&priv->klist_devices, NULL, NULL);
693     priv->driver = drv;
694     drv->p = priv;
695     priv->kobj.kset = bus->p->drivers_kset;
696     error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL,
697                                  "%s", drv->name);
698     if (error)
699         goto out_unregister;
700
```

```
713     module_add_driver(drv->owner, drv);
714
715     error = driver_create_file(drv, &driver_attr_uevent);
716     if (error) {
717         printk(KERN_ERR "%s: uevent attr (%s) failed\n",
718                __func__, drv->name);
719     }
720     error = driver_add_groups(drv, bus->drv_groups);
721     if (error) {
722         /* How the hell do we get out of this pickle? Give up */
723         printk(KERN_ERR "%s: driver_create_groups(%s) failed\n",
724                __func__, drv->name);
725     }
726
727     if (!drv->suppress_bind_attrs) {
728         error = add_bind_files(drv);
729         if (error) {
730             /* Ditto */
731             printk(KERN_ERR "%s: add_bind_files(%s) failed\n",
732                    __func__, drv->name);
733         }
734     }
735
```

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。