第 8 章 驱动模型与总线驱动

Linux 内核将设备视为文件，每个设备由一个设备文件表示，内核采用与普通文件相同的接口操作设备，即 file_operations 结构体实例。内核为每个设备赋予一个系统内唯一的设备号，保存在设备文件 inode 中。内核建立了设备驱动数据库，用于管理设备驱动程序（驱动数据结构实例）。

设备文件是特殊文件，在打开设备文件时，将做特殊处理，通过设备号检索设备驱动数据库，找到设备对应的驱动数据结构实例，并将设备驱动数据结构实例中包含的 file_operations 实例，赋予 file 实例，从而可通过此实例操作设备。

内核根据设备与系统之间交换数据的方法，将设备分为字符设备和块设备。字符设备一般只能顺序访问，数据传输量较低，如：鼠标、键盘等。块设备是可以按固定数目字节数（数据块）随机访问的设备，主要是存储设备，如：硬盘、U 盘、光盘等。

字符设备、块设备驱动程序在内核设备驱动数据库中由特定的数据结构表示（cdev 或 gendisk），并分别进行管理。设备驱动程序的主要工作就是定义设备专属的 file_operations 实例，定义驱动数据结构实例，并向数据库注册。

系统中的设备不是固定不变的，是可以动态地接入或移出系统，因此设备驱动程序也需要具有动态注册/移出的机制。内核定义了通用驱动模型用于管理内核中的设备和驱动程序（驱动数据结构实例），当向系统接入设备时，驱动模型需要将相应的驱动程序注册进内核，移出设备时将其从设备驱动数据库移除。

通用驱动模型中定义了总线、设备类、设备和驱动等概念，设备用于描述设备硬件信息，硬件信息将传递给驱动，驱动用于匹配设备，匹配成功则注册设备驱动程序、激活设备等。通用驱动模型中的驱动是驱动程序的管理者，负责注册/移除匹配设备的驱动数据结构实例，注意其与驱动程序的区别。

设备和驱动挂接在总线上，在向总线注册设备/驱动时，会触发设备与驱动之间的匹配，匹配成功将完成设备驱动数据结构实例的注册，可理解成加载设备驱动程序到内核。

本章先介绍设备在内核中的表示方法、设备驱动数据库以及通用驱动模型的实现，然后介绍几种常用总线驱动的实现，具体设备驱动程序在后面章节中再做介绍。

8.1 驱动模型概述

设备驱动程序的主要工作是实现设备驱动对应的 cdev 或 gendisk 结构体实例并向内核注册。现代计算机系统中外部设备都通过总线与 CPU 连接，简单的设备驱动数据库没有解决设备与总线的连接问题及总线驱动的问题。另外，外部设备可以动态地从系统中接入或移出，相应的设备驱动数据结构实例也应该能够动态地添加或移除。

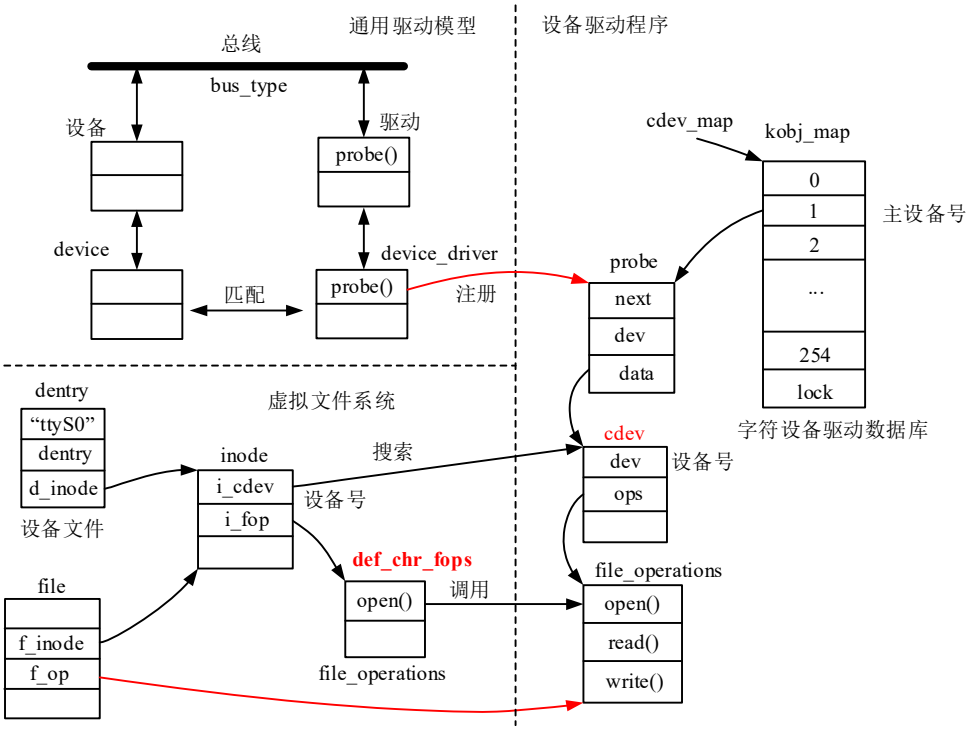
为此，内核在 2.6 版本引入了通用驱动模型的概念，驱动模型中定义了总线、设备类、设备、驱动等概念，用于管理系统中的实际总线、硬件设备和驱动程序（设备驱动数据结构实例）。总线管理着其下设备和驱动，添加设备或驱动时，会触发两者之间的匹配，匹配成功，将由驱动注册设备驱动数据结构实例，并激活设备。

8.1.1 组织结构

通用驱动模型中定义了总线 bus_type、设备类 class、设备 device 和驱动 device_driver 等数据结构。总线 bus_type 表示系统中实际的总线（也可以是虚拟总线，芯片内部总线等），如：USB 总线、SPI 总线等，设备类 class 用于管理同一类型的设备，device 表示设备的硬件信息，device_driver 表示驱动（用于管理设备驱动程序）。下图示意了字符设备的通用驱动模型框架，块设备也类似，只不过它也有专门的设备驱动数据库。

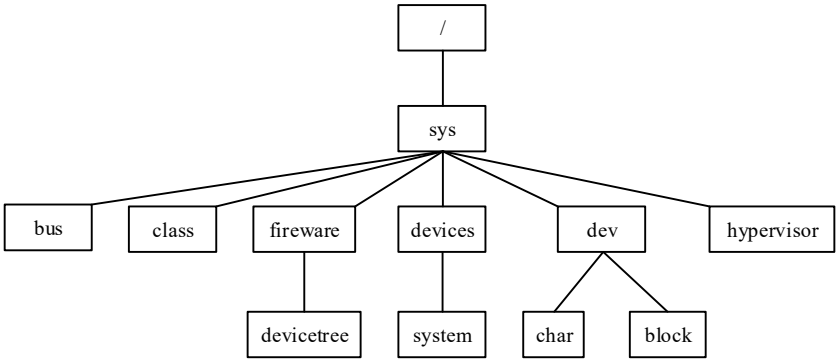
同一条总线上的设备 device 实例挂接到对应总线 bus_type 实例的设备链表上，相应的驱动 device_driver

实例挂接在总线的驱动链表上。当向总线注册设备或驱动时（注册设备时可能动态创建设备文件），将触发总线的匹配操作，寻找匹配的驱动或设备（扫描驱动或设备链表）。若匹配成功则调用总线 bus_type 实例（或驱动 device_driver 实例）中定义的 probe()函数，probe()函数完成设备驱动数据结构实例的定义和注册。进程可通过设备文件操作设备，设备文件中包含设备号，依此找到设备驱动数据结构实例，从而实现对设备的操作。



8.1.2 初始化

内核通过 kobject 结构体跟踪和管理驱动模型中的各数据结构实例，并导出到 sysfs 文件系统。在初始化函数中将创建各目录项，用于管理驱动模型中的数据结构实例，如下图所示。



以上目录项在驱动模型初始化函数 driver_init()中创建，函数调用关系如下：
kernel_init->kernel_init_freeable()->do_basic_setup()->driver_init()。

driver_init()函数定义在/drivers/base/init.c 文件内，代码如下：

```
void __init driver_init(void)
{
    devtmpfs_init();        /*注册 devtmpfs 文件系统类型，创建内核线程，/drivers/base/devtmpfs.c*/
    devices_init();         /*设备初始化，/drivers/base/core.c*/
```

```
    buses_init();              /*总线初始化，/drivers/base/bus.c*/
    classes_init();            /*设备类初始化，/drivers/base/class.c*/
    firmware_init();           /*固件初始化，/drivers/base/firmware.c*/
    hypervisor_init();         /*/drivers/base/hypervisor.c*/
    platform_bus_init();       /*平台总线初始化，/drivers/base/platform.c*/
    cpu_dev_init();            /*注册 cpu_subsys 总线，挂载 cpu 设备到此总线，/drivers/base/cpu.c*/
    memory_dev_init();         /*注册 memory_subsys 总线和内存设备，/drivers/base/memory.c*/
    container_dev_init();      /*注册 container_subsys 总线，/drivers/base/container.c*/
    of_core_init();            /*设备树初始化，见下文，/drivers/of/base.c*/
}
```

以上初始化函数简介如下：

●**devtmpfs_init()**：在/drivers/base/devtmpfs.c 文件内实现，需选择 DEVTMPFS 配置选项，否则为空操作。函数完成 devtmpfs 文件系统类型的注册以及创建设备文件守护线程 kdevtmpfs 的创建和启动，此线程用于在注册 device 实例时自动创建设备文件。

●**devices_init()**：在/drivers/base/core.c 文件内实现，代码详见本章下文。函数在 sysfs 文件系统中创建以下目录：

/sys/devices/：导出驱动模型中 device 实例。

/sys/dev/block/：导出块设备，其下文件名称为“主设备号：从设备号”，是到/sys/devices/下设备的符号链接。

/sys/dev/char/：导出字符设备，其下文件名称为“主设备号：从设备号”，是到/sys/devices/下设备的符号链接。

●**buses_init()**：总线初始化函数，定义在/drivers/base/bus.c 文件内，详见本章下文。函数在 sysfs 文件系统内创建以下目录：/sys/bus/，/sys/devices/system，前者用于导出注册的总线。

●**classes_init()**：设备类初始化函数，在/drivers/base/class.c 文件内实现，在 sysfs 文件系统中创建目录/sys/class/，用于导出注册的设备类，详见本章下文。

●**firmware_init()**：在/drivers/base/firmware.c 文件内实现，代码如下：

```
int __init firmware_init(void)
{
    firmware_kobj = kobject_create_and_add("firmware", NULL);
    ...
    return 0;
}
```

函数在 sysfs 文件系统中创建目录/sys/fireware，用于导出固件信息。

●**hypervisor_init()**：在/drivers/base/hypervisor.c 文件内实现，代码如下：

```
int __init hypervisor_init(void)
{
    hypervisor_kobj = kobject_create_and_add("hypervisor", NULL);
    ...
    return 0;
}
```

函数在 sysfs 文件系统中创建目录/sys/hypervisor。

●**platform_bus_init()：**平台 platform 总线初始化函数，在/drivers/base/platform.c 文件内实现，主要完成平台总线的注册，详见本章下文。

● **cpu_dev_init()：**在/drivers/base/cpu.c 文件内实现，代码如下：

```
void __init cpu_dev_init(void)
{
  if (subsys_system_register(&cpu_subsys, cpu_root_attr_groups))  /*注册子系统，/drivers/base/bus.c*/
        panic("Failed to register CPU subsystem");

   cpu_dev_register_generic();    /*挂载 cpu 设备至 cpu_subsys 总线，/drivers/base/cpu.c*/
}
```

cpu_dev_init()函数完成 cpu_subsys 总线的注册和 cpu 设备的挂载。

注册子系统的函数在/drivers/base/bus.c 文件内实现，它先注册总线，再将总线以设备形式注册到内核。

若选择了 GENERIC_CPU_DEVICES 配置选项，CPU 将会当作设备（由 cpu 结构体表示）挂载到总线 cpu_subsys 上。

●**memory_dev_init()：**函数在/drivers/base/memory.c 文件内实现，主要完成内存总线类型 memory_subsys 及内存设备的注册。

●**container_dev_init()：**在/drivers/base/container.c 文件内实现，主要完成 container_subsys 总线（子系统）的注册。

●**of_core_init()：**在/drivers/of/base.c 文件内实现，需选择 OF 配置选项，表示内核支持设备树，否则为空操作。函数内主要工作是创建/sys/fireware/devicetree/目录，并将设备树节点导出到此目录下。

8.2 设备管理

在介绍通用驱模型前，先介绍一下设备在内核中的表示（设备文件），以及内核对设备驱动程序的管理（设备驱动数据库）。

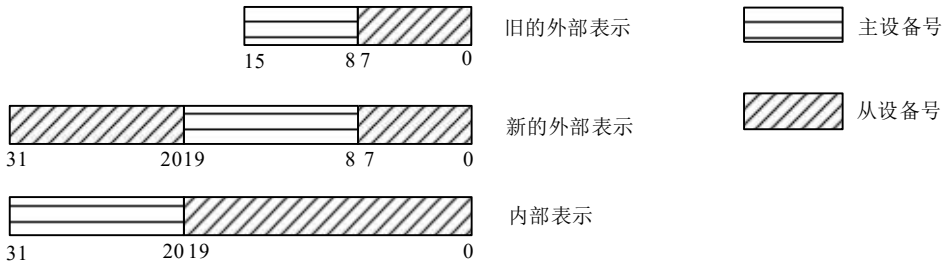
设备在 Linux 系统中由设备文件表示，设备文件保存的主要信息有设备类型和设备号等。设备类型即字符设备或块设备，设备号用于在打开设备文件时在设备驱动数据库中搜索设备驱动。

8.2.1 设备号

设备号由主设备号和从设备号组成，字符设备与块设备分开编号，也就是说这两种类型设备可以使用相同的设备号，内核通过设备类型进行区分。一般同一类型的设备具有相同的主设备号，通过从设备号来区分具体的设备。

最初设备号由 16 位整数表示，高 8 位表示主设备号，低 8 位表示从设备号。后因设备增多设备号不够用，因此将设备号扩展为 32 位，高 12 位表示主设备号，低 20 位表示从设备号。

保存在旧的文件系统（介质文件系统）中的设备文件，其节点中保存的设备号只使用了 16 个比特位，为与旧的布局相兼容，在设备号需要在外部表示时（如：外部文件系统中的设备文件）则采用外部表示方法。如下图所示，外部表示方式中 0~7 位及 20~31 位合并表示从设备号，而 8~19 位表示主设备号。



具体设备的设备号在/Documentation/devices.txt 文件内定义，内核在/include/uapi/linux/major.h 头文件内定义了表示主设备号的宏。

设备号在内核内部由 dev_t 数据类型表示（32 位无符号整数），内核在/include/linux/kdev_t.h 头文件中定义了以上设备号表示方式之间的转换函数，例如：

```
#define   MINORBITS   20
#define   MINORMASK     ((1U << MINORBITS) - 1)

#define   MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))      /*获取 dev_t 中主设备号*/
#define   MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))     /*获取 dev_t 中从设备号*/
#define   MKDEV(ma,mi)   (((ma) << MINORBITS) | (mi))          /*主、从设备号合成 dev_t 实例*/
```

设备文件中保存了设备号和设备类型，在打开设备文件时，内核通过设备号和设备类型查找设备驱动数据库。在以前的 Linux 系统中，设备文件静态保存在外部文件系统中（需要用户创建），因此需要保证驱动程序注册的设备号与设备文件中的设备号相同，否则打开设备文件时将无法找到正确的设备驱动数据结构。

后来 Linux 系统引入了 udev（mdev）机制，在向内核添加设备时（注意是添加设备，添加驱动时不行）将触发 uevent 事件，通过环境变量向用户空间传递设备信息，内含设备号和类型信息，然后由用户守护进程 udev 根据设备信息自动创建设备文件。

现在 Linux 内核引入了更智能化和简单化的方法，即内核在调用 device_add(dev)函数添加 device（设备）实例时，如果 device 实例设置了设备号，内核将自动唤醒内核守护线程在 devtmpfs 文件系统（/dev/）中自动创建设备文件（见本章下文）。驱动中设备号与设备文件中设备号由内核自动实现统一，驱动程序和用户进程只需关注设备文件名称，而不再需要将某个设备固定赋予某个设备号。

8.2.2 驱动数据库

在设备驱动数据库中，字符设备由 cdev 结构体表示，块设备由 gendisk 结构体表示，这两个结构体的定义在介绍字符设备和块设备驱动程序时再做介绍。内核对字符设备和块设备驱动数据结构实例的管理采用了相同的数据结构，称之为设备驱动数据库。

1 数据结构

内核定义了 kobj_map 结构体用于管理设备驱动数据结构实例，结构体定义在/drivers/base/map.c 文件内：

```
struct kobj_map {
    struct probe {
        struct probe *next;          /*指向下一个 probe 实例，构成单链表*/
        dev_t dev;                   /*设备号，包含起始从设备号*/
```

```
        unsigned long    range;      /*从设备号数量*/
        struct module *owner;        /*模块指针*/
        kobj_probe_t *get;           /*获取设备驱动数据结构中 kobject 结构体成员指针函数*/
        int    (*lock)    (dev_t, void *);
        void *data;                  /*指向 cdev 或 gendidk 结构体*/
    } *probes[255];                  /*probe 结构指针数组*/
    struct mutex *lock;              /*互斥量*/
};
```

kobj_map 结构体实际上是一个 probe 结构体指针数组，数组项数为 255。probe 结构体中各成员定义如下：

●**next**：指向下一个 probe 实例，构成单链表。

●**dev**：probe 实例所适配设备的设备号，包含主设备号和起始从设备号。

●**range**：从设备号数量，从设备号取值范围为[MINORS(dev)，MINORS(dev)+range-1]。probe 实例管理的 cdev 或 gendisk 实例可适用于同一主设备号的多个从设备，即多个从设备共用一个驱动程序。

●**owner**：模块指针。

●**get**：kobj_probe_t 类型函数指针，函数返回设备驱动数据结构中 kobject 结构体成员指针。通常 kobject 结构内嵌在被跟踪数据结构内部，获取 kobject 指针后，通过容器机制可获取被跟踪数据结构实例指针。

kobj_probe_t 函数类型定义在/include/linux/kobj_map.h 头文件内：

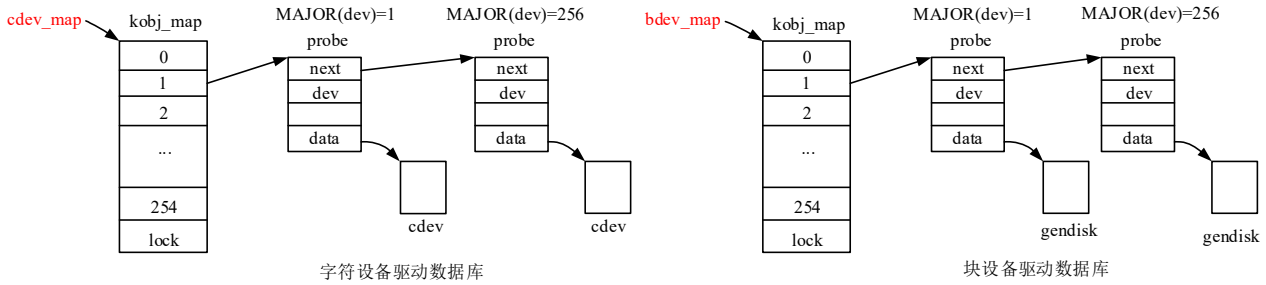
typedef struct kobject *kobj_probe_t(dev_t, int *, void *);

●**data**：指向设备驱动数据结构实例，指向 **cdev** 或 **gendisk** 结构体，data 将作为 kobj_probe_t(dev_t, int *, void *)函数的 void 指针参数。

每个 probe 结构实例管理的设备驱动数据结构实例（cdev 或 gendisk），可适用于同一主设备号下的多个从设备。一个主设备号也可对应多个 probe 实例表示，每个实例适用一段范围的从设备号。

probe 实例由所适用设备的主设备号确定添加到 probe 指针数组中哪一项，即 probe 实例关联的数组项为 probes[MAJOR(dev)%255]。同一指针数组项下的 probe 实例通过 next 指针成员组成单链表。

内核为字符设备和块设备分别创建了 kobj_map 实例，用于管理设备驱动数据结构实例，称之为设备数据库，如下图所示。



2 驱动数据库操作

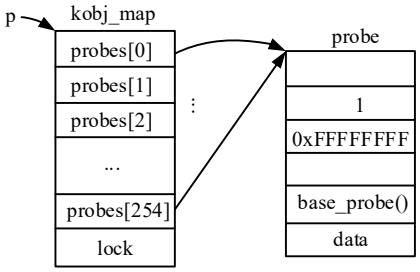
内核在初始化阶段为字符设备和块设备分别创建了 kobj_map 结构体实例，注册设备驱动时需要向设备驱动数据库添加 probe 实例，实例 data 成员指向 cdev 或 gendisk 实例。在打开设备文件时，打开操作函数根据设备类型和设备号搜索设备驱动数据库，找到设备对应的驱动数据结构实例。

■初始化

kobj_map_init()函数用于创建和初始化 kobj_map 结构体实例，函数代码如下（/drivers/base/map.c）：

```
struct kobj_map *kobj_map_init(kobj_probe_t *base_probe, struct mutex *lock)
/*base_probe：kobj_probe_t 函数指针*/
{
    struct kobj_map *p = kmalloc(sizeof(struct kobj_map), GFP_KERNEL);   /*创建 kobj_map 实例*/
    struct probe *base = kzalloc(sizeof(*base), GFP_KERNEL);             /*创建一个初始 probe 实例*/
    int i;

    if ((p == NULL) || (base == NULL)) {
        ...
    }

    base->dev = 1;                  /*主设备号为 0，从设备号为 1*/
    base->range = ~0;               /*从设备号范围是无符号整数最大值*/
    base->get = base_probe;         /*函数指针，由参数传递*/
    for (i = 0; i < 255; i++)
        p->probes[i] = base;        /*所有指针数组项指向初始 probe 实例*/
    p->lock = lock;
    return p;          /*返回 kobj_map 实例指针*/
}
```

kobj_map_init()函数比较简单，需要向函数传递 kobj_probe_t 函数指针和互斥量指针参数。函数内创建一个 kobj_map 和 probe 结构体实例，并对实例进行初始化，最后返回 kobj_map 实例指针。新创建 kobj_map 实例内 probes[]指针数组项都指向初始的 probe 实例，如下图所示：



■添加设备驱动

向设备驱动数据库添加新项时将调用 kobj_map()函数，函数定义如下（/drivers/base/map.c）：

```
int kobj_map(struct kobj_map *domain, dev_t dev, unsigned long range,struct module *module,   \
                                    kobj_probe_t *probe,int (*lock)(dev_t, void *), void *data)
/*
* domain：kobj_map 实例指针，dev：设备号（含起始从设备号），range：从设备号数量，
* probe：kobj_probe_t 函数指针，data：cdev 或 gendisk 实例指针，module：模块指针。
*/
{
    unsigned n = MAJOR(dev + range - 1) - MAJOR(dev) + 1;     /*需创建 probe 实例数量*/
```

```
    unsigned index = MAJOR(dev);                /*主设备号*/
    unsigned i;
    struct probe *p;

    if (n > 255)
        n = 255;

    p = kmalloc_array(n, sizeof(struct probe), GFP_KERNEL);      /*创建 probe 结构实例数组*/
    if (p == NULL)
        return -ENOMEM;

    for (i = 0; i < n; i++, p++) {          /*初始化每个 probe 实例，每个实例成员值相同*/
        p->owner = module;
        p->get = probe;        /*获取 kobject 函数指针*/
        p->lock = lock;
        p->dev = dev;           /*设备号*/
        p->range = range;       /*从设备号数量*/
        p->data = data;         /*指向 cdev 或 gendisk 实例*/
    }
    mutex_lock(domain->lock);
    for (i = 0, p -= n; i < n; i++, p++, index++) {       /*由主设备号将 probe 实例添加到 kobj_map 中*/
        struct probe **s = &domain->probes[index % 255];  /*指针数组项，每次循环 index 加 1*/
        while (*s && (*s)->range < range)            /*probe 实例在单链表中按 range 值从小到大排序*/
            s = &(*s)->next;
        p->next = *s;
        *s = p;
    }
    mutex_unlock(domain->lock);
    return 0;          /*成功返回 0*/
}
```

kobj_map()函数比较简单，主要是创建 probe 实例并初始化，由主设备号确定将实例添加到 kobj_map 中指针数组中的哪一项，实例以 range 值从小到大，在单链表中从左至右排序。需要注意的是，如果起始从设备号加上从设备号数量大于 20 位从设备号能表示的最大值，则需要创建多个 probe 实例，每个 probe 实例各成员值是一样的，但是添加到不同的 probe 指针数组项中。

设备驱动程序的主要工作就是构建设备驱动数据结构（cdev 或 gendisk）实例，在通用驱动模型中驱动的 probe()函数中将其添加到设备驱动数据库。

■查找设备驱动

在打开设备文件时，需要根据设备文件中保存的设备号，查找设备驱动数据库获取设备驱动数据结构实例，查找函数返回跟踪 cdev 或 gendisk 实例的 kobject 实例指针，调用者可通过容器等机制获取 cdev 或 gendisk 实例指针。

查找设备数据库的 kobj_lookup()函数定义如下（/drivers/base/map.c）：

```
struct kobject *kobj_lookup(struct kobj_map *domain, dev_t dev, int *index)
```

```
/*
*domain：kobj_map 实例指针，dev：查找设备号，含主从设备号，
*index：*index 保存查找从设备号基于 probe 实例起始从设备号的偏移量。
*/
{
    struct kobject *kobj;
    struct probe *p;
    unsigned long best = ~0UL;

retry:
    mutex_lock(domain->lock);
    for (p = domain->probes[MAJOR(dev) % 255]; p; p = p->next) {     /*遍历 probe 单链表*/
        struct kobject *(*probe)(dev_t, int *, void *);
        struct module *owner;
        void *data;

        if (p->dev > dev || p->dev + p->range - 1 < dev)    /*查找设备号不在 probe 表示的范围内*/
            continue;
        if (p->range - 1 >= best)
            break;
        if (!try_module_get(p->owner))
            continue;

        /*找到含盖 dev_t 设备号的 probe 实例*/
        owner = p->owner;
        data = p->data;             /*cdev 或 gendisk 实例指针*/
        probe = p->get;
        best = p->range - 1;
        *index = dev - p->dev;              /*从设备号相对于 p->dev 起始从设备号的偏移量*/
        if (p->lock && p->lock(dev, data) < 0) {
            module_put(owner);
            continue;
        }
        mutex_unlock(domain->lock);
        kobj = probe(dev, index, data);     /*调用 p->get()函数获取 kobject 实例指针*/
        module_put(owner);
        if (kobj)
            return kobj;          /*返回 kobject 实例指针*/
        goto retry;
    }
    mutex_unlock(domain->lock);
    return NULL;
}
```

查找函数根据设备号 dev 中的主设备号索引 kobj_map 指针数组，扫描 probes[MAJOR(dev) % 255]表项中的 probe 实例链表，找到含盖 dev 设备号的 probe 实例，获取从设备号偏移量后，调用 p->get()函数获取跟踪设备驱动数据结构实例的 kobject 实例，并返回 kobject 实例指针。

*index 保存参数 dev 中从设备号相对于 probe 实例中的起始从设备号的偏移量。例如，假设参数 dev 的从设备号为 2，而 probe 实例表示的从设备号范围为[0，12]，则*index 为 2，若从设备号范围是[1，12]，则*index 为 1。

void kobj_unmap(struct kobj_map *domain, dev_t dev, unsigned long range)函数负责从设备驱动数据库中删除设备对应的 probe 结构实例，源代码请读者自行阅读。

8.2.3 创建设备文件

设备文件是进程操作设备的接口，曾经设备文件需要静态保存在外部根文件系统/dev/目录下，后来使用了 udev 机制，在向内核注册设备时通过 uevent 机制向用户空间发送信息，由用户守护进程 udev 创建设备文件。现在内核采用了更简单的方法，内核在启动阶段将 devtmpfs 文件系统（ramfs 或 tmpfs 实例）挂载到/dev/目录下，并创建内核守护线程。在向内核添加设备时，由守护线程在/dev/目录下自动创建设备文件。

使用 devtmpfs 需选择 DEVTMPFS 配置选项，devtmpfs 文件系统是 ramfs 或 tmpfs 文件系统实例，在添加设备 device 实例时创建设备文件，系统关机时消失。因为自动创建的设备文件其设备号由内核在设备 device 实例中提取，可保证两个设备号是相同的，因此驱动程序并不需要特别在意设备使用哪个设备号，用户进程只是通过设备文件名称识别设备。

1 devtmpfs 文件系统

devtmpfs 文件系统通常挂载到/dev/目录下，用于存放设备文件，devtmpfs 是 ramfs 或 tmpfs 实例，内核保证只有一个 devtmpfs 文件系统挂载到根文件系统，以保证设备文件的唯一性。

devtmpfs 文件系统类型定义在/drivers/base/devtmpfs.c 文件内：

```
static struct file_system_type dev_fs_type = {
        .name = "devtmpfs",                 /*文件类型名称*/
        .mount = dev_mount,                 /*挂载函数*/
        .kill_sb = kill_litter_super,
};
```

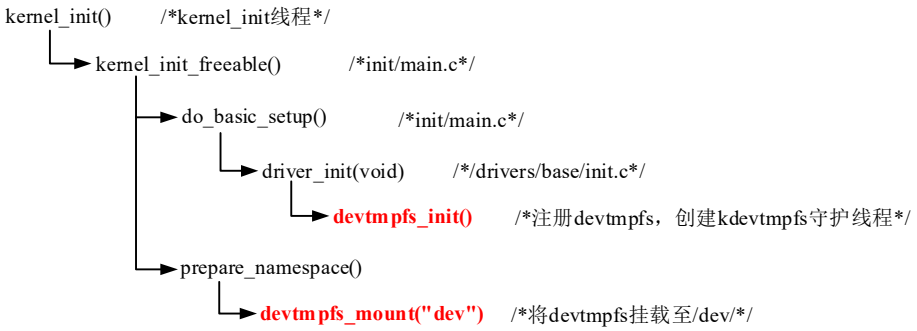
devtmpfs 文件系统类型的挂载函数定义如下（/drivers/base/devtmpfs.c）：

```
static struct dentry *dev_mount(struct file_system_type *fs_type, int flags,const char *dev_name, \
                                                                        void *data)
{
  #ifdef CONFIG_TMPFS
     return mount_single(fs_type, flags, data, shmem_fill_super);   /*套用 tmpfs 文件系统类型*/
  #else
     return mount_single(fs_type, flags, data, ramfs_fill_super);    /*套用 ramfs 文件系统类型*/
  #endif
}
```

mount_single()函数保证挂载的文件系统在内核中只有一个实例（超级块实例），即使进行多次挂载，

各挂载点挂载的也是同一个文件系统实例。

devtmpfs 文件系统初始化流程如下图所示：



devtmpfs_init()函数向内核注册 devtmpfs 文件系统类型，并创建 kdevtmpfs 内核守护线程，用于自动创建设备文件。

在创建 kernel_init 线程之前内核已经挂载了 rootfs 作为初始的根文件系统。kdevtmpfs 守护线程首次运行时，将为线程创建新的挂载命名空间，并将 devtmpfs 文件系统挂载到根文件系统根目录项，并设 devtmpfs 根目录项为守护线程根目录和当前工作目录。kdevtmpfs 线程创建设备文件时，在其当前工作目录下创建。

由于 kdevtmpfs 线程是在新挂载命名空间中挂载 devtmpfs，因此挂载只对守护线程可见，对其它线程不可见。

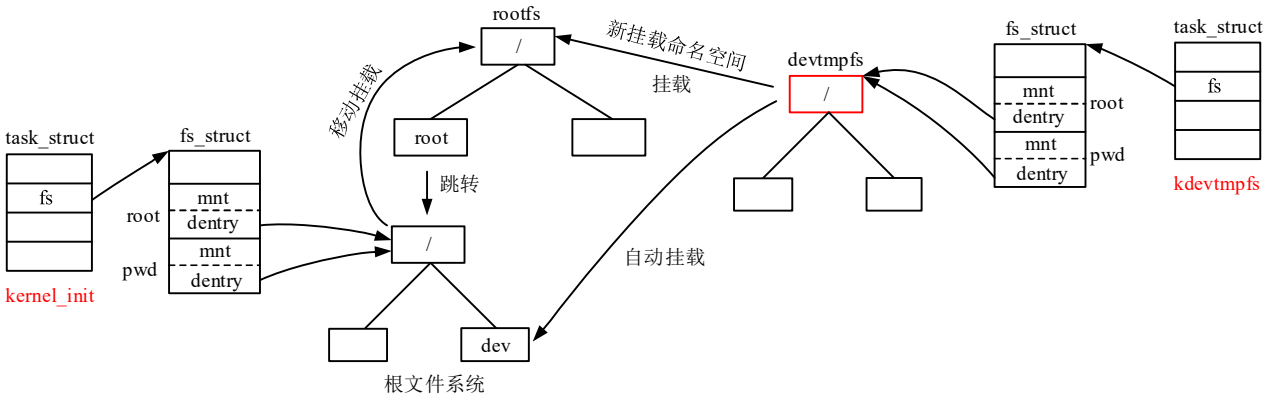
如果内核挂载了外部块设备作为根文件系统，prepare_namespace()函数中调用 **devtmpfs_mount("dev")** 函数，可能自动挂载 devtmpfs 至/dev/目录。自动挂载的条件是（满足其一即可）：

●选择了 DEVTMPFS_MOUNT 配置选项。

●传递了“devtmpfs.mount=1”命令行参数。

如果内核没有挂载外部块设备作为根文件系统（仍使用 rootfs）或传递了“devtmpfs.mount=0”命令行参数（不管有没有选择 DEVTMPFS_MOUNT 配置选项），devtmpfs_mount()函数中将不会挂载 devtmpfs，需要用户在启动后手动挂载。

devtmpfs 自动挂载结果如下图所示：



kdevtmpfs 线程创建设备文件时，在其当前工作目录下创建，即在 devtmpfs 根目录下创建。由于 devtmpfs 同时挂载到/dev/目录下（同一个文件系统实例），因此其对内核根文件系统也可见。

初始化函数 devtmpfs_init()和 devtmpfs_mount()定义如下（/drivers/base/devtmpfs.c）：

```
int __init devtmpfs_init(void)
{
```

```
    int err = register_filesystem(&dev_fs_type);        /*注册 devtmpfs 文件系统类型*/
    ...

    thread = kthread_run(devtmpfsd, &err, "kdevtmpfs");
                                /*创建内核守护线程 kdevtmpfs，线程执行函数为 devtmpfsd()*/
    if (!IS_ERR(thread)) {
        wait_for_completion(&setup_done);
                                /*kernel_init 线程进入睡眠，等待 kdevtmpfs 线程将其唤醒*/
    } else {
        ...
    }
    ...
    return 0;
}
```

devtmpfs_init()函数由 kernel_init 内核线程调用，函数内注册 devtmpfs 文件系统类型，并创建 kdevtmpfs 内核守护线程，然后 kernel_init 线程进入睡眠等待，等待 kdevtmpfs 守护线程首次运行时将其唤醒。

devtmpfs_mount()函数由 prepare_namespace()函数调用，负责将 devtmpfs 挂载到/dev/目录，函数定义如下：

```
int devtmpfs_mount(const char *mntdir)
/*mntdir：挂载目录为"dev"，基于当前工作目录（/root/）*/
{
    int err;

    if (!mount_dev)          /*是否自动挂载*/
        return 0;

    if (!thread)         /*kdevtmpfs 守护线程指针*/
        return 0;

    err = sys_mount("devtmpfs", (char *)mntdir, "devtmpfs", MS_SILENT, NULL);      /*挂载*/
    if (err)          /*输出信息*/
        printk(KERN_INFO "devtmpfs: error mounting %i\n", err);
    else
        printk(KERN_INFO "devtmpfs: mounted\n");
    return err;
}
```

2 守护线程

kdevtmpfs 内核守护线程执行函数 devtmpfsd()定义如下（/drivers/base/devtmpfs.c）：

```
static int devtmpfsd(void *p)
{
    char options[] = "mode=0755";
```

```
    int *err = p;
    *err = sys_unshare(CLONE_NEWNS);          /*为守护线程创建新挂载命名空间*/
    if (*err)
        goto out;
    *err = sys_mount("devtmpfs", "/", "devtmpfs", MS_SILENT, options);
                                              /*首次运行将 devtmpfs 挂载到 rootfs 文件系统根目录项*/
                                              /*在新挂载命名空间中的挂载，对 kernel_init 线程不可见*/
    if (*err)
        goto out;
    sys_chdir("/..");     /*设置 kdevtmpfs 线程当前工作目录和根目录为 devtmpfs 根目录*/
    sys_chroot(".");
    complete(&setup_done);       /*首次运行至此，唤醒 kernel_init 线程*/

    /*线程进入无限循环，用于创建设备文件（详见下文）*/
    while (1) {
        spin_lock(&req_lock);
        while (requests) {   /*扫描 requests 指向的请求 req 实例链表，逐个为 req 实例创建设备文件*/
            struct req *req = requests;
            requests = NULL;
            spin_unlock(&req_lock);
            while (req) {
                struct req *next = req->next;
                req->err = handle(req->name, req->mode,req->uid, req->gid, req->dev);
                                          /*调用函数 handle()创建设备文件，/drivers/base/devtmpfs.c*/
                complete(&req->done);       /*唤醒在请求上等待的进程*/
                req = next;
            }
            spin_lock(&req_lock);
        }
        /*requests 链表为空，守护线程睡眠等待*/
        __set_current_state(TASK_INTERRUPTIBLE);       /*线程进入睡眠*/
        spin_unlock(&req_lock);
        schedule();          /*进程调度，唤醒后继续循环*/
    }
    return 0;
out:
    complete(&setup_done);
    return *err;
}
```

kdevtmpfs 线程首次运行将创建新挂载命名空间，将 devtmpfs 文件系统挂载至 rootfs 文件系统根目录。kdevtmpfs 线程的根目录和当前工作目录都指向 devtmpfs 文件系统根目录。

随后，kdevtmpfs 线程进入无限循环，在循环中扫描 requests 指向的请求 req 实例链表，逐个为 req 实例创建设备文件，并唤醒在请求上睡眠的进程，requests 链表为空时，线程睡眠。

3 创建设备文件

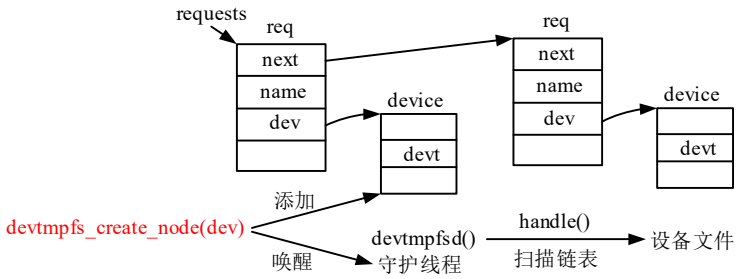
内核需要创建设备文件时，需要构建请求 req 结构体实例，添加到 requests 链表，唤醒 kdevtmpfs 守护线程。kdevtmpfs 线程扫描 requests 链表，根据请求 req 实例信息在 devtmpfs 根目录下创建设备文件。

请求 req 结构体定义在/drivers/base/devtmpfs.c 文件内：

```
static struct req {
    struct req *next;              /*下一个请求，请求组成单链表*/
    struct completion done;        /*完成量，等待设备文件创建完成，由 kdevtmpfs 内核线程唤醒*/
    int err;                       /*创建设备文件返回值*/
    const char *name;              /*设备文件名称，如：hd1*/
    umode_t mode;                  /*文件模式，  0  表示删除设备文件*/
    kuid_t uid;
    kgid_t gid;
    struct device *dev;            /*设备 device 实例指针，device 结构体见下文*/
} *requests;
```

requests 是一个全局变量，指向 req 实例单链表，向 kdevtmpfs 内核线程传递请求。

在向内核注册表示设备的 device 实例时，将调用 **devtmpfs_create_node**(struct device *dev)函数触发设备文件的创建（device 实例中赋予了主设备号），此函数内将构建请求 req 实例，将其添加到 requests 指向的单链表，唤醒 kdevtmpfs 内核守护线程，由守护线程扫描单链表逐个创建设备文件，如下图所示。



devtmpfs_create_node()函数定义如下（/drivers/base/devtmpfs.c）：

```
int devtmpfs_create_node(struct device *dev)
{
    const char *tmp = NULL;
    struct req req;        /*请求 req 实例*/

    if (!thread)
        return 0;

    req.mode = 0;
    req.uid = GLOBAL_ROOT_UID;        /*uid，gid 设为 0*/
    req.gid = GLOBAL_ROOT_GID;
    req.name = device_get_devnode(dev, &req.mode, &req.uid, &req.gid, &tmp);
                                                    /*获取设备文件名称，/drivers/base/core.c*/
    if (!req.name)
        return -ENOMEM;
```

```
    if (req.mode == 0)                  /*如果 mode 为 0*/
        req.mode = 0600;                /*设备文件访问权限，默认只有文件主有读写权限*/
    if (is_blockdev(dev))               /*dev->class == &block_class??*/
        req.mode |= S_IFBLK;            /*块设备文件，设置文件类型*/
    else
        req.mode |= S_IFCHR;            /*字符设备文件，设置文件类型*/

    req.dev = dev;      /*device 实例*/
    init_completion(&req.done);

    spin_lock(&req_lock);
    req.next = requests;
    requests = &req;                    /*req 添加到 requests 单链表头部*/
    spin_unlock(&req_lock);

    wake_up_process(thread);            /*唤醒 kdevtmpfs 内核线程*/
    wait_for_completion(&req.done);     /*等待设备文件创建完成时，唤醒本进程*/
    kfree(tmp);
    return req.err;
}
```

devtmpfs_create_node()函数构建 req 结构体实例，将其添加到 requests 单链表的头部，然后唤醒守护线程 kdevtmpfs，当前进程在 req.done 完成量上睡眠等待，当 kdevtmpfs 守护线程创建完设备文件后，将唤醒在 req.done 上睡眠等待的进程。

kdevtmpfs 线程被唤醒后，如果 requests 链表不为空，则为链表中每个请求 req 实例创建设备文件，直至链表为空线程进入睡眠。

kdevtmpfs 线程中创建设备文件的 handler()函数定义如下：

```
static int handle(const char *name, umode_t mode, kuid_t uid, kgid_t gid,struct device *dev)
/*name：设备文件名称，mode：文件模式，dev：device 实例指针，内含设备号*/
{
    if (mode)           /*非 0，创建设备文件*/
        return handle_create(name, mode, uid, gid, dev);    /*创建设备文件，/drivers/base/devtmpfs.c*/
    else                /*为 0，删除设备文件*/
        return handle_remove(name, dev);                    /*删除设备文件*/
}
```

handle_create()函数根据设备名称、设备号、访问权限等信息，调用 devtmpfs 文件系统 inode_operations 结构实例中的 mknod()函数（ramfs_mknod()）创建设备文件对应的 dentry 和 inode 实例。对于 ramfs/tmpfs 文件系统，创建设备节点的操作就是直接创建 dentry 和 inode 实例，文件系统没有后备存储设备。

在创建设备文件时，文件名至关重要，因为用户进程只能通过设备文件访问设备，设备号由 device 实例传递。设备文件名由 device_get_devnode()函数获取，函数定义如下（/drivers/base/core.c），读者可学习完下面的驱动模型相关知识后再回过头来阅读此函数。

```
const char *device_get_devnode(struct device *dev,umode_t *mode, kuid_t *uid, kgid_t *gid,   \
                                                                    const char **tmp)
{
```

```
    char *s;

    *tmp = NULL;

    /*优先调用设备类型 device_type 实例提供的设备文件名*/
    if (dev->type && dev->type->devnode)
        *tmp = dev->type->devnode(dev, mode, uid, gid);
    if (*tmp)
        return *tmp;        /*tmp 不为 NULL，返回*/

    /*tmp 为 NULL，继续往下执行，设备类 class 可能提供设备文件名*/
    if (dev->class && dev->class->devnode)
        *tmp = dev->class->devnode(dev, mode);
    if (*tmp)
        return *tmp;        /*tmp 不为 NULL，返回*/

    /*如果设备类型和设备类没有提供设备名称，则通过 dev_name()获取设备名称*/
    if (strchr(dev_name(dev), '!') == NULL)     /*名称字符串中如果不包含‘!’字符*/
        return dev_name(dev);           /*返回设备名称，dev->init_name 或 dev->kobject 名称*/

    /*如果名称字符串中包含 '!' 则用 '/' 代替*/
    s = kstrdup(dev_name(dev), GFP_KERNEL);
    if (!s)
        return NULL;
    strreplace(s, '!', '/');
    return *tmp = s;    /*dev->init_name 或 dev->kobject 名称*/
}
```

删除设备文件的接口函数为 **devtmpfs_delete_node**(struct device *dev)，删除设备文件也是由 kdevtmpfs 守护线程完成（req.mode=0），源代码请读者自行阅读。

8.2.4 设备文件操作

设备文件是特殊文件，VFS 在打开文件时，当发现打开的是特殊文件时，将调用 init_special_inode(inode, mode, dev)函数对文件 inode 实例进行初始化。

init_special_inode()函数定义如下（/fs/inode.c）：

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
/*inode：inode 实例指针，mode：文件访问模式，rdev：设备号*/
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {                /*是否是字符设备*/
        inode->i_fop = &def_chr_fops;       /*字符设备文件操作结构实例，/fs/char_dev.c*/
        inode->i_rdev = rdev;               /*设备号*/
    } else if (S_ISBLK(mode)) {             /*块设备文件*/
```

```
        inode->i_fop = &def_blk_fops;          /*块设备文件操作结构实例，/fs/block_dev.c*/
        inode->i_rdev = rdev;                  /*设备号*/
    } else if (S_ISFIFO(mode))                 /*命名管道*/
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))          /*套接字*/
        ;
    else
        ...     /*输出信息*/
}
```

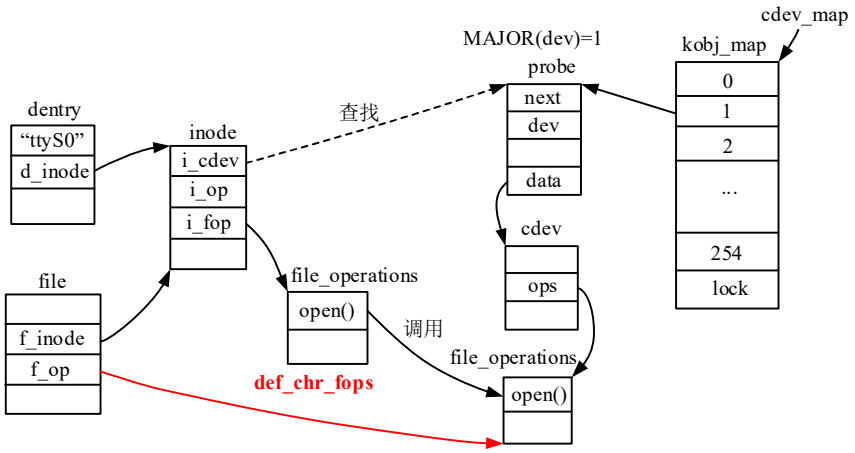
字符设备文件默认的文件操作结构实例定义如下（/fs/char_dev.c）：

```
const struct file_operations def_chr_fops = {
    .open = chrdev_open,                /*打开函数*/
    .llseek = noop_llseek,
};
```

块设备文件默认的文件操作结构实例定义如下（/fs/block_dev.c）：

```
const struct file_operations def_blk_fops = {
    .open       = blkdev_open,
    .release    = blkdev_close,
    .llseek     = block_llseek,
    .read_iter  = blkdev_read_iter,
    .write_iter     = blkdev_write_iter,
    .mmap           = generic_file_mmap,
    .fsync          = blkdev_fsync,
    .unlocked_ioctl     = block_ioctl,
#ifdef CONFIG_COMPAT
    .compat_ioctl   = compat_blkdev_ioctl,
#endif
    .splice_read    = generic_file_splice_read,
    .splice_write   = iter_file_splice_write,
};
```

以字符设备文件为例，在打开文件的系统调用最后将调用 **inode->i_fop->open()**函数完成文件的底层打开操作。字符设备 def_chr_fops 实例中定义的打开函数为 **chrdev_open()**，函数内根据设备号搜索字符设备驱动数据库，找到对应的 cdev 实例，调用 ops 成员指向的 file_operations 实例中的 open()函数，并将 ops 指向实例赋予文件 file 实例 f_op 成员，如下图所示，此后对字符设备文件的操作将由驱动程序中定义的 file_operations 实例中的函数完成。



块设备文件的打开操作与此类似，在通用的打开函数 blkdev_open()中根据设备号查找块设备驱动数据结构 gendisk 实例，再调用底层块设备定义的打开操作函数激活设备，完成设备的打开操作。

块设备文件 file 实例 f_op 成员指向默认的 def_blk_fops 实例，也就是说所有的块设备文件采用相同的文件操作结构实例，而不像字符设备一样由驱动程序定义文件操作结构实例。通过块设备文件访问的是裸块设备，而内核通常是通过文件系统间接地访问块设备。

后面两章讲解字符设备和块设备驱动程序时再详细介绍设备文件的操作。

8.3 总线与设备类

通用驱动模型中总线由 bus_type 结构体表示，也可以表示虚拟的总线。总线实例由内核（驱动程序）静态定义，总线 bus_type 管理挂载在总线上的设备 device 和驱动 device_driver 实例。

当向总线注册设备或驱动时，总线负责查找匹配的驱动或设备，匹配成功则调用驱动中的 probe()函数，创建并注册设备驱动数据结构实例（加载设备驱动程序）。

设备类用于管理同一类型的设备，同一类型设备可能挂接在不同的总线上。设备类定义了同类型设备的公共属性等。

本节介绍总线和设备类的定义（创建）和注册。

8.3.1 总线

1 数据结构

总线 bus_type 结构体定义在/include/linux/device.h 头文件：

```
struct bus_type {
    const char          *name;                  /*总线名称，导出到 sysfs 文件系统*/
    const char          *dev_name;              /*默认设备名称*/
    struct device       *dev_root;              /*指向表示总线自身的 device 实例*/
    struct device_attribute    *dev_attrs;      /*总线设备的默认属性，device_attribute 数组*/
    const struct attribute_group **bus_groups;  /*默认的总线属性，注册总线时添加*/
    const struct attribute_group **dev_groups;  /*总线设备的默认属性（attribute 指针数组）*/
    const struct attribute_group **drv_groups;  /*总线驱动的默认属性（attribute 指针数组）*/

    int (*match) (struct device *dev, struct device_driver *drv);
                                    /*检查指定 device 和 device_driver 实例是否匹配，匹配成功返回非零值*/
```

```
    int (*uevent) (struct device *dev, struct kobj_uevent_env *env);
                                        /*添加、移除设备等时机调用，用于向 uevent 事件添加环境变量*/
    int (*probe) (struct device *dev);
                                        /*设备驱动匹配成功时调用，用于注册设备驱动数据结构实例*/
    int (*remove) (struct device *dev);        /*设备从总线移除时调用*/
    void (*shutdown) (struct device *dev);   /*关机时调用，用于退出设备*/
    int (*online)(struct device *dev);          /*重新连接设备*/
    int (*offline)(struct device *dev);         /*热插拔移除设备*/
    int (*suspend)(struct device *dev, pm_message_t state);     /*设备进入睡眠状态时调用*/
    int (*resume)(struct device *dev);            /*唤醒设备时调用*/

    const struct dev_pm_ops *pm;              /*总线功耗管理操作，调用驱动定义的 dev_pm_ops 操作*/
    const struct iommu_ops *iommu_ops;     /*总线 IOMMU 特定操作，用于具有 MMU 的外设*/

    struct subsys_private *p;              /*私有数据结构，用于管理设备和驱动等*/
    struct lock_class_key lock_key;       /*锁*/
};
```

bus_type 结构体中主要成员简介如下：

●**dev_root**：指向表示总线自身的 device 实例，在调用 subsys_system_register()函数注册子系统时，在 sysfs 文件系统/sys/devices/system/目录下创建表示总线的 device 实例。

●**match()**：在向总线添加设备时，将扫描总线下的驱动，对每个驱动调用 match()函数判断设备与扫描的驱动是否匹配，匹配成功 match()返回非零值，否则返回 0。同理，当向总线添加驱动时，扫描总线上的设备，对每个设备调用 match()函数，匹配成功返回非零值，否则返回 0。

●**probe()**：向总线添设备或驱动时，设备和驱动匹配成功则调用此函数，完成驱动程序的加载。如果总线没有定义 probe()函数，设备和驱动匹配成功后将调用 device_driver 实例中的 probe()函数。

驱动程序加载主要是向设备驱动数据库注册 cdev 或 gendisk 实例。硬件设备的激活（初始化）在打开设备文件时，调用设备文件操作结构 file_operations 实例中的 open()函数完成。

●**pm**：设备功耗管理结构 dev_pm_ops 指针，dev_pm_ops 结构体定义在/include/linux/pm.h 头文件，总线 dev_pm_ops 实例内函数将回调驱动 device_driver 实例中关联的 dev_pm_ops 实例中的函数。

●**bus_groups**：指向 attribute_group 结构体指针数组，表示总线的默认属性组，属性文件添加到总线目录项下。attribute_group 结构体定义如下（/include/linux/sysfs.h）：

```
struct attribute_group {
    const char          *name;      /*属性组名称*/
    umode_t             (*is_visible)(struct kobject *,struct attribute *, int);      /*判断属性是否可见*/
    struct attribute **attrs;       /*指向属性指针数组*/
    struct bin_attribute  **bin_attrs;      /*指向二进制属性指针数组*/
};
```

attribute_group 结构体中包含的是一组属性，若 name 不为 NULL，则向 sysfs 添加属性时创建名称为 name 的目录项，属性文件添加到此目录项下。

int **sysfs_create_groups**(struct kobject *kobj,const struct attribute_group **groups)：函数用于向 kobj 实例（目录项）下添加属性组，函数定义在/fs/sysfs/group.c 文件内，源代码请读者自行阅读。

●**p**：subsys_private 结构体指针，subsys_private 用于表示总线 bus_type 和设备类 class 的私有数据，结

构体定义在/drivers/base/base.h 头文件：

```
struct subsys_private {
    struct kset    subsys;                  /*内嵌 kset 实例，在 sysfs 内表示总线/设备类*/
    struct kset    *devices_kset;           /*指向设备集合 kset 实例，管理总线下设备 device 实例*/
    struct list_head    interfaces;         /*链接 subsys_interface 或 class_interface 实例*/
    struct mutex    mutex;                  /*互斥量，保护设备及 interfaces 链表*/
    struct kset    *drivers_kset;           /*指向驱动集合 kset 实例，跟踪总线上驱动 device_driver 实例*/
    struct klist    klist_devices;          /*总线上设备 device 实例链表*/
    struct klist    klist_drivers;          /*总线上驱动 device_driver 实例链表*/
    struct blocking_notifier_head    bus_notifier;      /*通知链*/
    unsigned int drivers_autoprobe:1;     /*添加设备时是否自动探测驱动（匹配驱动），默认是*/
    struct bus_type    *bus;                /*指向总线 bus_type 实例*/
    struct kset    glue_dirs;               /**/
    struct class    *class;                 /*指设备类 class 实例*/
};
```

subsys_private 结构体中 subsys 成员用于将总线/设备类实例导出到 sysfs 文件系统，并作为其下设备和驱动的父目录项。devices_kset 和 drivers_kset 指向 kset 实例分别用于跟踪总线下的 device 和 device_driver 实例，并在 sysfs 文件系统中做为设备和驱动实例的父目录。链表成员 klist_devices 和 klist_drivers 分别管理 device 和 device_driver 实例，在匹配操作中便于扫描 device 和 device_driver 实例。

2 注册总线

总线 bus_type 实例必须由内核（总线驱动）代码静态定义，内核没有提供动态创建 bus_type 实例的方法。总线驱动程序中创建 bus_type 实例后需要向内核注册。在讲解总线注册前，先看一下总线初始化函数。

总线初始化函数 buses_init()在/drivers/base/bus.c 文件内实现，代码如下：

```
int __init buses_init(void)
{
    bus_kset = kset_create_and_add("bus", &bus_uevent_ops, NULL);    /*创建目录/sys/bus*/
    ...
    system_kset = kset_create_and_add("system", NULL, &devices_kset->kobj);
                                                    /*创建目录/sys/devices/system*/
    ...
    return 0;
}
```

buses_init()函数创建名称为"bus"的 kset 实例，导出到在/sys/目录下，即/sys/bus/为所有总线实例的父目录，用于跟踪注册的总线 bus_type 实例。创建名称为"system"的 kset 实例，导出到在/sys/devices 目录下，即创建/sys/devices/system 目录。

"bus"名称的 kset 实例，其指向的 kset_uevent_ops 结构体实例 bus_uevent_ops 定义在/drivers/base/bus.c 文件内：

```
static const struct kset_uevent_ops bus_uevent_ops = {
    .filter = bus_uevent_filter,
};
```

bus_uevent_filter()函数定义如下：

```
static int bus_uevent_filter(struct kset *kset, struct kobject *kobj)
```

```
{
    struct kobj_type *ktype = get_ktype(kobj);
    if (ktype == &bus_ktype)      /*若为总线实例总返回 1，表示发送 uevent 事件，否则返回 0*/
        return 1;
    return 0;
}
```

在向内核注册总线实例时，进行以下赋值：

bus_type.p.**subsys.kobj.kset = bus_kset**

bus_type.p.**subsys.kobj.ktype = &bus_ktype**

因此对于 bus_type.p.**subsys** 实例 bus_uevent_filter()函数总是返回 1，表示内核不过滤总线的 uevent 事件，总是允许发送 uevent 事件。

注册总线 **bus_register()**函数定义在/drivers/base/bus.c 文件内，注册成功返回 0，否则返回错误码：

```
int bus_register(struct bus_type *bus)
{
    int retval;
    struct subsys_private *priv;
    struct lock_class_key *key = &bus->lock_key;

    priv = kzalloc(sizeof(struct subsys_private), GFP_KERNEL);     /*创建 subsys_private 实例*/
    if (!priv)
        return -ENOMEM;

    priv->bus = bus;     /*指向总线实例*/
    bus->p = priv;       /*指向 subsys_private 实例*/

    BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);   /*初始化通知链*/

    retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);   /*设置总线 kobject 名称*/
    if (retval)
        goto out;

    priv->subsys.kobj.kset = bus_kset; /*父节点 bus_set 在初始化时创建，总线位于/sys/bus/目录下*/
    priv->subsys.kobj.ktype = &bus_ktype;   /*设置总线 kobject 类型为 bus_type*/
    priv->drivers_autoprobe = 1;            /*允许自动探测驱动*/

    retval = kset_register(&priv->subsys);      /*注册总线 kset 实例，导出路径为/sys/bus/bus->name*/
    if (retval)
        goto out;

    retval = bus_create_file(bus, &bus_attr_uevent);     /*添加 uevent 总线属性，总线属性后面介绍*/
    ...     /*错误处理*/

    priv->devices_kset = kset_create_and_add("devices", NULL, &priv->subsys.kobj);
```

```
                                        /*创建总线设备“devices”子目录，/sys/bus/bus->name/devices/*/
    ...    /*错误处理*/

    priv->drivers_kset = kset_create_and_add("drivers", NULL, &priv->subsys.kobj);
                                        /*创建总线驱动“drivers”子目录，/sys/bus/bus->name/drivers/*/
    ...    /*错误处理*/

    INIT_LIST_HEAD(&priv->interfaces);          /*初始化子系统接口链表头*/
    __mutex_init(&priv->mutex, "subsys mutex", key);
    klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);    /*初始化 device 链表*/
                                        /*函数参数用于增加/减小 device 实例引用计数值*/
    klist_init(&priv->klist_drivers, NULL, NULL);      /*初始化 device_driver 链表*/

    retval = add_probe_files(bus);      /*添加总线 drivers_probe 和 drivers_autoprobe 属性*/
    if (retval)
        goto bus_probe_files_fail;

    retval = bus_add_groups(bus, bus->bus_groups);   /*添加总线默认属性，bus->bus_groups*/
    ...
    return 0;
    ...
}
```

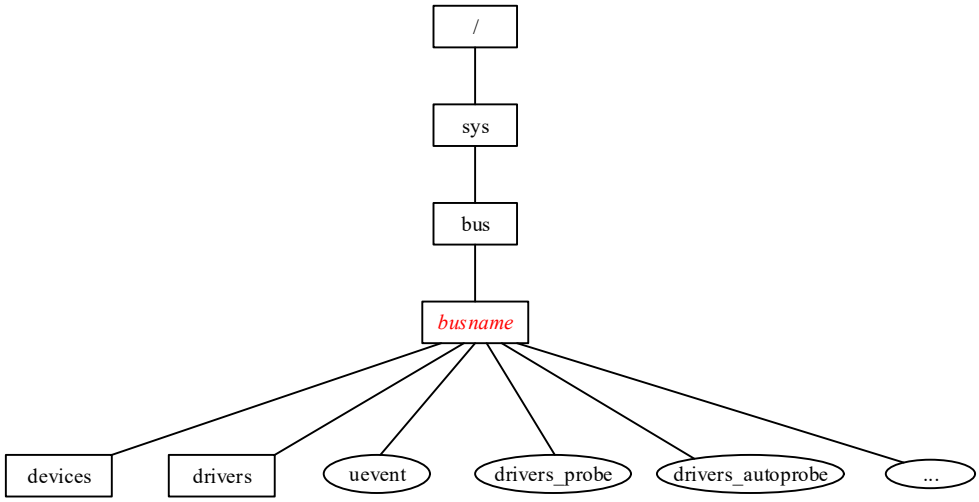
注册总线 bus_register()函数主要完成的工作如下：

（1）创建 subsys_private 结构体实例，初始化，将 subsys_private.subsys.kobj 导出到/sys/bus/目录下。

（2）在 subsys_private.subsys 下分别创建跟踪设备和驱动实例的 kset，导出/sys/bus/*bus*->*name*/devices 和/sys/bus/*bus*->*name*/drivers。

（3）向总线 **subsys_private.subsys.kobj** 添加 uevent，drivers_probe，drivers_autoprobe 属性以及总线实例定义的默认属性组 bus->bus_groups 中定义的属性。

下图示意了注册总线在 sysfs 中导出的目录项和文件：



内核在/drivers/base/bus.c 文件内定义了注册子系统的函数 int **subsys_system_register**(struct bus_type

*subsys,const struct attribute_group **groups)，参数 subsys 表示总线实例指针，groups 指向总线属性指针数组。此函数内完成 subsys 总线的注册，创建表示总线的 device 实例（将总线视为设备），导出到 sysfs 文件系统/sys/devices/system/目录下，device 实例指针赋予 subsys->dev_root 成员，函数源代码请读者自行阅读。

void bus_unregister(struct bus_type *bus)函数用于注销总线。

3 总线属性

在注册总线时，内核会在跟踪总线的 subsys_private.subsys.kobj 实例下添加 uevent、drivers_autoprobe、和 drivers_probe 属性以及总线默认属性组 bus->bus_groups 中定义的属性。属性在 sysfs（kernfs）文件系统中表现为文件，用户可对属性文件进行读写，从而获取总线信息或对总线进行控制。

总线属性定义如下（/include/linux/device.h）：

```
struct bus_attribute {
    struct attribute    attr;                   /*通用属性结构体成员*/
    ssize_t (*show) (struct bus_type *bus, char *buf);                  /*总线属性读函数*/
    ssize_t (*store) (struct bus_type *bus, const char *buf, size_t count);    /*总线属性写函数*/
};
```

内核在/include/linux/device.h 头文件内实现了定义总线属性的宏，例如：

```
#define BUS_ATTR(_name, _mode, _show, _store)    \      /*定义读写属性，_show、_store 为读写函数*/
        struct bus_attribute    bus_attr_##_name = __ATTR(_name, _mode, _show, _store)
                                        /*__ATTR()宏定义在/include/linux/sysfs.h 头文件*/

#define BUS_ATTR_RW(_name) \        /*定义读写属性，只需指定属性名称，只有文件属主在写权限*/
        struct bus_attribute    bus_attr_##_name = __ATTR_RW(_name)
                                        /*读写函数名为_name##_show(), _name##_store()*/

#define BUS_ATTR_RO(_name) \                /*定义只读属性，读函数名为_name##_store()*/
    struct bus_attribute    bus_attr_##_name = __ATTR_RO(_name)
```

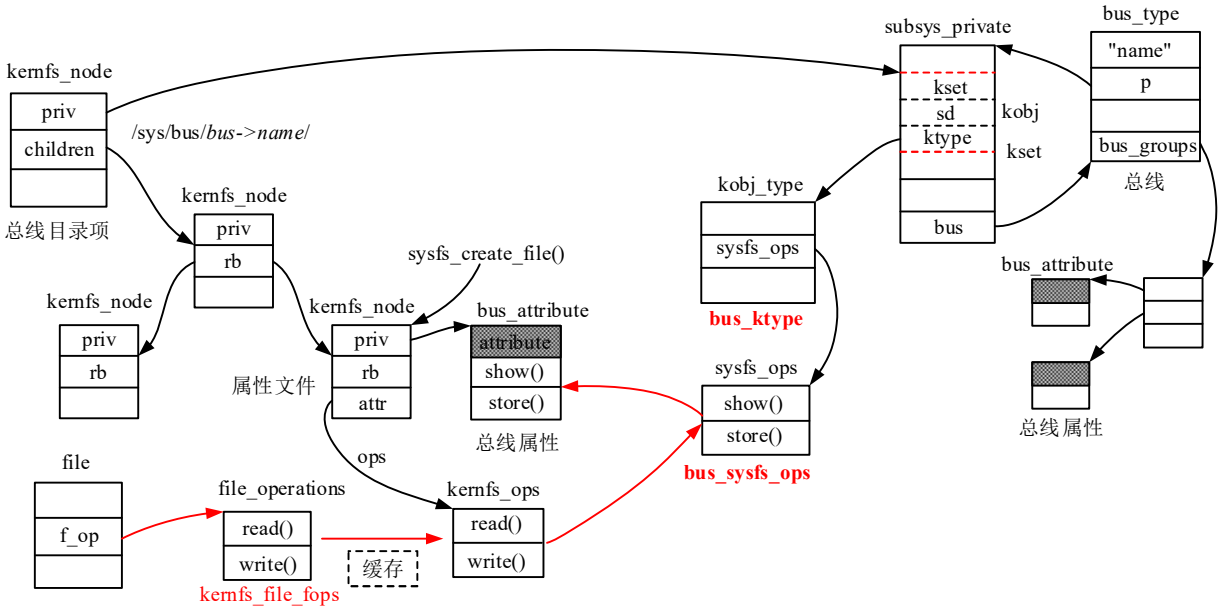
BUS_ATTR()等宏只是定义 bus_attribute 实例，其读写函数需要另外定义（后面设备类、设备、驱动的属性也是这样）。

在注册总线函数中跟踪总线的 kobect 类型定义为 subsys.kobj.ktype = **&bus_ktype**，bus_ktype 实例定义在/drivers/base/bus.c 文件内：

```
static struct kobj_type bus_ktype = {
        .sysfs_ops = &bus_sysfs_ops,        /*读写属性操作结构*/
        .release    = bus_release,
};

static const struct sysfs_ops bus_sysfs_ops = {
    .show     = bus_attr_show,          /*读属性函数*/
    .store    = bus_attr_store,         /*写属性函数*/
```

};

用户读写总线属性文件时，将先调用 bus_sysfs_ops 实例中的 show()和 store()函数，再调用具体总线属性 bus_attribute 实例中定义的 show()和 store()函数，如下图所示。



bus_sysfs_ops 实例中的 show()和 store()函数定义如下：

```
static ssize_t bus_attr_show(struct kobject *kobj, struct attribute *attr,char *buf)
{
    struct bus_attribute *bus_attr = to_bus_attr(attr);          /*attribute 指针转 bus_attribute 指针*/
    struct subsys_private *subsys_priv = to_subsys_private(kobj);
                                                    /*容器机制，获取 subsys_private 实例指针*/
    ssize_t ret = 0;
    if (bus_attr->show)
        ret = bus_attr->show(subsys_priv->bus, buf);   /*调用具体总线属性读函数*/
    return ret;
}

static ssize_t bus_attr_store(struct kobject *kobj, struct attribute *attr,const char *buf, size_t count)
{
    struct bus_attribute *bus_attr = to_bus_attr(attr);
    struct subsys_private *subsys_priv = to_subsys_private(kobj);
    ssize_t ret = 0;

    if (bus_attr->store)
        ret = bus_attr->store(subsys_priv->bus, buf, count);     /*调用具体总线属性写函数*/
    return ret;
}
```

读和写属性函数通过容器机制，由属性 attribute 实例获取总线属性 bus_attribute 实例指针，由 kobject 实例指针获取 subsys_private 实例指针（进而获取总线实例指针），然后调用 bus_attribute 实例中定义的具

体属性读写函数完成属性的读写操作。

添加总线属性的 add_probe_files()和 bus_create_file()函数等，最终都是调用 sysfs_create_file()函数，在总线目录项下添加属性文件。

在注册总线时默认添加 uevent、drivers_probe、drivers_autoprobe 总线属性以及总线定义的默认属性。

●uevent：只写属性，用于发送 uevent 事件。

●drivers_probe：只写属性，用于为指定名称的设备探测驱动。

●drivers_autoprobe：可读写属性，用于读写 bus->p->drivers_autoprobe 成员值，写入非零值表示设置成员值，否则清零。drivers_autoprobe 成员值用于控制向总线添加设备时是否自动探测驱动。

4 子系统接口

子系统接口通常用于对总线下的设备执行某项特殊操作。子系统接口由 subsys_interface 结构体表示，定义如下（/include/linux/device.h）：

```
struct subsys_interface {
    const char *name;         /*名称*/
    struct bus_type *subsys;     /*指向总线*/
    struct list_head node;         /*双链表节点，将实例添加到总线 subsys->p->interfaces 双链表*/
    int (*add_dev)(struct device *dev, struct subsys_interface *sif);       /*添加设备到接口*/
    int (*remove_dev)(struct device *dev, struct subsys_interface *sif);
};
```

注册子系统接口的 subsys_interface_register()函数定义如下（/drivers/base/bus.c）：

```
int subsys_interface_register(struct subsys_interface *sif)
{
    struct bus_type *subsys;
    struct subsys_dev_iter iter;     /*/include/linux/device.h*/
    struct device *dev;
    ...
    subsys = bus_get(sif->subsys);       /*总线*/
    ...
    mutex_lock(&subsys->p->mutex);
    list_add_tail(&sif->node, &subsys->p->interfaces);     /*添加到 subsys_private 中双链表*/
    if (sif->add_dev) {
        subsys_dev_iter_init(&iter, subsys, NULL, NULL);   /*遍历总线下设备*/
        while ((dev = subsys_dev_iter_next(&iter)))
            sif->add_dev(dev, sif);       /*添加设备到接口*/
        subsys_dev_iter_exit(&iter);
    }
    mutex_unlock(&subsys->p->mutex);

    return 0;
}
```

注册子系统接口时将对总线下所有设备调用 sif->add_dev(dev, sif)函数。

注销子系统接口的函数为 subsys_interface_unregister()，将对总线下所有设备调用 sif->remove_dev(dev, sif)函数，请读者自行阅读源代码。

8.3.2 设备类

设备类是对设备的高层次抽象，由 class 结构体表示，用于按功能对设备进行分类，一个设备类管理同一类型的设备，例如：输入设备类，块设备类等。

1 数据结构

设备类 class 结构体定义在/include/linux/device.h 头文件：

```
struct class {
        const char          *name;          /*设备类名称*/
        struct module       *owner;         /*模块指针*/
        struct class_attribute          *class_attrs;       /*设备类默认属性*/
        const struct attribute_group  **dev_groups;     /*设备类中设备的默认属性（组）*/
        struct kobject      *dev_kobj;                  /*默认指向/sys/dev/char/对应的 kobject 实例*/
        int (*dev_uevent) (struct device *dev, struct kobj_uevent_env *env);
                                    /*添加、移除设备时，用于向 uevent 事件添加环境变量*/
        char *(*devnode) (struct device *dev, umode_t *mode);
                                                /*用于自动创建设备文件时，确定文件名称*/

        void (*class_release) (struct class *class);    /*释放设备类时调用此函数*/
        void (*dev_release) (struct device *dev);       /*释放设备时调用此函数*/

        int (*suspend) (struct device *dev, pm_message_t state);
                                                /*用于使设备睡眠，一般为使设备进入低功耗状态*/
        int (*resume) (struct device *dev);         /*用于从睡眠模式唤醒设备*/

        const struct    kobj_ns_type_operations    *ns_type;    /*sysfs 用于确定命名空间*/
        const void *(*namespace) (struct device *dev);          /*返回命名空间标签*/
        const struct dev_pm_ops *pm;        /*设备类中设备默认功耗管理操作结构*/

        struct subsys_private *p;           /*设备类私有数据结构，与 bus 中私有数据结构相同*/
};
```

2 创建设备类

内核提供了动态创建设备类的接口函数，函数内负责创建 class 实例并向内核注册。在介绍创建设备类函数之前，先简单看一下设备类的初始函数 classes_init()，函数在/drivers/base/class.c 文件内定义：

```
int __init classes_init(void)
{
    class_kset = kset_create_and_add("class", NULL, NULL);      /*/sys/class*/
     ...
    return 0;
```

```
}
```

初始化函数很简单，主要是创建名称为 class 的 kset 结构体实例，并导出到 sysfs 文件系统根目录下，即创建/sys/class/目录，表示设备类的集合。

创建设备类的接口函数 class_create()声明在/include/linux/device.h 头文件：

```
#define class_create(owner, name)            \
 ({                                  \
      static struct lock_class_key __key;\
      __class_create(owner, name, &__key);      \
 })
```

参数 owner 表示模块指针，name 表示设备类名称。

class_create()函数内调用的__class_create()函数在/drivers/base/class.c 内实现，代码如下：

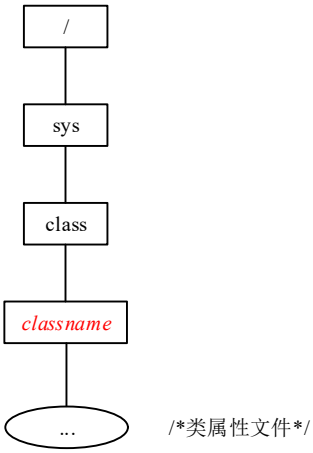
```
struct class *__class_create(struct module *owner, const char *name,struct lock_class_key *key)
{
      struct class *cls;
      int retval;

      cls = kzalloc(sizeof(*cls), GFP_KERNEL);     /*创建 class 结构体实例*/
      ...          /*错误处理*/

      cls->name = name;              /*设备类名称*/
      cls->owner = owner;
      cls->class_release = class_create_release;   /*默认的释放设备类函数*/

      retval = __class_register(cls, key);              /*注册设备类*/
      ...
      return cls;
      ...
}
```

注册设备类的__class_register()函数定义如下：

```
int __class_register(struct class *cls, struct lock_class_key *key)
{
    struct subsys_private *cp;
    int error;

    pr_debug("device class '%s': registering\n", cls->name);

    cp = kzalloc(sizeof(*cp), GFP_KERNEL);   /*创建 subsys_private 实例*/
    if (!cp)
        return -ENOMEM;
    klist_init(&cp->klist_devices, klist_class_dev_get, klist_class_dev_put);      /*初始化设备链表*/
```

```
    INIT_LIST_HEAD(&cp->interfaces);        /*初始化链表*/
    kset_init(&cp->glue_dirs);              /*初始化链表成员*/
    __mutex_init(&cp->mutex, "subsys mutex", key);
    error = kobject_set_name(&cp->subsys.kobj, "%s", cls->name);
                                            /*设备类名称设为 subsys.kobj 名称*/
    ...    /*错误处理*/

    if (!cls->dev_kobj)      /*dev_kobj 为空则指向字符设备 kobject，/sys/dev/char/*/
        cls->dev_kobj = sysfs_dev_char_kobj;     /*/sys/dev/char/*/

    #if defined(CONFIG_BLOCK)                    /*如果选择了支持块设备配置选择项*/
        if (!sysfs_deprecated || cls != &block_class)
            cp->subsys.kobj.kset = class_kset;   /*不是 block_class 则上级目录为/sys/class/*/
    #else
            cp->subsys.kobj.kset = class_kset;   /*没有配置支持块设备时，上级目录为/sys/class/*/
    #endif

    cp->subsys.kobj.ktype = &class_ktype;     /*设备类 kobject 类型*/
    cp->class = cls;                          /*指向设备类*/
    cls->p = cp;                              /*指向 subsys_private 实例*/

    error = kset_register(&cp->subsys);       /*注册 kset，导出目录为/sys/class/class_name*/
    ...    /*错误处理*/
    error = add_class_attrs(class_get(cls));
                              /*添加设备类默认属性，在/sys/class/class_nam/下创建属性文件*/
    class_put(cls);
    return error;
}
```

设备类的注册比总线的注册简单许多，函数内创建设备类私有数据结构 subsys_private 实例，并初始化，将跟踪 class 实例的 class.subsys.kobj 实例添加到 class_kset 集合下（块设备类外），设备类在 sysfs 文件系统中导出目录/sys/class/*class_name*/，函数最后创建设备类的默认属性文件，如下图所示。



设备类 class 实例也可以静态创建，随后调用 **class_register(class)**函数向内核注册，此函数内直接调用

__class_register(cls, key)函数完成设备类实例的注册。

void class_unregister(struct class *cls)函数用于注销设备类。

3 设备类属性

设备类属性由 class_attribute 结构表示，定义如下（/include/linux/device.h）：

```
struct class_attribute {
    struct attribute attr;          /*通用属性结构体*/
    ssize_t (*show) (struct class *class, struct class_attribute *attr,char *buf);     /*具体属性读函数*/
    ssize_t (*store) (struct class *class, struct class_attribute *attr,const char *buf, size_t count);
                                                                                        /*具体属性写函数*/
};
```

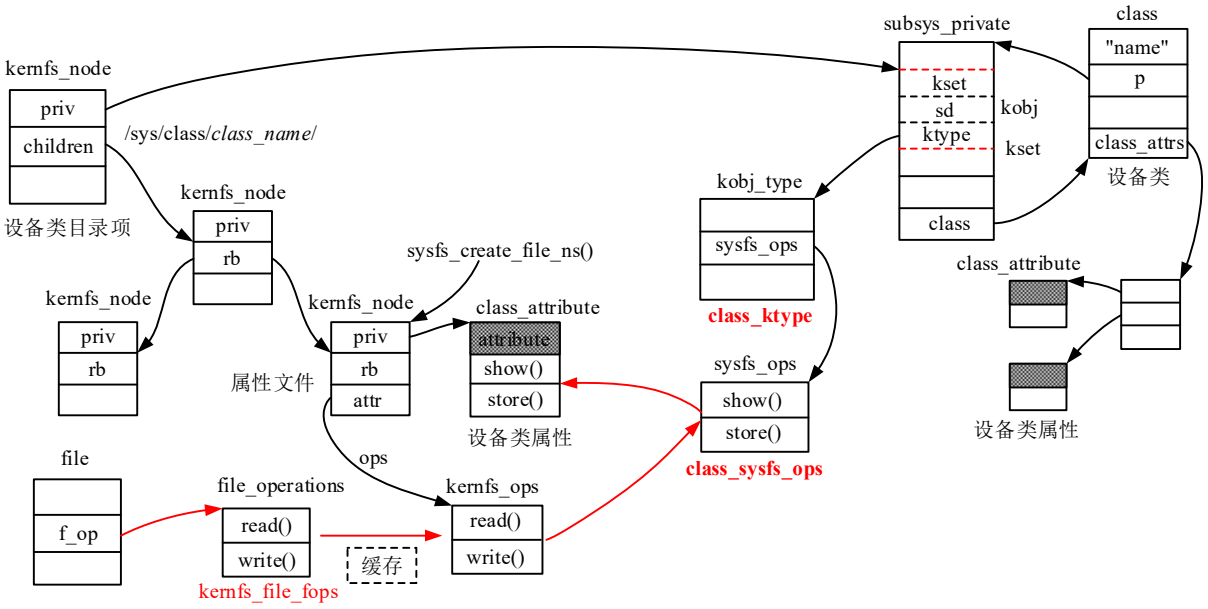
内核在/include/linux/device.h 头文件中实现了定义设备类属性的宏，例如：

```
#define   CLASS_ATTR(_name, _mode, _show, _store) \      /*指定读写函数的读写属性*/
        struct class_attribute class_attr_##_name = __ATTR(_name, _mode, _show, _store)

#define   CLASS_ATTR_RW(_name) \                  /*默认函数名称的读写属性*/
        struct class_attribute class_attr_##_name = __ATTR_RW(_name)

#define   CLASS_ATTR_RO(_name) \                   /*默认函数名称的只读属性*/
        struct class_attribute class_attr_##_name = __ATTR_RO(_name)
```

内核在注册设备类 class 实例时，其 kobject 实例类型为 **subsys.kobj.ktype = &class_ktype**，class_ktype 实例定义如下：

```
static struct kobj_type class_ktype = {
        .sysfs_ops= &class_sysfs_ops,
        .release  = class_release,
        .child_ns_type      = class_child_ns_type,
};

static const struct sysfs_ops class_sysfs_ops = {
        .show    = class_attr_show,          /*设备类属性读函数*/
        .store    = class_attr_store,          /*设备类属性写函数*/
};
```

class_sysfs_ops 实例中定义的 show()和 store()函数将调用设备类属性 class_attribute 中的 show()和 store()函数，如下图所示：



设备类属性读写函数与总线属性的读写函数类似，函数定义在/drivers/base/class.c 文件内，源代码请读者自行阅读。

向设备类添加属性的函数最终调用 sysfs_create_file_ns()函数，在设备类对应的目录项下创建属性文件。

4 设备类接口

设备类与总线类似，也支持设备类接口。设备类接口由 class_interface 结构体表示，定义如下：

```
struct class_interface {        /*/include/linux/device.h*/
    struct list_head    node;       /*添加到设备类 subsys->p->interfaces 双链表*/
    struct class        *class;     /*指向设备类*/

    int (*add_dev)      (struct device *, struct class_interface *);    /*向接口添加设备*/
    void (*remove_dev)      (struct device *, struct class_interface *);    /*从接口移除设备*/
};
```

注册、注销设备类接口的函数定义在/drivers/base/class.c 文件内，原型如下，源代码请读者自行阅读：

int __must_check **class_interface_register**(struct class_interface *)：注册设备类接口，成功返回 0。

void **class_interface_unregister**(struct class_interface *)：注销设备类接口。

8.4 设备与驱动

在通用驱动模型中设备由 device 结构体表示，用于向驱动传递设备硬件信息，驱动由 device_driver 结构体表示，用于驱动程序和设备的管理。

设备 device 和驱动 device_driver 必须挂接在总线上，当向总线注册设备/驱动时，将扫描总线中的驱动/设备链表，寻找匹配的项。若设备与驱动匹配成功，则调用 device_driver 实例中定义的探测函数 probe()，注册设备驱动程序，即创建设备驱动数据结构实例并注册。

8.4.1 创建管理目录

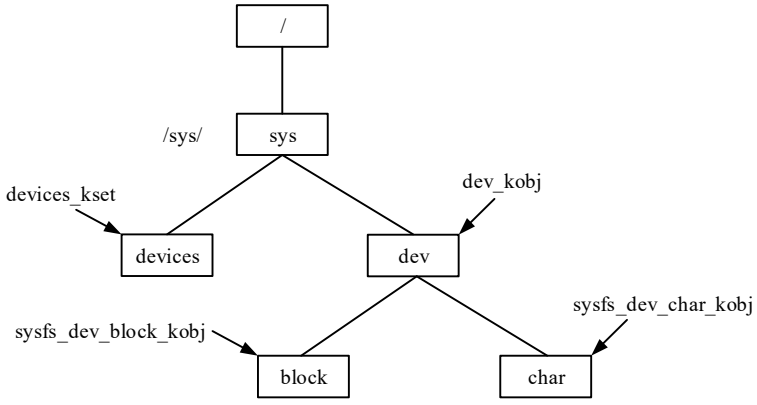
在通用驱动模型初始化函数中，将调用 devices_init()函数在 sysfs 中创建管理设备和驱动的目录，函数在/drivers/base/core.c 文件内实现，代码简列如下：

```
int __init devices_init(void)
{
   devices_kset = kset_create_and_add("devices", &device_uevent_ops, NULL); /*创建/sys/devices/目录*/
   ...
   dev_kobj = kobject_create_and_add("dev", NULL);              /*创建/sys/dev/目录*/
   ...
   sysfs_dev_block_kobj = kobject_create_and_add("block", dev_kobj);   /*创建/sys/dev/block/目录*/
   ...
   sysfs_dev_char_kobj = kobject_create_and_add("char", dev_kobj);     /*创建/sys/dev/char/目录*/
   ...
   return 0;
   ...
}
```

devices_init()函数比较简单主要是在 sysfs 文件系统中创建 kset 和 kobject 实例。

这里需要注意的是 devices_kset 实例的创建，在向内核添加 device 实例时，默认将 device 实例添加到 **devices_kset** 集合中，总线和设备类下的 device 是到 devices_kset 下 device 实例的符号链接。

devices_init()函数在 sysfs 中创建的目录项如下图所示：



devices_init()函数中创建的 devices_kset 实例，其 uevent_ops 成员被赋予 device_uevent_ops，此实例定义在/drivers/base/core.c 文件内：

```
static const struct kset_uevent_ops device_uevent_ops = {
    .filter =   dev_uevent_filter,          /*对于 device 实例，返回 1，不屏蔽 uevent 事件*/
    .name =   dev_uevent_name,              /*获取产生 uevent 事件的总线或设备类的名称*/
    .uevent =dev_uevent,                    /*向 uevent 事件添加环境变量的函数*/
};
```

在添加/删除设备 device 实例时，会调用 kobject_uevent()函数发送 uevent 事件。由前面第 7 章介绍的 kobject_uevent()函数可知，将调用 device 实例所在集合 kset 实例关联 kset_uevent_ops 实例中的 uevent()函数，即 devices_kset->uevent_ops->uevent()函数，向 uevent 事件添加环境变量，也就是调用 dev_uevent()函数。

dev_uevent()函数定义如下（/drivers/base/core.c）：

```
static int dev_uevent(struct kset *kset, struct kobject *kobj, struct kobj_uevent_env *env)
{
    struct device *dev = kobj_to_dev(kobj);
    int retval = 0;

    if (MAJOR(dev->devt)) {        /*如果 device 实例 devt 成员中赋值了主设备号*/
        const char *tmp;
        const char *name;
        umode_t mode = 0;
        kuid_t uid = GLOBAL_ROOT_UID;
        kgid_t gid = GLOBAL_ROOT_GID;

        add_uevent_var(env, "MAJOR=%u", MAJOR(dev->devt));     /*向 uevent 事件添加环境变量*/
        add_uevent_var(env, "MINOR=%u", MINOR(dev->devt));     /*传递主从设备号信息*/
        name = device_get_devnode(dev, &mode, &uid, &gid, &tmp);   /*设备名称*/
        if (name) {
            add_uevent_var(env, "DEVNAME=%s", name);
            if (mode)
                add_uevent_var(env, "DEVMODE=%#o", mode & 0777);
            if (!uid_eq(uid, GLOBAL_ROOT_UID))
                add_uevent_var(env, "DEVUID=%u", from_kuid(&init_user_ns, uid));
            if (!gid_eq(gid, GLOBAL_ROOT_GID))
                add_uevent_var(env, "DEVGID=%u", from_kgid(&init_user_ns, gid));
            kfree(tmp);
        }
    }

    if (dev->type && dev->type->name)        /*设备类型名称*/
        add_uevent_var(env, "DEVTYPE=%s", dev->type->name);

    if (dev->driver)          /*关联了驱动*/
        add_uevent_var(env, "DRIVER=%s", dev->driver->name);      /*驱动名称*/

    of_device_uevent(dev, env);              /*设备树相关*/

    if (dev->bus && dev->bus->uevent) {
        retval = dev->bus->uevent(dev, env);     /*调用总线 uevent 事件函数，添加环境变量*/
        ...
    }

    if (dev->class && dev->class->dev_uevent) {
        retval = dev->class->dev_uevent(dev, env);     /*调用设备类 uevent 事件函数，添加环境变量*/
        ...
```

```
    }

    if (dev->type && dev->type->uevent) {          /*设备类型定义的 uevent 函数，添加环境变量*/
        retval = dev->type->uevent(dev, env);
        ...
    }
    return retval;
}
```

dev_uevent()函数主要用于向 uevent 事件添加环境变量，包括设备所在总线、设备类、设备类型等需要传递的环境变量，各环境变量通过 uevent 机制传递到用户空间，用户空间进程（udev/mdev）获取环境变量后，可执行相应的操作，如加载模块、创建设备文件等。

8.4.2 设备

在通用驱动模型中设备由 device 结构体表示，通常由板级（平台）相关代码定义并注册设备 device 实例。

1 数据结构

设备 device 结构体定义在/include/linux/device.h 头文件内：

```
struct device {
    struct device     *parent;          /*指向父设备，通常是表示总线或主机控制器的 device 实例*/
    struct device_private   *p;         /*设备私有数据结构指针*/
    struct kobject    kobj;             /*跟踪 device 实例的 kobject 实例，导出到 sysfs 文件系统*/
    const char        *init_name;       /*设备初始名称，可用于设备文件名称*/
    const struct device_type   *type;   /*设备类型，包含设备类型特定信息*/
    struct mutex          mutex;        /*互斥量*/
    struct bus_type       *bus;         /*所在总线实例指针*/
    struct device_driver   *driver;     /*匹配驱动实例指针*/
    void        *platform_data;         /*平台相关硬件信息，在板级文件中定义，如 IO 地址等*/
    void        *driver_data;           /*驱动程序中使用的数据结构*/
    struct   dev_pm_info   power;       /*电源管理结构，/include/linux/pm.h*/
    struct   dev_power_domain   *pm_domain;   /*提供关机、停机等时机的回调函数*/
#ifdef CONFIG_PINCTRL
    struct dev_pin_info *pins;          /*引脚信息*/
#endif
#ifdef CONFIG_NUMA
    ...
#endif
    u64  *dma_mask;                 /* dma mask*/
    u64  coherent_dma_mask;         / * */
    unsigned long dma_pfn_offset;     /*DMA 内存偏移量相对于 RAM*/
    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 memoverride */
#ifdef CONFIG_DMA_CMA
    ...
#endif
    struct dev_archdata   archdata;
                                /*体系结构相关数据结构实例，/arch/mips/include/asm/device.h*/
    struct device_node  *of_node;          /*对应的设备树节点*/
    struct fwnode_handle     *fwnode;      /*固件相关设备节点*/
    dev_t      devt;                       /*设备号*/
    u32        id;                         /*设备在总线中的编号*/
    spinlock_t              devres_lock;   /*自旋锁用于保护设备资源*/
    struct list_head        devres_head;   /*设备资源 devres 实例双链表*/
    struct klist_node       knode_class;          /*用于将设备链入设备类中链表*/
    struct class            *class;               /*设备所属设备类指针*/
    const struct attribute_group **groups;        /*设备属性指针数组  */
    void (*release)    (struct device *dev);      /*释放设备时调用*/
    struct iommu_group      *iommu_group;         /*IOMMU group*/
    bool       offline_disabled:1;          /*置 1 表示设备持久在线*/
    bool       offline:1;                   /*设备下线时置 1*/
};
```

device 结构体主要成员简介如下：

●**parent**：指向父设备，通常是表示总线或主机控制器的 device 实例。内核中将总线、设备控制器等也视为设备注册到内核，device 实例在内核中组成层次的父子结构。

●**kobj**：跟踪 device 实例的 kobject 结构体成员，导出到 sysfs 文件系统。

●**driver**：指向匹配驱动 device_driver 实例。

●**platform_data**：指向表示设备硬件信息的数据结构，如 IO 地址、中断号等，结构体在板级文件中定义，传递给驱动程序。

●**driver_data**：指向驱动程序中数据结构。

●**devt**：设备号，含主从设备号。

●**type**：设备类型 device_type 结构体指针。device_type 结构体定义在/include/linux/device.h 头文件：

```
struct device_type {
    const char    *name;                /*设备类型名称*/
    const struct attribute_group **groups;       /*设备属性指针数组*/
    int (*uevent) (struct device *dev, struct kobj_uevent_env *env); /*向 uevent 事件添加环境变量函数*/
    char *(*devnode) (struct device *dev, umode_t *mode,kuid_t *uid, kgid_t *gid);
                                                          /*获取设备文件名称的函数*/
    void (*release) (struct device *dev);   /*释放设备函数*/
    const struct dev_pm_ops *pm;          /*电源管理相关数据结构*/
};
```

device_type 结构体表示的设备类型不是前面介绍的设备类，它是比设备类更细的划分，如鼠标、分区等。

●**p**：device 私有数据结构 device_private 指针，device_private 结构体定义在/drivers/base/base.h 头文件，

表示设备私有信息：

```
struct device_private {
    struct klist klist_children;          /*链接子设备的链表头*/
    struct klist_node knode_parent;       /*链接同一父设备下兄弟设备*/
    struct klist_node knode_driver;       /*链接到驱动 device_driver 中链表*/
    struct klist_node knode_bus;          /*链接到总线下的设备链表*/
    struct list_head deferred_probe;      /*用于重试绑定驱动*/
    struct device *device;                /*指向表示设备 device 实例*/
};
```

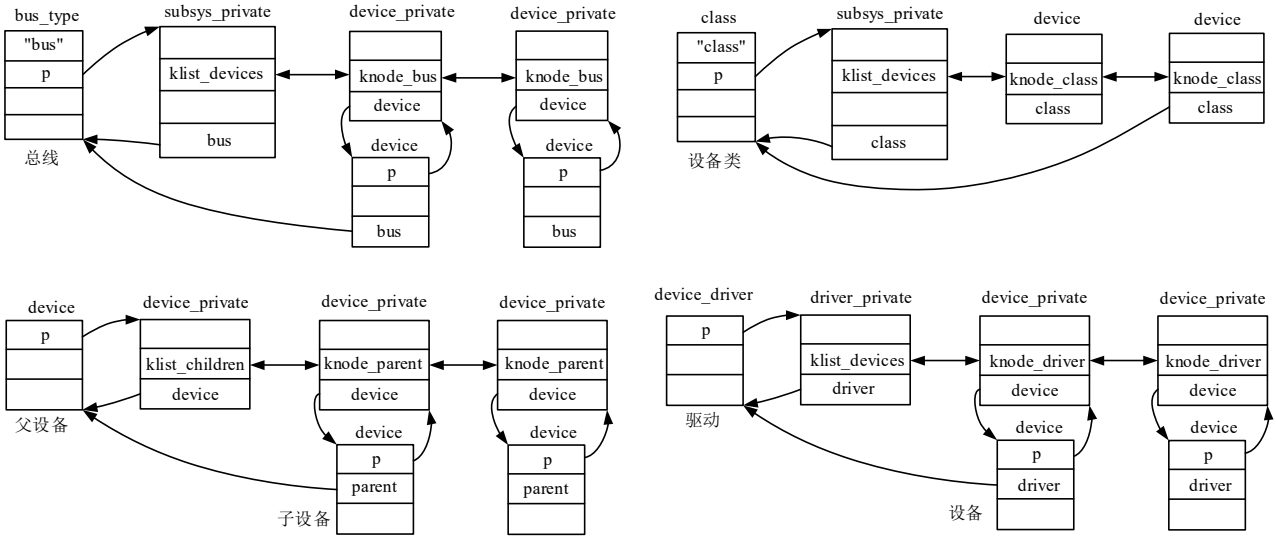
由上可知 device 实例存在于以下管理结构中（如下图所示）：

（1）通过 device_private.knode_parent 成员添加到父设备 device_private.klist_children 链表。

（2）通过 device_private.knode_bus 添加到总线的设备链表。

（3）通过 knode_class 成员添加到设备类的设备链表。

（4）通过 device_private.knode_driver 添加到匹配驱动的设备链表。



2 注册设备

device 实例通常在板级（平台）相关代码中定义，或根据设备树节点（见下节）创建，然后注册到内核。

device_register(struct device *dev)函数用于向内核注册设备。注册设备就是将 device 实例添加到各种管理结构中，导出到 sysfs 文件系统，并探测总线上匹配的驱动，匹配成功则调用总线或驱动的 probe()函数，完成设备驱动程序的注册。

device_register(struct device *dev)函数定义在/drivers/base/core.c 文件内，代码如下：

```
int device_register(struct device *dev)
{
    device_initialize(dev);          /*初始化设备，/drivers/base/core.c*/
    return device_add(dev);          /*添加设备，/drivers/base/core.c*/
}
```

注册设备 device 实例主要分两步，一是初始化设备 device 实例，二是将设备 device 实例添加到各种管理结构中，导出到 sysfs 文件系统，并探测驱动等。

注销 device 实例的函数为 void **device_unregister**(struct device *dev)。

■初始化 device

初始化 device 实例 device_initialize(dev)函数定义在/drivers/base/core.c 文件内，代码如下：

```
void device_initialize(struct device *dev)
{
    dev->kobj.kset = devices_kset;          /*指定集合为 devices_kset（父节点）*/
    kobject_init(&dev->kobj, &device_ktype);    /*kobj_type 类型指定为 device_ktype*/
    INIT_LIST_HEAD(&dev->dma_pools);
    mutex_init(&dev->mutex);
    lockdep_set_novalidate_class(&dev->mutex);
    spin_lock_init(&dev->devres_lock);
    INIT_LIST_HEAD(&dev->devres_head);
    device_pm_init(dev);        /*电源管理结构初始化，/drivers/base/power/power.h*/
    set_dev_node(dev, -1);      /*设置内存节点，没有选择 NUMA 选项为空操作*/
}
```

初始化函数主要完成 device 实例各成员的初始化，需要注意的是跟踪实例的 kobject 结构体成员，其 kset 成员指向 devices_kset，kobj_type 类型为 device_ktype。

device_initialize()初始化的 device 实例在没有指定父设备的情况下将导出到/sys/devices/目录下，指定了父设备则导出到父设备对应的目录项下。

●设备属性

device 实例导出到 sysfs 为目录项，其下属性表现为文件。属性文件的读写操作由 kobj_type 实例实现，device.kobj 关联的 kobj_type 实例为 device_ktype。

device_ktype 实例定义在/drivers/base/core.c 文件内：

```
static struct kobj_type device_ktype = {
    .release    = device_release,
    .sysfs_ops  = &dev_sysfs_ops,       /*读写设备属性操作结构*/
    .namespace  = device_namespace,
};
```

读写设备属性的通用操作 dev_sysfs_ops 实例定义如下（/drivers/base/core.c）：

```
static const struct sysfs_ops dev_sysfs_ops = {
    .show= dev_attr_show,       /*调用具体设备属性定义的读函数*/
    .store= dev_attr_store,     /*调用具体设备属性定义的写函数*/
};
```

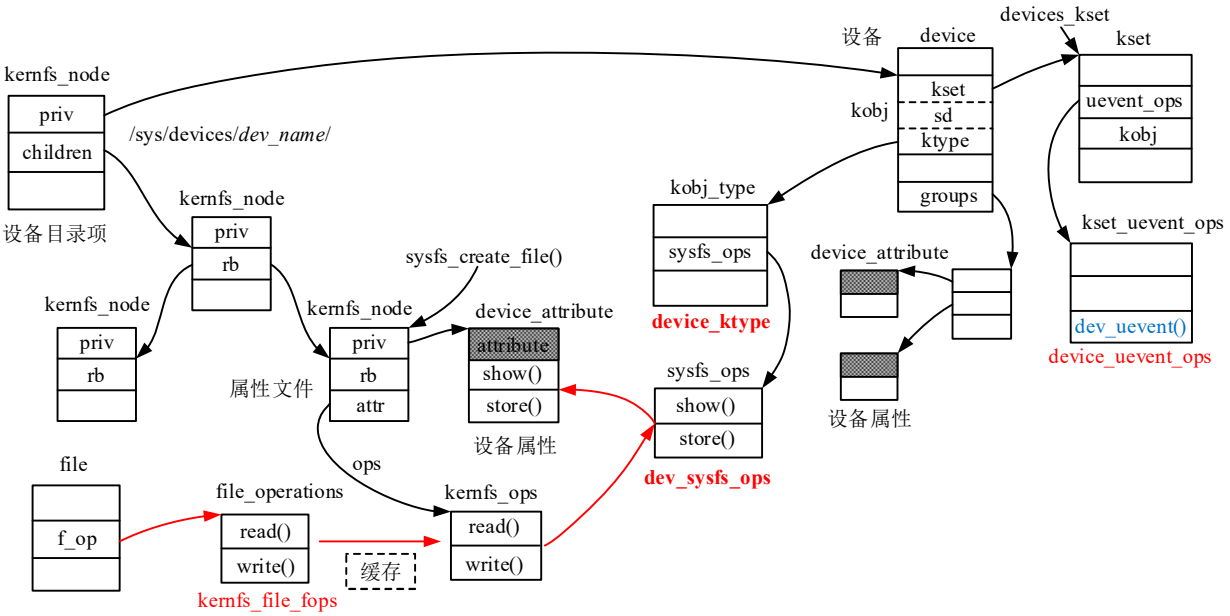
设备属性由 device_attribute 结构体表示，定义如下（/include/linux/device.h）：

```
struct device_attribute {
    struct attribute    attr;               /*通用属性结构体*/
    ssize_t (*show) (struct device *dev, struct device_attribute *attr,char *buf);
    ssize_t (*store) (struct device *dev, struct device_attribute *attr, const char *buf, size_t count);
};
```

在/include/linux/device.h 头文件内实现了定义设备属性的宏，例如：

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \     /*指定读写函数的设备属性*/
        struct device_attribute    dev_attr_##_name = __ATTR(_name, _mode, _show, _store)

#define DEVICE_ATTR_RW(_name) \       /*采用默认读写函数名称的设备属性*/
      struct device_attribute dev_attr_##_name = __ATTR_RW(_name)
```

dev_sysfs_ops 实例中的设备属性读写函数与总线、设备类属性读写函数相似，也是调用具体设备属性中定义的读写函数完成操作，如下图所示。



注意上图中标注的 dev_uevent()函数，在对 device 实例进行添加、删除等操作时，触发的 uevent 事件将调用 dev_uevent()函数向 uevent 事件添加环境变量，函数内又将调用设备所在总线、所属设备类和设备类型等实例中的 uevent()函数向 uevent 事件添加环境变量，详见本节开头。

■添加 device

注册 device 实例的第二步是调用 **device_add(dev)**函数向内核添加 device 实例，device_add(dev)函数是一个非常重要的函数，不光是在 device_register(dev)函数中会被调用。

device_add(dev)函数完成的主要工作如下：

（1）为设备分配并初始化 device_private 实例。

（2）确定设备名称，可能用于设备文件命名。

（3）确定父设备，并向 sysfs 添加 device.kobj 实例。如果存在父设备，则 device.kobj 导出到父设备在 sysfs 中的目录项下，否则导出到/sys/devices/目录下。假设设备导出目录项名称为 dev_name。

（4）在设备目录项下创建到设备类的符号链接 subsystem（可能被后面的总线覆盖），创建到父设备的符号链接 device（如果指定了父设备），在设备类下创建到设备的符号链接 dev_name。

（5）在设备目录项下创建到总线的符号链接 subsystem， 在总线/sys/bus/*bus_name*/devices/目录下创建到设备的符号链接 dev_name。

（6）在/sys/dev/char/或/sys/dev/block/目录下创建到设备的符号链接，名称为“主设备号：从设备号”。

（7）向设备目录项下添加属性文件，包括设备自带的默认属性，以及所在总线，所属设备类，设备类型赋予设备的默认属性。

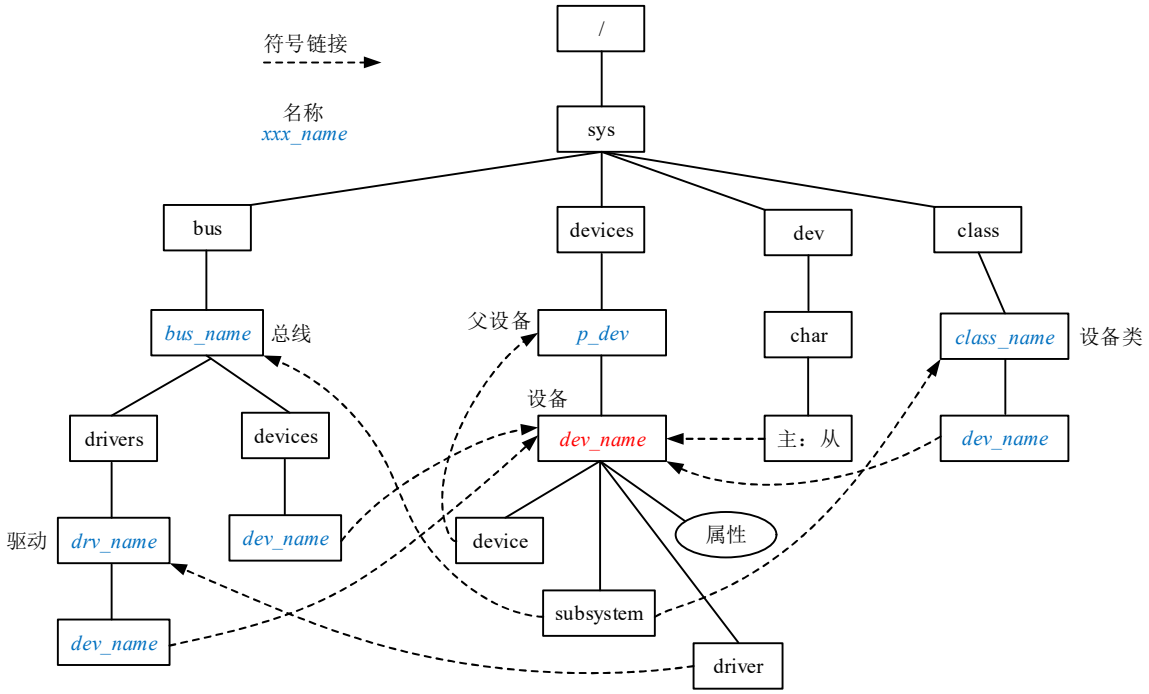
（8）将 device 实例添加到各管理结构中，包括总线、设备类中的设备链表，父设备中的子设备链表

等。

（9）自动创建设备文件，触发 uevent 事件。

（10）探测设备驱动，找到匹配驱动则加载设备驱动程序，并创建设备与驱动之间的符号链接，将 device 实例添加到驱动 device_driver 实例中设备链表。

device_add(dev)函数创建的目录项、符号链接、属性文件如下图所示：



下面来看一下 device_add(dev)函数的实现，函数定义在/drivers/base/core.c 文件内：

```
int device_add(struct device *dev)
{
    struct device *parent = NULL;
    struct kobject *kobj;
    struct class_interface *class_intf;
    int error = -EINVAL;

    dev = get_device(dev);      /*增加 dev->kobj 引用计数*/
    ...
    if (!dev->p) {
        error = device_private_init(dev);    /*分配并初始化 device_private 实例*/
        ...
    }

    if (dev->init_name) {          /*设置了名称，可能用于设备文件名称*/
        dev_set_name(dev, "%s", dev->init_name);    /*设置 init_name 至 dev->kobj.name 成员*/
        dev->init_name = NULL;
    }
```

```
if (!dev_name(dev) && dev->bus && dev->bus->dev_name)     /*如果没有设置设备名称*/
    dev_set_name(dev, "%s%u", dev->bus->dev_name, dev->id);
                                                    /*名称设为 bus->dev_name+dev->id*/
...     /*错误处理*/

parent = get_device(dev->parent);
kobj = get_device_parent(dev, parent);
if (kobj)                                   /*如果存在父设备*/
    dev->kobj.parent = kobj;        /*指向父设备 kobject*/

if (parent)
    set_dev_node(dev, dev_to_node(parent));       /*没有选择 NUMA 选项为空操作*/

error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
            /*向内核添加 dev->kobj 实例，创建设备对应的目录项，名称为 init_name*/
...

/* notify platform of device entry */
if (platform_notify)
    platform_notify(dev);

error = device_create_file(dev, &dev_attr_uevent);
                            /*在设备目录项下添加 uevent 属性文件，写操作触发 uevent 事件*/
...
error = device_add_class_symlinks(dev);          /*/drivers/base/core.c*/
             /*在设备类创建下创建到设备的符号链接，创建到设备类、父设备的符号链接*/
...
error = device_add_attrs(dev);          /*/drivers/base/core.c*/
        /*添加 class->dev_groups，device_type->groups，device->groups 指定的设备属性文件*/
...
error = bus_add_device(dev);     /*向总线添加设备，见下文，/drivers/base/bus.c*/
...
error = dpm_sysfs_add(dev);
              /*配置了 PM 选项时注册设备电源管理属性，/drivers/base/power/sysfs.c*/
...
device_pm_add(dev);          /*将设备添加到电源管理结构中，/drivers/base/power/main.c*/

if (MAJOR(dev->devt)) {     /*如果设置了主设备号，则自动创建设备文件*/
    error = device_create_file(dev, &dev_attr_dev);     /*增加 dev 属性，读属性显示设备号*/
    ...
    error = device_create_sys_dev_entry(dev);   /*/drivers/base/core.c*/
            /*在/sys/dev/char 或/sys/dev/block 目录下创建到 device 实例的符号链接*/
    ...
```

```
        devtmpfs_create_node(dev);      /*在/dev/（devtmpfs）目录下创建设备文件，见本章上文*/
    }

    if (dev->bus)       /*执行总线中通知链*/
      blocking_notifier_call_chain(&dev->bus->p->bus_notifier,BUS_NOTIFY_ADD_DEVICE, dev);

    kobject_uevent(&dev->kobj, KOBJ_ADD);   /*向用户空间发出 uevent 事件，传递环境变量*/
    bus_probe_device(dev);          /*探测设备驱动，见下文，/drivers/base/bus.c*/
    if (parent)
        klist_add_tail(&dev->p->knode_parent,&parent->p->klist_children); /*添加到父设备的子链表*/

    if (dev->class) {
        mutex_lock(&dev->class->p->mutex);
        klist_add_tail(&dev->knode_class,&dev->class->p->klist_devices);
                                                    /*将设备添加到设备类的设备链表*/
        list_for_each_entry(class_intf,&dev->class->p->interfaces, node)
            if (class_intf->add_dev)
                class_intf->add_dev(dev, class_intf);          /*将设备添加到设备类接口*/
        mutex_unlock(&dev->class->p->mutex);
    }
done:
    put_device(dev);
    return error;
    ...
}
```

device_add(dev)函数的主要工作前面已经介绍过了。下面介绍一下向总线添加设备的 bus_add_device()函数和探测设备驱动的 bus_probe_device()函数的定义。

●向总线添加设备

bus_add_device()函数用于将设备添加到总线，函数定义在/drivers/base/bus.c 文件内，代码如下：

```
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);      /*/drivers/base/bus.c*/
                    /*添加总线定义的设备默认属性，bus->dev_attrs（device_attribute 数组）*/
        if (error)
            goto out_put;
        error = device_add_groups(dev, bus->dev_groups);
                        /*添加总线定义的设备属性，attribute 指针数组，/drivers/base/core.c*/
        if (error)
```

```
            goto out_id;
        error = sysfs_create_link(&bus->p->devices_kset->kobj,&dev->kobj, dev_name(dev));
                        /*创建/sys/bus/bus_name/devices/dev_name 符号链接，到设备目录项*/
        if (error)
            goto out_groups;
        error = sysfs_create_link(&dev->kobj,&dev->bus->p->subsys.kobj, "subsystem");
                    /*在设备目录项下创建到总线的符号链接，/sys/devices/dev_name/subsystem*/
        if (error)
            goto out_subsys;
        klist_add_tail(&dev->p->knode_bus, &bus->p->klist_devices);
                                /*将设备添加到总线 bus->p->klist_devices 设备链表*/
    }
    return 0;
    ...
}
```

bus_add_device()函数主要是向设备目录项下添加总线定义的设备属性，创建符号链接，将设备添加到总线中的设备链表等，这里还没有将设备添加到子系统接口（在 bus_probe_device()函数中添加）。

●匹配驱动

device_add(dev)函数中调用 **bus_probe_device()**函数完成添加设备最后一项非常重要的工作，那就是为设备探测驱动。

bus_probe_device()函数将扫描总线上的驱动 device_driver 实例链表，逐个调用 **bus->match(dev, drv)** 函数判断驱动与设备是否匹配，若匹配则调用总线 **bus->probe()**函数或驱动 **device_drive->probe()**函数完成设备驱动程序的加载（不再扫描后面的 device_driver 实例），即注册设备驱动数据结构实例。

bus_probe_device()函数定义在/drivers/base/bus.c 文件内，代码如下：

```
void bus_probe_device(struct device *dev)
{
    struct bus_type *bus = dev->bus;
    struct subsys_interface *sif;

    if (!bus)
        return;

    if (bus->p->drivers_autoprobe)          /*如果总线允许自动探测驱动*/
        device_initial_probe(dev);          /*匹配设备驱动，/drivers/base/dd.c*/

    mutex_lock(&bus->p->mutex);
    list_for_each_entry(sif, &bus->p->interfaces, node)
      if (sif->add_dev)
            sif->add_dev(dev, sif);             /*将设备添加到子系统接口*/
    mutex_unlock(&bus->p->mutex);
}
```

bus_probe_device()函数判断总线是否设置了允许自动探测驱动，如果是则调用 device_initial_probe()函数完成驱动的探测，随后还将设备添加到总线中子系统接口。

device_initial_probe(struct device *dev)函数定义在/drivers/base/dd.c 文件内，代码如下：

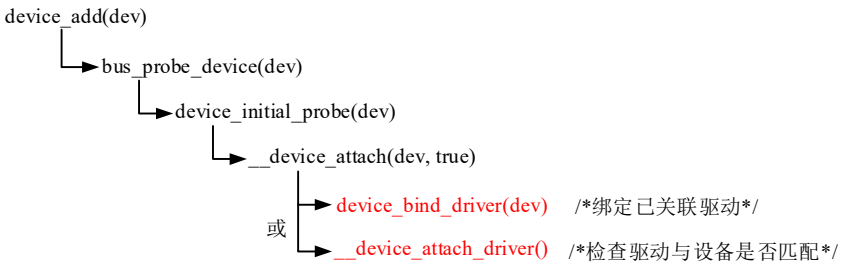
```
void device_initial_probe(struct device *dev)
{
    __device_attach(dev, true);    /*/drivers/base/dd.c*/
}
```

__device_attach(dev, true)函数用于从总线探测匹配的驱动，函数定义在/drivers/base/dd.c 文件内：

```
static int __device_attach(struct device *dev, bool allow_async)
/*allow_async：true*/
{
    int ret = 0;

    device_lock(dev);                /*锁定设备*/
    if (dev->driver) {               /*若设备已经关联了驱动 device_drvier 实例*/
        if (klist_node_attached(&dev->p->knode_driver)) {
                                     /*如果 dev->p 已经添加到 device_drvier 中链表，返回 1*/
            ret = 1;
            goto out_unlock;
        }
        /*设备还没有绑定驱动*/
        ret = device_bind_driver(dev);       /*主要完成设备与驱动绑定，/drivers/base/dd.c*/
        if (ret == 0)
            ret = 1;          /*绑定成功*/
        else {
            dev->driver = NULL;        /*绑定不成功*/
            ret = 0;
        }
    }
    else {                   /*尚未关联驱动*/
        struct device_attach_data data = {        /*/drivers/base/dd.c*/
            .dev = dev,
            .check_async = allow_async,
            .want_async = false,
        };

        ret = bus_for_each_drv(dev->bus, NULL, &data,__device_attach_driver);
                    /*扫描总线上驱动实例，调用__device_attach_driver()检查驱动是否匹配*/
        if (!ret && allow_async && data.have_async) {
            dev_dbg(dev, "scheduling asynchronous probe\n");
            get_device(dev);
            async_schedule(__device_attach_async_helper, dev);
        } else {
            pm_request_idle(dev);
```

```
            }
        }
    out_unlock:
        device_unlock(dev);
        return ret;             /*成功返回 0*/
}
```

__device_attach(dev, true)函数调用关系如下图所示：



函数首先判断 device 实例是否已经关联了 device_driver 实例，如果是且已绑定驱动则函数返回，如果没有绑定则调用 device_bind_driver(dev)函数绑定驱动，主要工作是创建设备与驱动之间的符号链接，以及将设备添加到驱动中的设备链表。

如果设务尚未关联驱动，则调用 bus_for_each_drv()函数逐个扫描总线驱动链表中的 device_driver 实例，调用__device_attach_driver()函数检查驱动是否与设备匹配，若匹配则调用总线驱动的 probe()函数，完成设备驱动程序的注册，__device_attach_driver()函数返回 0。

__device_attach_driver()函数暂且放一下，因为后面注册驱动时要扫描总线中的设备链表，也要执行匹配操作，因此介绍完驱动之后再一并介绍。

3 动态创建设备

前面介绍的注册 device 实例的 device_register(dev)函数用于注册静态定义或已经创建好的 device 实例，内核还提供了动态创建并添加 device 实例的接口函数 **device_create()**。

device_create()函数定义在/drivers/base/core.c 文件内，代码如下：

```
struct device *device_create(struct class *class, struct device *parent,dev_t devt, void *drvdata, \
                                                                const char *fmt, ...)
/*class：所属设备类，不能为空，parent：父设备，devt：设备号，drvdata：传递给驱动的数据*/
{
    va_list vargs;
    struct device *dev;

    va_start(vargs, fmt);
    dev = device_create_vargs(class, parent, devt, drvdata, fmt, vargs);     /*/drivers/base/core.c*/
    va_end(vargs);
    return dev;
}

struct device *device_create_vargs(struct class *class, struct device *parent,      \
                                   dev_t devt, void *drvdata, const char *fmt,va_list args)
{
```

```
    return device_create_groups_vargs(class, parent, devt, drvdata, NULL, fmt, args);
}

device_create_groups_vargs()函数定义如下（/drivers/base/core.c）：
static struct device *device_create_groups_vargs(struct class *class, struct device *parent,     \
                dev_t devt, void *drvdata, const struct attribute_group **groups, const char *fmt, va_list args)
{
    struct device *dev = NULL;
    int retval = -ENODEV;

    if (class == NULL || IS_ERR(class))       /*必须指定设备类*/
        goto error;

    dev = kzalloc(sizeof(*dev), GFP_KERNEL);       /*分配 device 实例*/
    ...       /*错误处理*/

    device_initialize(dev);       /*初始化 device 实例，同 device_register()中调用的初始化函数*/
    dev->devt = devt;             /*赋予设备号*/
    dev->class = class;           /*设备类*/
    dev->parent = parent;         /*父设备*/
    dev->groups = groups;         /*属性组，这里为 NULL*/
    dev->release = device_create_release;          /*释放 device 实例的函数*/
    dev_set_drvdata(dev, drvdata);       /*dev->driver_data = data，/include/linux/device.h*/

    retval = kobject_set_name_vargs(&dev->kobj, fmt, args);     /*设置 device.kobj 名称*/
    if (retval)
        goto error;

    retval = device_add(dev);     /*添加 device 实例，见上文*/
    if (retval)
        goto error;
    return dev;
    ...
}
```

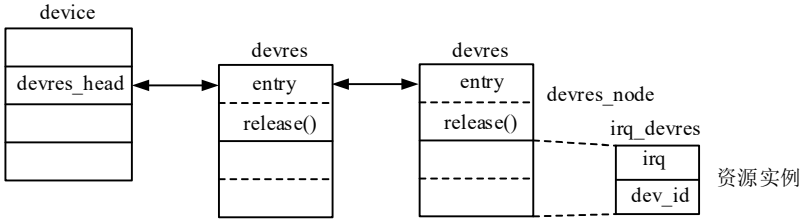
device_create()函数不难理解，函数内动态分配 device 实例，初始化后添加到内核。需要注意的是函数参数必须指定设备类 class（class 参数不能为 NULL）， drvdata 参数指向的是传递给驱动程序的数据（结构体实例），将赋予 dev->driver_data 成员。

新创建 device 实例并没有指定所属总线，也就是说添加设备时不会探测驱动，但会动态创建设备文件（devt 主设备号不为 0 时，这是一个非常重要的功能）。如果内核已经加载了此设备的驱动程序（设备类中设备共用驱动），则可以正常地访问设备。

4 设备资源

通用驱动模型中由 devres 结构体表示设备资源，device 结构体双链表成员 devres_head 管理了设备拥

有的资源 devres 实例。设备资源管理数据结构组织关系如下图所示：



设备资源 devres 结构体定义在/drivers/base/devres.c 文件内：

```
struct devres {
    struct    devres_node     node;          /*devres_node 结构体成员*/
    unsigned long long        data[];        /*具体资源的数据结构*/
};

struct devres_node {
    struct list_head          entry;     /*添加到 device.devres_head 双链表*/
    dr_release_t              release;   /*释放资源函数指针，/include/linux/device.h*/
#ifdef CONFIG_DEBUG_DEVRES
    ...
#endif
};
```

devres 结构体是一个通用的数据结构，可表示设备的各种资源，如中断等。data 成员一个占位符，实际是表示具体资源的数据结构，如表示中断资源的 irq_devres 结构体。

devres_node 结构体中 entry 成员用于添加到设备的资源双链表，release 为释放资源的函数指针，函数原型定义如下：

```
typedef void (*dr_release_t)(struct device *dev, void *res);
typedef int (*dr_match_t)(struct device *dev, void *res, void *match_data);   /*查找资源时的匹配函数*/
```

分配设备资源 devres 实例的接口函数定义如下（drivers/base/devres.c）：

```
void * devres_alloc(dr_release_t release, size_t size, gfp_t gfp)
/*release：释放资源的函数指针，size：资源数据结构大小，gfp：分配掩码*/
{
    struct devres *dr;

    dr = alloc_dr(release, size, gfp | __GFP_ZERO);
                        /*分配 devres 实例，后接 size 大小的表示具体资源结构体大小*/
    if (unlikely(!dr))
        return NULL;
    return dr->data;      /*返回表示具体资源的数据结构指针*/
}
```

devres_alloc()函数参数 release 表示释放资源的函数指针（函数需由具体资源的代码实现），size 是具体资源数据结构大小。

alloc_dr()函数用于分配 devres 结构体实例，其后紧接具体资源数据结构实例。devres_alloc()函数返回具体资源数据结构实例指针，即 dr->data。

分配资源 devres 实例后，需要调用接口函数 **devres_add(struct device *dev, void *res)**将其添加到设备 device 实例资源双链表的末尾，见/drivers/base/devres.c 文件。

内核在/drivers/base/devres.c 文件内还提供了管理（操作）设备资源的接口函数，例如：

●void * devm_kmalloc(struct device *dev, size_t size, gfp_t gfp)：为设备分配并添加内存资源，大小为 size，返回内存指针。设备解绑驱动时会自动释放内存资源。

●void * devres_find(struct device *dev, dr_release_t release,dr_match_t match, void *match_data)：查找设备资源，返回资源指针，若不存在返回 NULL。

●void * devres_get(struct device *dev, void *new_res,dr_match_t match, void *match_data)：获取设备资源，若不存在则创建，返回资源指针。

●int devres_release(struct device *dev, dr_release_t release,dr_match_t match, void *match_data)：查找并释放资源 devres，match()为在设备资源双链表中查找资源的匹配函数（match_data 为此函数参数），此参数可选。

●int devres_release_all(struct device *dev)：释放设备持有的所有资源。

8.4.3 驱动

驱动可以认为是驱动程序和设备的管理者。驱动由 device_driver 结构体表示，当设备与驱动匹配时，将会调用 device_driver 实例中的 probe()函数完成设备驱动程序的加载。device_driver 结构体还定义了设备移出、睡眠、唤醒时的回调函数。device_driver 实例的管理比 device 实例要简单一些，device_driver 实例只挂接到总线 bus_type 的驱动链表下。

在内核启动或加载模块时，将向内核注册 device_driver 实例，注册时将驱动添加到总线上的驱动链表，并在总线上查找匹配的设备，匹配成功则加载设备驱动程序。

1 数据结构

驱动 device_driver 结构体定义在/include/linux/device.h 头文件：

```
struct device_driver {
    const char          *name;      /*驱动名称，匹配设备时，可能与设备名称进行匹配*/
    struct bus_type     *bus;       /*所属总线*/

    struct module       *owner;     /*模块指针*/
    const char          *mod_name;  /*模块名称*/

    bool    suppress_bind_attrs;        /*禁止通过 sysfs 文件系统绑定/解绑*/
    enum    probe_type  probe_type;     /*探测设备的方式，同步或异步*/

    const struct of_device_id       *of_match_table;    /*匹配列表，用于匹配设备树中设备节点*/
    const struct acpi_device_id     *acpi_match_table;  /*ACPI 匹配列表*/

    int (*probe)    (struct device *dev);           /*探测设备函数（加载设备驱动程序）*/
    int (*remove)   (struct device *dev);           /*解绑设备时调用*/
    void (*shutdown)    (struct device *dev);       /*关机时用于退出设备*/
    int (*suspend)  (struct device *dev, pm_message_t state);   /*使设备进入睡眠时调用（低功耗）*/
```

```
    int (*resume) (struct device *dev);                 /*从睡眠模式唤醒设备时调用*/
    const struct attribute_group **groups;              /*驱动属性*/

    const struct dev_pm_ops *pm;        /*电源管理操作结构指针，/include/linux/pm.h*/

    struct driver_private *p;                /*驱动私有数据结构指针*/
};
```

device_driver 结构体主要成员简介如下：

●**name**：驱动名称，在匹配设备时，通常匹配操作最后通过此名称与设备名称比对（是否相同）。

●**of_match_table**：指向 of_device_id 结构体数组，用于匹配设备树中的设备节点。of_device_id 结构体定义如下（/include/linux/mod_devicetable.h）：

```
struct of_device_id {
    char name[32];
    char type[32];
    char compatible[128];          /*兼容属性*/
    const void *data;
};
```

●acpi_match_table：指向 acpi_device_id 结构体，用于 ACPI 匹配列表。

●**pm**：指向设备电源管理 dev_pm_ops 结构体，定义如下（/include/linux/pm.h）：

```
struct dev_pm_ops {
    int (*prepare)(struct device *dev);
    void (*complete)(struct device *dev);
    int (*suspend)(struct device *dev);
    int (*resume)(struct device *dev);
    int (*freeze)(struct device *dev);
    int (*thaw)(struct device *dev);
    int (*poweroff)(struct device *dev);
    int (*restore)(struct device *dev);
    int (*suspend_late)(struct device *dev);
    int (*resume_early)(struct device *dev);
    int (*freeze_late)(struct device *dev);
    int (*thaw_early)(struct device *dev);
    int (*poweroff_late)(struct device *dev);
    int (*restore_early)(struct device *dev);
    int (*suspend_noirq)(struct device *dev);
    int (*resume_noirq)(struct device *dev);
    int (*freeze_noirq)(struct device *dev);
    int (*thaw_noirq)(struct device *dev);
    int (*poweroff_noirq)(struct device *dev);
    int (*restore_noirq)(struct device *dev);
    int (*runtime_suspend)(struct device *dev);
    int (*runtime_resume)(struct device *dev);
    int (*runtime_idle)(struct device *dev);
```

};

●**prob()**：探测函数指针，非常重要的函数，在设备与驱动匹配上时将会调用此函数加载设备驱动程序。

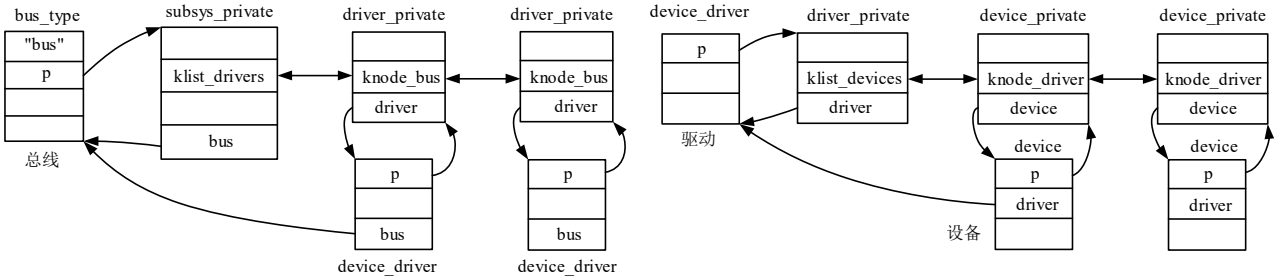
●**probe_type**：probe_type 枚举类型成员，列举了驱动注册时探测设备的方式，枚举值定义如下：

```
enum probe_type {
    PROBE_DEFAULT_STRATEGY,          /*创建设备时的默认值*/
    PROBE_PREFER_ASYNCHRONOUS,
    PROBE_FORCE_SYNCHRONOUS,
};
```

●**p**：驱动私有数据结构 driver_private 指针，结构体定义在/drivers/base/base.h 头文件：

```
struct driver_private {
    struct kobject    kobj;              /*跟踪 device_driver 实例的 kobject 实例*/
    struct klist    klist_devices;       /*链接可驱动设备的 device 实例*/
    struct klist_node    knode_bus;      /*将实例添加到总线的驱动链表*/
    struct module_kobject    *mkobj;
    struct device_driver    *driver;      /*指向 device_driver 实例*/
};
```

device_driver 实例的管理比 device 实例要简单，它只添加到总线的驱动链表中，如下图所示：



2 注册驱动

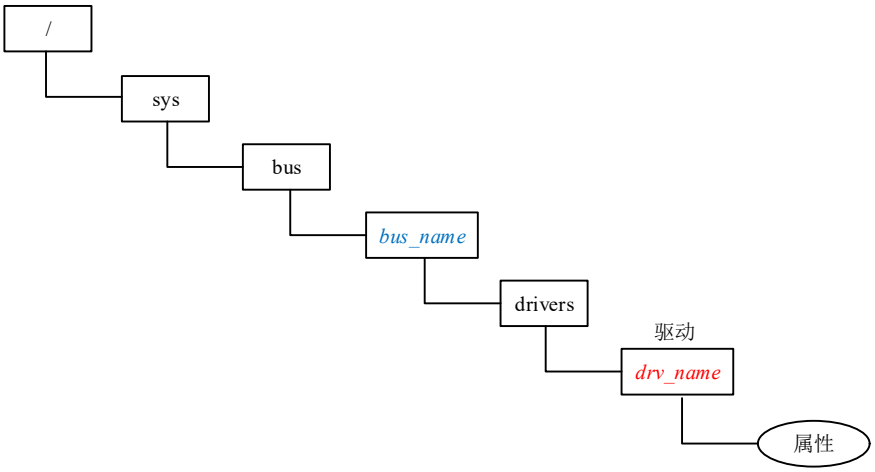
注册驱动与注册设备相比要简单许多，主要工作如下：

（1）将 device_driver 实例添加到总线的驱动链表，在总线对应目录项的 drivers 目录项下创建驱动对应的目录项。

（2）向驱动对应的目录项下添加驱动属性文件。

（3）匹配设备，匹配上则绑定设备到驱动，加载驱动程序。

注册 device_driver 实例在 sysfs 中创建的目录项和文件如下图所示：



注册驱动的 **driver_register**(struct device_driver *drv)函数定义在/drivers/base/driver.c 文件内：

```
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))
        ...       /*输出警告信息，总线中定义的函数有优先权*/

    other = driver_find(drv->name, drv->bus);
                /*在总线中以驱动名称查找同名驱动是否存在，存在返回驱动指针，否则返回 NULL*、
    if (other) {
        printk(KERN_ERR "Error: Driver '%s' is already registered, ""aborting...\n", drv->name);
        return -EBUSY;                  /*已经存在同名驱动，返回错误码*/
    }

    ret = bus_add_driver(drv);   /*将驱动添加到总线，/drivers/base/bus.c*/
    ...
    ret = driver_add_groups(drv, drv->groups);    /*向驱动添加属性（数组），/drivers/base/driver.c*/
    ...
    kobject_uevent(&drv->p->kobj, KOBJ_ADD);    /*最后触发 uevent 事件*/

    return ret;
}
```

注册函数首先在总线驱动链表中查找是否已经存在同名驱动，如果已经存在则返回错误码，否则调用函数 **bus_add_driver(drv)**向总线添加驱动，然后添加驱动中定义的属性组，最后调用 kobject_uevent()函数向用户空间发送 uevent 事件。

下面看一下向总线添加驱动的 bus_add_driver()函数实现，代码如下（/drivers/base/bus.c）：

```
int bus_add_driver(struct device_driver *drv)
{
    struct bus_type *bus;
    struct driver_private *priv;
    int error = 0;

    bus = bus_get(drv->bus);      /*获取驱动所属总线实例，增加总线引用计数*/
    ...
    priv = kzalloc(sizeof(*priv), GFP_KERNEL);    /*分配 driver_private 实例*/
    ...
    klist_init(&priv->klist_devices, NULL, NULL);    /*初始化设备链表头*/
    priv->driver = drv;
```

```
    drv->p = priv;
    priv->kobj.kset = bus->p->drivers_kset;          /*kobjedct 集合，父目录项*/
    error = kobject_init_and_add(&priv->kobj, &driver_ktype, NULL, "%s", drv->name);
                                                     /*添加 priv->kobj，kobject 类型为 driver_ktype*/
    ...

    klist_add_tail(&priv->knode_bus, &bus->p->klist_drivers);
                                                     /*driver_private 添加到总线驱动链表末尾*/
    if (drv->bus->p->drivers_autoprobe) {            /*允许自动探测设备，总线默认允许*/
        if (driver_allows_async_probing(drv)) {  /*判断是否异步探测，/drivers/base/dd.c*/
            pr_debug("bus: '%s': probing driver %s asynchronously\n",drv->bus->name, drv->name);
            async_schedule(driver_attach_async, drv);  /*异步绑定设备，/kernel/async.c*/
        }
        else {
            error = driver_attach(drv);  /*探测匹配设备，/drivers/base/dd.c*/
            ...
        }
    }
    module_add_driver(drv->owner, drv);

    error = driver_create_file(drv, &driver_attr_uevent);  /*添加 uevent 属性*/
    ...
    error = driver_add_groups(drv, bus->drv_groups);  /*添加总线默认的驱动属性*/
    ...
    if (!drv->suppress_bind_attrs) {
        error = add_bind_files(drv);    /*添加 unbind、bind 属性，/drivers/base/bus.c*/
        ...
    }
    return 0;
    ...
}
```

bus_add_driver()函数主要工作是为驱动创建 driver_private 实例并初始化，将 driver_private 添加到总线的驱动链表中，并添加 priv->kobj 实例，在总线 drivers 目录项下创建驱动的目录项；向驱动目录项下添加驱动属性文件；探测匹配的设备。

■驱动属性

驱动属性由 driver_attribute 结构体表示，定义如下（/include/linux/device.h）：

```
struct driver_attribute {
    struct attribute attr;              /*通用属性*/
    ssize_t    (*show) (struct device_driver *driver, char *buf);       /*读特定属性函数*/
    ssize_t    (*store) (struct device_driver *driver, const char *buf,size_t count);    /*写特定属性函数*/
};
```

在同一头文件内还给出了定义驱动属性的宏，例如：

```
#define    DRIVER_ATTR(_name, _mode, _show, _store) \
    struct driver_attribute driver_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define    DRIVER_ATTR_RW(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RW(_name)
```

在前面介绍的添加 priv->kobj 实例函数中，实例 kobj->ktype 赋值为 **driver_ktype** 实例指针，其中包含通用读写驱动属性文件的函数。

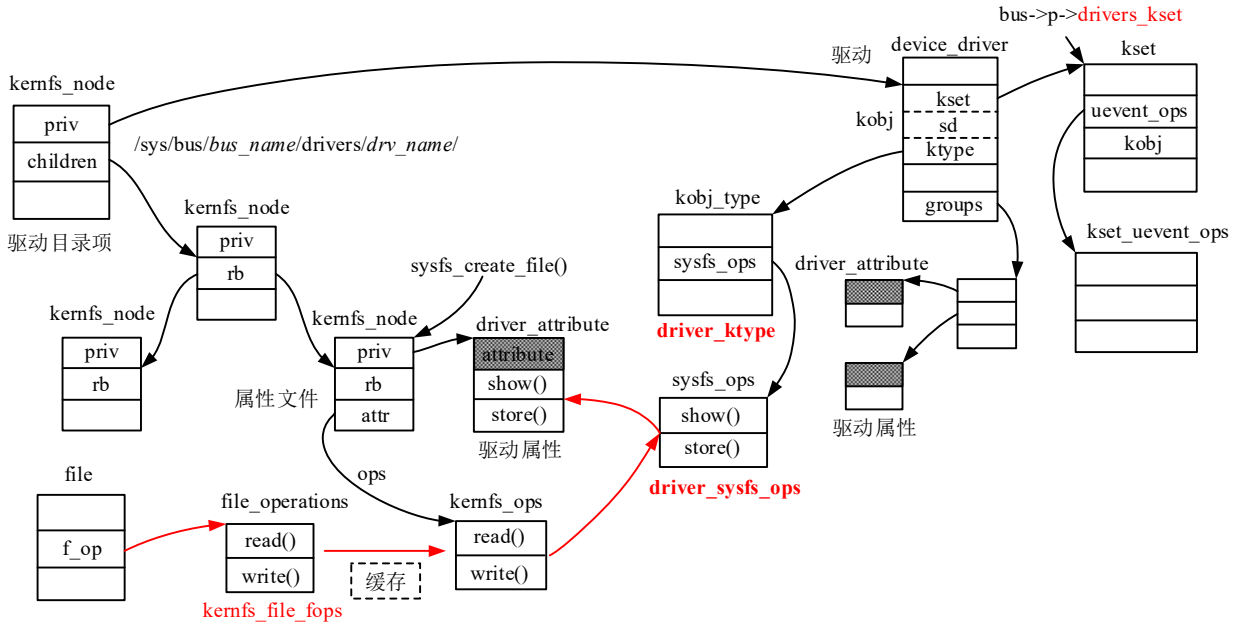
driver_ktype 实例定义在/drivers/base/bus.c 文件内：

```
static struct kobj_type driver_ktype = {
    .sysfs_ops      = &driver_sysfs_ops,
    .release    = driver_release,
};
```

driver_sysfs_ops 实例定义如下：

```
static const struct sysfs_ops driver_sysfs_ops = {
    .show     = drv_attr_show,        /*读驱动属性函数*/
    .store    = drv_attr_store,       /*写驱动属性函数*/
};
```

通用驱动读写函数与之前介绍的属性读写函数相似，最终调用具体 driver_attribute 实例内定义的读写函数读写属性，如下图所示。



■匹配设备

在向总线添加驱动的 bus_add_driver()函数中，将扫描总线下的设备链表，查找匹配的设备，匹配成功则将设备绑定驱动，加载驱动程序，此项工作通常由 **driver_attach(drv)**函数完成。

driver_attach()函数定义如下（/drivers/base/dd.c）：

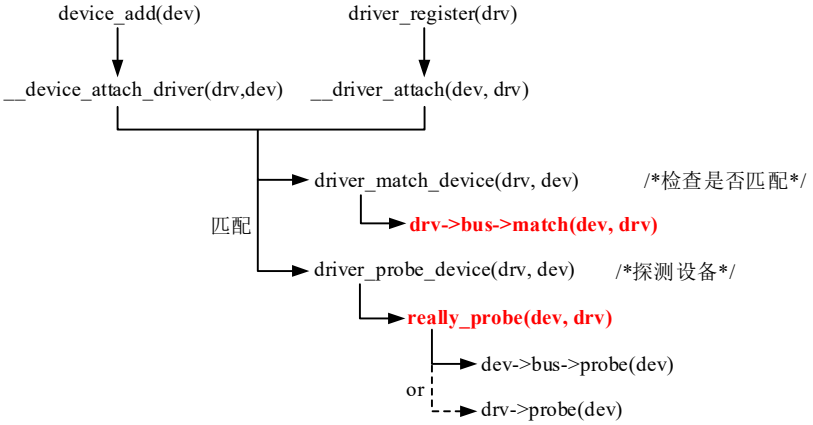
```
int driver_attach(struct device_driver *drv)
{
```

```
    return bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);    /*/drivers/base/bus.c*/
}
```

bus_for_each_dev()函数扫描总线中的设备链表，对每个设备调用__driver_attach(dev, data)函数判断设备是否与驱动匹配，若匹配则调用探测函数向内核注册设备驱动程序，这与注册设备时扫描驱动链表类似。下一小节将专门介绍设备与驱动的匹配。

8.4.4 匹配与探测

在注册设备时将扫描总线的驱动链表并调用__device_attach_driver(drv,dev)函数判断当前驱动与设备是否匹配；在注册驱动时，将扫描总线中的设备链表，对每个设备调用__driver_attach(dev, drv)函数，检查设备与驱动是否匹配，函数调用关系如下图所示。



匹配与探测主要分两步，一是调用总线的 match()函数检查设备与驱动是否匹配，若匹配则进行第二步，二是调用总线或驱动中定义的 probe()函数，加载设备驱动程序。

__device_attach_driver(drv,dev)和__driver_attach(dev, drv)函数都定义在/drivers/base/dd.c 文件内，代码如下：

```
static int __device_attach_driver(struct device_driver *drv, void *_data)
{
    struct device_attach_data *data = _data;
    struct device *dev = data->dev;          /*设备 device 实例*/
    bool async_allowed;

    if (dev->driver)          /*设备已经关联了驱动，返回错误码*/
        return -EBUSY;

    if (!driver_match_device(drv, dev))    /*判断设备与驱动是否匹配，/drivers/base/base.h*/
        return 0;          /*不匹配返回 0*/

    async_allowed = driver_allows_async_probing(drv);

    if (async_allowed)
        data->have_async = true;

    if (data->check_async && async_allowed != data->want_async)
        return 0;
```

```
        return driver_probe_device(drv, dev);     /*调用总线或驱动 probe()函数，/drivers/base/dd.c*/
}

static int __driver_attach(struct device *dev, void *data)
{
        struct device_driver *drv = data;

        if (!driver_match_device(drv, dev))       /*判断设备与驱动是否匹配，/drivers/base/base.h*/
                return 0;                         /*不匹配返回 0*/

        if (dev->parent)      /* Needed for USB */
                device_lock(dev->parent);
        device_lock(dev);
        if (!dev->driver)
                driver_probe_device(drv, dev);    /*调用总线或驱动 probe()函数，/drivers/base/dd.c*/
        device_unlock(dev);
        if (dev->parent)
                device_unlock(dev->parent);

        return 0;
}
```

1 匹配函数

driver_match_device(drv, dev)函数定义在/drivers/base/base.h 头文件内，用于检查驱动与设备是否匹配：

```
static inline int driver_match_device(struct device_driver *drv, struct device *dev)
{
        return drv->bus->match ? drv->bus->match(dev, drv) : 1;       /*调用总线匹配函数，否则返回 1*/
}
```

如果总线定义了 match()函数，则调用它检查是否匹配，否则返回 1（默认匹配）。函数返回非零值表示匹配，返回 0 表示不匹配。

2 探测函数

若驱动与设备匹配，将继续调用 driver_probe_device(drv, dev)函数绑定设备与驱动，函数内调用总线或驱动的 probe()函数。

driver_probe_device()函数定义如下（/drivers/base/dd.c）：

```
int driver_probe_device(struct device_driver *drv, struct device *dev)
{
        int ret = 0;

        if (!device_is_registered(dev))   /*device 是否已注册（导出到 sysfs），/include/linux/device.h*/
                return -ENODEV;
```

```
    ...     /*输出信息*/

    pm_runtime_barrier(dev);
    ret = really_probe(dev, drv);          /*/drivers/base/dd.c*/
    pm_request_idle(dev);

    return ret;
}
```

really_probe(dev, drv)函数定义如下：

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
    int ret = 0;
    int local_trigger_count = atomic_read(&deferred_trigger_count);

    atomic_inc(&probe_count);
    ...             /*输出信息*/
    WARN_ON(!list_empty(&dev->devres_head));

    dev->driver = drv;          /*关联驱动*/

    ret = pinctrl_bind_pins(dev);       /*pinctrl 子系统，/drivers/base/pinctrl.c*/
    if (ret)
        goto probe_failed;

    if (driver_sysfs_add(dev)) {          /*创建符号链接等，/drivers/base/dd.c**/
        ...         /*输出信息*/
        goto probe_failed;
    }

    if (dev->pm_domain && dev->pm_domain->activate) {
        ret = dev->pm_domain->activate(dev);       /*pm_domain 中激活函数*/
        ...
    }

    if (dev->bus->probe) {          /*优先调用总线的 probe()函数*/
        ret = dev->bus->probe(dev);
        ...
    } else if (drv->probe) {        /*总线没有定义 probe()函数，则调用驱动的 probe()函数*/
        ret = drv->probe(dev);
        ...
    }
```

```
    if (dev->pm_domain && dev->pm_domain->sync)
        dev->pm_domain->sync(dev);        /*调用 pm_domain 中同步函数*/

    driver_bound(dev);    /*绑定设备与驱动，设备添加到驱动中链表等，/drivers/base/dd.c*/
    ret = 1;
    ...         /*输出信息*/
    goto done;

probe_failed:
    ...      /*输出信息*/
done:
    atomic_dec(&probe_count);
    wake_up(&probe_waitqueue);          /*唤醒在 probe_waitqueue 队列睡眠等待进程*/
    return ret;
}
```

really_probe()函数中将关联设备与驱动，如果总线 bus_type 实例定义了 probe()函数，则调用总线定义的 probe()函数，否则调用驱动定义的 probe()函数。在 probe()函数中，将完成设备驱动程序的加载，即设备驱动数据结构（cdev 或 gendisk）实例的定义和注册。

really_probe()函数中调用 driver_sysfs_add()函数创建驱动与设备在 sysfs 中的符号链接，函数定义如下：

```
static int driver_sysfs_add(struct device *dev)
{
    int ret;

    if (dev->bus)          /*执行通知链*/
        blocking_notifier_call_chain(&dev->bus->p->bus_notifier,BUS_NOTIFY_BIND_DRIVER, dev);

    ret = sysfs_create_link(&dev->driver->p->kobj, &dev->kobj,kobject_name(&dev->kobj));
                                    /*在驱动目录项下创建到设备的符号链接，名称为设备名称*/
    if (ret == 0) {
        ret = sysfs_create_link(&dev->kobj, &dev->driver->p->kobj,"driver");
                                    /*在设备目录项下创建到驱动的符号链接，名称为“driver”*/
        if (ret)
            sysfs_remove_link(&dev->driver->p->kobj,kobject_name(&dev->kobj));
    }
    return ret;
}
```

设备 device 结构体和驱动 device_driver 结构体通常嵌入到具体总线定义的设备和驱动数据结构中。设备（device 实例）可以在平台（板级）文件中定义，并在初始化函数中向内核注册。驱动在设备驱动程序文件中定义，在初始化函数或加载模块时，调用注册函数向内核注册。

在平台（板级）代码中定义并注册设备实例，会产生很多“垃圾”代码，并且每次修改都需要重新编译内核，更现代的方法是通过设备树向内核传递设备硬件信息，详见下节。

8.5 设备树

在通用驱动模型中，每个外部设备需要由 device 结构体（或封装它的数据结构）表示，并向内核注册。这些代码位于体系结构相关的平台（板级）文件中（例如：/arch/mips/loongson32/common/platform.c），另外还需要定义描述硬件信息的数据结构实例，并关联到表示设备的数据结构中，以便驱动程序获取硬件信息，如控制寄存器地址、中断号等。

由于 Linux 内核支持的体系结构、平台众多，以及外部设备类型的繁杂，导致在内核中描述硬件信息的代码数量众多，而这些代码对内核来说是没有什么技术含量的“垃圾”代码，为此内核采用了设备树，将描述硬件信息的代码从内核代码中分离出来。

设备树是用于描述系统硬件信息的文件，分为源文件和目标文件。源文件（DTS，.dts）相当于 C 语言的源文件，它用规定的语法格式描述系统硬件信息，通过设备树文件编译器（DTC）编译后生成设备树目标文件（DTB，.dtb），相当于 C 语言目标文件，这是个二进制文件。

设备树目标文件可以先编译好（不编入内核），保存在固件中，然后传递给内核，内核即可从设备树目标文件中获取硬件信息。设备树源文件也可以存放在内核源文件中，连同内核一起编译（内核提供设备树编译器），并将目标文件嵌入到内核可执行文件中。内核在启动阶段，将从设备树目标文件中获取设备信息，并在内核中用 device_node 实例（相当于硬件设备库）表示硬件信息，设备驱动程序可从此数据结构中获取指定硬件的信息，用其创建设备实例，并向内核注册。

设备树说明文档《Devicetree Specifification》可从 https://www.devicetree.org/specifications/网站下载。

8.5.1 设备树源文件

设备树源文件位于体系结构相关的/arch/mips/boot/bts/目录下，设备树源文件由设备节点组成，每个设备对应源文件中一个节点，设备节点组成父子关系的层次结构。设备节点由节点名称（设备名称）和属性组成。属性由形如“property=value”的属性名称和属性值组成，"property"为属性名称，"value"为属性值。属性包含设备兼容属性、寄存器属性、中断号属性等。

下面用一个简单的例子来演示设备树源文件的组成：

```
/{                                                          //根节点，名称为“/”
    compatible = "nvidia,harmony", "nvidia,tegra20";        //平台兼容属性
    #address-cells = <1>;
    #size-cells = <1>;
    interrupt-parent = <&intc>;      //中断控制器节点

    chosen { };       //chosen 节点，传递命令行参数等
    aliases { };      //别名节点

    memory {              //物理内存节点
        device_type = "memory";
        reg = <0x00000000 0x40000000>;      //物理内存起始地址和长度
    };

    soc {       //soc 节点开始
        compatible = "nvidia,tegra20-soc", "simple-bus";
        #address-cells = <1>;
        #size-cells = <1>;
```

```
        ranges;

        intc: interrupt-controller@50041000 {        //中断控制器，intc 为标号，等同节点地址
            compatible = "nvidia,tegra20-gic";
            interrupt-controller;              //标记为中断控制器
            #interrupt-cells = <1>;
            reg = <0x50041000 0x1000>, < 0x50040100 0x0100 >;     //控制寄存器起始地址和长度
        };

        serial@70006300 {       //串口设备节点
            compatible = "nvidia,tegra20-uart";
            reg = <0x70006300 0x100>;
            interrupts = <122>;         //中断号
        };

        i2s1: i2s@70002800 {
            compatible = "nvidia,tegra20-i2s";
            reg = <0x70002800 0x100>;
            interrupts = <77>;
            codec = <&wm8903>;
        };

        i2c@7000c000 {
            compatible = "nvidia,tegra20-i2c";
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <0x7000c000 0x100>;
            interrupts = <70>;

            wm8903: codec@1a {
                compatible = "wlf,wm8903";
                reg = <0x1a>;
                interrupts = <347>;
            };
        };     //i2c@7000c000 节点结束
    };   //soc 节点结束

    sound {       //sound 节点
        compatible = "nvidia,harmony-sound";
        i2s-controller = <&i2s1>;
        i2s-codec = <&wm8903>;
    };
};
```

设备树源文件中各节点组成父子层次结构，必须有一个根节点，名称为“/”，它是层次结构中最顶层节点。根节点下有子节点，如上所示，根节点下存在 chosen、aliases、memory、soc 等子节点，而 soc 节点下又存在子节点 interrupt-controller@50041000、i2c@7000c000 等。

节点内容是形如“property=value”的属性、值列表，表示硬件的具体信息。下文中将介绍设备树源文件中节点和属性相关的内容。

1 节点

设备树源文件中节点名称一般形式为：

node-name@unit-address

node-name 为设备名称，最长为 31 个字符，第一个字符需是字母，字符可以是数字和字母，以及有限的几个其它字符，详见说明文档。设备名称通常表示设备类型，如 cpu、memory、i2c 等。

unit-address 也由字符组成，表示设备在总线上的位置（地址），它必须与该节点中 reg 属性中第一个地址值相同，例如：

```
serial@70006300 {       //串口设备节点，70006300  前没有加 0x
        compatible = "nvidia,tegra20-uart";
        reg = <0x70006300 0x100>;       //起始地址为 0x70006300
        interrupts = <122>;         //中断号
};
```

如果节点属性中不存在 reg 属性，则节点名称中@unit-address 字符必须省略，只剩 node-name 字符。

推荐使用的节点名称如下（详见说明文档）：

- adc
- accelerometer
- atm
- audio-codec
- audio-controller
- backlight
- bluetooth
- bus
- cache-controller
- camera
- can
- charger
- clock
- clock-controller
- compact-flflash
- cpu
- cpus
- crypto
- disk

...

设备树源文件中的节点可以用路径来表示，路径由从根节点到本节点的节点名称组成，中间用‘/’字符隔开，类似文件路径。例如：上例中 i2c@7000c000 节点的路径为/soc/i2c@7000c000。

2 属性

节点下的内容是属性，用于描述硬件的物理信息，属性通常具有值，但是值也可以为空。属性名称与节点名称类似，最多也是由 31 个字符组成。属性值由 0 个或多个字节组成，4 个字节（32 位）表示一个单元（cell）。

常用的属性值有以下几种：

●**空**：不需要等号和属性值，只需要属性名称，通常用于 bool 量，属性名称存在代表 true，属性名称不存在代表 false。例如：interrupt-controller；（表示设备节点是中断控制器）

●**32 位整数**：整数值在目标文件中以大端模式存放，整数值放在< >尖括号内。例如：interrupts = <122> 表示设备中断编号为 122。

●**64 位整数**：与 32 位整数类似，也是大端存放。

●**字符串**：与 C 语言中字符串相似，以‘\0’字符结束，例如：compatible = "arm,cortex-a9"，字符串放在双引号内。

●**字符串列表**：属性值由多个字符串组成，例如：string-list = "red fish", "blue fish";

●**节点地址**：实际为 32 位（无符号）整数，表示节点在目标文件中的地址，用于节点引用。例如：i2s-controller = <&i2s1>;表示中断控制器节点的地址，i2s1 为中断控制器节点标号。在节点名称前面可以加标号（类似 C 语言中标号）表示节点的地址。

下面简要介绍几个常用标准属性及其属性值类型，更多的属性请读者参考说明文档：

●**compatible**：兼容属性，属性值为字符串或字符串列表，表示节点与设备驱动的匹配属性。属性值推荐格式为"manufacturer,model"，前者表示设备生产商，后者表示型号。例如：compatible = "fsl,mpc8641";

●**model**：model 属性与 compatible 属性类似，但其属性值只能是字符串，而不是字符串列表。

●**phandle**：句柄属性，属性值为节点编号，用于其它节点引用本节点，例如：phandle = <1>;

●**status**：状态属性，属性值为字符串，属性值取值："okay" 、"disabled" 、"reserved" 、"fail"、"fail-sss" 。

●**#address-cells, #size-cells**：#address-cells 表示 reg 属性值中地址值由多少个 cell 表示，如果地址是 32 位，则属性值为 1。#size-cells 表示 reg 属性值中长度值由多少个 cell 表示，如果长度 32 位值，则属性值为 1。这里表示的 reg 属性是指当前节点的子节点中的属性，而不是本节点 reg 属性（本节点的在父节点中指定）。

●**reg**：地址区间属性，属性值为<address length address length...>，每个 address length 对表示一个地址区间，address 表示起始地址，length 表示长度。前面介绍的#address-cells 和 #size-cells 属性表示 address 和 length 分别由多少个 cell（32 位）表示。例如：

```
#address-cells = <1>;     //32 位地址
#size-cells = <1>;        //32 位长度值

memory {                  //物理内存节点
    device_type = "memory";
    reg = <0x00000000 0x40000000>;      //物理内存起始地址和长度值都为 32 位
};
```

●**interrupts**：中断属性，属性值通常表示设备产生的中断编号，例如：

```
interrupts = <0xA>;
```

●**#interrupt-cells**：中断控制器 interrupt-controller 节点中的属性，表示描述中断域需要的 cell 数量。

●**interrupt-controller**：中断控制器 interrupt-controller 节点中的属性，属性值为空，存在此属性则表示节点为中断控制器。

3 常用节点及属性

下面简要介绍几个设备树源文件中常用的节点及其属性，更详细的信息请参考说明文档。

●根节点

根节点为‘/’，必须的属性如下：

#address-cells

#size-cells

model

compatible

●aliases

别名节点，通常是根节点下的子节点。节点内属性用于给其它节点取别名，例如：

```
aliases {
    serial0 = "/simple-bus@fe000000/serial@llc500";       //serial0 代表 serial@llc500 节点
    ethernet0 = "/simple-bus@fe000000/ethernet@31c000";
};
```

●memory

物理内存节点，表示物理内存起始地址及长度。例如：

```
#address-cells = <2> ;       //起始地址为 64 位
#size-cells = <2>;           //长度值为 64 位

memory@0 {
    reg = <0x000000000 0x00000000 0x00000000 0x80000000     //64 位，高位在前
           0x000000001 0x00000000 0x00000001 0x00000000>;
};
```

上面表示的物理内存包含两段分别是：起始地址 0x0 长度 0x0x80000000（2GB）、起始地址 0x1 00000000，长度 0x1 00000000 (4 GB)。

●chosen

chosen 节点并不表示设备，其属性值表示命令行参数等信息，通常是根节点下的子节点。例如：

```
chosen {
    bootargs = "root=/dev/nfs rw nfsroot=192.168.1.1 console=ttyS0,115200";     //命令行参数
};
```

●cpus/cpu

cpus 表示系统中 CPU 核的共用信息，其子节点 cpu 表示某一个 CPU 核的信息。cpus 节点常用属性有：

```
#address-cells     //表示子节点 cpu 中 reg 属性起始地址
#size-cells        //属性值通常为 0
```

cpu 节点为 cpus 节点的子节点，表示 CPU 核的物理信息，常用属性请参考设备树说明文档。下面看一个简单的例子：

```
cpus {        //cpus 节点下存在一个 cpu 子节点
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
        reg = <0>;              //提供给线程的 CPU 核编号
        d-cache-block-size = <32>;     // L1 数据缓存块大小 32 bytes
        i-cache-block-size = <32>;     // L1 指令缓存块大小 32 bytes
        d-cache-size = <0x8000>;     // L1 数据缓存大小 32K
        i-cache-size = <0x8000>;     // L1 指令缓存块大小 32K
        timebase-frequency = <82500000>;   //  时钟计数器频率 82.5 MHz
        clock-frequency = <825000000>;     // CPU 核时钟频率 825 MHz
    };
};
```

另外 cpu 节点中还需要包含 TLB 信息等，详情请参考说明文档。

设备树源文件中需要注明源文件采用的 DTS 版本号，否则默认为版本 0。源文件也可以像 C 源文件一样通过头文件引入其它文件，例如：

```
/dts-v1/;        //采用版本 1
/include/ "bcm6328.dtsi"     //bcm6328.dtsi 必须与当前文件在同一目录下
```

内核在/Documentation/devicetree/bindings/目录下保存了设备节点的说明文档，例如属性值含义等。如果在设备树中增加新的设备节点名称，需要在此目录下添加说明文档。

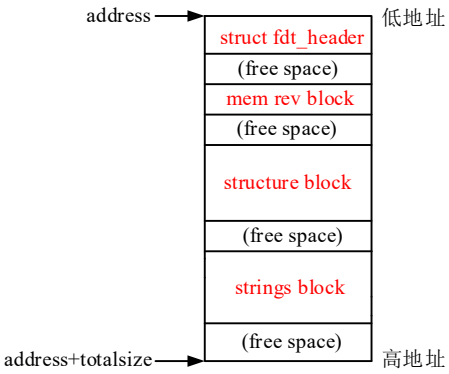
8.5.2 设备树目标文件

设备树源文件需经过设备树编译器 DTC 编译后生成具有规定格式的目标文件，后缀为.dtb，相当于 C 编译器输出的目标文件。本小节简要介绍设备树目标文件的结构以及如何将目标文件传递给内核。

1 目标文件格式

设备树目标文件格式如下图所示，目标文件主要包含以下部分：设备树目标文件头数据结构，节点结构区域，属性字符串区域，保留内存区域等。

free space 是为满足对齐要求空闲的区域。目标文件加载到物理内存中时，文件开头位于低地址。



设备树目标文件格式

●fdt_header:

目标文件头是一个 fdt_header 结构体实例，结构体中成员都是 32 位整数（大端存放），结构体定义如下：

```
struct fdt_header {
    uint32_t  magic;     /*魔数，用于判定是否是设备树目标文件，0xd00dfeed*/
    uint32_t  totalsize;       /*目标文件总大小，字节数*/
    uint32_t  off_dt_struct;    /*structure block 相对于文件头的偏移量，字节数*/
    uint32_t  off_dt_strings;    /*strings block 相对文件头的偏移量，字节数*/
    uint32_t  off_mem_rsvmap;    /*memory reservation block 相对文件头的偏移量，字节数*/
    uint32_t  version;         /*目标文件版本号*/
    uint32_t  last_comp_version;    /*最低兼容的文件版本号*/
    uint32_t  boot_cpuid_phys;      /*启动 CPU 核 ID 号*/
    uint32_t  size_dt_strings;       /*structure block 部分大小，字节数*/
    uint32_t  size_dt_struct;    /*strings block 部分大小，字节数*/
};
```

●memory reservation block:

memory reservation block 区域保存的是 fdt_reserve_entry 结构体实例数组，结构体定义如下：

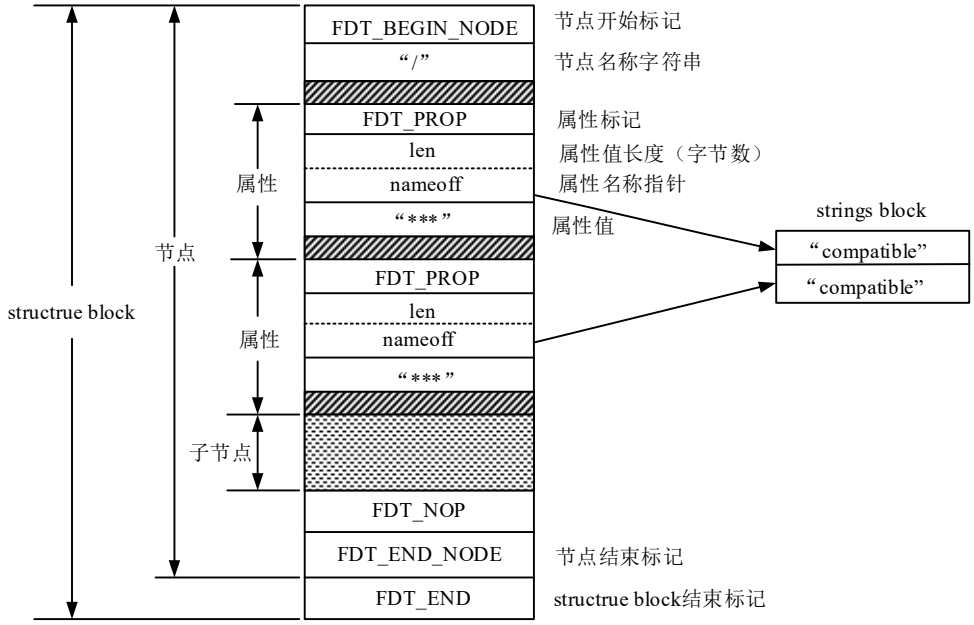
```
struct fdt_reserve_entry {
    fdt64_t address;    /*64 位地址，大端存放*/
    fdt64_t size;       /*64 位长度，大端存放*/
};
```

fdt_reserve_entry 结构体表示一段物理内存，起始地址和长度值都是 64 位，即使在 32 位系统中也一样，在 32 位系统中高 32 位被忽略。数组的最末尾项 address 和 size 成员值都为 0。

数组项中表示的物理内存段是保留的物理内存，只能由引导加载程序访问，内核等用户程序不能访问。保留物理内存段在 memory 节点中设置。

●structure block：

structure block 部分描述了设备树中节点信息，其结构如下图所示：



每个节点由以下部分组成（斜线阴影部分表示为满足对齐要求预留的空间）：

●FDT_BEGIN_NODE：节点开始标记，值为 0x00000001（大端存放），后跟节点名称字符串，为满足对齐要求，可能预留有空间。

●FDT_PROP：属性标记，值为 0x00000003，表示一个属性的开始，后接以下结构体实例：

```
struct {
   uint32_t len;            /*属性值的长度，字节数*/
   uint32_t nameoff;     /*属性名称指针，属性名称保存在 strings block 区域*/
}
```

在结构体实例之后保存的是属性值。

●FDT_NOP：空标记（可选），取值为 0x00000004，没有实际的意义，表示删除节点。

●FDT_END_NODE：节点结束标记，取值 0x00000002。

子节点与父节点的属性并列，子节点结构也如上所述，也就是说子节点是嵌套在父节点内部的。节点在 structure block 中是层层嵌套的。

structure block 区域最后是 FDT_END (0x00000009)标记，标记 structure block 区域结束。

●strings block:

strings block 区域保存的是设备树所有节点属性名称字符串，此区域没有对齐要求。

内核在/scripts/dtc/libfdt/目录下的文件内定义了设备树目标文件底层的操作接口函数，在/drivers/of/fdt.c 文件内定义了更高层次的操作接口函数。例如：

●void *__init **of_get_flat_dt_prop**(unsigned long node, const char *name,int *size)：node 表示目标文件加载到内存后节点地址，name 表示属性名称字符串，函数返回属性值指针，函数调用后 size 保存属性值长度，字节数。

●nt __init of_flat_dt_is_compatible(unsigned long node, const char *compat)：node 指向节点是否匹配参数 compat 指向的兼容属性。

2 链接目标文件

如果内核需要支持设备树，需要选择 OF 配置选项（/drivers/of/Kconfig），这个选项是在体系结构（板级）相关的配置文件中直接选择，而不是通过配置界面选择。例如，MIPS 体系结构在/arch/mips/Kconfig 配置文件中定义了设备树相关的配置选项：

```
config USE_OF     /*支持设备树*/
    bool
    select OF
    select OF_EARLY_FLATTREE     /*早期初始化调用*/
    select IRQ_DOMAIN

config BUILTIN_DTB    /*使用/arch/mips/boot/dts/目录下设备树源文件，目标文件嵌入内核镜像文件*/
    bool
```

如果平台需要使用/arch/mips/boot/dts/目录下的设备树源文件，则需要选择 USE_OF 和 BUILTIN_DTB 配置选项，例如（/arch/mips/Kconfig）：

```
config CAVIUM_OCTEON_SOC       /*平台类型选项*/
    ...
    select USE_OF
    ...
    select BUILTIN_DTB
    ...
```

如果选择了 BUILTIN_DTB 配置选项，内核会在**/arch/mips/boot/dts/**目录下根据配置选项选择设备树源文件进行编译（自带 DTC 编译器），编译生成的设备树目标文件嵌入到内核镜像文件初始化数据段中。

在体系结构无关的头文件/include/asm-generic/vmlinux.lds.h 中定义了设备树目标文件段：

```
#define KERNEL_DTB()                    \
    STRUCT_ALIGN();                     \
    VMLINUX_SYMBOL(__dtb_start) = .;    \      /*设备树目标文件起始地址*/
    *(.dtb.init.rodata)                 \
    VMLINUX_SYMBOL(__dtb_end) = .;            /*设备树目标文件结束地址*/

#define INIT_DATA                       \      /*初始化数据段*/
    *(.init.data)                       \
    ...
    KERNEL_DTB()                  \     /*设备树目标文件*/
    ...
```

另外也可以将设备树目标文件打包到内核镜像的末尾，需要选择相应的配置选项（/arch/mips/Kconfig）：

```
choice
    prompt "Kernel appended dtb support" if OF
    default MIPS_NO_APPENDED_DTB        /*不将设备树目标文件与内核目标文件打包*/
```

```
        config MIPS_NO_APPENDED_DTB
            bool "None"
            help
              Do not enable appended dtb support.

        config MIPS_RAW_APPENDED_DTB    /*设备树目标文件打包到 vmlinux.bin 内核目标文件末尾*/
            bool "vmlinux.bin"          /*没有压缩的目标文件*/
            help
              ...

        config MIPS_ZBOOT_APPENDED_DTB
                                                /*设备树目标文件打包到 vmlinuz.bin 内核目标文件末尾*/
            bool "vmlinuz.bin"                  /*压缩的目标文件*/
            depends on SYS_SUPPORTS_ZBOOT
            help
              ...
endchoice

endmenu
```

在链接文件/arch/mips/kernel/vmlinux.lds.S 中为设备树目标文件预留了空间：

```
SECTIONS
{
   ...
#ifdef CONFIG_MIPS_RAW_APPENDED_DTB       /*内核镜像的末尾*/
    __appended_dtb = .;          /*初始化完成后将释放此空间*/
    /*为 DTB 预留空间*/
    . += 0x100000;
#endif
    ...
    __init_end = .;

    BSS_SECTION(0, 0x10000, 0)

    _end = . ;
    ...
    }
}
```

如果设备树目标文件由引导加载程序（固件）加载并传递给内核，则引导加载程在将设备树目标文件加载到内存后，需要将设备树目标文件地址写入 a1 通用寄存器传递给内核。

如果在内核镜像末尾拼接上设备树目标文件，则此设备树目标文件具有优先权（不使用引导加载程序传入的目标文件），其地址将写入 a1 寄存器（head.S）。在 head.S 源文件中，通用寄存器 a1 值将保存至

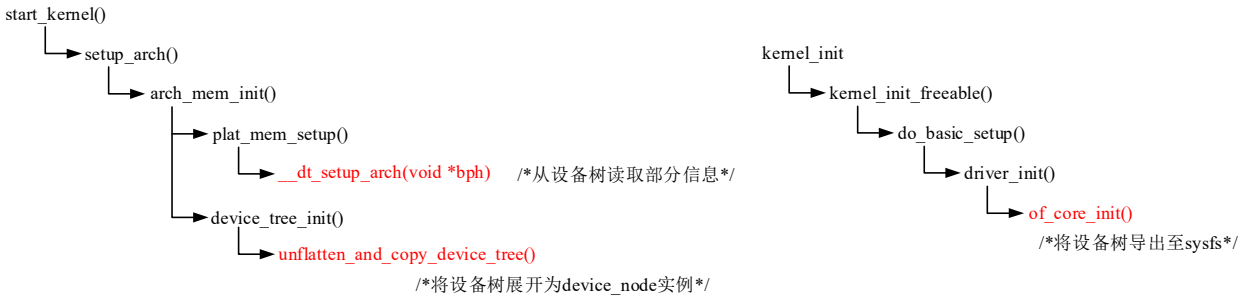
全局变量 fw_arg1，表示命令行参数或设备树目标文件地址，如果使用的是打包的设备树目标文件，a0 寄存器值将赋值-2。

内核是使用内嵌的设备树目标文件还是拼接在内核镜像末尾（或外部传入）的设备树目标文件以及优先顺序，由平台相关代码中的 plat_mem_setup()函数确定。

8.5.3 使用设备树

设备树目标文件可以由引导加载程序传入内核，也可以将设备树目标文件拼接在内核镜像末尾，也可以直接将目标文件嵌入到内核镜像初始化数据段中。

内核在启动阶段早期会从设备树目标文件中获取信息，例如：命令行参数等，随后将设备树目标文件中节点在内核展开，由 device_node 结构体实例表示设备节点，最后还需要将节点信息导出到 sysfs 文件系统，函数调用关系如下图所示。



__dt_setup_arch(void *bph)函数用于在早期读取设备树中信息至内核，例如：命令行参数等，通常将读取信息保存至全局变量。

unflatten_and_copy_device_tree()函数用于将设备树目标文件在内核中展开成 device_node 实例。

of_core_init()函数用于将设备树节点和属性信息导出至 sysfs 文件系统。

设备树在内核中最终由 device_node 结构体实例组成的层次结构表示，内核提供了从层次结构中查找节点、读取属性值等接口函数，驱动程序可调用这些接口函数查找并获取设备的物理信息。

1 设备树初始化

内核在初始化阶段的早期需要从设备树文件中读取信息至内核全局变量中，函数调用关系如下：

```
static void __init arch_mem_init(char **cmdline_p)
{
    ...
    plat_mem_setup();   /*读取设备树中部分信息*/
    ...
    device_tree_init();    /*设备树目标文件展开成 device_node 结构体实例，平台（板级）实现*/
    ...
}
```

驱动模型初始化函数中与设备树相关的函数如下：

```
void __init driver_init(void)
{
    ...
```

```
    of_core_init();          /*将设备树节点信息导出到/sys/fireware/devicetree/，/drivers/of/base.c*/
}
```

of_core_init()函数负责将设备树中节点信息导出到/sys/fireware/devicetree/目录下，节点表现为目录项，属性表现为文件，源代码请读者自行阅读。

■早期初始化

在平台（板级）实现的 **plat_mem_setup()**函数中需要调用__dt_setup_arch(void *bph)函数读取设备树中信息，参数 bph 表示设备树目标文件地址。参数 bph 值可以是 fw_arg1 或__dtb_start，以确定使用哪个设备树目标文件。

__dt_setup_arch(void *bph)函数定义如下（/arch/mips/kernel/prom.c 体系结构相关的函数）：

```
void __init __dt_setup_arch(void *bph)     /*bph 表示设备树目标文件地址*/
{
    if (!early_init_dt_scan(bph))    /*/drivers/of/fdt.c*/
        return;

    mips_set_machine_name(of_flat_dt_get_machine_name());   /*从设备树中读取机器名称*/
}
```

early_init_dt_scan()函数定义如下：

```
bool __init early_init_dt_scan(void *params)    /*params 为设备树目标文件地址*/
{
    bool status;

    status = early_init_dt_verify(params);
        /*设备树目标文件地址保存至全局变量 initial_boot_params，/drivers/of/fdt.c*/
    if (!status)
        return false;

    early_init_dt_scan_nodes();     /*读取设备树部分节点属性信息，/drivers/of/fdt.c*/
    return true;
}
```

early_init_dt_verify(params)函数检查设备树目标文件的有效性后，将设备树目标文件地址赋予全局变量 **initial_boot_params**，后面的函数中将使此变量访问设备树目标文件。

early_init_dt_scan_nodes()函数用于读取设备树中信息，函数定义如下（体系结构无关的函数）：

```
void __init early_init_dt_scan_nodes(void)
{
    /*从 /chosen 节点读取命令行参数复制到 boot_command_line 指向空间*/
    of_scan_flat_dt(early_init_dt_scan_chosen, boot_command_line);     /*/drivers/of/fdt.c*/

    /*读取根节点"#address-cells"和"#size-cells"属性值，保存至全局变量*/
    of_scan_flat_dt(early_init_dt_scan_root, NULL);        /*/drivers/of/fdt.c*/
```

```
    /*读取物理内存信息，调用 memblock_add(base, size)函数，memory 节点必须定义设备类型属性*/
    of_scan_flat_dt(early_init_dt_scan_memory, NULL);       /*/drivers/of/fdt.c*/
}
```

of_scan_flat_dt(int (*it)(),void *data)函数用于扫描设备树中所有节点，对每个节点都调用 **it()**函数。

■展开设备树

在体系结构相关的 arch_mem_init()函数中，在初始化自举分配器后将调用 **device_tree_init()**函数将设备树目标文件中节点导入到内核，在内核中由 device_node 结构体实例的层次结构表示设备节点。

下面先看一下 device_node 结构体的定义和实例的组织结构。

device_node 结构体定义在/include/linux/of.h 头文件内：

```
struct device_node {
    const char *name;     /*设备节点名称*/
    const char *type;       /*设备节点类型*/
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct      property *properties;     /*属性链表*/
    struct      property *deadprops;     /*删除的属性链表*/
    struct      device_node *parent;     /*父节点指针*/
    struct      device_node *child;       /*子节点指针*/
    struct      device_node *sibling;     /*兄弟节点指针*/
    struct      kobject kobj;       /*导出到 sysfs 文件系统中的 kobject 实例*/
    unsigned long _flags;
    void *data;
    ...
};
```

设备属性由 property 结构体表示，定义如下：

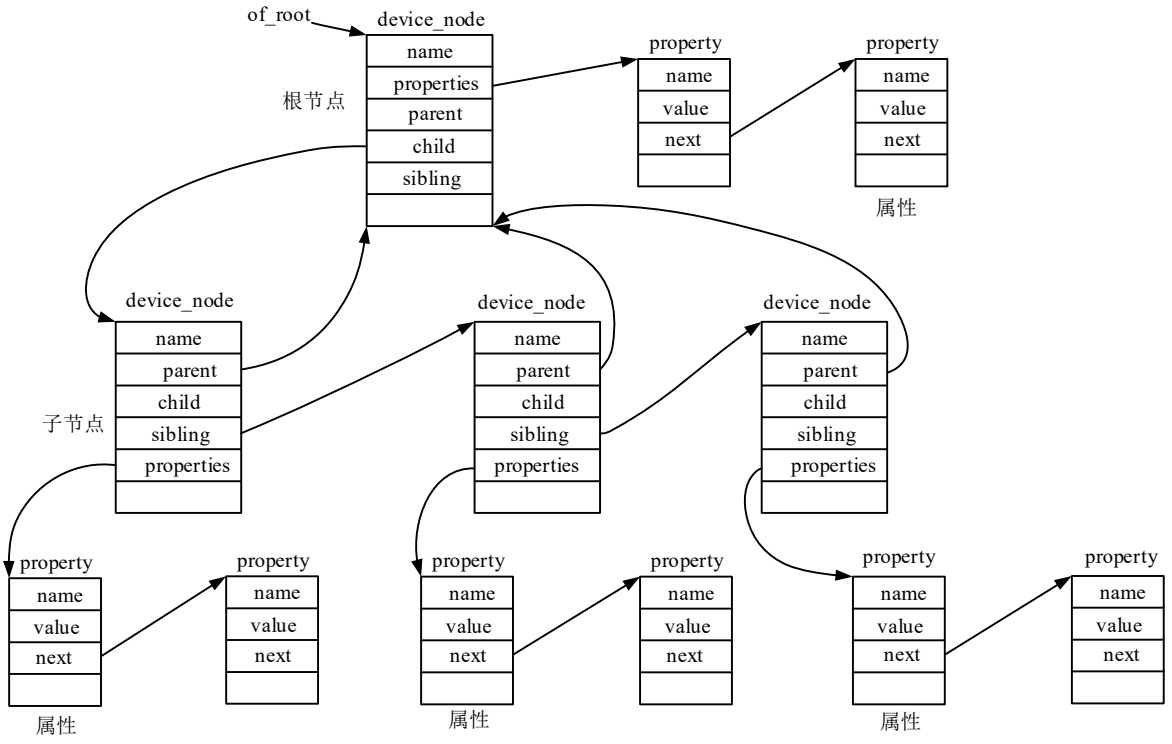
```
struct property {
    char *name;         /*属性名称*/
    int    length;        /*属性值长度*/
    void *value;         /*属性值长度*/
    struct property *next;     /*下一个属性，构成属性链表*/
    unsigned long _flags;
    unsigned int unique_id;
    struct bin_attribute attr;     /*属性导出到节点在 sysfs 文件系统目录下（导出为文件）*/
};
```

在体系结构相关的 device_tree_init()函数中通常调用 **unflatten_and_copy_device_tree()**函数将设备树目标文件导入内核，函数定义如下（/drivers/of/fdt.c）：

```
void __init unflatten_and_copy_device_tree(void)
```

```
{
    int size;
    void *dt;

    if (!initial_boot_params) {     /*initial_boot_params 保存设备树地址*/
        ...
    }

    size = fdt_totalsize(initial_boot_params);     /*设备树文件大小*/
    dt = early_init_dt_alloc_memory_arch(size,roundup_pow_of_two(FDT_V17_SIZE));
                                                                    /*为设备树分配空间*/
    if (dt) {
        memcpy(dt, initial_boot_params, size);     /*复制设备树目标文件*/
        initial_boot_params = dt;        /*新复制的设备树目标文件地址*/
    }
    unflatten_device_tree();   /*展开设备树*/
}
```

unflatten_and_copy_device_tree()函数先复制目标文件至内核空间，然后调用 unflatten_device_tree()函数将设备树目标文件中节点展开成 device_node 实例组成的层次结构，如下图所示：



2 设备树接口函数

基于 device_node 实例的层次结构，内核提供了从 device_node 实例中获取设备节点信息的接口函数，这些函数声明在/include/linux/of.h 头文件，定义在/drivers/of/base.c 文件内，驱动程序中可调用这些接口函数获取设备信息。例如：

●struct device_node ***of_find_node_by_name***(struct device_node *from,const char *name)：通过节点名称

查找 device_node 实例，name 表示名称，from 表示查找起点，为 NULL 表示从根节点开始查找。

●struct device_node ***of_find_compatible_node**(struct device_node *from,const char *type, const char *compatible)：按照兼容属性查找节点，compatible 指向兼容属性字符串。

●struct of_device_id ***of_match_node**(const struct of_device_id *matches,const struct device_node *node)：检查 node 节点与 of_device_id 列表各项的匹配性，返回匹配的 of_device_id 项指针。

●struct property ***of_find_property**(const struct device_node *np,const char *name,int *lenp)：查找指定节点的指定属性。

●int **of_property_read_string**(struct device_node *np, const char *propname,const char **out_string)：查找并读取指定节点指定字符串属性值，np 指定节点，propname 为属性名称，*out_string 指向属性值字符串。

●int **of_property_read_u32_array**(const struct device_node *np,const char *propname, u32*out_values,size_t sz)：读取 32 位属性值（数组），np 指定节点，propname 为属性名称，*out_values 指向属性值数组，sz 为数组项数（32 位整数数量）。

●int **of_property_read_u32_index**(const struct device_node *np,const char *propname,u32 index, u32 *out_value)：读取 index 指定的 32 位属性值。

8.6 平台总线

在了解了驱动模型框架之后，从本节开始介绍几种具体总线（驱动）的实现。

下面先介绍在嵌入式系统中常用的平台（platform）总线。平台总线是内核定义的虚拟总线，用于管理没有挂接到实际总线上（如：SPI、USB 总线）的设备，也可以将 platform 总线理解成处理器的地址/数据总线（芯片内总线）。

8.6.1 platform 总线

1 总线实例

platform 总线实例定义在/drivers/base/platform.c 文件：

```
struct bus_type   platform_bus_type = {
    .name       = "platform",                   /*总线名称为"platform"*/
    .dev_groups     = platform_dev_groups,      /*总线下设备默认属性*/
    .match          = platform_match,           /*设备与驱动匹配函数，见下文*/
    .uevent         = platform_uevent,          /*向 uevent 事件添加环境变量的函数*/
    .pm         = &platform_dev_pm_ops,     /*电源管理操作*/
};      /*总线没有定义探测函数*/
```

platform_bus_type 实例中部分成员简介如下：

●**platform_dev_groups**：默认设备属性指针数组，定义在/drivers/base/platform.c 文件内，表示默认添加到 platform 总线下设备的属性。

```
static struct attribute *platform_dev_attrs[] = {
    &dev_attr_modalias.attr,            /*modalias 只读属性，获取模块别名*/
    &dev_attr_driver_override.attr,     /*driver_override 读写属性，属性值为 pdev->driver_override*/
    NULL,
};
```

总线设备默认属性包括 modalias 和 driver_override，它们都定义在/drivers/base/platform.c 文件内，属性

读写函数请读者自行阅读。

●**platform_dev_pm_ops**：电源管理数据结构，实例定义在/drivers/base/platform.c 文件内：

```
static const struct dev_pm_ops platform_dev_pm_ops = {
    .runtime_suspend = pm_generic_runtime_suspend,        /*/drivers/base/power/generic_ops.c*/
    .runtime_resume = pm_generic_runtime_resume,
    USE_PLATFORM_PM_SLEEP_OPS
 };
```

platform_dev_pm_ops 实例中各函数，在没有选择 PM 配置选项时直接返回 0，否则各函数将调用设备驱动 device_driver->pm 指向实例中定义对应函数，如果驱动中没有定义对应的函数则返回 0。

●**platform_match()**：判断总线下指定设备与指定驱动是否匹配的函数，若匹配函数返回 1，否则返回 0，函数实现后面再做介绍。

●**platform_uevent()**：向总线添加设备触发 uevent 事件时，调用此函数添加环境变量。此函数定义在/drivers/base/platform.c 文件内，代码如下：

```
static int platform_uevent(struct device *dev, struct kobj_uevent_env *env)
{
    struct platform_device *pdev = to_platform_device(dev);
    int rc;
    rc = of_device_uevent_modalias(dev, env);       /*添加模块别名的环境变量*/
    if (rc != -ENODEV)
        return rc;

    rc = acpi_device_uevent_modalias(dev, env);
    if (rc != -ENODEV)
        return rc;

    add_uevent_var(env, "MODALIAS=%s%s", PLATFORM_MODULE_PREFIX,pdev->name);
    return 0;
}
```

2 初始化

platform 总线初始化函数 platform_bus_init(void)完成总线的注册，函数由 driver_init()函数调用，初始化函数定义在/drivers/base/platform.c 文件内，代码如下：

```
int __init platform_bus_init(void)
{
    int error;

    early_platform_cleanup();

    error = device_register(&platform_bus);     /*将总线当成设备向内核注册，名称为"platform"*/
    if (error)
        return error;
    error = bus_register(&platform_bus_type);     /*注册总线实例*/
    if (error)
```

```
        device_unregister(&platform_bus);
    of_platform_register_reconfig_notifier();  /*注册通知*/
    return error;
}
```

初始化函数比较简单，platform 总线被当成设备（或视其为主机控制器）注册为/sys/devices/*platform* 目录，作为总线下设备的父设备，然后注册总线 **platform_bus_type** 实例。

8.6.2 platform 设备

platform_device 结构体表示挂接在 platform 总线上的设备，它是 device 结构体的包装器。

1 数据结构

内核在/include/linux/platform_device.h 头文件内定义了 platform_device 结构体，表示平台总线设备：

```
struct platform_device {
    const char      *name;          /*设备名称*/
    int             id;             /*设备编号，由内核管理设备编号，platform_devid_ida*/
    bool            id_auto;
    struct device   dev;            /*内嵌 device 实例*/
    u32             num_resources;  /*资源实例数量*/
    struct resource *resource;      /*指向资源结构实例数组*/
    const struct platform_device_id   *id_entry;   /*指向驱动 platform_device_id 列表中的匹配项*/
    char *driver_override;          /*强制匹配的驱动名称，只进行与驱动名称的匹配*/

    struct mfd_cell *mfd_cell;
    struct pdev_archdata    archdata;   /*体系结构相关的数据结构，MIPS 为空*/
};
```

platform_device 结构体中主要成员简介如下：

●**name**：设备名称，在总线匹配函数中用于匹配驱动，在添加设备时将赋予 device 实例成员（或尾部加上编号）。

●**id**：设备在平台总线上的编号，设为 **PLATFORM_DEVID_NONE**(-1)表示向总线添加设备时不分配编号，为 PLATFORM_DEVID_AUTO(-2)表示由内核自动分配编号，并赋予 id 成员。在注册设备时 name 加上编号 id 将赋予 dev 实例作为设备在 sysfs 文件系统中的名称。

●**dev**：内嵌 device 结构体实例。

●**resource**：资源结构 resource 实例指针，实例指向的是资源数组，后面再作介绍。

●**driver_override**：字符串指针，如果设置了此成员，则在匹配驱动时，只用此字符串与驱动名称进行匹配，不再进行其它方式的匹配。

●**id_entry**：指向匹配的 platform_device_id 实例。在 platform 驱动中如果定义了 platform_device_id 实例数组，则在匹配设备与驱动的函数中，会将设备名称与驱动中 platform_device_id 实例数组项（name 成员）进行比较，名称相同则表示匹配上。id_entry 成员指向与本设备匹配的数组项（platform_device_id 实例），详见下文件 platform 驱动中的介绍。

2 资源

资源由 resource 结构体表示，是设备具有的硬件资源，如内存、中断号等。

■数据结构

资源 resource 结构体定义在/include/linux/ioport.h 头文件：

```
struct resource {
    resource_size_t   start;      /*起始值*/
    resource_size_t   end;        /*结束值*/
    const char    *name;          /*资源名称*/
    unsigned long    flags;       /*标记*/
    struct resource *parent, *sibling, *child;     /*资源结构实例组成父子层次结构*/
};
```

resource 结构体是描述各种硬件资源的通用数据结构，主要成员简介如下：

●start、end：通用的资源的起始和结束值，如内存的起始、结束地址。

●name：资源名称。

●parent、sibling、child：resource 实例指针，用于将实例组织成父子层次结构。父子资源之间是包含关系，即父资源的起止值包含了其下子资源的起止值。

●flags：标记。flags 成员为 32 位无符号整数，各标记位语义定义在/include/linux/ioport.h 头文件。其中 bit[7..0]共 8 位表示特定于总线的特性，bit[12..8]共 5 位表示资源的类型，其它标记位表示属性的其它特性，如下所示。

```
#define   IORESOURCE_BITS          0x000000ff   /*总线特性掩码，共 8 位*/
#define   IORESOURCE_TYPE_BITS     0x00001f00   /*资源类型掩码，共 5 位*/

#define   IORESOURCE_IO            0x00000100   /* PCI/ISA I/O ports，IO 端口*/
#define   IORESOURCE_MEM           0x00000200   /*内存资源*/
#define   IORESOURCE_REG           0x00000300   /* IO 寄存器偏移量*/
#define   IORESOURCE_IRQ           0x00000400   /*中断编号*/
#define   IORESOURCE_DMA           0x00000800
#define   IORESOURCE_BUS           0x00001000

#define   IORESOURCE_PREFETCH      0x00002000   /*没有副作用*/
#define   IORESOURCE_READONLY      0x00004000   /*只读资源*/
#define   IORESOURCE_CACHEABLE     0x00008000   /*可缓存*/
#define   IORESOURCE_RANGELENGTH   0x00010000
#define   IORESOURCE_SHADOWABLE    0x00020000

#define   IORESOURCE_SIZEALIGN     0x00040000   /*大小（长度）值对齐*/
#define   IORESOURCE_STARTALIGN    0x00080000   /*起始值对齐*/

#define   IORESOURCE_MEM_64        0x00100000
#define   IORESOURCE_WINDOW        0x00200000   /* forwarded by bridge */
#define   IORESOURCE_MUXED         0x00400000   /* Resource is software muxed */
```

```
#define   IORESOURCE_EXCLUSIVE    0x08000000   /*独占式资源*/
#define   IORESOURCE_DISABLED     0x10000000
#define   IORESOURCE_UNSET        0x20000000   /* No address assigned yet */
#define   IORESOURCE_AUTO         0x40000000
#define   IORESOURCE_BUSY         0x80000000   /*驱动程序已标记资源忙*/

/*  以下是 PnP  各类型资源的特性标记，低 8 位*/
...      （略）

/* PCI ROM 控制器资源特性位*/
#define   IORESOURCE_ROM_ENABLE       (1<<0)
#define   IORESOURCE_ROM_SHADOW       (1<<1)
#define   IORESOURCE_ROM_COPY         (1<<2)
#define   IORESOURCE_ROM_BIOS_COPY    (1<<3)

/* PCI 控制位  */
#define   IORESOURCE_PCI_FIXED      (1<<4)     /* Do not move resource */
```

■初始化

资源管理是为了更好地分配资源，防止冲突，设备需要使用资源时需先向内核申请。资源管理相关代码位于/kernel/resource.c 文件内，先来看初始化函数。

内核在/kernel/resource.c 文件内定义了 IO 端口和物理内存两个资源类型的根节点：

```
struct resource ioport_resource = {
    .name     = "PCI IO",
    .start    = 0,
    .end      = IO_SPACE_LIMIT,
    .flags    = IORESOURCE_IO,       /*类型，IO 资源*/
};
```

IO_SPACE_LIMIT 表示端口结束地址，它显然应该是一个与体系结构相关的参数。MIPS 体系结构下，定义在以/arch/mips/include/asm/io.h 头文件内：

```
#define   IO_SPACE_LIMIT   0xffff
```

物理内存资源根节点定义如下：

```
struct resource iomem_resource = {
    .name     = "PCI mem",
    .start    = 0,
    .end      = -1,
    .flags    = IORESOURCE_MEM,    /*类型，物理内存资源*/
};
```

内核在启动初期需要对资源管理数据结构进行初始化，资源管理结构其实就是由 resource 实例组成的层次树状结构，每种类型资源由一个层次结构管理。父子资源包含子资源。

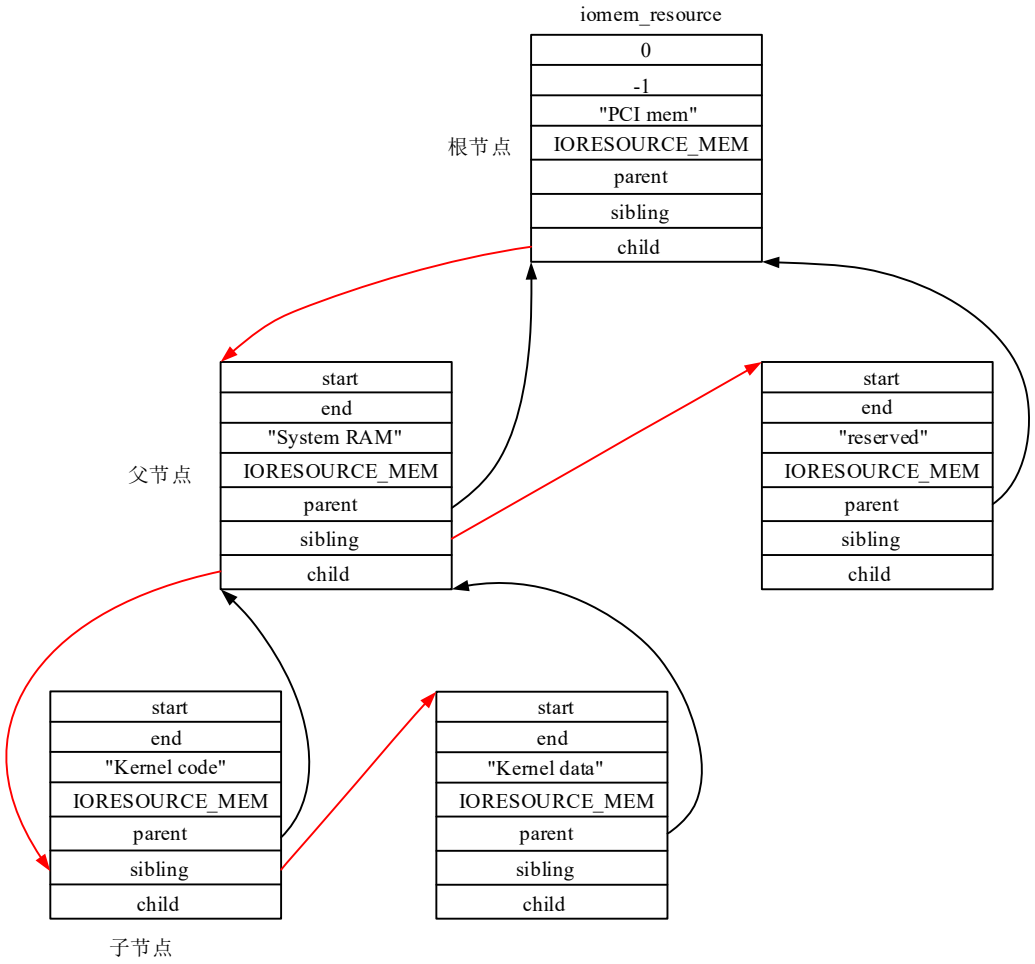
资源初始化函数调用关系为 setup_arch()->resource_init()，resource_init()函数在/arch/mips/kernel/setup.c 文件内实现，代码如下：

```
static struct resource code_resource = { .name = "Kernel code", };       /*内核代码所占内存*/
static struct resource data_resource = { .name = "Kernel data", };       /*内核数据所占内存*/

static void __init resource_init(void)
{
    int i;
    if (UNCAC_BASE != IO_BASE)
        return;
    code_resource.start = __pa_symbol(&_text);     /*设置内核代码、数据内存的起止地址*/
    code_resource.end = __pa_symbol(&_etext) - 1;
    data_resource.start = __pa_symbol(&_etext);
    data_resource.end = __pa_symbol(&_edata) - 1;

    for (i = 0; i < boot_mem_map.nr_map; i++) {        /*扫描物理内存段信息，注册内存资源*/
      struct resource *res;
      unsigned long start, end;

      start = boot_mem_map.map[i].addr;
      end = boot_mem_map.map[i].addr + boot_mem_map.map[i].size - 1;
      if (start >= HIGHMEM_START)
          continue;
      if (end >= HIGHMEM_START)
          end = HIGHMEM_START - 1;              /*剔除高端内存*/

      res = alloc_bootmem(sizeof(struct resource));   /*从自举分配器中分配资源结构实例*/
      switch (boot_mem_map.map[i].type) {
          case BOOT_MEM_RAM:                  /*设置资源名称*/
          case BOOT_MEM_ROM_DATA:
              res->name = "System RAM";
              break;
          case BOOT_MEM_RESERVED:
          default:
              res->name = "reserved";
      }

      res->start = start;
      res->end = end;
      res->flags = IORESOURCE_MEM | IORESOURCE_BUSY;        /*资源实例初始化*/
      request_resource(&iomem_resource, res);       /*在 iomem_resource 下注册新资源实例*/
      request_resource(res, &code_resource);        /*code_resource 注册到新资源实例下*/
      request_resource(res, &data_resource);        /*data_resource 注册到新资源实例下*/
    }
```

}

resource_init()函数扫描 boot_mem_map 实例保存的物理内存段信息，对每个内存段创建 resource 实例（内存类型），然后将 resource 实例注册到 **iomem_resource** 物理内存资源实例下。新创建的 resource 实例名称为"System RAM"或"reserved"，最后尝试将 code_resource 和 data_resource 实例注册到新创建的 resource 实例下。

resource_init()函数执行后资源数据结构实例组织关系如下图所示：



在/kernel/resource.c 文件还定义了 ioresources_init()函数，在 proc 文件系统下创建两个文件，如下所示：

```
static int __init ioresources_init(void)
{
    proc_create("ioports", 0, NULL, &proc_ioports_operations);    /*ioports 文件*/
    proc_create("iomem", 0, NULL, &proc_iomem_operations);    /*iomem 文件*/
    return 0;
}
__initcall(ioresources_init);
```

ioports 文件和 iomem 文件是只读文件，用于输出 IO 和内存资源信息。

■申请资源

在前面介绍的初始化中，内核定义了 IO 和内存资源的根节点 ioport_resource 和 iomem_resource 实例，所有 IO 及内存资源都应位于此根节点之下，并且不能超过根节点对资源的限制值。

在根节点下申请（注册）资源的函数为 request_resource(struct resource *root, struct resource *new)，函

数定义在/kernel/resource.c 文件内，代码如下：

```
int request_resource(struct resource *root, struct resource *new)
/*root：资源根节点，new：申请的新资源实例*/
{
    struct resource *conflict;

    conflict = request_resource_conflict(root, new);      /*/kernel/resource.c*/
    return conflict ? -EBUSY : 0;        /*申请成功返回 0，否则返回错误码-EBUSY*/
}
```

root 为 new 资源实例的根节点，new 表示申请的资源。new 表示资源的起止值必须在 root 表示资源的起止值范围内，也就是说 root 必须包含 new，且 new 不与 root 下已有所有子资源值范围有重叠。资源申请成功返回 0，不成功返回错误码。

request_resource()函数直接调用 request_resource_conflict(root, new)函数完成具体的工作，此函数获取锁后调用__request_resource(root, new)函数完成资源的申请，随后释放锁。

__request_resource(root, new)函数定义如下：

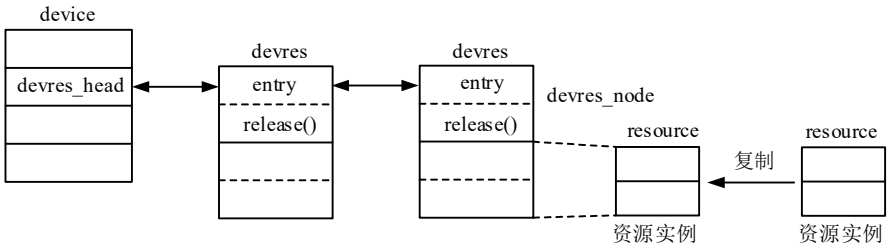
```
static struct resource * __request_resource(struct resource *root, struct resource *new)
{
    resource_size_t start = new->start;   /*新资源起始值*/
    resource_size_t end = new->end;      /*结束值*/
    struct resource *tmp, **p;

    if (end < start)
        return root;
    if (start < root->start)
        return root;
    if (end > root->end)
        return root;          /*判断资源信息是否有效，无效返回 root*/
    p = &root->child;          /*指向 root 的子节点*/
    for (;;) {            /*遍历子节点，找到 new 合适的插入点，并插入父节点的子节点链表*/
        tmp = *p;
        if (!tmp || tmp->start > end) {        /*没有重叠*/
            new->sibling = tmp;            /*插入子链表*/
            *p = new;
            new->parent = root;
            return NULL;          /*申请成功（没有检查重叠）*/
        }
        p = &tmp->sibling;
        if (tmp->end < start)      /*与当前子节点没有重叠，遍历下一节点*/
           continue;
        return tmp;      /*返回 tmp 表示 new 与 tmp 有重叠，申请失败*/
    }
}
```

__request_resource()函数只遍历 root 下的子资源实例（不遍历子节点的子节点），查找 new 实例合适

的插入点。子资源实例在链表中以起始值从小到大在链表中从左至右排列。如果 new 与现有 root 下子资源实例值范围有重叠，函数将返回有重叠资源实例的指针，申请失败。如果没有重叠，则将 new 实例插入正 root 下的子资源链表，函数返回 0，申请成功。

在指定父节点下申请（注册）资源的函数为 int **insert_resource(struct resource *parent, struct resource *new)**，parent 为父节点，new 为申请（注册）节点，此函数会逐层遍历 parent 下的子孙节点，如果 new 与现在节点没有重叠则直接添加，如果 new 包含于某个节点内部，则添加为此节点的子节点，如果与某个节点部分重叠则返回错误码，函数源代码请读者自行阅读。

int **devm_request_resource**(struct device *dev, struct resource *root,struct resource *new)：用于将 resource 实例作为 device 实例的资源向 device 实例添加，devres 实例后面接的是 new 指向 resource 实例的副本，如下图所示。new 指向实例添加到管理结构中，添加到 root 下子节点链表中（调用 request_resource_conflict() 函数），源代码请读者自行阅读。



int **platform_device_add_resources**(struct platform_device *pdev,const struct resource *res, unsigned int num)：用于将 platform_device 实例关联资源数组，res 指向资源数组，num 为数组项数。

3 注册设备

在体系结构相关的代码（板级相关的代码）可静态定义 platform_device 实例，在内核启动阶段，向 platform 总线注册设备。

如果使用了设备树，将根据设备节点 device_node 实例动态创建和注册 platform_device 实例。

■静态注册

如果没有使用设备树，注册 platform 设备的函数为 platform_device_register(pdev)，定义在 /drivers/base/platform.c 文件内：

```
int platform_device_register(struct platform_device *pdev)
{
    device_initialize(&pdev->dev);          /*初始化内嵌 device 实例*/
    arch_setup_pdev_archdata(pdev);         /*空操作*/
    return platform_device_add(pdev);       /*添加 platform_device 设备*/
}
```

device_initialize(&pdev->dev)函数在前面介绍过，用于初始化 platform_device 内嵌的 device 实例。

platform_device_add(pdev)函数完成向内核添加 platform_device 设备的工作，代码如下：

```
int platform_device_add(struct platform_device *pdev)
{
    int i, ret;
```

```
if (!pdev)
    return -EINVAL;

if (!pdev->dev.parent)
    pdev->dev.parent = &platform_bus;   /*父设备为表示 platform_bus 总线的设备*/

pdev->dev.bus = &platform_bus_type;          /*platform_bus 总线*/

switch (pdev->id) {       /*如果设置了 id 成员，需要设置 pdev->dev 名称*/
default:                  /*静态指定 id 编号*/
    dev_set_name(&pdev->dev, "%s.%d", pdev->name,  pdev->id);  /*device 名称为 name+id*/
    break;
case PLATFORM_DEVID_NONE:       /*不使用 id 编号（-1）*/
    dev_set_name(&pdev->dev, "%s", pdev->name);         /*直接使用 pdev->name 作为设备名称*/
    break;
case PLATFORM_DEVID_AUTO:       /*由内核自动分配 id 编号*/
    ret = ida_simple_get(&platform_devid_ida, 0, 0, GFP_KERNEL);
    if (ret < 0)
        goto err_out;
    pdev->id = ret;
    pdev->id_auto = true;
    dev_set_name(&pdev->dev, "%s.%d.auto", pdev->name, pdev->id);      /*设置名称*/
    break;
}

for (i = 0; i < pdev->num_resources; i++) {          /*添加资源，扫描资源数组*/
    struct resource *p, *r = &pdev->resource[i];

    if (r->name == NULL)
        r->name = dev_name(&pdev->dev);        /*设置资源名称*/
    p = r->parent;
    if (!p) {       /*如果资源没有指定父节点，则根据资源类型设置父节点*/
        if (resource_type(r) == IORESOURCE_MEM)
            p = &iomem_resource;
        else if (resource_type(r) == IORESOURCE_IO)
            p = &ioport_resource;
    }

    if (p && insert_resource(p, r)) {      /*存在父节点，并执行插入资源操作*/
        ...
    }
}
...
```

```
    ret = device_add(&pdev->dev);     /*向内核添加设备 device 实例*/
    if (ret == 0)
        return ret;
    ...
}
```

platform_device_add()函数主要完成内嵌 device 实例父节点的设置、设置所属总线、设置名称等，向内核注册设备资源，最后调用 device_add()向内核添加内嵌的 device 实例。

void platform_device_unregister(struct platform_device *pdev)：注销 platform_device 实例。

int **platform_add_devices**(struct platform_device **devs, int num)：注册 platform_device 实例数组，参数 devs 指向 platform_device 实例的指针数组，num 为需要添加的实例数量，调用 platform_device_register()函数执行注册设备操作。

int **platform_device_add_data**(struct platform_device *pdev, const void *data,size_t size)：将设备的平台信息 data 指向的数据结构赋予 pdev->dev.platform_data 成员，size 为数据大小。

■动态注册

如果使用了设备树，用户通过设备树文件向内核传递设备硬件信息。外部设备在设备树中由节点表示，在内核中由 device_node 结构体实例表示。

内核在/drivers/of/platform.c 文件内定义了为指定设备节点创建 platfrom_device 实例的函数：

```
struct platform_device *of_platform_device_create(struct device_node *np,const char *bus_id,
                                                  struct device *parent);
                                /*调用 platform_device_alloc()函数分配 platform_device 实例*/
```

struct platform_device ***platform_device_alloc**(const char *name, int id)：用于动态分配 platform_device 实例（/drivers/base/platform.c）。

struct platform_device ***platform_device_register_full**(const struct platform_device_info *pdevinfo)：创建并注册带资源和平台数据的 platform_device 实例。

platform_device_info 结构体定义如下（/include/linux/platform_device.h）：

```
struct platform_device_info {
        struct device *parent;          /*父设备*/
        struct fwnode_handle *fwnode;
        const char *name;               /*名称*/
        int id;
        const struct resource *res;     /*资源列表*/
        unsigned int num_res;           /*资源数量*/
        const void *data;               /*平台数据*/
        size_t size_data;               /*平台数据大小*/
        u64 dma_mask;
};
```

在后面介绍具体设备驱动程序时，还将涉及到 platform_device 的操作函数。

如果平台通过设备树传递 platform 总线上的设备信息，在平台定义的初始化函数中将调用函数：

int **of_platform_bus_probe**(struct device_node *root,const struct of_device_id *matches,

struct device *parent); /*/drivers/of/platform.c*/

扫描设备树节点，检查节点与 matches 指定 of_device_id 实例是否匹配，若匹配则创建并添加表示设备的 platform_device 实例，将从节点中获取设备信息填充至 platform_device 实例。

其它总线在注册主机控制器时，将扫描设备树中控制器下的设备节点，为其创建并设置 xxx_device 实例。

8.6.3 platform 驱动

平台总线驱动由 platform_driver 结构体表示，它是通用驱动模型中 device_driver 结构体的包装器。

1 数据结构

platform 总线设备驱动 platform_driver 结构体定义如下（/include/linux/platform_device.h）：

```
struct platform_driver {
    int (*probe) (struct platform_device *);          /*探测设备函数，匹配设备后调用*/
    int (*remove) (struct platform_device *);         /*移除 platform 设备时调用*/
    void (*shutdown) (struct platform_device *);      /*关闭设备时调用*/
    int (*suspend) (struct platform_device *, pm_message_t state);     /*设备睡眠时调用*/
    int (*resume) (struct platform_device *);                          /*唤醒设备时调用*/
    struct device_driver    driver;                   /*内嵌 device_driver 结构体实例*/
    const struct platform_device_id    *id_table;     /*匹配列表*/
    bool    prevent_deferred_probe;
};
```

platform_driver 结构体主要成员简介如下：

●**probe()**：探测函数，设备与驱动匹配后调用此函数，加载设备驱动程序。

●**id_table**：指向 platform_device_id 实例数组，每个数组项保存匹配本驱动的设备名称，用于与 platform_device 实例进行名称匹配，列表最后一项必须为空。

platform_device_id 结构体定义在/include/linux/mod_devicetable.h 头文件：

```
#define    PLATFORM_NAME_SIZE    20
#define    PLATFORM_MODULE_PREFIX    "platform:"
struct platform_device_id {
    char name[PLATFORM_NAME_SIZE];          /*匹配字符串，用于与设备名称匹配*/
    kernel_ulong_t driver_data;
};
```

2 注册驱动

platform_driver 实例通常在设备驱动程序中静态定义，在内核初始化函数或模块初始化函数中调用注册函数 **platform_driver_register(drv)**向内核注册实例。

注册函数定义在/include/linux/platform_device.h 头文件内：

```
#define platform_driver_register(drv) \
    __platform_driver_register(drv, THIS_MODULE)
```

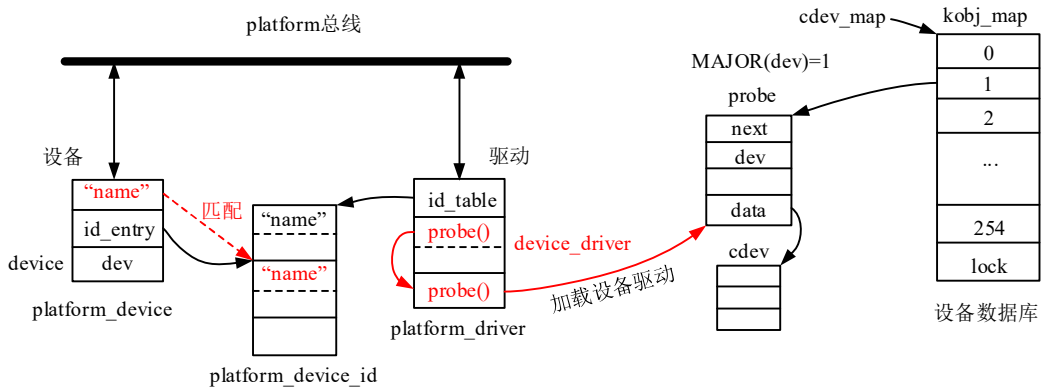
__platform_driver_register()函数在/drivers/base/platform.c 文件内实现，代码如下：

```
int    __platform_driver_register(struct platform_driver *drv,struct module *owner)
```

```
{
    drv->driver.owner = owner;
    drv->driver.bus = &platform_bus_type;        /*指向总线实例*/
    if (drv->probe)      /*如果 platform_driver 定义了 probe()函数*/
        drv->driver.probe = platform_drv_probe;   /*device_driver.probe 赋值通用探测函数*/
     if (drv->remove)
        drv->driver.remove = platform_drv_remove;   /*device_driver.remove 赋值通用函数*/
     if (drv->shutdown)
        drv->driver.shutdown = platform_drv_shutdown;   /*device_driver.shtdown 赋值通用函数*/
     return driver_register(&drv->driver);                    /*注册驱动 device_driver 实例*/
}
```

注册函数比较简单，需要注意的是 platform_driver 结构体中定义了与 device_driver 结构体中同名的函数，通常实现 platform_driver 实例中的函数，而 device_driver 实例中相应的函数赋值为通用函数。通用函数内部调用 platform_driver 实例中的同名的函数完成操作，因为通用驱动模型中调用的是 device_driver 实例中函数。

如下图所示，假设 platform_driver 实例定义了 probe()函数，则内嵌 device_driver 实例的 probe()函数赋值为通用函数 platform_drv_probe()。在通用驱动模型中，当设备与驱动匹配时，将调用 device_driver 实例的 probe()函数（platform 总线没有定义 probe()函数），进而调用 platform_driver 实例中的 probe()函数，加载设备驱动程序。



platform_drv_probe()函数定义如下：

```
static int platform_drv_probe(struct device *_dev)
/*_dev：platform_driver 结构体内嵌的 device 实例指针*/
{
    struct platform_driver *drv = to_platform_driver(_dev->driver);
    struct platform_device *dev = to_platform_device(_dev);
    int ret;

    ret = of_clk_set_defaults(_dev->of_node, false);
    if (ret < 0)
        return ret;

    ret = dev_pm_domain_attach(_dev, true);
    if (ret != -EPROBE_DEFER) {
```

```
        ret = drv->probe(dev);          /*调用 platform_driver 实例 probe()函数*/
        if (ret)
            dev_pm_domain_detach(_dev, true);
    }

    if (drv->prevent_deferred_probe && ret == -EPROBE_DEFER) {
        ...
    }

    return ret;
}
```

platform_drv_remove()和 platform_drv_shutdown()函数与 platform_drv_probe()函数类似，请读者自行阅读源代码。

void platform_driver_unregister(struct platform_driver *drv)：用于注销 platform 驱动。

在/include/linux/platform_device.h 头文件和/drivers/base/platform.c 文件内还定义了其它 platform_driver 实例的操作函数，例如：

●int __init_or_module **__platform_driver_probe**(struct platform_driver *drv,int (***probe**)(struct platform_device *), struct module *module)：为不支持热插拔的设备注册驱动，只在注册驱动时会匹配设备，之后设备不能再与本驱动匹配了（probe()函数设为 NULL），probe()为探测函数指针。

●**platform_driver_probe**(drv, probe)：__platform_driver_probe()函数的包装器，module 参数为 THIS_MODULE。

●void platform_set_drvdata(struct platform_device *pdev,void *data)：用 data 设置 pdev->dev->driver_data 成员，表示驱动数据。

●void *platform_get_drvdata(const struct platform_device *pdev)：获取 pdev->dev->driver_data 值。

●**module_platform_driver**(__platform_driver)：宏定义，__platform_driver 为 platform_driver 实例，定义模块初始化函数为 platform_driver_register()，表示在加载模块时注册 platform_driver 实例，模块卸载函数为 platform_driver_unregister()，用于卸载模块时注销 platform_driver 实例。

●**module_platform_driver_probe**(__platform_driver, **__platform_probe**)：宏定义，__platform_driver 为 platform_driver 实例，定义模块初始化函数为 platform_driver_probe()，表示在加载模块时注册 platform_driver 实例，其 pobe()函数为__platform_probe()，模块卸载函数为 platform_driver_unregister()，用于卸载模块时注销 platform_driver 实例。

●**builtin_platform_driver_probe**(__platform_driver, __platform_probe)：宏定义，表示在内核初始化时调用 platform_driver_probe()函数，注册 platform_driver 实例，其 pobe()函数为__platform_probe()，此宏不适用于模块。

8.6.4 总线匹配函数

在向总线添加设备/驱动时，将扫描总线的驱动/设备链表，对各实例调用总线定义的 match()函数判断设备与驱动是否匹配，如果匹配则调用总线或驱动的 probe()函数，加载设备驱动程序。

下面看一下 platform 总线匹配函数的定义（/drivers/base/platform.c）：

```
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);

    if (pdev->driver_override)   /*如果设置了 driver_override，则只将其与驱动名称进行匹配*/
        return !strcmp(pdev->driver_override, drv->name);   /*strcmp()字符相等返回 0*/

    if (of_driver_match_device(dev, drv))   /*匹配设备树节点，/include/linux/of_device.h*/
        return 1;

    if (acpi_driver_match_device(dev, drv))        /*需选择 ACPI 配置选项，/drivers/acpi/scan.c*/
        return 1;

    if (pdrv->id_table)       /*platform_device_id 实例列表匹配*/
        return platform_match_id(pdrv->id_table, pdev) != NULL;

    return (strcmp(pdev->name, drv->name) == 0);     /*最后进行设备与驱动名称的匹配*/
}
```

platform_match()函数判断设备与驱动是否匹配的顺序如下（有一项匹配则返回 1）：

（1）如果 platform_device 设置了 **driver_override** 成员则只将其与驱动名称进行匹配，匹配返回 1，不匹配返回 0，不再进行下面的匹配。下面步骤以 driver_override 成员为 NULL 为前提。

（2）通过 device_driver->driver->of_match_table 列表检查与 platform_device->dev->of_node 是否匹配，比较兼容属性。

（3）ACPI 列表匹配。

（4）将 platform_device 实例与 platform_driver 实例中 platform_device_id 实例数组项匹配，见下文。

（5）如果以上匹配都不成功，最后检查设备与驱动名称的是否相同，相同则表示匹配，返回 1，不相同表示不匹配，返回 0。

下面看一下检查 platform_device 实例与 platform_driver 实例中 platform_device_id 实例数组项匹配的函数。platform_match_id()函数定义如下（/drivers/base/platform.c）：

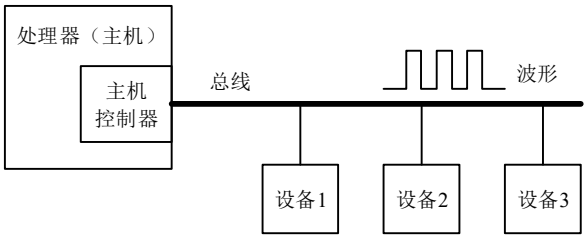
```
static const struct platform_device_id *platform_match_id(const struct platform_device_id *id, \
                                                          struct platform_device *pdev)
/*id：驱动 platform_device_id 实例数组指针，pdev：platform_device 指针*/
{
    while (id->name[0]) {                /*直至扫描到空的数组项*/
        if (strcmp(pdev->name, id->name) == 0) {          /*将设备名称与列表项 name 成员比较*/
            pdev->id_entry = id;       /*相同则 pdev->id_entry 指向匹配的 platform_device_id 实例*/
            return id;
        }
        id++;
    }
    return NULL;
}
```

platform_match_id()函数将 platform_device 名称依次与驱动中的 platform_device_id 实例数组项的名称进行比较，若相同则表示匹配成功，并将匹配项赋予 platform_device 实例的 id_entry 成员，函数返回匹配

列表项指针，否则返回 NULL。

8.7 SPI 总线

SOC 系统中，SPI、I2C、USB 等总线结构如下图所示，处理器（主机）具有总线的主机控制器，负责通过总线连接外接设备，控制器以特定的协议在总线上产生波形，实现与外接设备的数据传输和进行设备控制。主机控制器挂载在芯片内部总线上，通常处理器通过操作控制器的寄存器来对控制器进行控制和数据传输，处理器对主机控制器的操作类似于对物理内存的操作。



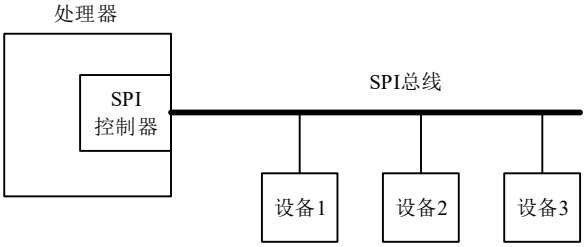
Linux 内核采用主机控制器驱动与外设驱动分离的思想。总线控制驱动只负责产生波形，传输给定的数据，数据是何用处，数据的内容是什么，主机控制器不管。主机控制器驱动向外设驱动程序提供传输数据的接口函数，数据如何解释和使用由外设驱动程序实现。

外设驱动程序调用主机控制器驱动提供的接口函数在总线上实现数据的传输。外设驱动只需要提供传输数据的内容，调用接口函数进行数据传输即可，它并不需要关心数据是如何进行传输的，这由主机控制器驱动实现。

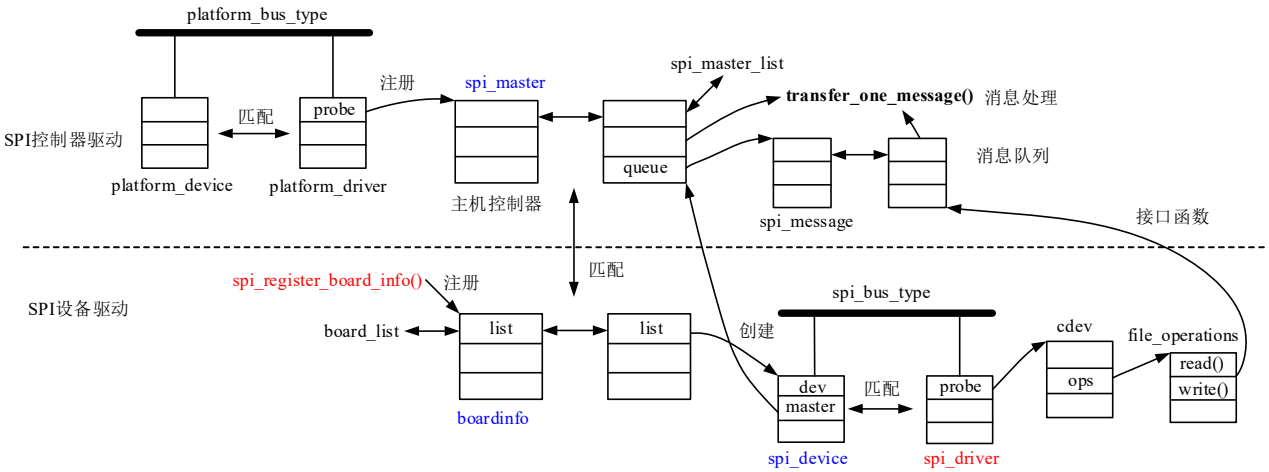
本节介绍 SPI 主机控制器（总线）驱动的实现，后面将介绍 I2C、USB 等总线（控制器）驱动的实现。

8.7.1 概述

SPI 总线是一种全双工串行通信总线，接口协议中包括时钟信号、片选信号、主机发送信号线及主机接收信号线等。SPI 总线结构如下图所示，处理器通过操作 SPI 控制器的寄存器来实现总线上数据的传输。SPI 总线上可能挂接了多个设备，它们共用总线进行数据传输，通过控制器的片选信号来确定 SPI 控制器与哪个外接设备进行数据传输。



SPI 总线驱动框架如下图所示：



图中上半部分表示 SPI 主机控制器驱动，下半部分表示 SPI 设备驱动。SPI 主机控制器在驱动框架中由 **spi_master** 结构体表示，结构体中包含控制器的物理信息及数据传输函数指针等成员，结构体实例在内核中由全局双链表管理。主机控制器作为一个设备挂接在平台总线上（或其它总线，下同），在添加表示控制器的设备时（platform_device），匹配控制器驱动（platform_driver），匹配成功调用驱动中的 probe()函数，函数内完成 spi_master 实例的创建和注册。

SPI 主机控制器的 spi_master 实例中具有需要传输数据的消息队列，spi_master 结构体中函数指针成员 transfer_one_message()用于完成一个消息的数据传输。SPI 控制器驱动程序的主要工作就是在驱动的 probe()函数中创建和设置 spi_master 实例，并向内核注册。

SPI 驱动框架中定义了 spi_bus_type 总线，SPI 总线设备和驱动挂载到此总线上。SPI 设备由 spi_device 结构体表示，SPI 设备驱动由 spi_driver 结构体表示。向 SPI 总线添加设备或驱动时，将会在总线下查找匹配的驱动或设备，匹配成功将调用驱动的 probe()函数，函数内完成具体字符/块设备驱动程序的加载（如：注册 cdev 实例）。

SPI 驱动框架中提供了向 spi_master 实例提交消息，执行数据传输的接口函数，SPI 设备驱动程序中文件操作结构 file_operations 实例中的函数调用此接口函数进行数据传输。

向 SPI 总线添加设备的操作与平台设备的添加稍有不同，用户需通过 **spi_register_board_info()**接口函数注册 **spi_board_info** 实例来完成 SPI 设备的添加。spi_board_info 结构体表示 SPI 设备的信息，注册此结构体实例时会查找匹配的 spi_master 实例（设备连接到的 SPI 主机控制器），若匹配成功则创建 **spi_device** 结构体实例，并挂载到 SPI 总线。在板级相关的文件内只需要为 SPI 设备创建和注册 spi_board_info 实例即可，而不需要创建和添加 spi_device 实例。

8.7.2 控制器驱动

SPI 主机控制器驱动主要完成 spi_master 实例的创建和注册，以及数据传输函数的实现，控制器驱动位于/drivers/spi/目录下。

1 数据结构

每个 SPI 主机控制器由 spi_master 结构体表示，结构体定义如下（/include/linux/spi/spi.h）：

```
struct spi_master {
    struct device    dev;          /*内嵌 device 实例，添加到通用驱动模型中*/
    struct list_head  list;        /*双链表成员，将 spi_master 实例添加到全局链表*/
    s16              bus_num;      /*主机控制器编号，从 0 开始编号，用于标识 SPI 主机控制器*/
    u16              num_chipselect;    /*片选信号最大编号，从 0 开始编号，用于区分外接设备*/
```

```
u16             dma_alignment;      /*DMA 对齐要求*/
u16             mode_bits;          /*识别 spi_device.mode 成员的位数*/
u32             bits_per_word_mask;    /* bitmask of supported bits_per_word for transfers */
...
u32             min_speed_hz;       /*最低传输速率*/
u32             max_speed_hz;       /*最高传输速率*/
u16             flags;              /*标记成员，取值定义如下*/
...
spinlock_t          bus_lock_spinlock;      /*保护自旋锁*/
struct mutex        bus_lock_mutex;         /*保护互斥量*/

bool            bus_lock_flag;     /*标记总线被独占使用*/

int    (*setup)(struct spi_device *spi);     /*修改传输速率和模式*/
int    (*transfer)(struct spi_device *spi,struct spi_message *mesg);        /*已弃用*/
                                        /*进行消息传输的函数，数据由 spi_message 结构体表示*/
void   (*cleanup)(struct spi_device *spi);              /*释放由 spi_master 提供的内存*/
bool   (*can_dma)(struct spi_master *master, struct spi_device *spi,struct spi_transfer *xfer);
                                                        /*是否支持 DMA*/
bool                queued;          /*是否提供内部消息队列*/
struct kthread_worker     kworker;          /*内核线程工作者*/
struct task_struct        *kworker_task;    /*内核工作者进程结构指针*/
struct kthread_work       pump_messages;    /*内核工作者工作*/
spinlock_t                queue_lock;   /*消息队列保护自旋锁*/
struct list_head          queue;        /*消息队列，管理 spi_messages 实例*/
struct spi_message        *cur_msg;     /*当前正在处理的消息*/
bool        idling;             /*设备进入空闲状态*/
bool        busy;               /*忙状态，正在处理消息*/
bool        running;            /*正在处理消息*/
bool        rt;                 /*处理消息队列的线程是否设为实时线程*/
bool        auto_runtime_pm;
bool        cur_msg_prepared;   /*是否调用 spi_prepare_message*/
bool        cur_msg_mapped;   /*消息是否已映射到 DMA*/
struct completion     xfer_completion;   /*完成量，由 transfer_one_message()使用*/
size_t      max_dma_len;        /*DMA 最大传输长度*/

int (*prepare_transfer_hardware)(struct spi_master *master);
int (*transfer_one_message)(struct spi_master *master,struct spi_message *mesg);
                                        /*传输单个 spi_message 的函数，数据格式为 spi_message*/
int (*unprepare_transfer_hardware)(struct spi_master *master);
int (*prepare_message)(struct spi_master *master,struct spi_message *message);
int (*unprepare_message)(struct spi_master *master,struct spi_message *message);
```

```
    void (*set_cs)(struct spi_device *spi, bool enable);     /*设置片选信号*/
    int (*transfer_one)(struct spi_master *master, struct spi_device *spi,struct spi_transfer *transfer);
                                                              /*传输单个 spi_transfer 实例数据*/
    void (*handle_err)(struct spi_master *master,struct spi_message *message);

    int    *cs_gpios;     /*指向 GPIO 引脚编号数组，表示片选信号的 GPIO 编号*/

    /* DMA channels for use with core dmaengine helpers */
    struct dma_chan          *dma_tx;          /*DMA 传输通道*/
    struct dma_chan          *dma_rx;

    /* dummy data for full duplex devices */
    void                *dummy_rx;
    void                *dummy_tx;
};
```

spi_master 结构体中主要成员简介如下：

●**list**：双链表成员，将 spi_master 实例添加到全局双链表 spi_master_list。

●**bus_num**：主机控制器编号，从 0 开始编号，用于标识 SPI 主机控制器。

●**setup**：函数指针，用于修改传输速率和模式。

●**transfer**：函数指针，如果不为 NULL，则不使用消息队列，直接调用此函数传输数据（已弃用）。

●**transfer_one_message**：函数指针，控制器调用此函数传输消息队列中单个消息（spi_message 实例）。

●**transfer_one**：函数指针，在处理单个消息的函数中调用此函数传输单个 spi_transfer 实例表示的数据。

●**queue**：双链表成员，表示消息队列，管理 spi_messages 实例。

●**cs_gpios**：指向 GPIO 引脚编号数组，表示片选信号的 GPIO 编号。

●**set_cs**：函数指针，用于设置片选。

2 注册控制器

SPI 主机控制器被视为设备，挂接到某一总线上，例如 platform 总线。主机控制器驱动需要定义控制器设备的驱动，如 platform_driver。板级相关文件内需要定义主机控制器设备，如 platform_devcie。在注册表示控制器的设备时将匹配驱动，匹配成功，调用驱动探测函数，函数内完成 spi_master 实例的创建、初始化和注册。

内核提供的创建 spi_master 实例的 **spi_alloc_master()**函数定义在/drivers/spi/spi.c 文件内：

```
struct spi_master *spi_alloc_master(struct device *dev, unsigned size)
/*dev：控制器父设备的 device 实例指针，不能为 NULL，size：接在 spi_master 之后的数据长度*/
{
    struct spi_master    *master;

    if (!dev)     /*父设备不能为 NULL*/
        return NULL;

    master = kzalloc(size + sizeof(*master), GFP_KERNEL);     /*为实例分配内存空间*/
    if (!master)
        return NULL;
```

```
    device_initialize(&master->dev);       /*初始化 device 成员*/
    master->bus_num = -1;
    master->num_chipselect = 1;
    master->dev.class = &spi_master_class;          /*设备类*/
    master->dev.parent = get_device(dev);     /*父设备*/
    spi_master_set_devdata(master, &master[1]);   /*master[1]表示 master 实例之后的内存空间*/
                                        /*master.dev.driver_data=master[1]，/include/linux/spi/spi.h*/
    return master;
}
```

spi_alloc_master()函数中 dev 参数表示 spi_master 实例父设备的 device 实例指针，函数分配的内存空间前半部分为 spi_master 实例，后半部分长度为 size，表示主机控制器驱动私有数据。master.dev.driver_data 成员指向私有数据，表示驱动程序中使用的数据。

SPI 主机控制器设备关联的设备类为 spi_master_class（初始化函数中将注册），定义如下：

```
static struct class spi_master_class = {
    .name           = "spi_master",           /*名称*/
    .owner          = THIS_MODULE,
    .dev_release    = spi_master_release,         /*释放 spi_master 实例的函数*/
};
```

在创建 spi_master 实例后，主机控制器驱动的 probe()函数还需要对其进行进一步的初始化，然后将其注册到 SPI 控制器驱动框架中，注册函数为 **spi_register_master()**，定义如下（/drivers/spi/spi.c）：

```
int spi_register_master(struct spi_master *master)
{
    static atomic_t     dyn_bus_id = ATOMIC_INIT((1<<15) - 1);
    struct device       *dev = master->dev.parent;          /*父设备*/
    struct boardinfo    *bi;
    int             status = -ENODEV;
    int             dynamic = 0;

    if (!dev)
        return -ENODEV;

    status = of_spi_register_master(master);      /*从设备树节点中提取片选引脚编号*/
    if (status)
        return status;

    if (master->num_chipselect == 0)
        return -EINVAL;

    /*如果实例中没有指定控制器编号，且存在设备树节点*/
    if ((master->bus_num < 0) && master->dev.of_node)
```

```
        master->bus_num = of_alias_get_id(master->dev.of_node, "spi");   /*从设备树节点获取编号*/

    if (master->bus_num < 0) {          /*如果 master->bus_num 小于 0，则动态分配控制器编号*/
        master->bus_num = atomic_dec_return(&dyn_bus_id);
        dynamic = 1;
    }

    INIT_LIST_HEAD(&master->queue);
    spin_lock_init(&master->queue_lock);
    spin_lock_init(&master->bus_lock_spinlock);
    mutex_init(&master->bus_lock_mutex);
    master->bus_lock_flag = 0;
    init_completion(&master->xfer_completion);          /*初始化完成量*/
    if (!master->max_dma_len)
        master->max_dma_len = INT_MAX;

    dev_set_name(&master->dev, "spi%u", master->bus_num);   /*控制器设备名称：spiX*/
    status = device_add(&master->dev);              /*添加设备*/
    ...

    if (master->transfer)       /*master->transfer 不为空表示不使用队列，直接由该函数完成数据传输*/
        dev_info(dev, "master is unqueued, this is deprecated\n");      /*已经弃用的方法*/
    else {                          /*使用消息队列*/
        status = spi_master_initialize_queue(master);
                                    /*初始化主机控制器消息队列，见下一小节，/drivers/spi/spi.c*/
        ...
    }

    mutex_lock(&board_lock);
    list_add_tail(&master->list, &spi_master_list);   /*spi_master 实例添加到全局双链表末尾*/
    list_for_each_entry(bi, &board_list, list)
        spi_match_master_to_boardinfo(master, &bi->board_info);     /*/drivers/spi/spi.c*/
                    /*扫描全局 boardinfo 双链表，查找匹配项并创建 spi_device 实例，见下文*/
    mutex_unlock(&board_lock);

    of_register_spi_devices(master);          /*注册设备树中 SPI 总线设备*/
    acpi_register_spi_devices(master);
done:
    return status;
}
```

注册 spi_master 实例函数的主要工作是初始化实例，创建消息队列（master->transfer 为 NULL），将实例添加到全局双链表末尾，查找匹配的 spi_board_info 实例，为匹配实例创建 **spi_device** 实例并添加到 SPI 总线。另外，如果支持设备树，还需要扫描设备树 spi 控制器节点下设备节点，完成设备的添加。

总之，SPI 主机控制器驱动探测函数主要工作是调用 spi_alloc_master()函数创建 spi_master 实例，然后对 spi_master 实例进行设置（如函数指针成员等），最后调用 spi_register_master()函数注册 spi_master 实例。

8.7.3 数据传输

SPI 主机控制器需要提供数据传输的功能，数据传输接口函数调用主机控制器数据传输函数实现数据的传输，需要传输的数据由 **spi_message** 结构体（SPI 消息）封装，然后提交给主机控制器，由其负责数据的传输。

如果主机控制器定义了 master->transfer()函数，则提交的消息由此函数负责传输，消息不由消息队列管理，这个方法已经弃用。如果主机控制器没有定义 master->transfer()函数，则主机控制器会创建消息队列，由队列管理消息，控制器从队列中提取消息执行传输。

1 消息队列

SPI 总线传输的数据由 spi_message 结构体表示，通常提交的 spi_message 实例插入到主机控制器管理的消息队列。在注册主机控制器时将初始化消息队列，控制器负责消息队列中消息的传输。

■数据结构

spi_messages 结构体定义如下（/include/linux/spi/spi.h）：

```
struct spi_message {
    struct list_head    transfers;    /*双链表头，管理 spi_transfers 实例*/

    struct spi_device   *spi;         /*指向请求消息传输的 SPI 设备（见下文）*/
    unsigned        is_dma_mapped:1;
    void            (*complete)(void *context);    /*传输完成的回调函数*/
    void            *context;                      /*complete 回调函数的参数*/
    unsigned        frame_length;                  /*消息总的传输数据长度*/
    unsigned        actual_length;                 /*实际成功传输的数据长度，字节数*/
    int             status;                        /*成功为 0，否则为负的错误码*/

    struct list_head    queue;                     /*添加到主机控制器的消息队列*/
    void            *state;                        /*驱动使用*/
};
```

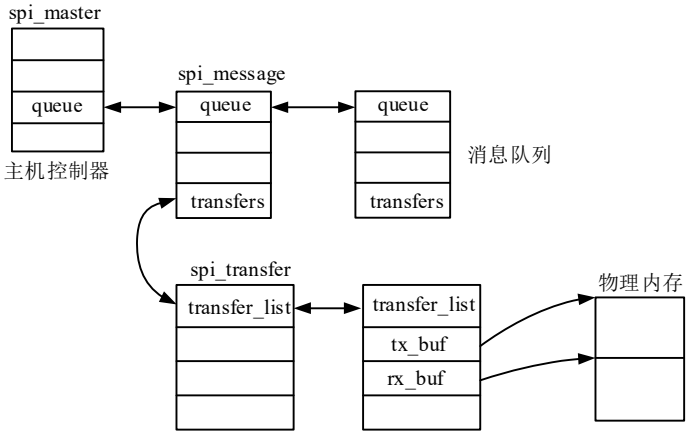
消息中实际传输的数据由 spi_transfer 结构体表示，spi_messages 结构体中双链表头 transfers 管理着 spi_transfer 实例。spi_transfer 结构体表示一段传输的数据，一个消息内可以由多段数据组成。

spi_transfer 结构体定义如下（/include/linux/spi/spi.h）：

```
struct spi_transfer {
    const void      *tx_buf;      /*发送缓存区指针*/
    void        *rx_buf;          /*接收缓存区指针*/
    unsigned    len;              /*缓存区长度，字节数*/
```

```
    dma_addr_t    tx_dma;    /*发送缓存区，DMA 地址*/
    dma_addr_t    rx_dma;    /*接收缓存区，DMA 地址*/
    struct sg_table    tx_sg;
    struct sg_table    rx_sg;

    unsigned cs_change:1;
    unsigned tx_nbits:3;
    unsigned rx_nbits:3;
  #define   SPI_NBITS_SINGLE      0x01    /* 1bit transfer */
  #define   SPI_NBITS_DUAL        0x02    /* 2bits transfer */
  #define   SPI_NBITS_QUAD        0x04    /* 4bits transfer */
    u8        bits_per_word;    /*每个字的比特位数，为 0 则采用 spi_device 的默认值*/
    u16       delay_usecs;
    u32       speed_hz;         /*选择传输速率，为 0 则采用 spi_device 的默认值*/

    struct list_head    transfer_list;    /*双链表成员，添加到 spi_message 实例中双链表*/
};
```

以上数据结构组织关系如下图所示：



■初始化消息队列

在注册主机控制器的 spi_register_master(struct spi_master *master)函数中将对主机控制器的消息队列进行初始化（前提条件是 master->transfer 为 NULL），初始化函数为 **spi_master_initialize_queue(master)**，函数定义在/drivers/spi/spi.c 文件内：

```
static int spi_master_initialize_queue(struct spi_master *master)
{
    int ret;

    master->transfer = spi_queued_transfer;        /*初始化 master->transfer 函数指针成员，队列传输*/
    if (!master->transfer_one_message)    /*如果没有定义 transfer_one_message()函数 */
        master->transfer_one_message = spi_transfer_one_message;   /*执行单个消息的传输*/
```

```
    ret = spi_init_queue(master);        /*初始化消息队列，/drivers/spi/spi.c*/
    ...
    master->queued = true;
    ret = spi_start_queue(master);       /*激活消息队列，/drivers/spi/spi.c*/
    ...
    return 0;
    ...
}
```

spi_master_initialize_queue()函数首先为控制器 transfer 和 transfer_one_message 函数指针成员赋值，请读者记住此处赋值的函数，在处理消息时需要调用这些函数；随后，调用 **spi_init_queue(master)**函数初始化消息队列，最后调用 **spi_start_queue()**函数激活对消息的处理，下面简要介绍一下这两个函数。

```
static int spi_init_queue(struct spi_master *master)
{
    struct sched_param param = { .sched_priority = MAX_RT_PRIO - 1 };

    master->running = false;
    master->busy = false;

    init_kthread_worker(&master->kworker);        /*初始化内核工作者，/include/linux/kthread.h*/
    master->kworker_task = kthread_run(kthread_worker_fn,     \
                                &master->kworker, "%s",dev_name(&master->dev));     /*创建内核线程*/
    ...
    init_kthread_work(&master->pump_messages, spi_pump_messages);

    if (master->rt) {          /*如果控制器消息需实时处理*/
        dev_info(&master->dev,"will run message pump with realtime priority\n");
        sched_setscheduler(master->kworker_task, SCHED_FIFO, &param);
                                                    /*设置处理消息线程为实时线程*/
    }

    return 0;
}
```

SPI 消息的处理采用 kthread_worker 机制（见第 5 章），简单地说就是建立内核工作者 kthread_worker 实例，它通过双链表管理需要完成的工作 kthread_work 实例，创建一个内核线程，线程内扫描 kthread_work 实例链表，调用其中的 func()函数，完成相关的工作。

这里使用的内核工作者是 spi_master.**kworker**，spi_master.**pump_messages** 是工作 kthread_work 实例，添加到工作者链表，其执行函数为 **spi_pump_messages()**，spi_master.**kworker_task** 指向内核线程结构实例。

spi_start_queue(master)函数用于将 spi_master.pump_messages 添加到工作者 spi_master.kworker 中双链表，并激活创建的内核线程，由内核线程调用工作中的 spi_pump_messages()函数处理消息队列中消息，函数源代码请读者自行阅读。

■**消息传输**

在初始化消息队列时，spi_master.pump_messages 工作中处理函数设为 **spi_pump_messages()**，用于处理消息队列中消息，函数定义如下：

```
static void spi_pump_messages(struct kthread_work *work)
/*work：指向工作实例*/
{
    struct spi_master *master =container_of(work, struct spi_master, pump_messages);

    __spi_pump_messages(master, true);    /*/drivers/spi/spi.c*/
}
```

__spi_pump_messages()函数代码简列如下：

```
static void __spi_pump_messages(struct spi_master *master, bool in_kthread)
{
    unsigned long flags;
    bool was_busy = false;
    int ret;

    /*锁定消息队列*/
    spin_lock_irqsave(&master->queue_lock, flags);

    /*确保当前不在处理消息*/
    if (master->cur_msg) {
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return;
    }

    /* If another context is idling the device then defer */
    if (master->idling) {
        queue_kthread_work(&master->kworker, &master->pump_messages);
        spin_unlock_irqrestore(&master->queue_lock, flags);
        return;
    }

    /*如果消息队列为空或当前主机控制器不在运行*/
    if (list_empty(&master->queue) || !master->running) {
        ...
    }

    /*从消息队列头部提取一个消息实例*/
    master->cur_msg =list_first_entry(&master->queue, struct spi_message, queue);
    list_del_init(&master->cur_msg->queue);     /*将消息从消息队列中移除*/
    if (master->busy)
```

```
            was_busy = true;
        else
            master->busy = true;        /*标记主机控制器忙*/
        spin_unlock_irqrestore(&master->queue_lock, flags);

        if (!was_busy && master->auto_runtime_pm) {
            ret = pm_runtime_get_sync(master->dev.parent);
            ...
        }

        if (!was_busy)
            trace_spi_master_busy(master);
        if (!was_busy && master->prepare_transfer_hardware) {
            ret = master->prepare_transfer_hardware(master);        /*准备传输函数*/
            ...
        }

        trace_spi_message_start(master->cur_msg);

        if (master->prepare_message) {
            ret = master->prepare_message(master, master->cur_msg);        /*准备传输消息*/
            ...
            master->cur_msg_prepared = true;
        }

        ret = spi_map_msg(master, master->cur_msg);     /*用于使用 DMA 的数据传输*/
        ...
        ret = master->transfer_one_message(master, master->cur_msg);   /*执行单个消息数据的传输*/
        ...
    }
```

__spi_pump_messages()函数简单地说就是从控制器中取出第一个消息，调用 **transfer_one_message()** 函数完成单个消息的数据传输。

在前面初始化消息队列的函数中，如果主机控制器没有定义 transfer_one_message()函数，将设为默认的 **spi_transfer_one_message()**函数，函数定义如下：

```
static int spi_transfer_one_message(struct spi_master *master,struct spi_message *msg)
{
    struct spi_transfer *xfer;
    bool keep_cs = false;
    int ret = 0;
    unsigned long ms = 1;

    spi_set_cs(msg->spi, true);     /*设置片选信号*/
```

```
        /*遍历 spi_message 实例下的 spi_transfer 实例双链表*/
        list_for_each_entry(xfer, &msg->transfers, transfer_list) {
            trace_spi_transfer_start(msg, xfer);
            if (xfer->tx_buf || xfer->rx_buf) {      /*存在发送或接收缓存区*/
                reinit_completion(&master->xfer_completion);    /*初始化完成量*/
                ret = master->transfer_one(master, msg->spi, xfer);   /*传输单个 spi_transfer 实例*/
                if (ret < 0) {
                    ...
                }

                if (ret > 0) {      /*延时*/
                    ret = 0;
                    ms = xfer->len * 8 * 1000 / xfer->speed_hz;
                    ms += ms + 100;       /* some tolerance */
                    ms = wait_for_completion_timeout(&master->xfer_completion,msecs_to_jiffies(ms));
                }
                ...
            } else {    /*发送和接收缓存区都不存在*/
                ...
            }
            ...
        }    /*遍历 spi_transfer 实例链表结束*/
        ...
        spi_finalize_current_message(master);    /*/drivers/spi/spi.c*/
                        /*当前消息处理结束，将 spi_master.pump_messages 工作重新添加到工作者
                        *spi_master.kworker 中双链，唤醒内核线程，调用 mesg->complete()函数等。
                        */
        return ret;
    }
```

spi_transfer_one_message()函数处理单个消息的流程是遍历 spi_message 实例下的 spi_transfer 实例双链表，对每个 spi_transfer 实例调用 master->transfer_one(master, msg->spi, xfer)函数进行数据传输。

最后在 spi_finalize_current_message(master)函数中会将 spi_master.pump_messages 工作重新添加到工作者 spi_master.kworker 中双链，唤醒内核线程，并调用 mesg->complete()函数等。

2 接口函数

SPI 控制器驱动为 SPI 设备驱动提供了数据传输的接口函数，设备驱动需要传输数据时，只需要调用接口函数，将需要传输的数据传递给控制器即可，由主机控制器实现数据的传输，设备驱动不必关心数据是如何实现传输的。

■写数据函数

SPI 主机控制器驱动提供的写数据函数为 spi_write()，定义如下（/include/linux/spi/spi.h）：

static inline int **spi_write**(struct **spi_device *spi**, const void *buf, size_t len)

```
/*spi：指向 SPI 设备，buf：缓存区指针，len：数据长度*/
{
    struct spi_transfer   t = {       /*创建 spi_transfer 实例*/
                .tx_buf    = buf,     /*发送缓存区*/
                .len       = len,     /*数据长度*/
            };
    struct spi_message  m;       /*消息实例*/

    spi_message_init(&m);     /*初始化消息实例*/
    spi_message_add_tail(&t, &m);     /*spi_transfer 添加到 spi_message 链表末尾*/
    return spi_sync(spi, &m);       /*同步读写操作函数，见下文*/
}
```

■读数据函数

SPI 主机控制器驱动提供的读数据函数为 spi_read()，定义如下（/include/linux/spi/spi.h）：

```
static inline int spi_read(struct spi_device *spi, void *buf, size_t len)
{
    struct spi_transfer   t = {
                .rx_buf          = buf,
                .len       = len,
            };
    struct spi_message  m;

    spi_message_init(&m);
    spi_message_add_tail(&t, &m);
    return spi_sync(spi, &m);
}
```

读写函数内部都是构建 spi_transfer 和 spi_message 实例，然后调用 **spi_sync(spi, &m)**函数完成数据的传输。

spi_sync()函数定义如下（/drivers/spi/spi.c）：

```
int spi_sync(struct spi_device *spi, struct spi_message *message)
{
    return    __spi_sync(spi, message, 0);     /*/drivers/spi/spi.c*/
}
```

__spi_sync()函数定义在/drivers/spi/spi.c 文件内，代码如下：

```
static int __spi_sync(struct spi_device *spi, struct spi_message *message, int bus_locked)
{
    DECLARE_COMPLETION_ONSTACK(done);        /*定义完成量*/
    int status;
    struct spi_master *master = spi->master;       /*SPI 主机控制器*/
```

```
    unsigned long flags;

    status = __spi_validate(spi, message);     /*检测 spi_message 实例有效性*/
    if (status != 0)
        return status;

    message->complete = spi_complete;   /*传输完成的回调函数，唤醒在 done 完成量上等待的进程*/
    message->context = &done;       /*spi_complete()函数参数*/
    message->spi = spi;

    if (!bus_locked)
        mutex_lock(&master->bus_lock_mutex);

    if (master->transfer == spi_queued_transfer) {
                                        /*默认值，如果控制器驱动没有定义 transfer()函数*/
        spin_lock_irqsave(&master->bus_lock_spinlock, flags);
        trace_spi_message_submit(message);

        status = __spi_queued_transfer(spi, message, false);   /*将消息添加到消息队列尾部，返回 0*/

        spin_unlock_irqrestore(&master->bus_lock_spinlock, flags);
    } else {          /*主机控制器驱动定义了 transfer()函数，弃用的方法*/
        status = spi_async_locked(spi, message);     /*异步执行数据传输，调用 master->transfer()函数*/
    }

    if (!bus_locked)
        mutex_unlock(&master->bus_lock_mutex);

    if (status == 0) {     /*处理消息*/
        if (master->transfer == spi_queued_transfer)     /*使用消息队列*/
            __spi_pump_messages(master, false);     /*处理消息队列，见上文*/

        wait_for_completion(&done);     /*等待消息传输完成调用 spi_complete()函数唤醒当前进程*/
        status = message->status;
    }
    message->context = NULL;
    return status;
}
```

__spi_sync()函数将 spi_message 实例添加到主机控制器消息队列末尾，调用__spi_pump_messages()函数执行消息的传输，这里首先处理的是消息队列的第一个消息，而不是本次提交的消息，处理完后会继续添加 spi_master.pump_messages 工作，处理队列中后面的消息。

调用 spi_sync()函数的进程将进入等待睡眠（同步传输），当提交的消息传输完成时，再唤醒当前进程。

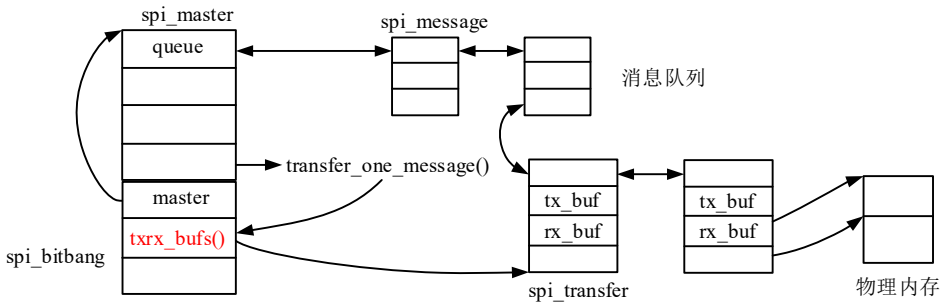
内核在/include/linux/spi/spi.h 头文件内还定义了创建、添加、同步消息的接口函数等，请读者自行阅读。

8.7.4 控制器驱动示例

本小节以龙芯 1B 开发板为例，介绍其 SPI 主机控制器驱动的实现。龙芯 1B 芯片内置两个 SPI 主机控制器，外部引脚可复用。SPI 主机控制器驱动套用了 Bitbang 框架，下面介绍 SPI 控制器驱动的实现。

1 Bitbang 框架

SPI 控制器驱动中提供了 Bitbang 框架用于采用 GPIO 模拟 SPI 控制器的情形或某些 SPI 控制器，进行逐字或 spi_transfer 实例的数据传输。Bitbang 框架如下图所示：



使用 Bitbang 框架时，在调用 spi_alloc_master()函数分配 spi_master 实例时，在其后附上 **spi_bitbang** 结构体实例。spi_bitbang 结构体中包含选择片选信号、实现 spi_transfer 实例数据传输的函数指针等成员。spi_master 实例中的 tranfer_one_message()函数赋值为 **spi_bitbang_transfer_one()**函数指针，而不是采用默认的函数，函数内调用 spi_bitbang 结构体中的 **txrx_bufs()**函数完成消息数据的传输。

spi_bitbang 结构体定义在/include/linux/spi/spi_bitbang.h 头文件内：

```
struct spi_bitbang {
    spinlock_t      lock;
    u8              busy;
    u8              use_dma;
    u8              flags;          /*额外的 spi->mode 支持*/

    struct spi_master  *master;     /*指向主机控制器 spi_master 实例*/
    int   (*setup_transfer)(struct spi_device *spi,struct spi_transfer *t);    /*更改时钟、字位数等*/
    void (*chipselect)(struct spi_device *spi, int is_on);      /*选择片选信号*/
    ...
    int   (*txrx_bufs)(struct spi_device *spi, struct spi_transfer *t);    /*发送接收消息函数*/

    /* txrx_word[SPI_MODE_*]() just looks like a shift register */
    u32  (*txrx_word[4])(struct spi_device *spi,unsigned nsecs,u32 word, u8 bits);
};
```

SPI 控制器驱动（probe()函数）的主要工作是实现 spi_bitbang 结构体中的函数，调用 spi_alloc_master()函数创建结构体 spi_master 和 spi_bitbang 实例，并对其进行初始化，最后调用 **spi_bitbang_start()**函数将创建的 spi_master 实例注册到内核。

spi_bitbang_start()函数定义如下（/drivers/spi/spi-bitbang.c）：

```
int spi_bitbang_start(struct spi_bitbang *bitbang)
{
    struct spi_master *master = bitbang->master;        /*SPI 主机控制器*/
    int ret;

    if (!master || !bitbang->chipselect)          /*必须定义 chipselect()函数*/
        return -EINVAL;

    spin_lock_init(&bitbang->lock);

    if (!master->mode_bits)
        master->mode_bits = SPI_CPOL | SPI_CPHA | bitbang->flags;

    if (master->transfer || master->transfer_one_message)   /*不能手工设置这两个函数*/
        return -EINVAL;

    master->prepare_transfer_hardware = spi_bitbang_prepare_hardware;    /*默认的实现函数*/
    master->unprepare_transfer_hardware = spi_bitbang_unprepare_hardware;
    master->transfer_one_message = spi_bitbang_transfer_one;    /*设置单个消息传输函数*/

    if (!bitbang->txrx_bufs) {            /*若 txrx_bufs()函数指针成员为空*/
        bitbang->use_dma = 0;
        bitbang->txrx_bufs = spi_bitbang_bufs;   /*赋值默认函数*/
        if (!master->setup) {          /*没有定义的函数，赋予默认值*/
            if (!bitbang->setup_transfer)
                bitbang->setup_transfer =spi_bitbang_setup_transfer;
            master->setup = spi_bitbang_setup;
            master->cleanup = spi_bitbang_cleanup;
        }
    }

    ret = spi_register_master(spi_master_get(master));    /*注册主机控制器 spi_master 实例*/
    if (ret)
        spi_master_put(master);

    return 0;
}
```

spi_bitbang_start()函数中将对 spi_bitbang 实例中各函数指针成员赋值，spi_master 实例中传输单个消息的函数设为 **spi_bitbang_transfer_one()**，函数定义如下：

```
static int spi_bitbang_transfer_one(struct spi_master *master,struct spi_message *m)
{
    struct spi_bitbang   *bitbang;
    unsigned        nsecs;
```

```
struct spi_transfer   *t = NULL;
unsigned        cs_change;
int             status;
int             do_setup = -1;
struct spi_device     *spi = m->spi;       /*SPI 总线设备*/

bitbang = spi_master_get_devdata(master);     /*master.dev.driver_data=bitbang*/

nsecs = 100;
cs_change = 1;
status = 0;

list_for_each_entry(t, &m->transfers, transfer_list) {     /*扫描消息实例中的 spi_transfer 实例链表*/
    /* override speed or wordsize? */
    if (t->speed_hz || t->bits_per_word)
        do_setup = 1;

    /* init (-1) or override (1) transfer params */
    if (do_setup != 0) {
        if (bitbang->setup_transfer) {
            status = bitbang->setup_transfer(spi, t);       /*启动数据传输*/
            if (status < 0)
                break;
        }
        if (do_setup == -1)
            do_setup = 0;
    }

    if (cs_change) {            /*使能片选信号*/
        bitbang->chipselect(spi, BITBANG_CS_ACTIVE);       /*设置片选信号有效*/
        ndelay(nsecs);          /*延时 100 纳秒*/
    }
    cs_change = t->cs_change;
    if (!t->tx_buf && !t->rx_buf && t->len) {
        status = -EINVAL;
        break;
    }

    if (t->len) {               /*执行传输数据*/
        if (!m->is_dma_mapped)
            t->rx_dma = t->tx_dma = 0;
        status = bitbang->txrx_bufs(spi, t);    /*传输 spi_transfer 实例中数据*/
    }
    if (status > 0)
```

```
                m->actual_length += status;
            if (status != t->len) {
                if (status >= 0)
                    status = -EREMOTEIO;
                break;
            }
            status = 0;

            /* protocol tweaks before next transfer */
            if (t->delay_usecs)
                udelay(t->delay_usecs);

            if (cs_change && !list_is_last(&t->transfer_list, &m->transfers)) {
                ndelay(nsecs);
                bitbang->chipselect(spi, BITBANG_CS_INACTIVE);   /*消息传输完了，片选失效*/
                ndelay(nsecs);
            }
        }       /*消息内数据传输结束*/

        m->status = status;

        if (!(status == 0 && cs_change)) {       /*片选失效*/
            ndelay(nsecs);
            bitbang->chipselect(spi, BITBANG_CS_INACTIVE);
            ndelay(nsecs);
        }
        spi_finalize_current_message(master);       /*完成消息数据传输完成的工作，见上文*/
        return status;
    }
```

spi_bitbang_transfer_one()函数遍历消息实例中的 spi_transfer 实例链表，对每个 spi_transfer 实例调用**bitbang->txrx_bufs(spi, t)**函数执行数据的传输。

2 驱动程序示例

龙芯 1B 处理器具有两个 SPI 主机控制器，因此在板级文件中定义了两个对应的 platform_device 实例：

```
static struct platform_device ls1x_spi0_device = {
    .name           = "spi_ls1x",
    .id         = 0,
    .num_resources      = ARRAY_SIZE(ls1x_spi0_resource),
    .resource = ls1x_spi0_resource,             /*资源*/
    .dev        = {
        .platform_data= &ls1x_spi0_platdata,       //&ls1x_spi_devices，平台数据
    },
```

```
};

static struct platform_device ls1x_spi1_device = {
    .name           = "spi_ls1x",
    .id         = 1,
    .num_resources      = ARRAY_SIZE(ls1x_spi1_resource),
    .resource = ls1x_spi1_resource,
    .dev        = {
        .platform_data= &ls1x_spi1_platdata,//&ls1x_spi_devices,
    },
};
```

在内核启动初始化子系统时会调用板级定义的初始化函数，初始化函数中会将 platform_device 实例添加到 platform 总线。

SPI 主机控制器驱动定义如下：

```
static struct platform_driver ls1x_spi_driver = {
    .probe = ls1x_spi_probe,       /*探测函数*/
    .remove = __devexit_p(ls1x_spi_remove),
    .driver = {
        .name = DRV_NAME,         /*"spi_ls1x"*/
        .owner = THIS_MODULE,
        .pm = NULL,
        .of_match_table = ls1x_spi_match,
    },
};

static int __init ls1x_spi_init(void)           /*初始化函数*/
{
    return platform_driver_register(&ls1x_spi_driver);       /*注册平台驱动*/
}
module_init(ls1x_spi_init);
```

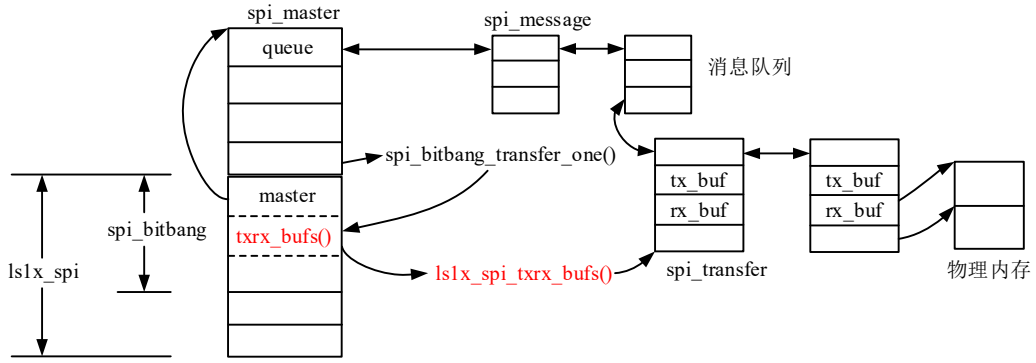
在添加/注册平台设备/驱动时会匹配总线上的驱动/设备，匹配成功将调用驱动的 probe()函数，此处为 **ls1x_spi_probe()**函数，函数内将会创建 spi_master 和 spi_bitbang 实例，并初始化，最后将 spi_master 实例注册到内核。

龙芯 1B 处理器 SPI 主机控制器驱动程序中定义了如下数据结构：

```
struct ls1x_spi {
    struct spi_bitbang    bitbang;      /*必须是第一个成员*/
    struct completion    done;        /*完成量*/

    void __iomem *base;          /*控制寄存器基地址*/
    int irq;
    unsigned int div;
    unsigned int speed_hz;
```

```
    unsigned int mode;
    unsigned int len;
    unsigned int txc, rxc;
    const u8 *txp;
    u8 *rxp;
#ifdef CONFIG_SPI_CS_USED_GPIO
    unsigned int gpio_cs_count;
    int *gpio_cs;
#endif
    struct clk *clk;
};
```

龙芯 1B 处理器 SPI 主机控制器驱动中数据结构组织关系如下图所示：



在控制器驱动的探测函数 **ls1x_spi_probe()**中将创建以上数据结构实例，函数代码简列如下：

```
static int __devinit ls1x_spi_probe(struct platform_device *pdev)
{
    struct ls1x_spi_platform_data *platp = pdev->dev.platform_data;
    struct ls1x_spi *hw;
    struct spi_master *master;
    struct resource *res;
    int err = -ENODEV;
#ifdef CONFIG_SPI_CS_USED_GPIO
    unsigned int i;
#endif

    master = spi_alloc_master(&pdev->dev, sizeof(struct ls1x_spi));
        /*spi_master 实例之后是 ls1x_spi 实例，且 master.dev.driver_data=ls1x_spi*/
    if (!master)
        return err;

    /* setup the master state. */
    master->bus_num = pdev->id;
    master->num_chipselect = 32;
    master->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
    master->setup = ls1x_spi_setup;
```

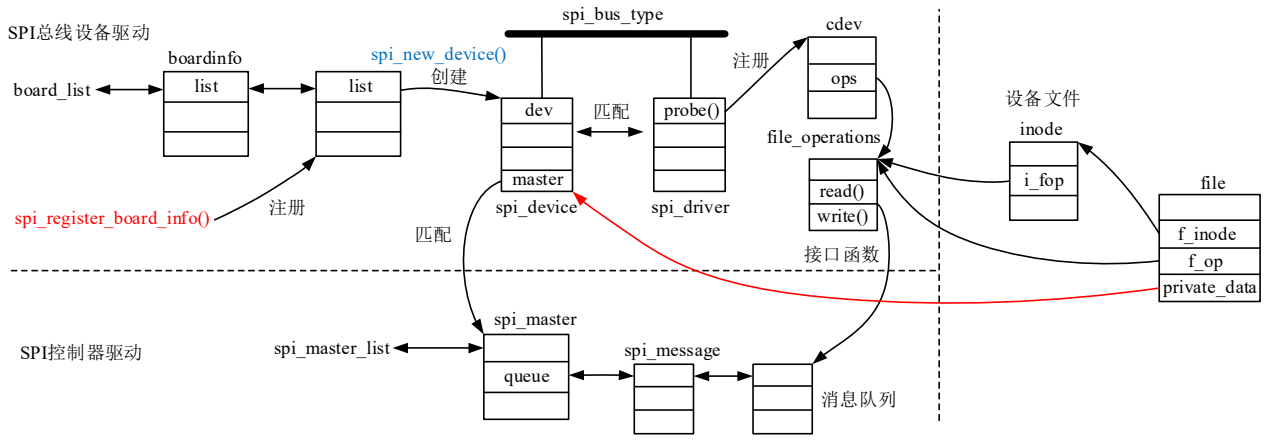
```
    hw = spi_master_get_devdata(master);      /*指向 spi_master 之后的 ls1x_spi 实例*/
    platform_set_drvdata(pdev, hw);     /*pdev->dev.driver_data=ls1x_spi*/

    /* setup the state for the bitbang driver */
    hw->bitbang.master = spi_master_get(master);          /*指向主机控制器*/
    if (!hw->bitbang.master)
        return err;
    hw->bitbang.setup_transfer = ls1x_spi_setup_transfer;        /*启动数据传输*/
    hw->bitbang.chipselect = ls1x_spi_chipselect;        /*片选*/
    hw->bitbang.txrx_bufs = ls1x_spi_txrx_bufs;   /*实现 spi_transfer 实例中数据传输的函数*/
    ...

    /*设置并注册 spi_master 实例等*/
    err = spi_bitbang_start(&hw->bitbang);
    ...
    return 0;
    ...
}
```

spi_bitbang 实例中的操作函数，通过操作 SPI 主机控制器的寄存器来实现，这里就不详细介绍了，有兴趣的读者可参考处理器手册阅读相关源代码。

8.7.5 设备与驱动

前面介绍了 SPI 主机控制器的驱动，下面介绍 SPI 总线设备驱动，驱动框架如下图所示。

内核定义了 SPI 总线实例 spi_bus_type，SPI 总线设备由 spi_device 结构体表示，SPI 总线设备驱动由 spi_driver 结构体表示。



SPI 设备的注册与前面介绍的平台总线设备的注册有所不同，板级相关代码中注册的是 spi_board_info 结构体实例。spi_board_info 结构体中包含 SPI 设备使用的 SPI 主机控制器的信息，如使用 SPI 控制器的片选信号、最大传输速率等。

spi_board_info 实例在内核中由双链表管理，在注册 spi_board_info 实例的 **spi_register_board_info()**函数中将查找 spi_board_info 实例匹配的 spi_master 实例。若匹配成功将创建表示 SPI 总线设备的 **spi_device** 实例（调用 spi_new_device()函数），并关联到主机控制器 spi_master 实例。spi_device 实例还将挂载到 SPI 总线，并查找匹配的设备驱动 **spi_driver** 实例。若与驱动匹配成功将调用其中的 probe()函数，在函数内将

完成字符设备（或块设备）数据结构实例的创建和注册。

注册 spi_driver 实例的函数为 **spi_register_driver(sdrv)**，函数内调用 SPI 总线的匹配函数查找总线下匹配的 spi_device 实例，匹配成功则调用 spi_driver 实例的 probe()函数，完成总线设备驱动程序的加载。

SPI 总线设备驱动中文件操作结构中的函数调用 SPI 主机控制器驱动提供的接口函数，通过向控制器提交消息来完成数据的传输。在打开 SPI 总线设备文件时，将会建立 file 实例与表示设备的 spi_device 实例之间的关联，以便在数据传输操作中将数据发送给正确的主机控制器和设备。

1 数据结构

SPI 总线设备由 spi_device 结构体表示，设备驱动由 spi_driver 结构体表示，在初始化时注册了 SPI 总线实例。

■设备与驱动

SPI 总线设备与驱动的数据结构定义如下（/include/linux/spi/spi.h）：

```
struct spi_device {
    struct device           dev;          /*内嵌 device 实例，将 spi_device 实例添加到通用驱动模型*/
    struct spi_master  *master;           /*指向主机控制器*/
    u32             max_speed_hz;         /*最大传输速率*/
    u8              chip_select;          /*片选信号编号*/
    u8              bits_per_word;
    u16             mode;                 /*数据传输模式*/
    ...             /*mode 成员取值*/
    int             irq;                  /*中断编号*/
    void            *controller_state;    /*主机控制器运行时状态*/
    void            *controller_data;     /*板级相关定义的控制器信息*/
    char            modalias[SPI_NAME_SIZE];    /*匹配驱动的名称，与驱动 spi_device_id 列表匹配*/
    int             cs_gpio;              /*片选信号使用的 GPIO 编号（可选）*/
};
```

SPI 总线设备驱动数据结构定义如下：

```
struct spi_driver {
    const struct    spi_device_id *id_table;    /*设备匹配列表，/include/linux/mod_devicetable.h*/
    int             (*probe)(struct spi_device *spi);     /*探测设备函数，由 driver.probe()调用*/
    int             (*remove)(struct spi_device *spi);    /*解绑设备*/
    void            (*shutdown)(struct spi_device *spi);  /*关机回调函数*/
    struct device_driver  driver;             /*内嵌 device_driver 实例*/
};
```

spi_device_id 结构体用于表示设备匹配列表，定义在/include/linux/mod_devicetable.h 头文件：

```
#define   SPI_NAME_SIZE       32
#define   SPI_MODULE_PREFIX   "spi:"

struct spi_device_id {
```

```
    char name[SPI_NAME_SIZE];     /*匹配字符串*/
    kernel_ulong_t    driver_data;        /* Data private to the driver */
};
```

在匹配设备与驱动时，将比较 spi_device.modalias 与 spi_driver.id_table 指向的列表成员，名称相同表示设备与驱动匹配成功。

■总线

SPI 总线实例定义在/drivers/spi/spi.c 文件内：

```
struct bus_type spi_bus_type = {
    .name       = "spi",
    .dev_groups     = spi_dev_groups,          /*默认的总线下设备属性*/
    .match          = spi_match_device,        /*匹配函数，/drivers/spi/spi.c*/
    .uevent         = spi_uevent,              /*添加环境变量，/drivers/spi/spi.c*/
};
```

SPI 总线实例中没有定义探测 probe()函数，在注册 spi_driver 实例时会对其 spi_driver.driver.probe()函数赋值，所赋函数将调用 spi_driver.probe()函数。

下面主要看一下总线匹配函数，函数定义如下（/drivers/spi/spi.c）：

```
static int spi_match_device(struct device *dev, struct device_driver *drv)
{
    const struct spi_device    *spi = to_spi_device(dev);      /*spi 设备*/
    const struct spi_driver    *sdrv = to_spi_driver(drv);     /*spi 驱动*/

    /* Attempt an OF style match */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then try ACPI */
    if (acpi_driver_match_device(dev, drv))
        return 1;

    if (sdrv->id_table)        /*驱动设置了 id_table 列表，则只匹配列表，不匹配驱动名称了*/
        return !!spi_match_id(sdrv->id_table, spi);     /*列表匹配*/

    return strcmp(spi->modalias, drv->name) == 0;     /*最后进行名称匹配，spi 别名与驱动名称*/
}
```

SPI 总线匹配函数与平台总线匹配函数类似，主要是最后的列表匹配有所区别。

列表匹配函数 spi_match_id()定义如下：

```
static const struct spi_device_id *spi_match_id(const struct spi_device_id *id,const struct spi_device *sdev)
{
    while (id->name[0]) {
```

```
            if (!strcmp(sdev->modalias, id->name)) /*设备别名与驱动 spi_device_id 列表成员名称匹配*/
                return id;
            id++;
        }
        return NULL;
    }
```

spi_match_id()函数内将 spi_device 结构体 modalias[]成员与 spi_device_id 列表项成员的名称字符串比较，相同则表示匹配成功，不相同则不匹配。

如果驱动没有定义 id_table 列表，最后会将 spi_device.modalias[]与驱动名称进行比较，相同表示匹配，不相同表示不匹配。

■初始化

内核在初始化后期调用 spi_init()函数对 SPI 总线驱动进行初始化，函数定义如下（/drivers/spi/spi.c）：

```
static int __init spi_init(void)
{
    int     status;

    buf = kmalloc(SPI_BUFSIZ, GFP_KERNEL);   /*申请缓存空间，/drivers/spi/spi.c*/
    ...
    status = bus_register(&spi_bus_type);              /*注册 SPI 总线*/
    if (status < 0)
        goto err1;

    status = class_register(&spi_master_class);     /*注册 SPI 主机控制器 spi_master_class 设备类*/
    if (status < 0)
        goto err2;

    if (IS_ENABLED(CONFIG_OF_DYNAMIC))
        WARN_ON(of_reconfig_notifier_register(&spi_of_notifier));
    return 0;
     ...
}
postcore_initcall(spi_init);
```

初始化函数内主要完成总线实例的注册和主机控制器设备类的注册。

2 注册设备信息

内核定义了 spi_board_info 结构体用于表示 SPI 设备与主机控制器的连接信息，一个控制器可连接多个 SPI 设备（通过片选信号确定）。

spi_board_info 结构体定义如下（/include/linux/spi/spi.h）：

```
struct spi_board_info {
    char        modalias[SPI_NAME_SIZE];      /*设备名称，赋予 spi_device 实例，用于匹配驱动*/
```

```
    const void      *platform_data;      /*平台数据结构，用于传递硬件信息，赋予 spi_device 实例*/
    void        *controller_data;       /*主机控制器信息，赋予 spi_device 实例*/
    int         irq;                    /*中断编号*/

    /* slower signaling on noisy or low voltage boards */
    u32         max_speed_hz;       /*最大传输速率，赋予 spi_device 实例*/
    u16         bus_num;            /*主机控制器（总线）编号，用于与 spi_master 实例匹配*/
    u16         chip_select;        /*片选信号，赋予 spi_device 实例*/
    u16         mode;               /*赋予 spi_device 实例*/
};
```

内核在/drivers/spi/spi.c 文件内定义了 boardinfo 结构体：

```
struct boardinfo {
    struct list_head      list;       /*双链表成员，将 boardinfo 实例添加到全局双链表*/
    struct spi_board_info     board_info;     /*内嵌 spi_board_info 结构体成员*/
};
```

boardinfo 结构体内嵌 spi_board_info 结构体成员，list 成员用于将实例添加到全局双链表 board_list。板级相关的代码中调用 **spi_register_board_info()**函数，向 SPI 驱动框架注册 spi_board_info 实例时，将会创建 boardinfo 实例，将 spi_board_info 实例复制到 boardinfo 实例 board_info 成员；查找匹配的 spi_master 实例，匹配成功将调用 **spi_new_device()**函数为设备创建 spi_device 实例，并添加到 SPI 总线（匹配驱动）。

spi_register_board_info()函数定义如下（/drivers/spi/spi.c）：

```
int spi_register_board_info(struct spi_board_info const *info, unsigned n)
/*info：指向 spi_board_info 实例数组，n：数组中实例数量（注册的实例数）*/
{
    struct boardinfo *bi;
    int i;
    ...
    bi = kzalloc(n * sizeof(*bi), GFP_KERNEL);     /*创建 boardinfo 实例数组*/
    ...
    for (i = 0; i < n; i++, bi++, info++) {
        struct spi_master *master;
        memcpy(&bi->board_info, info, sizeof(*info));
                                        /*复制 spi_board_info 实例至 boardinfo 实例数组*/
        mutex_lock(&board_lock);
        list_add_tail(&bi->list, &board_list);          /*boardinfo 实例添加到全局链表末尾*/
        list_for_each_entry(master, &spi_master_list, list)     /*扫描 spi_master 链表找到匹配项*/
            spi_match_master_to_boardinfo(master, &bi->board_info);     /*匹配 spi_master 实例*/
        mutex_unlock(&board_lock);
    }
    return 0;
}
```

注册 spi_board_info 实例函数内部创建 boardinfo 实例数组，从参数指向数组中复制数据至 boardinfo

实例数组，并将实例添加到全局双链表。对每个 boardinfo 实例扫描 spi_master 双链表，对每个 spi_master 实例调用 spi_match_master_to_boardinfo()函数，判断是否匹配，若匹配成功则调用 **spi_new_device()**函数创建并添加 spi_device 实例。

spi_match_master_to_boardinfo()函数定义如下：

```
static void spi_match_master_to_boardinfo(struct spi_master *master,struct spi_board_info *bi)
{
    struct spi_device *dev;

    if (master->bus_num != bi->bus_num)     /*两者总线编号不相同，则不匹配，返回*/
        return;

    dev = spi_new_device(master, bi);     /*总线编号相同，匹配，创建 spi_device 实例*/
    if (!dev)
        dev_err(master->dev.parent, "can't create new device for %s\n",bi->modalias);
}
```

创建 spi_device 实例的 spi_new_device()函数定义如下：

```
struct spi_device *spi_new_device(struct spi_master *master,struct spi_board_info *chip)
{
    struct spi_device    *proxy;
    int                  status;

    proxy = spi_alloc_device(master);     /*分配 spi_device 实例，并建立与 master 关联*/
    ...
    proxy->chip_select = chip->chip_select;       /*初始化 spi_device 实例*/
    proxy->max_speed_hz = chip->max_speed_hz;
    proxy->mode = chip->mode;
    proxy->irq = chip->irq;
    strlcpy(proxy->modalias, chip->modalias, sizeof(proxy->modalias));    /*复制名称字符串*/
    proxy->dev.platform_data = (void *) chip->platform_data;       /*板级数据结构的传递*/
    proxy->controller_data = chip->controller_data;
    proxy->controller_state = NULL;       /*从 spi_board_info 获取信息*/

    status = spi_add_device(proxy);    /*向 SPI 总线添加设备，/drivers/spi/spi.c*/
    ...
    return proxy;
}
```

向 SPI 总线添加 SPI 设备的函数 spi_add_device()定义如下：

```
int spi_add_device(struct spi_device *spi)
{
    static DEFINE_MUTEX(spi_add_lock);
    struct spi_master *master = spi->master;
    struct device *dev = master->dev.parent;
```

```
    int status;

    /*片选信号是否有效*/
    if (spi->chip_select >= master->num_chipselect) {
        dev_err(dev, "cs%d >= max %d\n",spi->chip_select,master->num_chipselect);
        return -EINVAL;
    }

    spi_dev_set_name(spi);        /*设置 device 名称*/
    mutex_lock(&spi_add_lock);
    status = bus_for_each_dev(&spi_bus_type, NULL, spi, spi_dev_check);
                        /*遍历 SPI 总线中设备，调用 spi_dev_check()函数，检查不能有重复的设备*/
    ...
    if (master->cs_gpios)        /*GPIO 引脚编号*/
        spi->cs_gpio = master->cs_gpios[spi->chip_select];

    status = spi_setup(spi);    /*设置片选信号，调用 master 实例 setup()函数等，/drivers/spi/spi.c*/
    ...
    status = device_add(&spi->dev);    /*添加 SPI 设备，匹配 spi_driver 成功则调用其 probe()函数*/
    ...
done:
    mutex_unlock(&spi_add_lock);
    return status;
}
```

3 注册设备驱动

SPI 总线设备驱动由 spi_driver 实例表示，**spi_register_driver()**函数用于向内核注册驱动，函数定义如下：

```
int spi_register_driver(struct spi_driver *sdrv)
{
    sdrv->driver.bus = &spi_bus_type;      /*设置总线实例*/
    if (sdrv->probe)
        sdrv->driver.probe = spi_drv_probe;     /*设置默认函数，调用 sdrv->probe()函数*/
    if (sdrv->remove)
        sdrv->driver.remove = spi_drv_remove;     /*设置默认函数，调用 sdrv->remove()函数*/
    if (sdrv->shutdown)
        sdrv->driver.shutdown = spi_drv_shutdown;     /*设置默认函数，调用 sdrv->shutdown()函数*/
    return driver_register(&sdrv->driver);      /*注册驱动实例*/
}
```

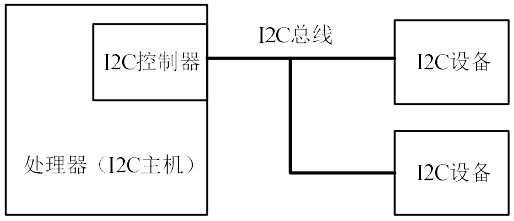
在通用的注册驱动的 driver_register()函数中将在总线中查找匹配的设备实例，若匹配成功将调用驱动中 spi_driver.driver.probe()函数，进而调用 spi_driver.probe()函数完成字符/块设备驱动数据结构实例的创建和注册。

SPI 总线设备驱动的一个例子是 SD 卡驱动程序（SPI 接口），请读者参考第 10 章。

8.8 I2C 总线

I2C 总线是一种由 Philips 公司开发的两线式串行总线，用于连接处理器与外围设备。在物理结构上，I2C 总线由数据线 SDA 和时钟 SCL 构成，每个器件都有一个唯一的识别地址。发送数据到总线的器件叫作发送器，从总线接收数据的器件称为接收器。初始化发送产生时钟信号和终止发送的器件叫作主机，被主机寻址的器件称为从机。

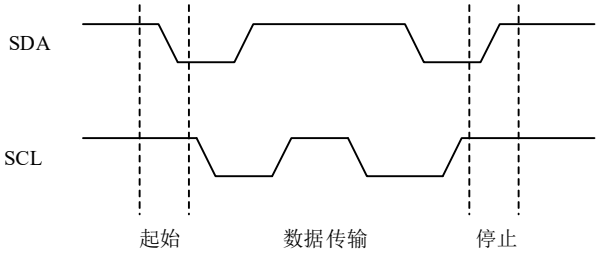
典型的 I2C 总线接口硬件架构如下图所示：



下面简单看一下 I2C 总线时序：

（1）开始/停止条件

如下图所示，在 SCL 处于高电平时，SDA 线从高电平向低电平切换表示一个开始信号（开始数据传输）。在 SCL 处于高电平时，SDA 线从低电平向高电平切换表示一个停止信号（停止数据传输）。

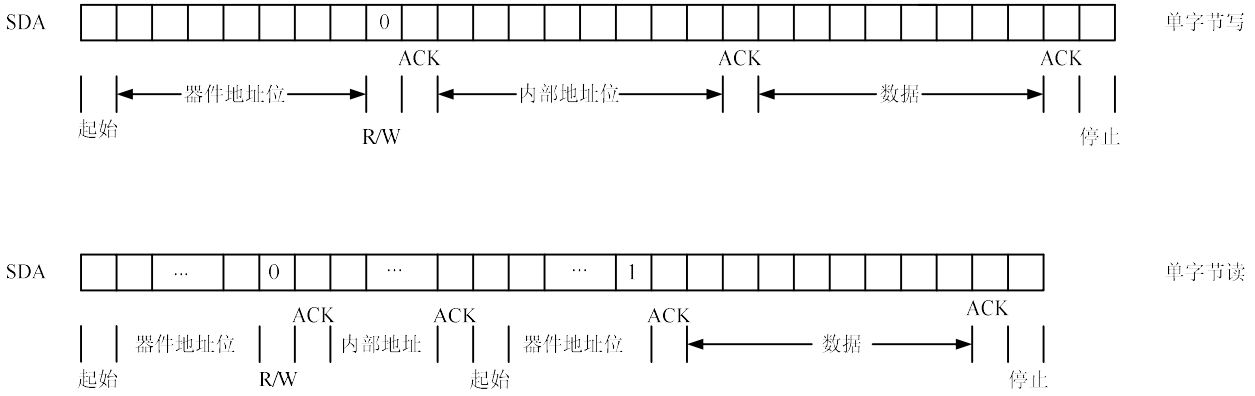


（2）数据传输

在数据传输过程中，数据线 SDA 上的数据在 SCL 时钟高电平周期内必须保持稳定，也就是说 SCL 为高电平时 SDA 线不能有电平切换，否则会被视为起始或停止条件。 SDA 线只能在 SCL 为低电平时其电平才能切换。SCL 脉冲可视为是对 SDA 线值的采样。

发送到 SDA 线上的每个字节必须是 8bit。每次传输可以发送的字节数量不受限制，每个字节后必须跟一个响应位（ACK），由接收方发送到 SDA 线上。

下图示意了单字节写/读的时序：



读写数据前都需要发送器件地址和器件内部地址，前者表示寻址总线上哪个设备，后者寻址器件内部哪个字节。器件地址由 8bit 组成，高 7bit 表示器件地址，最低位表示读写位（高为读，低为写）。

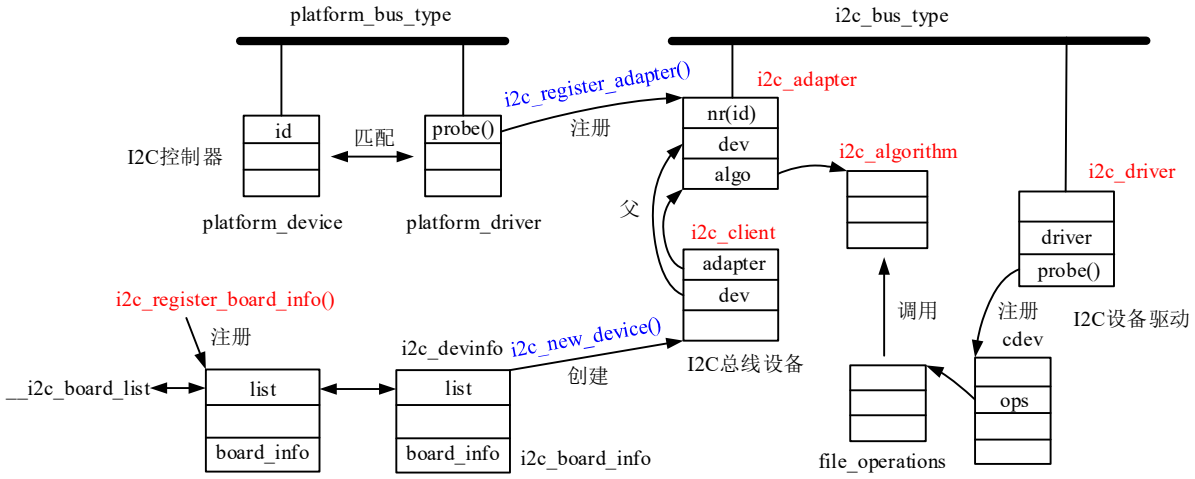
写数据时先发送器件地址，再发送内部地址，最后发送数据，每个字节后需要有应答。读数据时先发送器件地址，再发送内部地址，然后还需要重新发送起始条件，再发送器件地址，最后才能从 SDA 线上

读取数据，同样每个字节后也要有应答。

主机中的 I2C 控制器用于实现 I2C 总线的时序，实现给定数据的收发。I2C 控制器可以是标准的控制器，也可以由 GPIO 模拟。

8.8.1 概述

Linux 内核中 I2C 主机控制器驱动、设备驱动与 SPI 总线都很相似，如下图所示：



I2C 主机控制器作为一个设备挂接到平台 platform 总线上，在对应的平台设备驱动 probe()函数中创建并初始化表示 I2C 主机控制器的 **i2c_adapter** 结构体实例。i2c_adapter 结构体中包含 I2C 主机控制器在系统内的编号（来自于 platform_device）、deveice 实例、控制器传输数据操作的 **i2c_algorithm** 结构体实例等成员，i2c_adapter 实例将挂载到 I2C 总线 i2c_bus_type 实例上。

I2C 总线设备由 i2c_client 结构体表示，内嵌 device 实例，挂接到 i2c 总线上，并且主机控制器内嵌的 deveice 实例设为其父设备。i2c_client 实例并不是直接创建的，而是根据 i2c_board_info 实例创建的，板级相关代码需要创建 i2c_board_info 实例并调用接口函数 i2c_register_board_info()向内核注册（添加到全局双链表）。

在向内核注册主机控制器 i2c_adapter 实例时，将扫描 i2c_board_info 实例（实际是 i2c_devinfo 实例）双链表，对挂接到本控制器上的设备（由 i2c_board_info 实例表示）创建相应的 **i2c_client** 实例，并挂接到 i2c_bus_type 总线上。

I2C 设备驱动由 i2c_driver 结构体表示，它与 i2c_client 匹配。i2c_client 实例与 i2c_driver 实例匹配上时将调用设备驱动中的 probe()函数，函数内完成具体字符设备、块设备驱动数据结构实例的创建和注册。

设备驱动程序中定义的数据传输操作，通过 I2C 主机控制器驱动框架提供的接口函数完成，最终调用 i2c_client 实例绑定 i2c_adapter 实例关联的 **i2c_algorithm** 实例中定义的函数完成数据传输。

I2C 主机控制器驱动主要是创建 i2c_adapter 和 i2c_algorithm 实例，并向内核注册。在板级相关的代码中需要定义表示 I2C 总线设备的 i2c_board_info 实例并向内核注册，I2C 总线设备驱动主要是实现 i2c_driver 实例并向内核注册，其 probe()函数负责设备驱动程序加载。

I2C 驱动相关代码位于/drivers/i2c/目录下。

8.8.2 I2C 总线

I2C 总线实例中挂载了主机控制器、设备和设备驱动。本小节先介绍 I2C 总线实例的定义和初始化，后面小节介绍主机控制器、设备、设备驱动的定义和注册。

I2C 总线实例定义在/drivers/i2c/i2c-core.c 文件内：

```
struct bus_type i2c_bus_type = {
    .name       = "i2c",        /*总线名称*/
    .match      = i2c_device_match,     /*匹配函数，见下文*/
    .probe      = i2c_device_probe,     /*注意总线定义了探测函数，见下文*/
    .remove     = i2c_device_remove,
    .shutdown   = i2c_device_shutdown,
};
```

总线匹配函数和探测函数后面讲解总线设备和设备驱动时再作介绍。

下面先看一下 I2C 总线驱动的初始化，内核在启动阶段初始化子系统时调用 i2c_init()函数初始化 I2C 总线驱动，函数定义如下：

```
static int __init i2c_init(void)
{
    int retval;

    retval = of_alias_get_highest_id("i2c");   /*I2C 控制器 ID 编号最大值*/

    down_write(&__i2c_board_lock);
    if (retval >= __i2c_first_dynamic_bus_num)
        __i2c_first_dynamic_bus_num = retval + 1;
    up_write(&__i2c_board_lock);

    retval = bus_register(&i2c_bus_type);     /*注册总线实例，成功返回 0*/
    if (retval)
        return retval;
#ifdef CONFIG_I2C_COMPAT
    i2c_adapter_compat_class = class_compat_register("i2c-adapter");
    if (!i2c_adapter_compat_class) {
        retval = -ENOMEM;
        goto bus_err;
    }
#endif
    retval = i2c_add_driver(&dummy_driver);
                    /*向总线添加设备驱动 dummy_driver 实例（此驱动什么也不做）*/
    if (retval)
        goto class_err;

    if (IS_ENABLED(CONFIG_OF_DYNAMIC))
        WARN_ON(of_reconfig_notifier_register(&i2c_of_notifier));

    return 0;
    ...
}
postcore_initcall(i2c_init);
```

i2c_init()函数主要是将 i2c_bus_type 实例注册到内核，并注册 dummy_driver 设备驱动。

8.8.3 I2C 主机控制器

系统中每个 I2C 主机控制器由 i2c_adapter 结构体实例表示，挂接到 i2c_bus_type 总线上，i2c_adapter 结构体中包含实现数据传输的 i2c_algorithm 结构体实例指针。

I2C 主机控制器驱动主要工作是创建 i2c_adapter 和 i2c_algorithm 结构体实例，并将 i2c_adapter 实例挂接到 i2c_bus_type 总线。

1 数据结构

I2C 主机控制器 i2c_adapter 结构体定义如下（/include/linux/i2c.h）：

```
struct i2c_adapter {
    struct module *owner;
    unsigned int class;                  /*I2C 总线类型编号，/include/linux/i2c.h*/
    const struct i2c_algorithm *algo;   /*指向控制器数据传输的数据结构*/
    void *algo_data;       /*i2c_algorithm 实例私有数据结构*/

    struct rt_mutex bus_lock;     /*实时互斥量*/

    int timeout;                 /*超时时间（jiffies）*/
    int retries;
    struct device dev;           /*内嵌 device  实例*/

    int nr;                      /*I2C 控制器编号，系统内可能有多个 I2C 主机控制器*/
    char name[48];               /*名称*/
    struct completion dev_released;       /*完成量*/

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;     /*链接表示总线下设备的 i2c_client 实例*/

    struct i2c_bus_recovery_info *bus_recovery_info;   /*/include/linux/i2c.h*/
    const struct i2c_adapter_quirks *quirks;     /*/include/linux/i2c.h*/
};
```

i2c_adapter 结构体中主要成员简介如下：

●**class**：I2C 总线类型，定义如下（/include/linux/i2c.h）：

```
#define   I2C_CLASS_HWMON          (1<<0)  /* lm_sensors, ... */
#define   I2C_CLASS_DDC            (1<<3)  /* DDC bus on graphics adapters */
#define   I2C_CLASS_SPD            (1<<7)  /* Memory modules */
#define   I2C_CLASS_DEPRECATED        (1<<8)
```

●**algo**：i2c_algorithm 结构体指针，表示控制器数据传输的结构体，见下文。

●**algo_data**：i2c_algorithm 结构体中操作函数使用的私有数据结构指针。

●**dev**：内嵌 device 实例，将 i2c_adapter 实例挂接到 I2C 总线。
●**nr**：主机控制器编号，系统中可能存在多个 I2C 主机控制器，用于标识主机控制器（总线编号）。
●**userspace_clients**：双链表成员，链接表示总线下设备的 i2c_client 实例。

i2c_algorithm 结构体中主要包含主机控制器进行数据传输的函数指针成员，定义如下：

```
struct i2c_algorithm {
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,int num);
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,unsigned short flags, char read_write,
                       u8 command, int size, union i2c_smbus_data *data);

    u32 (*functionality) (struct i2c_adapter *);   /*返回控制器支持的特性*/

  #if IS_ENABLED(CONFIG_I2C_SLAVE)      /*控制器可作为从设备*/
    int (*reg_slave)(struct i2c_client *client);
    int (*unreg_slave)(struct i2c_client *client);
  #endif
};
```

i2c_algorithm 结构体中主要操作函数指针成员简介如下：

●**functionality**：返回控制器支持的特性，返回的标记值定义在/include/uapi/linux/i2c.h 头文件，例如：

```
#define   I2C_FUNC_I2C                    0x00000001
#define   I2C_FUNC_10BIT_ADDR             0x00000002
#define   I2C_FUNC_PROTOCOL_MANGLING      0x00000004   /* I2C_M_IGNORE_NAK etc. */
#define   I2C_FUNC_SMBUS_PEC              0x00000008
#define   I2C_FUNC_NOSTART                0x00000010   /* I2C_M_NOSTART */
#define   I2C_FUNC_SLAVE                  0x00000020
...
```

●**master_xfer**：控制器作为主设备传输数据的函数，传输数据由 i2c_msg 结构体表示，num 表示 i2c_msg 实例的数量，这是 I2C 主机控制器驱动需要实现的主要函数。

i2c_msg 结构体用于表示要传输的数据，结构体定义在/include/uapi/linux/i2c.h 头文件：

```
struct i2c_msg {
    __u16 addr;   /*从设备地址*/
    __u16 flags;    /*标记，取值定义如下*/
    #define   I2C_M_TEN         0x0010   /*10 位寻址，写数据*/
    #define   I2C_M_RD          0x0001   /*主设备自从设备读数据*/
    #define   I2C_M_STOP        0x8000   /* if I2C_FUNC_PROTOCOL_MANGLING */
    #define   I2C_M_NOSTART     0x4000       /* if I2C_FUNC_NOSTART */
    #define   I2C_M_REV_DIR_ADDR  0x2000   /* if I2C_FUNC_PROTOCOL_MANGLING */
    #define   I2C_M_IGNORE_NAK    0x1000   /* if I2C_FUNC_PROTOCOL_MANGLING */
    #define   I2C_M_NO_RD_ACK     0x0800   /* if I2C_FUNC_PROTOCOL_MANGLING */
    #define   I2C_M_RECV_LEN      0x0400   /* length will be first received byte */
    __u16 len;          /*buf 指向缓存区字节数量，须小于 64KB*/
```

```
    __u8 *buf;          /*读写数据缓存区指针*/
};
```

2 数据传输

I2C 主机控制器驱动提供了执行数据传输的接口函数，函数内调用 i2c_algorithm 结构体中 **master_xfer()** 函数传输数据，接口函数供具体 I2C 总线设备驱动程序调用。具体 I2C 总线设备驱动不需要关心数据是如何进行传输的，只需要调用接口函数传输数据即可。传输的数据由 i2c_msg 结构体封装。

发送接收数据接口函数如下：

●int **i2c_transfer**(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)：传输由 num 个 i2c_msg 实例组成的数据，返回传输的 i2c_msg 实例数量，函数内调用 adap->algo->master_xfer(adap, msgs, num)函数。消息的传输没有进行排队处理，而是在持有锁（实时互斥锁）的情况下直接进行传输。

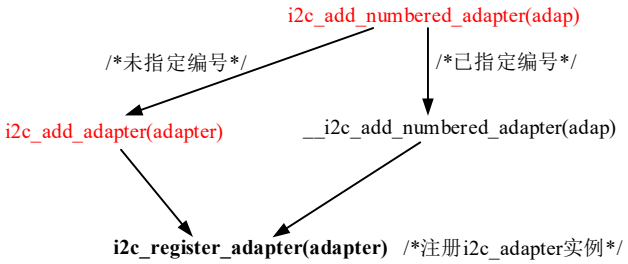
●int **i2c_master_send**(const struct i2c_client *client, const char *buf, int count)：发送（写）数据，buf 指向发送数据缓存区，count 表示发送数据字节数（小于 64KB），返回发送数据字节数。函数内构建一个消息 i2c_msg 实例，调用 i2c_transfer()函数发送数据。

●int **i2c_master_recv**(const struct i2c_client *client, char *buf, int count)：接收（读）数据，buf 指向接收数据缓存区，count 表示接收数据字节数（小于 64KB），返回实际接收数据字节数。函数内构建一个消息 i2c_msg 实例，调用 i2c_transfer()函数接收数据。

3 注册控制器

在板级相关的文件中需要为 I2C 主机控制器创建 platform_device 实例，并向内核注册。用户需要在驱动程序目录/drivers/i2c/busses/下添加主机控制器驱动，驱动程序中主要是创建 platform_driver 实例，并注册。platform_driver 实例的 probe()函数中需要完成控制器硬件的初始化，创建和初始化 i2c_adapter 实例，并向内核注册。另外还有一项最重要的工作就是实现 i2c_algorithm 实例，定义其中的数据传输函数，并将实例指针赋予 i2c_adapter 实例。

向内核注册 i2c_adapter 实例的接口函数调用关系如下：



i2c_add_adapter()与 i2c_add_numbered_adapter()接口函数的区别在于前者添加的 i2c_adapter 实例中未指定编号（nr 成员值为-1），而由内核分配编号，后者添加的 i2c_adapter 实例中指定了编号，并保存在 nr 成员中。两个接口函数最后都是调用 **i2c_register_adapter**(struct i2c_adapter *adap)函数向内核注册主机控制器 i2c_adapter 实例，函数代码简列如下（/drivers/i2c/i2c-core.c）：

```
static int i2c_register_adapter(struct i2c_adapter *adap)
{
    int res = 0;
    ...
    /*安全性、有效性检查*/

    rt_mutex_init(&adap->bus_lock);
```

```
    mutex_init(&adap->userspace_clients_lock);
    INIT_LIST_HEAD(&adap->userspace_clients);  /*初始化双链表成员*/

    if (adap->timeout == 0)
        adap->timeout = HZ;  /*超时时间设为 1s*/

    dev_set_name(&adap->dev, "i2c-%d", adap->nr);    /*设置内嵌 device 名称*/
    adap->dev.bus = &i2c_bus_type;    /*总线类型*/
    adap->dev.type = &i2c_adapter_type;  /*设备类型，定义了一组默认属性*/
    res = device_register(&adap->dev);    /*注册设备*/
    if (res)
        goto out_list;

    dev_dbg(&adap->dev, "adapter [%s] registered\n", adap->name);

    pm_runtime_no_callbacks(&adap->dev);

#ifdef CONFIG_I2C_COMPAT
    ...
#endif

    /* bus recovery specific initialization */
    if (adap->bus_recovery_info) {
        struct i2c_bus_recovery_info *bri = adap->bus_recovery_info;

        if (!bri->recover_bus) {
            dev_err(&adap->dev, "No recover_bus() found, not using recovery\n");
            adap->bus_recovery_info = NULL;
            goto exit_recovery;
        }

        /* Generic GPIO recovery */
        if (bri->recover_bus == i2c_generic_gpio_recovery) {
            if (!gpio_is_valid(bri->scl_gpio)) {
                dev_err(&adap->dev, "Invalid SCL gpio, not using recovery\n");
                adap->bus_recovery_info = NULL;
                goto exit_recovery;
            }

            if (gpio_is_valid(bri->sda_gpio))
                bri->get_sda = get_sda_gpio_value;
            else
                bri->get_sda = NULL;
```

```
                bri->get_scl = get_scl_gpio_value;
                bri->set_scl = set_scl_gpio_value;
            } else if (!bri->set_scl || !bri->get_scl) {
                /* Generic SCL recovery */
                dev_err(&adap->dev, "No {get|set}_gpio() found, not using recovery\n");
                adap->bus_recovery_info = NULL;
            }
        }

exit_recovery:
        /* create pre-declared device nodes */
        of_i2c_register_devices(adap);    /*查找设备树中传递的 I2C 设备创建并挂接 i2c_client 实例*/
        acpi_i2c_register_devices(adap);
        acpi_i2c_install_space_handler(adap);

        if (adap->nr < __i2c_first_dynamic_bus_num)
            i2c_scan_static_board_info(adap);
                /*扫描 i2c_board_info 实例，为控制器下设备创建并挂接 i2c_client 实例，见下一小节*/

        /* Notify drivers */
        mutex_lock(&core_lock);
        bus_for_each_drv(&i2c_bus_type, NULL, adap, __process_new_adapter);
          /*遍历总线下驱动，调用__process_new_adapter()函数探测驱动匹配的设备，见下一小节*/
        mutex_unlock(&core_lock);

        return 0;
        ...
}
```

注册 i2c_adapter 实例函数的主要工作是将表示控制器的 adap->dev 实例添加到 i2c_bus_type 总线，设备类型设为 i2c_adapter_type；为连接到控制器的设备创建 i2c_client 实例并添加到总线，设备信息可能来自设备树文件、i2c_board_info 实例双链表等。I2C 控制器下连接的设备由 i2c_client 实例表示，实例是动态创建的，控制器设备为挂接设备的父设备。

4 控制器驱动示例

下面以龙芯 1B 开发板源代码中 I2C 主机控制器驱动为例说明 I2C 控制器驱动的实现（源代码来自于开发板资料）。I2C 主机控制器驱动中定义了 ls1x_i2c 结构体，如下：

```
struct ls1x_i2c {
    void __iomem *base;      /*控制寄存器基地址*/
    struct i2c_adapter adap;     /*内嵌 i2c_adapter 实例*/
};
```

驱动程序中定义的 i2c_algorithm 结构体实例如下，主要包含数据传输的函数：

```
static const struct i2c_algorithm ls1x_algorithm = {
```

```
    .master_xfer    = ls1x_xfer,      /*通过操作控制寄存器实现数据传输，请读者自行阅读源代码*/
    .functionality  = ls1x_functionality,    /*返回控制器特性*/
};
```

I2C 主机控制器驱动 platform_driver 实例定义如下：

```
static struct platform_driver ls1x_i2c_driver = {
    .probe          = ls1x_i2c_probe,      /*探测函数，初始化并注册 I2C 控制器实例*/
    .remove         = __devexit_p(ls1x_i2c_remove),
    .suspend  = ls1x_i2c_suspend,
    .resume         = ls1x_i2c_resume,
    .driver         = {
        .owner      = THIS_MODULE,
        .name       = "ls1x-i2c",     /*名称用于匹配 platform_device 实例*/
    },
};
```

探测函数 **ls1x_i2c_probe()**负责创建 ls1x_i2c 结构体实例，并初始化和注册 i2c_adapter 实例。

在内核初始化时将调用 ls1x_i2c_init()函数注册主机控制器驱动 **ls1x_i2c_driver** 实例。

```
static int __init ls1x_i2c_init(void)
{
    return platform_driver_register(&ls1x_i2c_driver);   /*注册 ls1x_i2c_driver 实例*/
}
arch_initcall(ls1x_i2c_init);     /*内核初始化阶段调用此函数*/
```

在板级相关的文件中需要定义表示 I2C 控制器的 platform_device 实例并向内核注册，例如：

```
static struct platform_device ls1x_i2c0_device = {
    .name           = "ls1x-i2c",     /*用于匹配驱动*/
    .id             = 0,          /*编号 0，还有另外两个 I2C 控制器*/
    .num_resources      = ARRAY_SIZE(ls1x_i2c0_resource),
    .resource = ls1x_i2c0_resource,
};
```

在添加 platform 总线设备和驱动时将查找匹配的驱动或设备，匹配成功将调用驱动的 probe()函数为 I2C 主机控制器创建并注册 i2c_adapter 实例。

下面简要看一下驱动探测函数 ls1x_i2c_probe()的定义：

```
static int __devinit ls1x_i2c_probe(struct platform_device *pdev)
{
    struct ls1x_i2c *i2c;
    struct resource *res;
    int rc;

    res =   platform_get_resource(pdev, IORESOURCE_MEM, 0);   /*内存资源，控制寄存器地址*/
    ...
```

```
    if (!(i2c = kzalloc(sizeof(struct ls1x_i2c), GFP_KERNEL))) {        /*分配 ls1x_i2c 实例*/
        ...
    }

    if (!request_mem_region(res->start, resource_size(res), "ls1x-i2c"))        /*申请资源*/
        return -EBUSY;

    i2c->base = ioremap(res->start, resource_size(res));          /*控制寄存器基地址*/
    ...
    strlcpy(i2c->adap.name, "loongson1", sizeof(i2c->adap.name));
    i2c->adap.owner     = THIS_MODULE;
    i2c->adap.algo      = &ls1x_algorithm;
    i2c->adap.class     = I2C_CLASS_HWMON;
    i2c->adap.algo_data = i2c;
    i2c->adap.dev.parent = &pdev->dev;      /*父设备*/
    i2c->adap.nr = pdev->id;

    ls1x_i2c_hwinit(i2c);           /*硬件初始化*/
    platform_set_drvdata(pdev, i2c);        /*驱动数据指向 ls1x_i2c 实例*/

    rc = i2c_add_numbered_adapter(&i2c->adap);          /*注册 i2c_adapter 实例*/
    ...
    return 0;
    ...
}
```

8.8.4 设备与驱动

I2C 总线设备由 i2c_client 结构体表示，挂接到 I2C 总线上，其父设备是表示主机控制器的 i2c_adapter 实例，设备驱动由 i2c_driver 实例表示。i2c_client 与 i2c_driver 匹配时将调用 i2c_driver 内的 probe()函数，注册设备驱动程序。

1 I2C 设备

在板级相关的文件中应当定义 i2c_board_info 结构体实例用于表示 I2C 总线设备并向内核注册（或通过设备树传递设备信息），在注册控制器 i2c_adapter 实例时，将为连接到控制器的设备创建 i2c_client 实例，并挂载到 I2C 总线。

■添加设备信息

表示 I2C 总线设备的 i2c_client 结构体定义如下（/include/linux/i2c.h）：

```
struct i2c_client {
    unsigned short flags;           /*标记*/
    unsigned short addr;            /*设备地址（低 7bit）*/
```

```
    char name[I2C_NAME_SIZE];     /*名称*/
    struct i2c_adapter *adapter;   /*指向主机控制器 i2c_adapter 实例*/
    struct device dev;          /*内嵌 device 实例*/
    int irq;                    /*中断编号*/
    struct list_head detected;     /*双链表成员，将实例添加到 i2c_driver 实例中的双链表*/
  #if IS_ENABLED(CONFIG_I2C_SLAVE)
    i2c_slave_cb_t slave_cb;       /* callback for slave mode*/
  #endif
};
```

i2c_client 结构体主要成员简介如下：

●**flags**：标记，取值定义如下：

```
#define   I2C_CLIENT_PEC      0x04          /* Use Packet Error Checking */
#define   I2C_CLIENT_TEN      0x10          /* we have a ten bit chip address */
#define   I2C_CLIENT_WAKE     0x80          /* for board_info; true iff can wake */
#define   I2C_CLIENT_SCCB     0x9000        /* Use Omnivision SCCB protocol */
```

i2c_client 结构体实例不是静态定义的，而是根据 i2c_board_info 结构体实例动态创建的。

板级相关的文件通过 i2c_board_info 结构体实例向内核传递 I2C 总线设备信息，i2c_board_info 结构体定义如下（/include/linux/i2c.h）：

```
struct i2c_board_info {
    char        type[I2C_NAME_SIZE];    /*名称，用于初始化 i2c_client.name 成员，并匹配驱动*/
    unsigned short    flags;                /*用于初始化 i2c_client.flags 成员*/
    unsigned short    addr;     /*赋予 i2c_client.addr 成员，设备地址（有保留地址，不是从 0 开始）*/
    void        *platform_data;     /*用于初始化 i2c_client.dev.platform_data*/
    struct dev_archdata *archdata;
    struct device_node *of_node;          /*设备节点*/
    struct fwnode_handle *fwnode;
    int         irq;        /*中断编号，保存至 i2c_client.irq*/
};
```

i2c_board_info 表示设备的硬件信息，其中 addr 表示设备地址，I2C 设备有一些保留的地址，不能使用，如下所示：

```
/*
* Reserved addresses per I2C specification:
*   0x00        General call address / START byte
*   0x01        CBUS address
*   0x02        Reserved for different bus format
*   0x03        Reserved for future purposes
*   0x04-0x07   Hs-mode master code
*   [0x08,0x77] 可用的地址
*   0x78-0x7b   10-bit slave addressing
*   0x7c-0x7f   Reserved for future purposes
*/
```

在板级相关文件中需要定义 i2c_board_info 实例（数组），并调用接口函数 **i2c_register_board_info()** 向内核注册，函数代码简列如下（/drivers/i2c/i2c-boardinfo.c）：

```
int __init i2c_register_board_info(int busnum,struct i2c_board_info const *info, unsigned len)
/*busnum：主机控制器编号，info：指向 i2c_board_info 数组，len：数组项数*/
{
    int status;

    down_write(&__i2c_board_lock);

    if (busnum >= __i2c_first_dynamic_bus_num)
        __i2c_first_dynamic_bus_num = busnum + 1;

    for (status = 0; len; len--, info++) {
        struct i2c_devinfo *devinfo;

        devinfo = kzalloc(sizeof(*devinfo), GFP_KERNEL);     /*创建 i2c_devinfo 实例*/
        ...

        devinfo->busnum = busnum;
        devinfo->board_info = *info;     /*复制 i2c_board_info 实例至 i2c_devinfo 实例*/
        list_add_tail(&devinfo->list, &__i2c_board_list); /*i2c_devinfo 实例插入到全局双链表末尾*/
    }
    up_write(&__i2c_board_lock);

    return status;
}
```

注册 i2c_board_info 实例（数组）函数的主要工作是将 i2c_board_info 实例转换成 i2c_devinfo 实例，并将 i2c_devinfo 实例插入到全局双链表__i2c_board_list 末尾。

i2c_devinfo 结构体定义如下（/drivers/i2c/i2c-core.h）：

```
struct i2c_devinfo {
    struct list_head      list;      /*双链表成员*/
    int                   busnum;    /*控制器编号*/
    struct i2c_board_info   board_info;     /*i2c_board_info 实例*/
};
```

全局双链表__i2c_board_list 用于管理 i2c_devinfo 实例。

■创建 i2c_client

在注册主机控制器 i2c_adapter 实例时将扫描__i2c_board_list 全局双链表，对挂接到本控制器的设备创建 i2c_client 实例，并挂接到 I2C 总线。执行函数定义如下：

```
static void i2c_scan_static_board_info(struct i2c_adapter *adapter)
{
    struct i2c_devinfo   *devinfo;
```

```
    down_read(&__i2c_board_lock);
    list_for_each_entry(devinfo, &__i2c_board_list, list) {
        if (devinfo->busnum == adapter->nr&& !i2c_new_device(adapter,&devinfo->board_info))
            ...
    }
    up_read(&__i2c_board_lock);
}
```

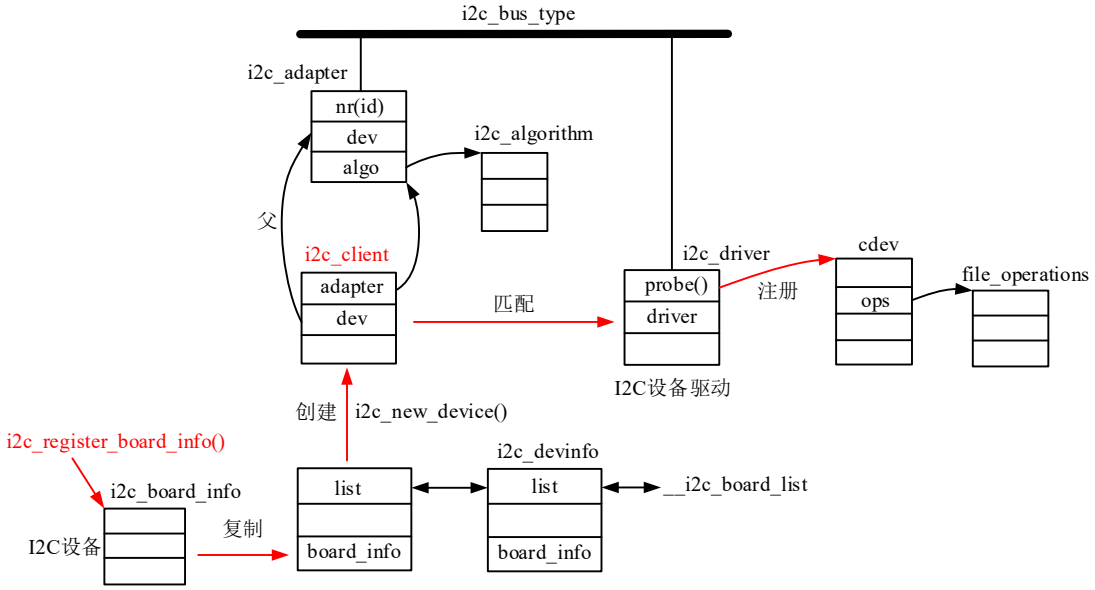
devinfo->busnum 与 adapter->nr 相等表示设备连接在本控制器上，**i2c_new_device()**函数用于创建和注册表示设备的 i2c_client 实例。

i2c_new_device()函数代码简列如下：

```
struct i2c_client *i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info)
{
    struct i2c_client    *client;
    int                  status;

    client = kzalloc(sizeof *client, GFP_KERNEL);    /*分配 i2c_client 实例*/
    if (!client)
        return NULL;

    client->adapter = adap;          /*指向主机控制器 i2c_adapter 实例*/
    client->dev.platform_data = info->platform_data;

    if (info->archdata)
        client->dev.archdata = *info->archdata;

    client->flags = info->flags;
    client->addr = info->addr;
    client->irq = info->irq;

    strlcpy(client->name, info->type, sizeof(client->name));      /*复制名称*/

    /*设备地址有效性检查*/
    status = i2c_check_client_addr_validity(client);
    ...

    /* Check for address business */
    status = i2c_check_addr_busy(adap, client->addr);
    if (status)
        goto out_err;
```

```
    client->dev.parent = &client->adapter->dev;     /*父设备*/
    client->dev.bus = &i2c_bus_type;          /*总线*/
    client->dev.type = &i2c_client_type;      /*设备类型*/
    client->dev.of_node = info->of_node;
    client->dev.fwnode = info->fwnode;

    i2c_dev_set_name(adap, client);           /*设置名称*/
    status = device_register(&client->dev);   /*注册设备*/
    ...
    return client;
    ...
}
```

注册 I2C 设备流程如下图所示（红色箭头）：



i2c_register_board_info()函数只是将 i2c_devinfo 实例添加到全局双链表，而没有创查找匹配的主机控制器，创建 i2c_client 实例。在注册主机控制器 i2c_adapter 实例时，将扫描 i2c_devinfo 实例链表，创建并注册设备 i2c_client 实例。是否意味着 i2c_devinfo 实例的注册必须在 i2c_adapter 实例的注册之前？

2 I2C 驱动

I2C 设备驱动由 i2c_driver 结构体表示，设备驱动程序中需要定义 i2c_driver 实例并向内核注册。在向内核注册 i2c_driver 实例时，将在 I2C 总线上查找匹配的 i2c_client 实例，匹配成功调用 i2c_driver 实例中的 probe()函数加载设备的驱动程序。

I2C 总线设备驱动由 i2c_driver 结构体表示，结构体定义如下（/include/linux/i2c.h）：

```
struct i2c_driver {
    unsigned int class;     /*与 adapter->class 取值相同*/

    int (*attach_adapter)(struct i2c_adapter *) __deprecated;     /*将被废弃的函数*/

    int (*probe)(struct i2c_client *, const struct i2c_device_id *);     /*探测函数，必须定义*/
```

```
    int (*remove)(struct i2c_client *);          /*解绑设备时调用的函数*/
    void (*shutdown)(struct i2c_client *);     /*设备关机时调用的函数*/
    void (*alert)(struct i2c_client *, unsigned int data);     /*改变传输协议等时调用的函数*/
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    struct device_driver driver;            /*通用 device_driver 实例*/
    const struct i2c_device_id *id_table;      /*匹配列表，用于驱动与设备匹配，必须定义*/

    int (*detect)(struct i2c_client *, struct i2c_board_info *);     /*自动探测控制器下匹配的设备*/
    const unsigned short *address_list;     /*自动探测设备地址数组*/
    struct list_head clients;   /*链接自动探测的设备的 i2c_client 实例*/
};
```

i2c_driver 结构体中主要成员简介如下：

●**probe**：设备与驱动匹配时调用此函数，必须定义，完成设备驱动程序的加载。

●**detect**：用于自动探测主机控制器下的适用本驱动的设备，自动探测的设备不需要注册 i2c_board_info 实例。

●**id_table**：i2c_device_id 结构体数组指针，用于匹配设备与驱动，必须定义，否则不能匹配设备。

i2c_device_id 结构体定义在/include/linux/mod_devicetable.h 头文件，如下：

```
struct i2c_device_id {
    char name[I2C_NAME_SIZE];     /*名称字符，用于匹配设备*/
    kernel_ulong_t driver_data;          /*驱动私有数据*/
};
```

■注册驱动

I2C 设备驱动程序需要创建 i2c_driver 实例，在 probe()函数中完成设备驱动程序的注册，然后调用函数 **i2c_register_driver()**向内核注册 i2c_driver 实例，函数定义如下：

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
{
    int res;
    ...
    driver->driver.owner = owner;
    driver->driver.bus = &i2c_bus_type;     /*总线类型*/

    res = driver_register(&driver->driver);   /*注册驱动，成功返回 0*/
    if (res)
        return res;

    pr_debug("i2c-core: driver [%s] registered\n", driver->driver.name);

    INIT_LIST_HEAD(&driver->clients);
    i2c_for_each_dev(driver, __process_new_driver);
                        /*遍历总线设备调用__process_new_driver()函数，自动探测控制器下设备*/
    return 0;
```

}

i2c_register_driver()函数主要是调用通用的 driver_register(&driver->driver)函数向内核注册驱动，然后扫描总线下设备，对主机控制器设备调用__process_new_driver()函数，自动探测主机控制器下设备。

●自动探测设备

前面介绍过，在板级相关文件内需要定义并注册表示总线设备的 i2c_board_info 实例，在注册主机控制器时，将根据 i2c_board_info 实例创建并注册表示设备的 i2c_client 实例。

I2C 设备驱动提供了自动探测控制器下设备的功能，不需要在代码中创建并注册 i2c_board_info 实例，而是在注册驱动时，对每个主机控制器调用**__process_new_driver()**函数，在此函数中调用驱动中的 **detect()** 函数，探测主机控制器下本驱动适用的设备，为其动态创建 i2c_board_info 实例，进而创建并注册表示设备的 i2c_client 实例。

__process_new_driver()函数定义如下：

```
static int __process_new_driver(struct device *dev, void *data)
/*dev：I2C 总线下设备，data：指向注册的 i2c_driver 实例*/
{
    if (dev->type != &i2c_adapter_type)   /*不是表示主机控制器的设备，跳过*/
        return 0;
    return i2c_do_add_adapter(data, to_i2c_adapter(dev));        /*对主机控制器调用此函数*/
}
```

__process_new_driver()函数只对主机控制器调用 i2c_do_add_adapter()函数，定义如下：

```
static int i2c_do_add_adapter(struct i2c_driver *driver,struct i2c_adapter *adap)
/*driver：指向 i2c_driver 实例，adap：指向 I2C 控制器 i2c_adapter 实例*/
{
    i2c_detect(adap, driver);     /*自动探测控制器下设备，创建并注册 i2c_client 实例*/

    if (driver->attach_adapter) {     /*将被废弃的函数*/
        ...
    }
    return 0;
}
```

i2c_detect()函数将调用 i2c_driver 实例中的 detect()函数，探测适用本驱动的设备（可以是多个设备），为其创建并注册 i2c_client 实例，实例还添加到 i2c_driver->clients 双链表，源代码请读者自行阅读。

3 总线匹配与探测函数

在注册表示总线的 i2c_client 实例和表示驱动的 i2c_driver 实例时，将会遍历驱动和设备，调用 I2C 总线匹配函数，检查两者的匹配性，若匹配，将会调用总线定义的 probe()函数。

■总线匹配函数

i2c_bus_type 总线匹配函数 **i2c_device_match()**定义如下：

```
static int i2c_device_match(struct device *dev, struct device_driver *drv)
```

```
{
    struct i2c_client     *client = i2c_verify_client(dev);
    struct i2c_driver     *driver;

    if (!client)
        return 0;

    if (of_driver_match_device(dev, drv))     /*基于设备树的匹配函数*/
        return 1;

    if (acpi_driver_match_device(dev, drv))
        return 1;

    driver = to_i2c_driver(drv);     /*指向 i2c_driver 实例*/
    if (driver->id_table)
        return i2c_match_id(driver->id_table, client) != NULL;
                                        /*i2c_device_id->name 与 client->name 匹配*/
    return 0;
}
```

i2c_match_id()函数将 id_table 指向的 i2c_device_id 数组项中的名称字符串与 client->name 名称比较，相同则表示匹配成功，函数返回 i2c_device_id 数组项指针，没有匹配的项返回 NULL。

注意：在 i2c_match_id()匹配函数中并没有比对 i2c_client 和 i2c_driver 名称，名称不做为检查匹配的条件，这与平台总线和 SPI 总线不同（这个总线最后会检查名称是否匹配）。

■总线探测函数

若设备 i2c_client 与驱动 i2c_driver 匹配成功，则将调用 i2c_bus_type 总线的探测函数，加载设备驱动程序。i2c_bus_type 总线实例定义了探测函数，因此会优先调用它（否则调用 device_driver 中探测函数），而平台总线和 SPI 总线实例没有定义探测函数。

I2C 总线探测函数 i2c_device_probe()定义如下：

```
static int i2c_device_probe(struct device *dev)
{
    struct i2c_client     *client = i2c_verify_client(dev);
    struct i2c_driver     *driver;
    int status;

    if (!client)
        return 0;

    if (!client->irq) {     /*如果 client->irq=0，获取中断号*/
        int irq = -ENOENT;
        if (dev->of_node)
            irq = of_irq_get(dev->of_node, 0);
        else if (ACPI_COMPANION(dev))
```

```
            irq = acpi_dev_gpio_irq_get(ACPI_COMPANION(dev), 0);

        if (irq == -EPROBE_DEFER)
            return irq;
        if (irq < 0)
            irq = 0;

        client->irq = irq;
    }

    driver = to_i2c_driver(dev->driver);    /*指向 i2c_driver 实例*/
    if (!driver->probe || !driver->id_table)        /*必须定义的成员*/
        return -ENODEV;

    if (!device_can_wakeup(&client->dev))    /*/include/linux/pm_wakeup.h*/
        device_init_wakeup(&client->dev,client->flags & I2C_CLIENT_WAKE);
                                        /*唤醒设备，/drivers/base/power/wakeup.c*/
    dev_dbg(dev, "probe\n");

    status = of_clk_set_defaults(dev->of_node, false);
    if (status < 0)
        return status;

    status = dev_pm_domain_attach(&client->dev, true);
    if (status != -EPROBE_DEFER) {
        status = driver->probe(client, i2c_match_id(driver->id_table,client));
                                                        /*调用 i2c_driver 中探测函数*/
        if (status)
            dev_pm_domain_detach(&client->dev, true);
    }
    return status;
}
```

i2c_device_probe()函数最终将调用 i2c_driver 实例中的 probe()函数，在此函数中将完成总线设备驱动程序的注册。

4 控制器驱动程序

主机控制器驱动程序是指将 I2C 主机控制器视为字符设备，导出为设备文件，用户进程通过对设备文件的读写，直接通过 I2C 总线实现数据的发送和接收。

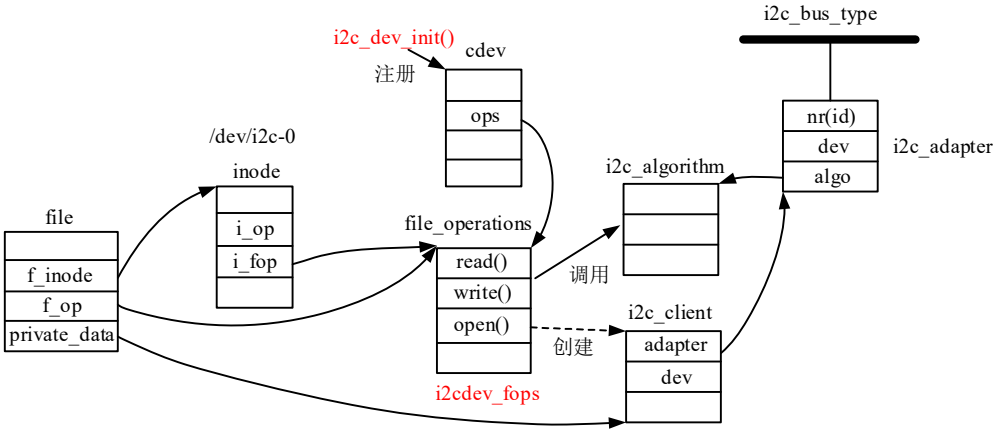
用户进程通过/dev/i2c-x 设备文件操作 I2C 主机控制器，x 表示主机控制器编号，相关驱动程序代码位于/drivers/i2c/i2c-dev.c 文件内（需选择 I2C_CHARDEV 配置选项）。

I2C 主机控制器驱动程序框架如下图所示，I2C 主机控制器视为字符设备，主设备号为 I2C_MAJOR（256），从设备号为控制器在系统内的编号。

内核在/drivers/i2c/i2c-dev.c 文件内定义了初始化函数 i2c_dev_init()用于为 I2C 控制器创建并注册 cdev

实例（字符设备驱动数据结构），为每个控制器创建设备文件/dev/i2c-x，创建设备类 i2c_dev_class 实例等。在打开主机控制器设备文件时，设备文件关联的 file_operations 实例为 i2cdev_fops，其 open()函数将为主机控制器创建 i2c_client 实例，并关联到设备文件 file 实例。

i2cdev_fops 实例中读写函数调用主机控制器驱动提供的接口函数 i2c_master_recv()/i2c_master_send()，完成总线数据的传输，相关源代码比较简单，请读者自行阅读。



8.9 USB 总线

USB（Universal Serial BUS）通用串行总线，是由 Compaq、DEC、IBM、Intel、NEC、Microsoft 等公司于是 1994 年 11 月共同提出的，主要目的是为了制定统一的 USB 标准。USB 设备使用方便，其文件传输速率快，而且具有热插拔性能，因此应用越来越广泛。常见的 USB 设备有 USB 鼠标、USB 键盘、U 盘、USB 打印机等。

虽然作者水平有限，但还是想简要讲解一下 USB 总线规范，仅当抛砖引玉，然后介绍内核中 USB 总线和 USB 设备驱动框架的实现。USB 规范文档可从 www.usb.org 网站下载。

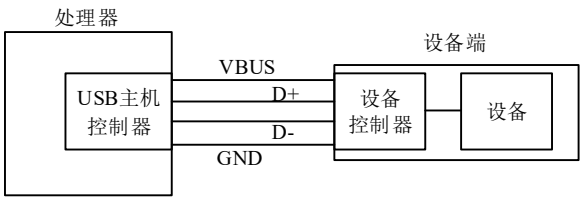
USB 总线及设备驱动源代码位于/drivers/usb/目录下，/drivers/usb/core/目录下包含 USB 总线驱动的核心代码，主要是主机控制器驱动的公共层代码，/drivers/usb/host/目录下是具体主机控制器的驱动，其它目录下是 USB 设备驱动程序。

8.9.1 总线规范

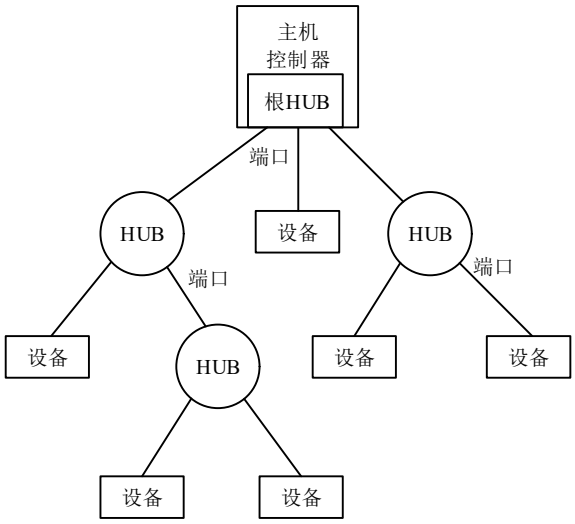
USB 总线规范已经到了 USB4，由于龙芯 1B 处理器 USB 主机控制器只支持到 USB2.0，更重要的是作者目前水平有限，因此本节仅限于讨论 USB2.0。

1 USB 总线拓扑

USB 总线使用 4 根电缆完成繁重要数据传输，如下图所示，它们分别是+5V 电源线 VBUS，地线 GND、两根差分线 D+和 D-。主机端和设备端都有一个控制器，负责实现 USB 总线规范的数据传输。设备控制器负责总线与具体设备的数据传输，设备控制器通常是一个 CPU（可带固件程序）。



USB 总线通过集线器可构成分层星型拓扑结构，如下图所示：

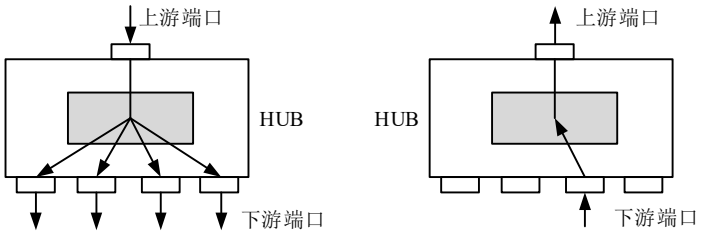


图中集线器（HUB）是一类特殊的 USB 设备，是星型拓扑结构的中心，它类似于网络中的路由器，负责接收上层的数据，并分发给下一层设备。HUB 通常有一个与上层连接的端口，多个连接下层设备的端口。主机控制器中嵌入一个 HUB，称之为根 HUB，主机通过根 HUB 提供多个端口。

通过集线器，USB 总线最多能连接 127 个设备，含集线器，因为集线器也是设备，只不过是一种特殊的设备。总线上连接的每个设备有一个 7 位的总线地址，地址 0 默认给新接入的设备使用，因此设备实际可用的地址是 127 个。主机控制器通过设备地址确定与哪个设备通信。

USB2.0 规范总共有 3 种数据传输速率：低速（Low Speed）1.5Mbit/s、全速（Full Speed）12Mbit/s、高速（High Speed）480Mbit/s（USB3.0 还有超高速）。

HUB 信号传输如下图所示，在数据向下游传输时，HUB 采用广播的方式。在数据包被检测到后，为每个接通的下游端口建立一条数据通道，使其能够接收数据。接收下游数据时，则只接通发送数据的下游端口。



HUB 必须支持 3 种速度模式。HUB 在上游端口接入高速环境时，必须工作于高速模式，上游端口接入全/低速环境时，必须工作于全/低速模式。在上下游端口接入的是同速度模式设备时，HUB 只是负责数据转发。当 HUB 工作于高速模式，下游接入全/低速设备时，HUB 内部需通过事务转换器，将高速事务分割成全/低速事务，发送给下游设备。当 HUB 工作于全/低速模式时，下游设备只能工作于于全/低速模式，不能工作于高速模式。

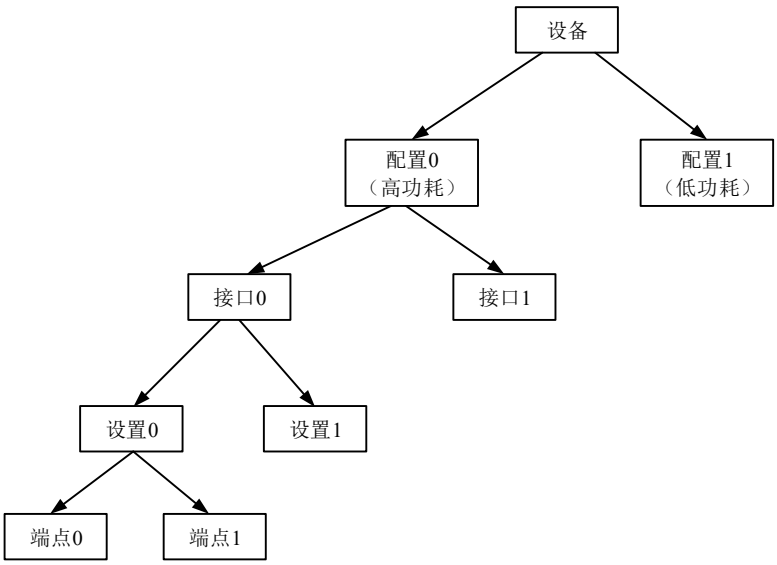
USB 设备逻辑视图如下图所示，一个 USB 设备可以有一个或多个配置状态。在任何时候只能有一个配置处于激活状态。

一个配置可以有一个或多个接口，在所处的配置被激活时，所有的接口也都处于激活状态。多重接口允许不同的主机方面的设备驱动与不同的 USB 设备的部分相关联。

接口可拥有一个或多个替代接口设置，任意时刻只能有一个接口设置被激活。一个接口设置下面拥有一个或多个端点。

设备、配置、接口、设置、端点都有对应的描述字，表示其信息，在主机枚举设备时（发送请求时）

由设备发送给主机。



每个 USB 设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个）；每个配置则由多个接口组成；接口下有多个设置；设置由多个端点组成。

每个接口代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个复杂的 USB 设备可以具有多个接口。通常说的 USB 设备驱动程序匹配的是 USB 接口。

端点是 USB 通信的最基本形式，对主机来说，每一个 USB 设备接口就是一组端点的集合。主机只有通过端点才能和设备进行通信，以使用设备的功能。在 USB 系统中每个端点都有独一无二的地址，该地址由设备地址和端点号组成。

端点就是一块典型的存储多个数据字节的缓存区。端点有一组属性，其中包括传输方向（IN/OUT）、传输类型（控制、中断、批量或等时）、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或从主机到设备（OUT 端点），或从设备到主机（IN 端点），因此端点传输是单向的。端点 0 是必须具有的端点，为控制端点，用于初始化设备参数。只要设备连接到 USB 上并且上电，端点 0 就可以被访问。端点 1、2 等一般用做数据端点，存放主机与设备间通信的数据。

2 传输类型

USB 设备插入总线时，主机要枚举（检测）设备，获取设备各描述字，为设备分配地址等，这些后面再介绍，先简要介绍一下 USB 的数据传输类型，因为检测设备时就要用到数据传输。

USB 数据传输类型共有 4 种，分别是控制传输、批量传输、中断传输和等时传输。每种类型都各具特点，适用于某种特殊的应用。

控制传输是 USB 规范所唯一规定的（必须的）传输类型，它能够使主机读取信息、设定设备地址并选取配置和采取其它安排。主机控制器枚举设备时使用的就是控制传输。

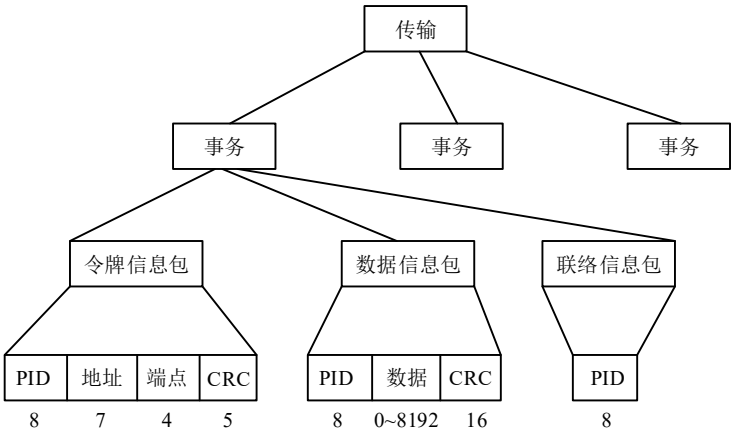
批量传输旨在那些对传输速率要求不严格的应用，如向打印机发送文件、从扫描仪接收数据或访问磁盘上的文件等工作。对于这些应用，快速传输当然更好，但若需要，数据还是可以等待的。若总线忙，则批量传输需要等待，而当总线空闲时，批量传输就是最快的传输类型。低速设备不支持批量传输端点。不要求设备一定支持批量传输，但某类设备需要这种传输。

中断传输用于那些必须周期性接收主机和其它设备动作的设备，或者用于短等待和短延时的设备。与控制传输不同，中断传输是低速设备用来传输数据的唯一方式。键盘和鼠标就使用中断传输来传递按键和鼠标动作数据。中断传输可使用任何速度模式。设备并不要求必须支持中断传输，但特定的设备可能会需要。

等时传输可保证传递时间，但不保证错误更正。使用等时传输的数据包括流音频和视频。等时传输是唯一在接到有效数据时不须回传的类型，因此使用这种传输必须允许偶尔出现错误。低速设备不支持等时传输。不要求设备必须支持等时传输，但特定设备会需要。

■传输

前面介绍的 4 种传输类型中，每个 USB 传输由一个或多个事务组成，每个事务由一个令牌信息包，一个数据信息包（可能没有）和一个联络信息包组成，如下图所示。



每个信息包开头为 PID（包标识符，8 位）。令牌信息包中包含 PID、7 位设备地址、4 位端点号和 CRC（错误检查），数据信息包中包含 PID、数据和 CRC，联络信息包中只包含 PID。

PID 提供了有关事务的信息，其值如下表所示（低 4 位为 PID，高 4 位为 PID 检测位）：

信息包类型	**PID** 名字	取值	适用传输类型	来源	总线速率	描述
令牌信息包	OUT	0001	全部	主机	全部	OUT 事务，需要设备地址和端点号
	IN	1001	全部	主机	全部	IN 事务，需要的设备地和端点号
	SOF	0101	帧开始	主机	全部	帧开始标记和帧数（计数值）
	SETUP	1101	控制	主机	全部	用于设置事务，需要设备和端点地址
数据信息包	DATA0	0011	全部	主机、设备	全部	偶数信息包 PID
	DATA1	1011	全部	主机、设备	全部	奇数信息包 PID
	DATA2	0111	等时	主机、设备	高速	数据信息包 PID
	MDATA	1111	等时、分割事务	主机、设备	高速	数据信息包 PID
联络信息包	ACK	0010	控制、批量、中断	主机、设备	全部	数据信息包接收无误（成功传输）
	NAK	1010	控制、批量、中断	设备	全部	接收端不能接收数据，或者发送端无法发送数据或无数据要发送
	STALL	1110	控制、批量、中断	设备	全部	控制请求不被支持或端点被停止
	NYET	0110	控制写、批量 OUT、分割事务	设备	高速	无误地接收了数据信息包，但还没准备好接收下一个；或者集线器还没有完成分割数据

特殊信息包	PRE	1100	控制、中断	主机	全速	主机发出的先行信号，表明下一个信息包为低速（仅低速或全速线路）
	ERR	1100	全部	集线器	高速	由集线器返回，在分割事务（仅高速线路段上）中的低速或全速错误
	SPLIT	1000	全部	主机	高速	先行于令牌信息包表明为分割事务
	PING	0100	控制写、批量 OUT	主机.	高速	批量/控制端点高速流探测
	EXT	0000	.无	主机	全部	协议拓展令牌

由上表可知，令牌信息包总是由主机发出的，也就是说每个事务总是由主机发起的，即使是中断传输也是由主机轮询发起的，而并是不像通常我们理解的设备硬件中断传输那样，由设备发给主机。

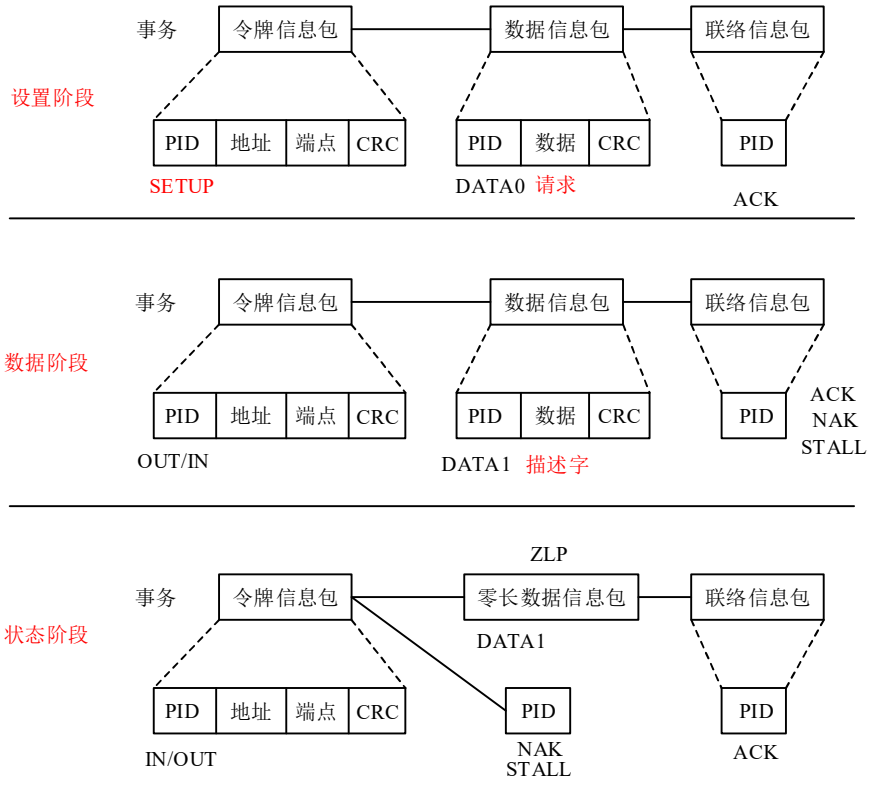
USB2.0 主机控制器通过把时间在低速、全速模式下分成 1ms 宽的帧，在高速模式下分为 125us 的微帧方式来管理传输。主机将每个帧或微帧的一部分分配给各个传输，每个事务会在一个帧或微帧内完成而不被中断，并且不会有其它信息包插到单个事务中。

在设备端点描述字中指示了端点的传输类型，如控制（端点 0）、批量、中断或等时，传输是通过端点来区分传输类型的，在传输的数据中并没有指示传输的类型。

■控制传输

控制传输有两个用途：一是对于所有设备，主机通过控制传输向设备传递用于了解和配置设备的标准请求，设备返回描述字，二是控制传输还可传递任何由类或厂商定义的请求，设备返回类或厂商描述字。每个设备都必须通过端点 0 管道（默认）完成控制传输。控制传输是 4 种传输类型中最复杂的一种。

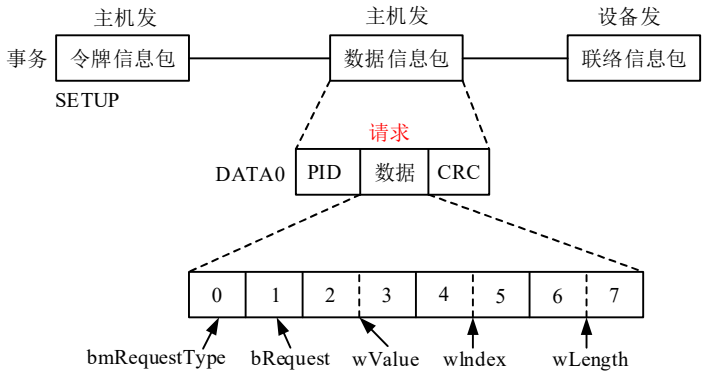
控制传输通常由 3 个阶段组成，分别是设置阶段、数据阶段和状态阶段，也可以没有数据阶段。数据阶段可以由一个或多个事务组成，设置和状态阶段包含一个事务，如下图所示。



●设置阶段

设置阶段由一个事务组成，令牌信息包 PID 为 **SETUP**。数据信息包 PID 为 DATA0，数据长度固定为 8 个字节，分成 5 个字段，主要表示请求的类型和代码。请求如果正确接收，设备返回 ACK。

数据信息包中 8 个字节的数据，5 个字段分布如下图所示：



●**bmRequestType**：1 个字节，位映射，确定数据流的方向、请求类型以及接收端。

bit7 表示数据阶段数据流的方向，0 表示（OUT 事务）从主机到设备或者无数据阶段，1 表示（IN 事务）从设备到主机。

bit6 到 bit5 表示请求类型：00 表示 USB 标准请求之一、01 表示专门的 USB 设备类定义的请求、10 表示是厂商特定请求。

bit4 到 bit0 表示请求接收者：00000 表示请求由设备接收，00001 表示由特定接口接收，00010 表示由端点接收，00011 表示设备中其他的元件。

●**bRequest**：1 个字节，请求代码（名字），受 bit6 到 bit5 表示的请求类型影响。为 00 时，bRequest 值为 USB 规范定义的 11 种标准请求；为 01 时，为 USB 设备类请求；为 10 时，为厂商特定请求。

●**wValue**：2 个字节，不同请求可传递不同含义的信息。每个请求可按照它自己的方式定义这两个字节的意义。例如，在 Set_Address()请求中，wValue 含有设备地址，在获取/设置描述字请求中，表示描述字类型和索引。

●**wIndex**：2 个字节，因请求不同意义不同，一般用于传递索引或偏移量，例如指定接口或端点。传递端点索引时，比特位 3 到 0 规定了端点号；且比特位 7 等于 0 代表控制或者 OUT 端点，等于 1 代表 IN 端点。传递接口索引时，比特位 7 到 0 是接口号。所有未被定义的比特位都是 0。

●**wLength**：2 个字节，指定本控制传输第二个阶段，即数据阶段中传输数据的长度。对于从主机到设备的传输，wLength 是主机打算传输的字节数。对于从设备到主机的传输，wLength 是可取得的最大值， 设备应返回此数目或少些的字节。如果此字段是 0，传输则没有数据阶段。

下表列出了对于 USB 规范定义的 11 种标准请求，以及字段的组合值：

bmRequestType	**bRequest** **（请求）**	**wValue**	**wIndex**	**wLength**	数据阶段传输的数据
1000 0000B 1000 0001B 1000 0010B （IN）	GET_STATUS （00h）	0	0 接口号 端点号	2	设备、接口、端点状态
0000 0000B 0000 0001B 0000 0010B	CLEAR_FEATURE （01h）	特征选择符	0 接口 端点	0	无
0000 0000B	SET_FEATURE	特征选择符	0	0	无

0000 0001B 0000 0010B	（03h）		接口号 端口号		
0000 0000B	SET_ADDRESS （05h）	设备地址	0	0	无
1000 0000B （IN）	GET_DESCRIPTOR （06h）	描述字类型和索引	0 或语言 ID	描述字长度	描述字
0000 0000B	SET_DESCRIPTOR （07h）	描述字类型和索引	0 或语言 ID	描述字长度	描述字
1000 0000B （IN）	GET_CONFIGURATION （08h）	0	0	1	配置值
0000 0000B	SET_CONFIGURATION （09h）	配置数值	0	0	无
1000 0001B （IN）	GET_INTERFACE （0Ah）	0	接口号	1	替换设置
0000 0001B	SET_INTERFACE （0Bh）	替换设置	接口号	0	无
1000 0010B （IN）	SYNCH_FRAME （0Ch）	0	端点号	2	帧数目

下面对部分请求做简要介绍：

（1）GET_STATUS：获取设备、接口、端点状态。

在数据阶段，设备返回 2 字节的状态值，对于设备、接口和端点，状态值意义不同。

设备：第一个字节的 bit0 表示设备是否自我供电，bit1 表示设备是否支持远程唤醒，其它位保留，值为 0。

接口：bit1 和 bit0 应由设备定义，其它位为 0。

端点：bit0 表示当前端点是否停止（Halt），1 表示停止，0 表示可用。bit1 可由设备定义，其它位为 0。

（2）CLEAR_FEATURE/SET_FEATURE：清除/设置设备、接口或端点特征。

接收者 （**bmRequestType**）	特征选择符（**wValue**）	接收者索引（**wIndex**）	描述
0000 0000B（设备）	DEVICE_REMOTE_WAKEUP （1）	0	远程唤醒
0000 0001B（接口）	TEST_MODE （2）	接口号	
0000 0010B（端点）	ENDPOINT_HALT （0）	端点号	暂停

（3）SET_ADDRESS：设置设备地址。

wValue 保存设备新地址，允许值为 01h-7Fh，总线上每个设备，包括集线器，都有一个总线下唯一的地址。

（4）**GET_DESCRIPTOR/SET_DESCRIPTOR**：获取/设置设备描述字。

wValue 保存描述字类型，wLength 表示在描述字的长度。标准描述字类型如表所示：

wValue	**描述字类型**	**是否需要**

0lh	设备	是
02h	配置	是
03h	字符串	否，除非驱动程序要求，为可选的描述内容
04h	接口	是
05h	端点	是，使用端点 0 以外的端点
06h	设备限定符	对于既支持全速又支持高速的设备，是。其他设备则否
07h	其他速率配置	对于既支持全速又支持高速的设备,足.其他设备则否
08h	接口功耗	否（推荐，但从未被普遍认可和实现）
09h	OTG	对于 OTG 设备，是
0Ah	调试	否
0Bh	接口联合	对某些复合设备，是
0Ch	安全性	对于无线设备
0Dh	密码钥匙	
0Eh	加密类型	
0Fh	二进制设备目标存储	对于超高速设备、无线设备以及支持链接功耗管理的设备，是
10h	设备性能	
11h	无线端点伙伴	对于无线设备
30h	超高速端点伙伴	对于超高速设备，是。对于其他设备，则否

在数据阶段的数据信息包中，将传递描述字的内容。每种类型的描述字由一个数据结构表示，所有数据结构中的第 2 个字节都是表示描述字类型的成员。具体类型描述字的定义这里就不介绍了，请读者自行参考相关资料。

下面对部分描述字的内容做简要介绍：

设备描述字：除复合设备外，所有设备有且仅有一个设备描述字，其中含有设备信息并确定了设备所支持的配置数量。设备描述字是设备连接到主机时主机读取的第一个描述字。设备描述字提供了关于设备、设备的配置以及任何设备所归属的类的信息。

配置描述字：每个配置，设有一个配置描述字，其中带有关于设备使用电源及配置所支持的接口数目的信息。设备接到对配置描述字的请求时，应当返回配置描述字，以及所有属于配置的接口、端点和其它多达所请求字节数的附属描述字。

接口描述字：每个接口，设备都有一个规定端点数目的接口描述字。接口描述字提供了关于设备所实现的功能和特性的信息。此描述字含有类、子类和协议的信息，以及接口所使用的端点数目。

接口可拥有一个或多个替代接口设置，任意时刻只能有一个接口设置被激活。每个设置含有一个接口描述字及其所需的附属描述字。接口描述字准确地说是接口设置描述字。

端点描述字：所有端点都有端点描述字，其中含有与端点通信所需的信息。没有端点描述字的接口，使用控制端点（0）完成全部通信，端点 0 从不会拥有描述字，因为所有设备必须包括端点 0。

字符串描述字：可存储诸如制造商、设备名称或者序列号等文字信息。其它描述字可能含有指向字符串描述字的索引值。

类和厂商描述字：为设备或接口提供了结构化的方式，以给出关于功能的更详细的信息。

（5）GET_CONFIGURATION/SET_CONFIGURATION：获取/设置配置。

获取配置时，wValue 值为 0，在数据阶段返回设备配置值，每个配置值由一个整数表示。

设置配置时，wValue 值表示设置设备配置。

（6）GET_INTERFACE/SET_INTERFACE：获取/设置接口。

wValue 保存设置编号（设置接口时，否则为 0），wlndex 保存接口号，获取/设置接口的设置。

（7）SYNCH_FRAME：帧同步请求。

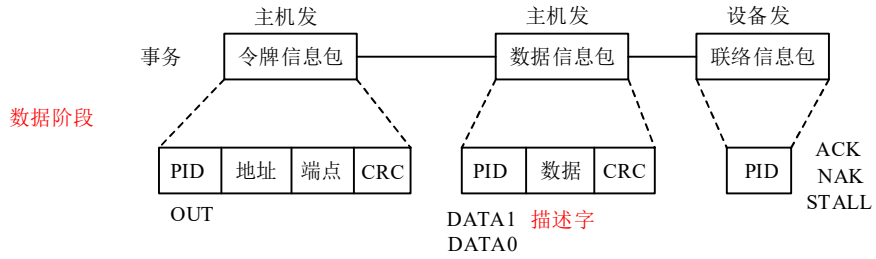
wValue 保存端点号，在数据阶段中的数据信息包中将传递帧号（2 字节），在接到帧同步请求后，端点会返回一个帧号，新的次序从这个帧号开始。

此请求很少使用，因为很少需要它所提供的信息。

●数据阶段

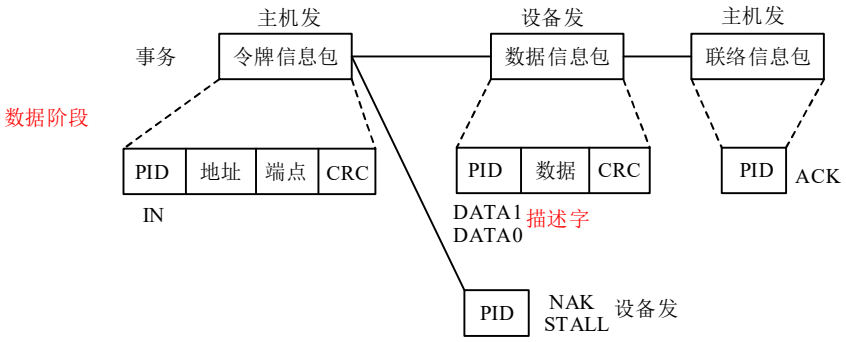
如果控制传输存在数据阶段，此阶段由一个或多个 IN 或 OUT 事务组成，如果在一个事务中不能传输所有要传输的数据则需要多个事务。在设置阶段数据信息包中的 **wLength** 字段，记录了数据阶段传输数据的长度。

数据阶段 OUT 事务如下所示（设置信息）：



令牌信息包 PID 为 OUT，数据信息包 PID 为 DATA1，若还有其它事务，则下一个事务中数据信息包 PID 为 DATA0，再下一个又为 DATA0，如此交替。联络信息包 PID 为 ACK、NAK 或 STALL。

数据阶段 IN 事务如下所示（获取信息）：

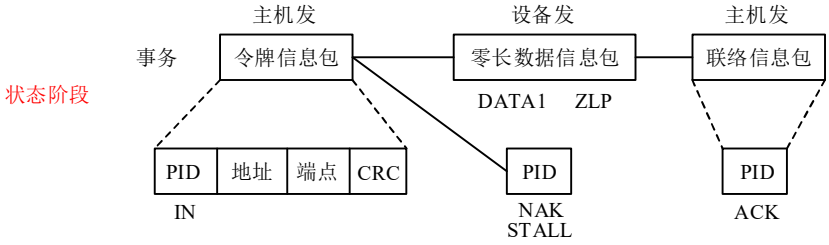


设备将会向主机发送主机请求的信息，如果设备不能发送数据，或不支持此请求或端点停止，将直接发回 NAK 或 STALL，而不会发送数据。

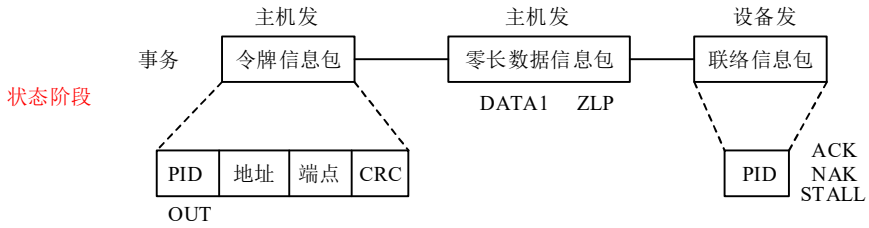
●状态阶段

状态阶段由一个 OUT 或 IN 事务组成，用于报告整个控制传输是否成功。

控制写传输（设置信息）状态阶段事务如下图所示：



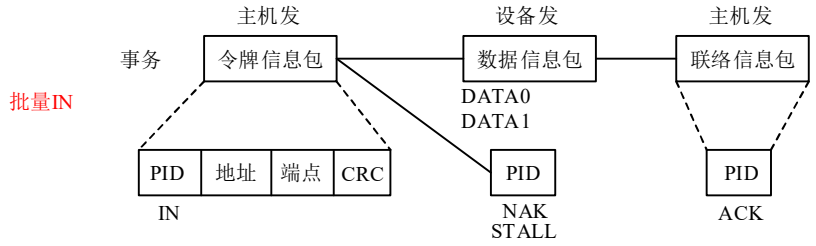
控制读传输（获取信息）状态阶段事务如下图所示：

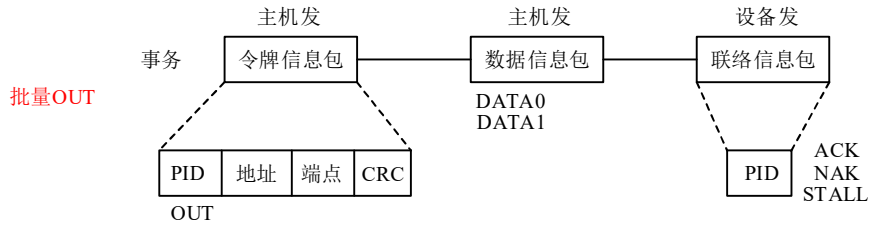


状态阶段接收端发送零长数据信息包（数据长度为 0）表示传输成功。

■其它传输类型

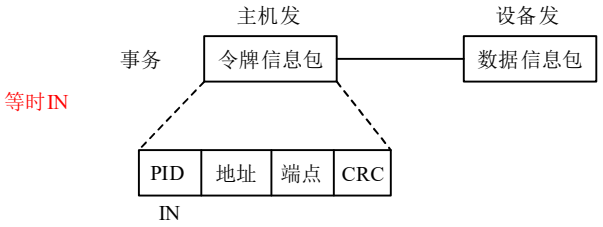
批量传输由一个或多个 IN 或 OUT 事务构成，所有数据沿同一方向传输，如下图所示：

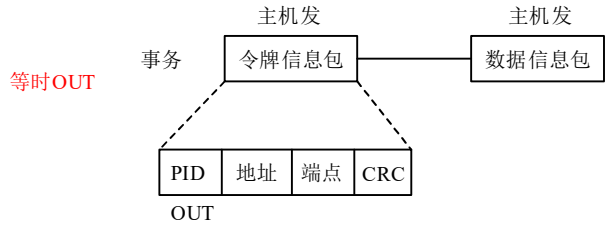




中断传输与批量传输的传输方式相同，不过中断传输并不由设备发起来，而是在主机轮询设备时发生，不同于设备触发的硬件中断。

等时传输更为简单，它不需要联络数据包，发送端不会检测接收方是否正确接收，传输方式如下图所示：





3 枚举设备

集线器的职责之一就是检测或移除处于下行端口处设备的连接。每个集线器都有一个中断 IN 端点，用来将这些事件报告给主机。系统启动时，集线器会通知主机是否有设备连接，包括额外的下行集线器和

任何连接到那些（下行）集线器的设备。开机完成后，主机将继续**周期性地轮询**（USB 2.0），来获悉有关新连接或已移除设备的信息。

获悉新设备时，主机将向设备所属的集线器发送请求（新接入设备地址为 0），使集线器建立一个处于主机和设备间的通信路径。然后主机会发布通向此设备、含有标准 USB 请求的控制传输，以尝试枚举此设备（获取设备信息）。所有的 USB 设备必须支持控制传输、标准请求和端点 0。对于成功的枚举，设备必须通过返回被请求信息的形式响应请求，并执行其他所请求的动作。

USB 2.0 规范定义了 6 个设备状态。枚举过程中，设备会先后经历开机（Powered）、缺省 （Default）、地址（Address）和配置（Configured）状态（另外两个状态是连接 Attached 状态和挂起 Suspend 状态）。在每个状态中，设备都定义了其性能和行为。

典型的 USB2.0 枚举顺序如下：

（1）系统拥有了新设备。用户将设备连接到 USB 端口，或者系统在连接有设备的情况下开机。端口可能处于主机内的根集线器上，或者处于连接主机和下行设备的其他集线器上。集线器提供端口电源，设备将处在开机状态（Powered）。设备可从总线获取 100 mA 的电流。

（2）集线器检测设备。集线器会监视其上每个端口的信号线（D+和 D-）电压。集线器在每条线上都有一个 14.25k~24.8kΩ的下拉电阻。设备还有 900~1575Ω的上拉电阻，对于全速设备该电阻在 D+上，对于低速设备则在 D-上。高速设备是以全速来连接的。在连接到某个端口时，设备的上拉电阻会将它的信号线电位抬高，使集线器检测到设备连接。检测到设备时，集线器会继续提供电源，但还不会对设备发出 USB 传输。

（3）主机获悉新设备。每个集线器都会使用它的中断端点来报告集线器上的事件。报告只表明是集线器还是端口（如果是端口，是哪一个端口）经历了事件。当获悉事件时，主机将发送给集线器一个 Get Port Status 请求，以了解更多信息。Get Port Status 请求和其他集线器请求都属于集线器所支持的标准请求。有新设备连接时，返回信息便会将此告知主机。

（4）集线器检测设备是低速还是全速。在复位（reset）设备之前，集线器通过检查两个信号线上的电压，确定设备是低速还是全速。集线器会确定闲置（idle）时哪条线有更高的电压，以此检测设备速度。然后，集线器将给主机发送信息，响应下一个 Get Port Status 请求。USB 1.x 集线器可能会在总线复位之后检测设备速度。USB 2.0 则要求在复位之前进行速度检测，以便使集线器在复位期间就知道(如下面所述的)被检测设备是否有高速能力。

（5）集线器复位设备。当主机获悉一新设备时，主机将向集线器发送一个 Set Port Feature 请求，此信号要求集线器复位端口。集线器会将设备的 USB 数据线置于复位情况下（至少 10ms）。复位是一种特殊的情况，这时 D+和 D-都为逻辑低（正常情况下，这两条线有相反的逻辑状态）。集线器只对新设备发送复位。总线上的其他集线器和设备都不会看到此复位信号。

（6）主机了解全速设备是否支持高速状态。检测设备是否支持高速状态使用两个特殊信号状态。在 Chirp J 状态只有 D+线被驱动，而在 Chirp K 状态只有 D-线被驱动。

复位期间，支持高速状态的设备会发送 Chirp K 信号。高速集线器检测到该 Chirp K 信号后，会响应一串交替的 Chirp K 与 Chirp J。当检测到 KJKJKJ 的样式时，设备会移除它的全速上拉电阻，然后以高速状态执行接下来的通信。如果集线器没有对设备的 Chirp K 信号作出响应，设备便知道它必须以全速状态继续后面的通信。所有的高速设备都必须能够响应全速状态下的控制请求。

（7）集线器在设备和总线间建立一条信号路径。主机会通过发送 Get Port Status 请求证明设备已经摆脱了复位状态。返回数据中的一个数据位将用于表明设备是否仍处于复位状态。若有必要，主机还会重复此请求直到设备离开复位状态。

当集线器的复位信号被移除时，设备便处于缺省状态（Default）。设备的 USB 寄存器都处于复位状态，且设备已准备好响应端点 0 处的控制传输。设备会使用缺省地址 **00h** 来与主机通信。

（8）主机发送 Get Descriptor 请求以了解缺省管道的最大信息包尺寸。主机向设备地址 00h，端点 0 发送此请求。由于主机每次只枚举一个设备，故也只有一个设备会响应发往设备地址 00h 的通信，即使此时有多个设备连接。设备描述字的第 8 字节含有端点 0 所支持的最大信息包尺寸。

（9）主机指定一个地址。复位完成时，主机控制器将通过发送 Set Address 请求，为设备指定唯一的地址。设备将以缺省地址完成请求的状态阶段，之后才开始执行新地址。设备现在处于地址状态（Address）。从这个端口所发出的所有通信都会指向新地址。地址会一直有效，直到设备断开连接、集线器复位端口或系统重启。下一次枚举时，主机可能会分给设备一个不同的地址。

（10）主机了解设备能力。主机向新地址发送 Get Descriptor 请求，读取设备描述字。这一次主机会取回整个描述字。描述字含有端点 0 的最大信息包尺寸、设备支持的配置数量，以及其他的设备基本信息。

主机会通过请求一个或多个指定在设备描述字内的配置描述字，继续了解设备。对配置描述字的请求，实际上就是对配置描述字和其后跟着它多达请求字节数的全部附属描述字的请求。

设备会通过发送配置描述字来做出响应，其后会跟有全部配置的附属描述字（包括接口描述字）。 而接口描述字后都跟着接口的所有端点描述字。有些配置也会有类或厂商专属描述字。

（11）主机指定并加载设备驱动程序（复合设备除外）。在根据描述字了解完设备信息后，主机会寻找最匹配的驱动程序，来管理与设备的通信。Windows 主机使用 INF 文件确认最优匹配。INF 文件可能是 USB 类的系统文件，或者厂商提供的含有厂商 ID 和产品 ID 的文件。

复合设备对这一顺序来说是个例外，它可在配置中含有指定到多个接口的不同驱动程序。主机只能在接口被使能后才可指定这些驱动程序，因此主机必须首先按下面所述的过程进行设备配置。

（12）主机的设备驱动程序会选择配置。从描述字那里了解完设备信息后，设备驱动程序会通过发送带有所需配置号的 Set Configuration 请求，来请求某一配置。许多设备只支持一种配置。若设备支持多个配置，驱动程序便可根据驱动程序所含有的设备信息，决定请求哪一配置；或者驱动程序也可询问用户或直接选择其中的第一个配置（许多驱动程序只会选择第一个配置）。接到请求时，设备会实现那个被请求的配置。至此设备就进入了（己）配置状态，设备的接口也就被使能了。

对于复合设备，主机现在就可以指定驱动程序了。而其他设备，主机使用从设备取得的信息来为每一个在配置中被激活了的接口找到驱动程序。之后，设备即为可用。

集线器也属于 USB 设备，且主机会以与其他设备相同的方式枚举新连接的集线器。如果集线器上还连接有设备，主机会在集线器通知主机它的存在后，对这些设备进行枚举。

连接状态：如果集线器不给设备的 VBUS 线提供电源，设备便处于连接状态（Attached）。若集线器检测到“过载电流”情况，或者主机请求集线器从此端口移除电源，不供给电源的情况就会发生。VBUS 上没有电源，主机和设备就不能通信，因此从它们的角度看，这与设备没有连接的情况是一样的。

挂起状态：设备会在检测到至少 3ms 没有总线动作的情况下进入挂起状态（Suspend），这里的总线动作也包括帧开端标记（SOF）。在挂起状态，设备应限制它对总线功耗的使用。配置与没配置过的设备都必须支持这种状态。

8.9.2 驱动框架

USB 总线及设备驱动程序框架如下图所示。

USB 主机控制器主要是负责实现主机与设备（端点）之间的数据传输。主机控制器由 usb_hcd 结构体表示，其关联的 hc_driver 结构体中的函数主要就是实现主机与设备（端点）之间的数据传输（按照 USB 规范执行）。USB 驱动中传输数据由 urb 结构体封装，后面将介绍 urb 的处理流程。

主机控制器驱动的主要工作就是实现 hc_driver 实例，调用内核提供的接口函数创建和添加 usb_hcd 实例。

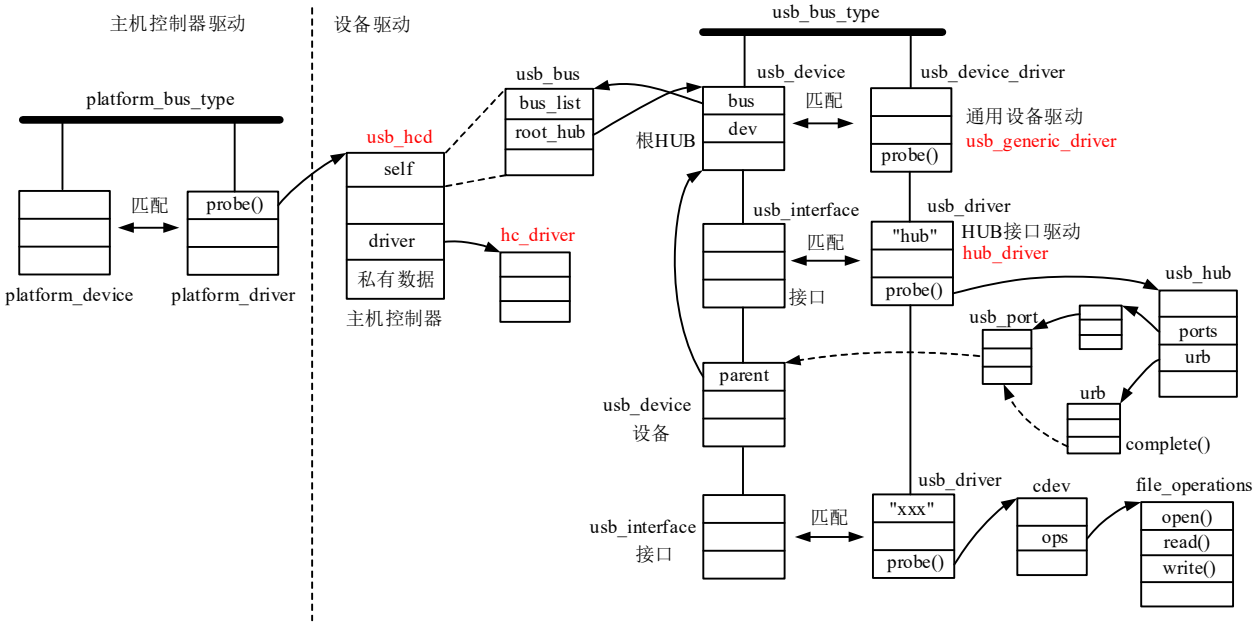
内核定义了 USB 总线实例 usb_bus_type。USB 设备由 usb_device 结构体表示，结构体中包含设备配置、

接口、端点（描述字）等信息。

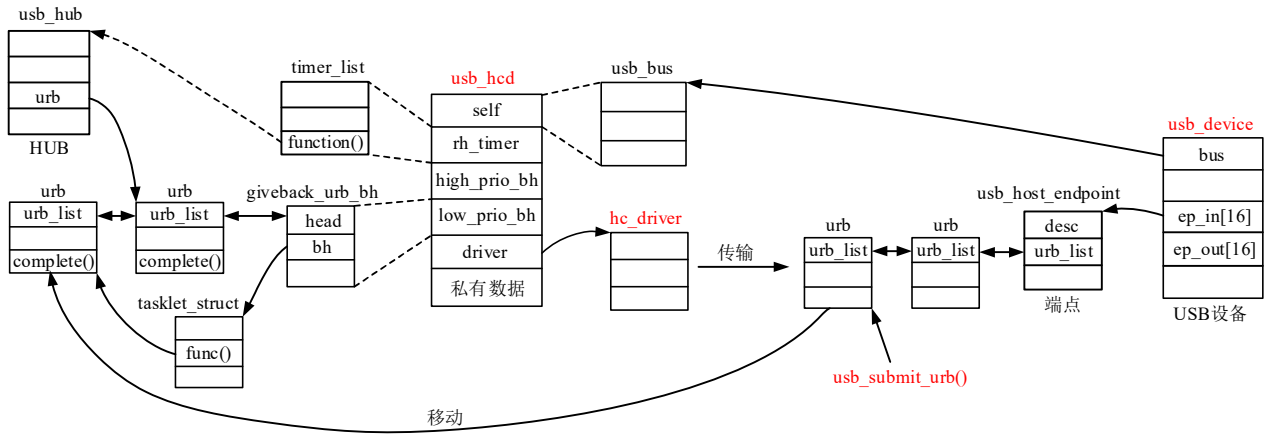
USB 主机控制器中内嵌一个根 HUB，HUB 也被视为 USB 设备。因此在添加 usb_hcd 实例时，将创建并注册表示根 HUB 的 usb_device 实例，挂载到 USB 总线上。

USB 设备 usb_device 实例将与 USB 设备驱动 usb_device_driver 实例匹配。内核定义并注册了通用设备驱动 usb_generic_driver 实例。在其 probe()函数中将选择设备配置，获取设备接口信息，为接口创建并注册 usb_interface 实例。usb_interface 实例也挂载到 USB 总线，与之匹配的是接口驱动 usb_driver 实例，在其 probe()函数中将加载接口驱动程序（请读者注意区分设备与接口，设备驱动与接口驱动）。

HUB 接口对应的驱动是 hub_driver 实例，其 probe()函数将为 HUB 创建、设置 usb_hub 结构体实例，并激活 HUB。usb_hub 结构体中包含 HUB 端口信息等。



USB 总线传输的数据由 urb 结构体封装，内核提供了分配、填充和提交 urb 实例的接口函数。提交的 urb 被添加到端点的 urb 实例双链表中，由主机控制器关联 hc_driver 实例中的函数负责数据传输（需要进行调度）。传输完成后，urb 实例被移动到 usb_hcd 实例中的 urb 实例双链表，通过 tasklet 机制处理传输完成的 urb 实例，主要是调用实例中的 complete()函数。



在创建 HUB 对应的 usb_hub 实例时，将创建一个用于中断传输的 urb 实例，用于探测设备。主机控制器 usb_hcd 实例中内嵌一个定时器（周期触发），定时器到期函数将检测根 HUB 是否有状态变化，如果有则将根 HUB 关联的中断传输 urb 添加到 usb_hcd 中的传输完成 urb 实例双链表。其 complete()函数将检测 HUB 端口的变化，枚举设备，为设备创建并添加 usb_device 实例。usb_device 实例将匹配通用 USB 设

备驱动，它为设备选择配置，获取接口信息，创建并注册 usb_interface 实例，并查找匹配的接口驱动 usb_driver 实例，在其 probe()函数中将加载用户实现的设备（接口）驱动程序，这与前面介绍的根 HUB 设备的创建相似。

8.9.3 主机控制器

USB 主机控制器主要负责主机与设备的数据传输。

USB 主机控制器作为设备挂接到某个总线上，例如：platform 总线。主机控制器由 usb_hcd 结构体表示，在主机控制器设备匹配驱动的 probe()函数中需要为控制器创建并添加 usb_hcd 实例，内核提供了创建和添加 usb_hcd 实例的接口函数。主机控制器驱动程序还有一项重要的工作就是实现 hc_driver 实例，它将赋予 usb_hcd 实例。

1 数据结构

USB 总线 bus_type 结构体实例 usb_bus_type，在内核中可认为是表示逻辑上的 USB 总线，即系统中所有的 USB 设备及驱动都挂接到此总线。系统内可能有多个 USB 主机控制器，即有多个 USB 物理总线。

每个主机控制器由 usb_hcd 结构体表示，其中内嵌的 usb_bus 结构体成员可认为是表示物理上的 USB 总线。usb_hcd 结构体关联的 hc_driver 结构体主要是操作函数指针，用于执行数据传输等操作。

下面将介绍 usb_hcd、usb_bus 和 hc_driver 结构体的定义。

■usb_hcd

USB 主机控制器在内核中由 **usb_hcd** 结构体表示，定义在/include/linux/usb/hcd.h 头文件：

```
struct usb_hcd {
    struct usb_bus    self;          /*表示物理总线*/
    struct kref       kref;          /*引用计数*/

    const char        *product_desc;     /*product/vendor string（厂商描述字符串）*/
    int               speed;             /*根 HUB 速率*/
    char              irq_descr[24];     /*driver + bus #*/

    struct timer_list     rh_timer;          /*根 HUB 轮询定时器*/
    struct urb        *status_urb;           /*用于传递根 HUB 状态变化的 urb*/
#ifdef CONFIG_PM
    struct work_struct   wakeup_work;       /*为远程唤醒*/
#endif

    /*硬件信息和状态*/
    const struct hc_driver    *driver;    /*由具体主机控制器驱动实现的操作接口，hc_driver 实例*/

    struct usb_phy      *usb_phy;       /*OTG，/include/linux/usb/phy.h*/
    struct phy        *phy;
    unsigned long         flags;     /*标记成员，取值，见下文*/
```

```
    /*HCD 注册和移除时设置的标记*/
    unsigned        rh_registered:1;      /*根 HUB 是否已经注册了*/
    unsigned        rh_pollable:1;        /*是否允许轮询根 HUB*/
    unsigned        msix_enabled:1;       /* driver has MSI-X enabled? */
    unsigned        remove_phy:1;         /* auto-remove USB phy */

    unsigned        uses_new_polling:1;
    unsigned        wireless:1;           /*无线 HCD*/
    unsigned        authorized_default:1;
    unsigned        has_tt:1;             /* Integrated TT in root hub */
    unsigned        amd_resume_bug:1;     /* AMD remote wakeup quirk */
    unsigned        can_do_streams:1;     /* HC supports streams */
    unsigned        tpl_support:1;        /* OTG & EH TPL support */
    unsigned        cant_recv_wakeups:1;

    unsigned int            irq;          /*中断编号*/
    void __iomem            *regs;        /*控制寄存器地址*/
    resource_size_t         rsrc_start;   /*memory/io resource start*/
    resource_size_t         rsrc_len;     /*memory/io resource length*/
    unsigned        power_budget;         /* in mA, 0 =无限制 */

    struct giveback_urb_bh  high_prio_bh;     /*giveback_urb_bh 结构体，/include/linux/usb/hcd.h*/
    struct giveback_urb_bh  low_prio_bh;
    struct mutex            *bandwidth_mutex;
    struct usb_hcd          *shared_hcd;
    struct usb_hcd          *primary_hcd;

  #define HCD_BUFFER_POOLS      4
    struct dma_pool         *pool[HCD_BUFFER_POOLS];   /*缓存（32，128，512，2048），/mm/dmapool.c*/

    int     state;              /*状态，取值见下文*/
    unsigned long hcd_priv[0] __attribute__ ((aligned(sizeof(s64))));/*具体主机控制器私有数据结构*/
};
```

usb_hcd 结构体主要成员简介如下：

●**flags**：标记成员，取值定义在结构体内，如下所示：

```
#define   HCD_FLAG_HW_ACCESSIBLE            0    /*主机控制器可访问（操作）*/
#define   HCD_FLAG_POLL_RH                  2    /* poll for rh status? */
#define   HCD_FLAG_POLL_PENDING             3    /*状态是否已经改变*/
#define   HCD_FLAG_WAKEUP_PENDING           4    /*根 HUB 正在唤醒吗? */
#define   HCD_FLAG_RH_RUNNING               5    /*根 HUB 正在运行吗？ */
#define   HCD_FLAG_DEAD                     6    /* controller has died? */

/*标记位检测宏如下*/
#define HCD_HW_ACCESSIBLE(hcd)      ((hcd)->flags & (1U << HCD_FLAG_HW_ACCESSIBLE))
```

```
#define HCD_POLL_RH(hcd)      ((hcd)->flags & (1U << HCD_FLAG_POLL_RH))
#define HCD_POLL_PENDING(hcd)         ((hcd)->flags & (1U << HCD_FLAG_POLL_PENDING))
#define HCD_WAKEUP_PENDING(hcd)    ((hcd)->flags & (1U << HCD_FLAG_WAKEUP_PENDING))
#define HCD_RH_RUNNING(hcd)     ((hcd)->flags & (1U << HCD_FLAG_RH_RUNNING))
#define HCD_DEAD(hcd)           ((hcd)->flags & (1U << HCD_FLAG_DEAD))
```

●**state**：状态成员，取值定义在结构体内，如下：

```
#define   __ACTIVE          0x01
#define    __SUSPEND       0x04
#define    __TRANSIENT          0x80

#define    HC_STATE_HALT     0
#define    HC_STATE_RUNNING     (__ACTIVE)
#define    HC_STATE_QUIESCING   (__SUSPEND|__TRANSIENT|__ACTIVE)
#define    HC_STATE_RESUMING    (__SUSPEND|__TRANSIENT)
#define    HC_STATE_SUSPENDED (__SUSPEND)
#define    HC_IS_RUNNING(state)    ((state) & __ACTIVE)
#define    HC_IS_SUSPENDED(state)    ((state) & __SUSPEND)
```

●**irq**：主机控制器中断编号。

●**high_prio_bh/low_prio_bh**：giveback_urb_bh 结构体成员，定义在/include/linux/usb/hcd.h 头文件：

```
struct giveback_urb_bh {
    bool running;
    spinlock_t lock;
    struct list_head    head;          /*管理 urb 实例*/
    struct tasklet_struct bh;          /*tasklet 实例*/
    struct usb_host_endpoint *completing_ep;
};
```

giveback_urb_bh 结构体用于管理传输完成的 urb 实例，执行传输完成后的工作。

●**pool[]**：dma_pool 结构体指针数组，表示 DMA 缓存池，项数为 4，结构体定义在/mm/dmapool.c 文件内：

```
struct dma_pool {          /* the pool */
    struct list_head page_list;
    spinlock_t lock;
    size_t size;          /*4 个数组项，size 大小依次为 32、128、512、2048*/
    struct device *dev;
    size_t allocation;
    size_t boundary;
    char name[32];
    struct list_head pools;
};
```

●**hcd_priv[0]**：主机控制器私有数据结构，后面介绍 EHCI 主机控制器驱动时将会涉及到。

●**self**：usb_bus 结构体成员，见下文。

●**driver**：指向 hc_driver 结构体，由具体主机控制器驱动程序实现，定义见下文。

■usb_bus

usb_hcd 结构体中 self 成员是 usb_bus 结构体，定义如下（/include/linux/usb.h）：

```
struct usb_bus {
    struct device *controller;      /*指向表示主机控制器的 device 实例，嵌入到 xxx_device 结构体中*/
    int busnum;                     /*主机控制器编号*/
    const char *bus_name;           /* stable id (PCI slot_name etc) */
    u8 uses_dma;                    /*主机控制器是否使用 DMA*/
    u8 uses_pio_for_control;
    u8 otg_port;                    /* 0, or number of OTG/HNP port */
    unsigned is_b_host:1;           /* true during some HNP roleswitches */
    unsigned b_hnp_enable:1;        /* OTG: did A-Host enable HNP? */
    unsigned no_stop_on_short:1;
    unsigned no_sg_constraint:1;    /* no sg constraint */
    unsigned sg_tablesize;          /* 0 or largest number of sg list entries */
    int devnum_next;
    struct usb_devmap devmap;       /*分配总线下设备地址的位图，/include/linux/usb.h*/
    struct usb_device *root_hub;    /*指向表示根 HUB 的 USB 设备 usb_device 实例*/
    struct usb_bus *hs_companion;   /* Companion EHCI bus, if any */
    struct list_head bus_list;      /*添加到 usb_bus_list 全局双链表*/

    struct mutex usb_address0_mutex;   /* unaddressed device mutex */

    int bandwidth_allocated;
    int bandwidth_int_reqs;         /*number of Interrupt requests */
    int bandwidth_isoc_reqs;        /*number of Isoc. requests */

    unsigned resuming_ports;        /*bit array: resuming root-hub ports */
#if defined(CONFIG_USB_MON) || defined(CONFIG_USB_MON_MODULE)
    struct mon_bus *mon_bus;        /* non-null when associated */
    int monitored;                  /* non-zero when monitored */
#endif
};
```

■hc_driver

usb_hcd 结构体中 driver 成员是 hc_driver 结构体指针。hc_driver 结构体是由具体 USB 主机控制器驱动程序实现的数据结构。

hc_driver 结构体定义如下（/include/linux/usb/hcd.h）：

```
struct hc_driver {
    const char      *description;       /*描述符，如"ehci-hcd"等*/
```

```
    const char      *product_desc;      /*厂商/产品字符串*/
    size_t          hcd_priv_size;      /*usb_hcd 实例中私有数据大小，例如：ehci_hcd*/

    irqreturn_t     (*irq) (struct usb_hcd *hcd);     /*主机控制器中断处理函数*/
    int    flags;                       /*标记成员，取值见下文*/

    int    (*reset) (struct usb_hcd *hcd);     /*初始化 HCD 或根 HUB*/
    int    (*start) (struct usb_hcd *hcd);

    int    (*pci_suspend)(struct usb_hcd *hcd, bool do_wakeup);
    int    (*pci_resume)(struct usb_hcd *hcd, bool hibernated);
    void (*stop) (struct usb_hcd *hcd);                 /*使 HCD 停止写内存和进行 I/O 操作*/
    void (*shutdown) (struct usb_hcd *hcd);             /*关闭 HCD*/
    int    (*get_frame_number) (struct usb_hcd *hcd);     /*返回当前帧编号*/

    /*管理 IO 请求和设备状态（读写请求）*/
    int    (*urb_enqueue)(struct usb_hcd *hcd,struct urb *urb, gfp_t mem_flags); /*urb 入队并执行传输*/
    int    (*urb_dequeue)(struct usb_hcd *hcd,struct urb *urb, int status);

    int    (*map_urb_for_dma)(struct usb_hcd *hcd, struct urb *urb,gfp_t mem_flags);
    void    (*unmap_urb_for_dma)(struct usb_hcd *hcd, struct urb *urb);

    void    (*endpoint_disable)(struct usb_hcd *hcd,struct usb_host_endpoint *ep); /*释放端点资源*/
    void   (*endpoint_reset)(struct usb_hcd *hcd,struct usb_host_endpoint *ep);       /*复位端点*/

    /*根 HUB 支持*/
    int    (*hub_status_data) (struct usb_hcd *hcd, char *buf);
    int    (*hub_control) (struct usb_hcd *hcd,u16 typeReq, u16 wValue, u16 wIndex,   \
                                                char *buf, u16 wLength);            /*实现控制传输*/
    int    (*bus_suspend)(struct usb_hcd *);
    int    (*bus_resume)(struct usb_hcd *);
    int    (*start_port_reset)(struct usb_hcd *, unsigned port_num);

    void (*relinquish_port)(struct usb_hcd *, int);
    int    (*port_handed_over)(struct usb_hcd *, int);
    void (*clear_tt_buffer_complete)(struct usb_hcd *,struct usb_host_endpoint *);
    /* xHCI 主机控制器回调函数（USB3.0） */
    ...
};
```

hc_driver 结构体主要成员简介如下：

●**flags**：标记，主要表示主机控制器支持的 USB 规范版本，如下所示：

```
#define    HCD_MEMORY          0x0001          /* HC regs use memory (else I/O) */
#define    HCD_LOCAL_MEM       0x0002          /*需要本地内存*/
```

```
#define   HCD_SHARED      0x0004    /*控制器共享硬件*/
#define   HCD_USB11       0x0010    /* USB 1.1 */
#define   HCD_USB2        0x0020    /* USB 2.0 */
#define   HCD_USB25       0x0030    /* Wireless USB 1.0 (USB 2.5)*/
#define   HCD_USB3        0x0040    /* USB 3.0 */
#define   HCD_MASK        0x0070    /*规范版本的掩码*/
#define   HCD_BH          0x0100    /* URB complete in BH context */
```

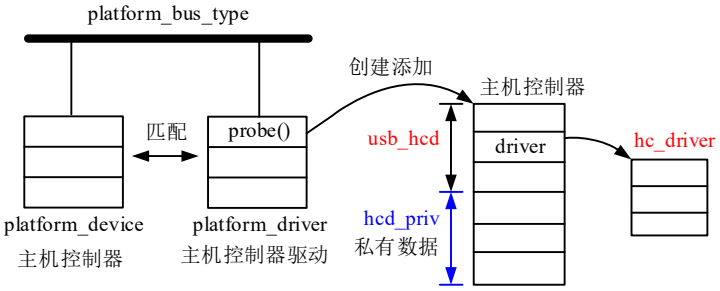
●**hub_control()：**实现控制传输的函数。

●**urb_enqueue()/urb_dequeue()：**其它传输类型时，将 urb 添加/移出队列。

2 主机控制器驱动

USB 主机控制器作为设备，挂接在某个总线上，如 platform 总线、PCI 总线等，因此需要定义并注册表示主机控制器的 xxx_device 实例（设备）和表示主机控制器驱动的 xxx_driver 实例。

主机控制器驱动程序需要实现 hc_driver 实例，xxx_driver 实例中的 probe()函数需要创建、设置并添加表示主机控制器的 usb_hcd 实例，如下图所示。



USB 驱动核心层提供了分配和添加主机控制器 **usb_hcd** 实例的接口函数，供 USB 主机控制器驱动程序中的 probe()函数调用。

■创建 usb_hcd 实例

创建主机控制器 usb_hcd 实例的接口函数如下：

struct usb_hcd ***usb_create_hcd**(const struct hc_driver *driver,struct device *dev, const char *bus_name)；

参数语义如下：

driver：指向具体控制器驱动实现的 hc_driver 实例。

dev：指向表示主机控制器设备的 device 实例，即 xxx_device 结构体中内嵌的 device 实例。

bus_name：指向控制器名称字符串。

usb_create_hcd()函数定义如下（/drivers/usb/core/hcd.c）：

```
struct usb_hcd *usb_create_hcd(const struct hc_driver *driver,struct device *dev, const char *bus_name)
{
    return usb_create_shared_hcd(driver, dev, bus_name, NULL);    /*/drivers/usb/core/hcd.c*/
}
```

usb_create_shared_hcd()函数代码简列如下，用于创建和初始化 usb_hcd 实例：

```
struct usb_hcd *usb_create_shared_hcd(const struct hc_driver *driver,   \
                                struct device *dev, const char *bus_name,struct usb_hcd *primary_hcd)
/*primary_hcd：NULL*/
{
    struct usb_hcd *hcd;

    hcd = kzalloc(sizeof(*hcd) + driver->hcd_priv_size, GFP_KERNEL);
                                                    /*分配 usb_hcd 实例，后接私有数据*/
    ...
    if (primary_hcd == NULL) {
        hcd->bandwidth_mutex = kmalloc(sizeof(*hcd->bandwidth_mutex),GFP_KERNEL);
        ...
        mutex_init(hcd->bandwidth_mutex);
        dev_set_drvdata(dev, hcd);      /*platform_device->dev->driver_data=hcd*/
    } else {
        ...
    }

    kref_init(&hcd->kref);          /*引用计数初始化*/

    usb_bus_init(&hcd->self);       /*初始化 usb_bus 结构体成员，/drivers/usb/core/hcd.c*/
    hcd->self.controller = dev;     /*指向表示主机控制器的 device 实例*/
    hcd->self.bus_name = bus_name;
    hcd->self.uses_dma = (dev->dma_mask != NULL);

    init_timer(&hcd->rh_timer);     /*初始化轮询根 HUB 定时器*/
    hcd->rh_timer.function = rh_timer_func;     /*定时器到期处理函数为 rh_timer_func()*/
    hcd->rh_timer.data = (unsigned long) hcd;       /*定时器数据指向 usb_hcd 实例*/
#ifdef CONFIG_PM
    INIT_WORK(&hcd->wakeup_work, hcd_resume_work);
#endif

    hcd->driver = driver;           /*关联 hc_driver 实例*/
    hcd->speed = driver->flags & HCD_MASK;          /*标识速率*/
    hcd->product_desc = (driver->product_desc) ? driver->product_desc :"USB Host Controller";
    return hcd;         /*返回 usb_hcd 实例指针*/
}
```

usb_create_shared_hcd()函数将分配 usb_hcd 实例及其私有数据，初始化和设置 usb_hcd 实例，关联参数传递的 hc_driver 实例，初始化内嵌定时器 rh_timer，其到期执行函数为 **rh_timer_func()**。表示主机控制器的 xxx_device->dev->driver_data 成员指向 hcd，即 usb_hcd 实例。

■添加 usb_hcd 实例

USB 主机控制器驱动 probe()函数在创建完 usb_hcd 实例后，还需要对实例进行特定于控制器的设置，

最后调用 **usb_add_hcd()**函数向 USB 驱动核心层添加 usb_hcd 实例，函数定义如下（/drivers/usb/core/hcd.c）：

```
int usb_add_hcd(struct usb_hcd *hcd,unsigned int irqnum, unsigned long irqflags)
/*hcd：usb_hcd 实例指针，irqnum：中断编号，irqflags：中断标记*/
{
    int retval;
    struct usb_device *rhdev;          /*表示根 HUB 的 usb_device*/

    if (IS_ENABLED(CONFIG_USB_PHY) && !hcd->usb_phy) {       /*OTG*/
        struct usb_phy *phy = usb_get_phy_dev(hcd->self.controller, 0);

        if (IS_ERR(phy)) {
            ...
        } else {
            retval = usb_phy_init(phy);
            ...
            hcd->usb_phy = phy;
            hcd->remove_phy = 1;
        }
    }

    if (IS_ENABLED(CONFIG_GENERIC_PHY) && !hcd->phy) {
        struct phy *phy = phy_get(hcd->self.controller, "usb");

        if (IS_ERR(phy)) {
            ...
        } else {
            retval = phy_init(phy);
            if (retval) {
                phy_put(phy);
                goto err_phy;
            }
            retval = phy_power_on(phy);
            ...
            hcd->phy = phy;
            hcd->remove_phy = 1;
        }
    }

    dev_info(hcd->self.controller, "%s\n", hcd->product_desc);

    if (authorized_default < 0 || authorized_default > 1)
        hcd->authorized_default = hcd->wireless ? 0 : 1;
    else
        hcd->authorized_default = authorized_default;
```

```
set_bit(HCD_FLAG_HW_ACCESSIBLE, &hcd->flags);          /*设置标记位*/

if ((retval = hcd_buffer_create(hcd)) != 0) {
                        /*为 pool[]指针数组创建 dma_pool 实例，/drivers/usb/core/buffer.c*/
    ...
}

if ((retval = usb_register_bus(&hcd->self)) < 0)       /*注册 usb_bus 实例，/drivers/usb/core/hcd.c*/
    goto err_register_bus;        /*分配编号，添加到全局双链表，执行 usb_notifier_list 通知链等*/

rhdev = usb_alloc_dev(NULL, &hcd->self, 0);       /*/drivers/usb/core/usb.c*/
    /*创建表示根 HUB 的 usb_device 实例，总线为 usb_bus_type，设备类型为 usb_device_type*/
...
mutex_lock(&usb_port_peer_mutex);
hcd->self.root_hub = rhdev;              /*指向表示根 HUB 的 usb_device 实例*/
mutex_unlock(&usb_port_peer_mutex);

switch (hcd->speed) {       /*主机控制器速度模式*/
case HCD_USB11:
    rhdev->speed = USB_SPEED_FULL;          /*全速*/
    break;
case HCD_USB2:
    rhdev->speed = USB_SPEED_HIGH;           /*高速*/
    break;
case HCD_USB25:
    rhdev->speed = USB_SPEED_WIRELESS;     /*无线*/
    break;
case HCD_USB3:
    rhdev->speed = USB_SPEED_SUPER;         /*超高速*/
    break;
default:
    retval = -EINVAL;
    goto err_set_rh_speed;
}

device_set_wakeup_capable(&rhdev->dev, 1);
                        /*dev->power.can_wakeup = 1 等，/drivers/base/power/wakeup.c*/
set_bit(HCD_FLAG_RH_RUNNING, &hcd->flags);       /*设置主机控制器标记位*/

if (hcd->driver->reset && (retval = hcd->driver->reset(hcd)) < 0) {    /*初始化控制器和根 HUB*/
    ...
}
hcd->rh_pollable = 1;
```

```
        if (device_can_wakeup(hcd->self.controller)&& device_can_wakeup(&hcd->self.root_hub->dev))
            dev_dbg(hcd->self.controller, "supports USB remote wakeup\n");

        /*初始化 giveback_urb_bh 结构体成员，tasklet 执行函数为 usb_giveback_urb_bh()*/
        init_giveback_urb_bh(&hcd->high_prio_bh);       /*/drivers/usb/core/hcd.c*/
        init_giveback_urb_bh(&hcd->low_prio_bh);

        if (usb_hcd_is_primary_hcd(hcd) && irqnum) {
            retval = usb_hcd_request_irqs(hcd, irqnum, irqflags);       /*/drivers/usb/core/hcd.c*/
              /*申请中断，中断处理函数为 usb_hcd_irq()，函数内调用 hcd->driver->irq(hcd)函数*/
            ...
        }

        hcd->state = HC_STATE_RUNNING;
        retval = hcd->driver->start(hcd);          /*启动主机控制器*/
        ...

        if ((retval = register_root_hub(hcd)) != 0)
          /*注册表示根 HUB 的 usb_device 实例，将匹配通用 USB 设备驱动，/drivers/usb/core/hcd.c*/
            ...
        retval = sysfs_create_group(&rhdev->dev.kobj, &usb_bus_attr_group);        /*添加属性文件*/
        ...
        if (hcd->uses_new_polling && HCD_POLL_RH(hcd))
            usb_hcd_poll_rh_status(hcd);              /*激活轮询定时器，检测根 HUB 状态变化*/

        return retval;
        ...
    }
```

usb_add_hcd()函数执行的主要工作简介如下：

（1）为 usb_hcd 结构体 pool[]指针数组分配 dma_pool 实例。

（2）注册 usb_hcd 内嵌的 usb_bus 结构体实例，主要工作是为其分配编号，添加到全局双链表，执行 usb_notifier_list 通知链等。

（3）创建表示根 HUB 的 usb_device 实例，挂载到 usb_bus_type 总线（USB 总线），设备类型设为 usb_device_type，并将实例赋予 hcd->self.root_hub。

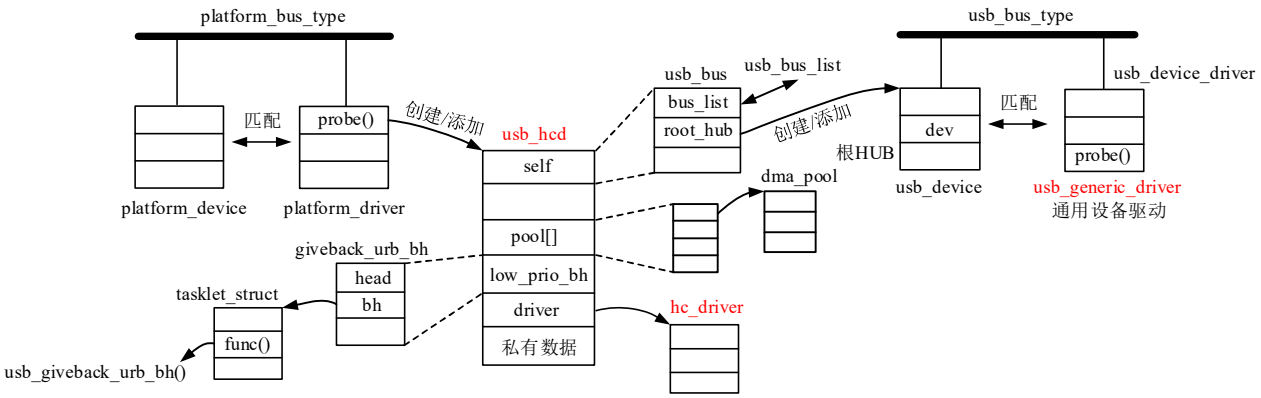
（4）复位主机控制器和根 HUB。

（5）初始化 usb_hcd 中 giveback_urb_bh 结构体成员 high_prio_bh 和 low_prio_bh，其内嵌 tasklet_struct 的执行函数为 usb_giveback_urb_bh()，用于处理 giveback_urb_bh 中管理的传输完成的 urb 实例。

（6）申请主机控制器中断，中断处理函数为 usb_hcd_irq()，函数内调用 hcd->driver->irq(hcd)函数。

（7）注册根 HUB，添加表示根 HUB 的 usb_device 实例（将匹配通用 USB 设备驱动），添加属性文件等。

创建并注册 usb_hcd 实例的结果简列如下图所示：



USB 主机控制器中内嵌一个根 HUB，它被视为 USB 设备，在添加 usb_hcd 实例时，将创建并注册表示根 HUB 的 usb_device 实例。此 usb_hcd 实例将匹配通用的 USB 设备驱动，其 probe()函数将选择设备配置，获取接口信息，为接口创建并注册 usb_interface 实例。usb_interface 实例又将匹配 usb_driver 实例，加载接口驱动程序，详细内容见下一小节（上图中未画出）。

3 EHCI 控制器驱动

USB 主机控制器驱动程序位于/drivers/usb/host/目录下。

EHCI 主机控制器是支持 USB2.0 规范的标准 USB 主机控制器，内嵌 OHCI 或 UHCI 控制器，下面以 EHCI 主机控制器驱动为例简要说明主机控制器驱动的实现。

EHCI 主机控制器包 Capability、Operational 和 EHCI 实现相关的寄存器，主机控制器的操作主要就是对其寄存器的操作，这里就不详细介绍了，请读者参考《Enhanced Host Controller Interface Specification for Universal Serial Bus》手册。下面主要介绍内核中 EHCI 主机控制器驱动框架。

EHCI 主机控制器驱动主要实现代码位于 ehci-hcd.c、ehci-hub.c、ehci-sched.c、ehci-mem.c 等文件内，主要是定义 hc_driver 实例 **ehci_hc_driver** 及其操作函数等。

在 ehci-xxx.c 文件内，主要是定义并注册表示主机控制器驱动的 xxx_driver 实例，实例 prob()函数中将创建、设置并添加表示主机控制器的 usb_hcd 实例。

例如，假设主机控制器挂接在平台总线上，在 ehci-platform.c 文件内定义并注册了表示主机控制器驱动的 platform_driver 实例 ehci_platform_driver，在其 probe()函数中将创建并添加 usb_hcd 实例，其关联的 hc_driver 实例设为 ehci_platform_hc_driver，它是 ehci_hc_driver 实例的副本，只不过加上了私有数据。

下面以挂接在平台总线的 EHCI 主机控制器为例，简要说明其驱动实现。

■设备

如果 ECHI 控制器挂接在 platform 总线上，则在/drivers/usb/host/ehci-platform.c 文件内实现了通用的主机控制器驱动程序（需选择 USB_EHCI_HCD_PLATFORM 配置选项）。

在板级相关文件（或设备树节点）中需要向驱动传递主机控制器的硬件信息，由 usb_ehci_pdata 结构体表示，定义如下（/include/linux/usb/ehci_pdriver.h）：

```
struct usb_ehci_pdata {
        int             caps_offset;
        unsigned has_tt:1;
        unsigned has_synopsys_hc_bug:1;
        unsigned big_endian_desc:1;
        unsigned big_endian_mmio:1;
```

```
    unsigned no_io_watchdog:1;
    unsigned reset_on_resume:1;
    unsigned dma_mask_64:1;

    int (*power_on)(struct platform_device *pdev);      /*开启所有供电及时钟*/
    void (*power_off)(struct platform_device *pdev);    /*关闭所有供电及时钟*/
    void (*power_suspend)(struct platform_device *pdev);
    int (*pre_setup)(struct usb_hcd *hcd);
};
```

在龙芯 1B 板级文件中，定义了以下 usb_ehci_pdata 结构体实例：

```
static struct usb_ehci_pdata ls1x_ehci_pdata = { };    /*usb_ehci_pdata 实例，为空*/

struct platform_device ls1x_ehci_pdev = {       /*表示主机控制器设备的 platform_device 实例*/
    .name           = "ehci-platform",          /*名称，用于匹配驱动*/
    .id         = -1,
    .num_resources      = ARRAY_SIZE(ls1x_ehci_resources),
    .resource = ls1x_ehci_resources,        /*资源*/
    .dev        = {
        .dma_mask = &ls1x_ehci_dmamask,
        .platform_data = &ls1x_ehci_pdata,      /*指向 usb_ehci_pdata 实例*/
    },
};
```

ls1x_ehci_resources 为资源数组定义如下：

```
static u64 ls1x_ehci_dmamask = DMA_BIT_MASK(32);
static struct resource ls1x_ehci_resources[] = {        /*主机控制器资源*/
    [0] = {
        .start= LS1X_EHCI_BASE,                 /*控制寄存器基址*/
        .end = LS1X_EHCI_BASE + SZ_32K - 1,
        .flags= IORESOURCE_MEM,     /*内存*/
    },
    [1] = {
        .start= LS1X_EHCI_IRQ,      /*中断编号*/
        .flags= IORESOURCE_IRQ,     /*中断*/
    },
};
```

在平台定义的初始化函数中将注册 ls1x_ehci_pdev 实例（platform_device），代码简列如下：

```
static struct platform_device *ls1b_platform_devices[] __initdata = {
    ...
    &ls1x_ehci_pdev,
    ...
};
```

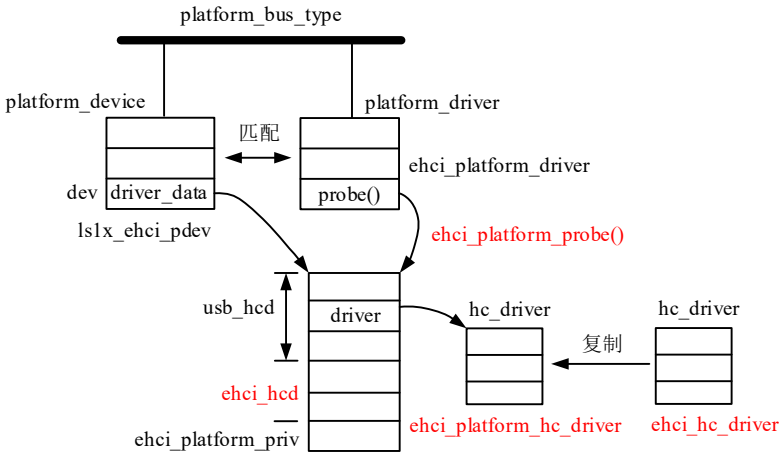
```
static int __init ls1b_platform_init(void)
{
    int err;

    ls1x_serial_setup(&ls1x_uart_pdev);
    err = platform_add_devices(ls1b_platform_devices,ARRAY_SIZE(ls1b_platform_devices));
    return err;
}
arch_initcall(ls1b_platform_init);
```

■驱动

在/drivers/usb/host/ehci-platform.c 文件内定义了匹配平台 EHCI 主机控制器设备的驱动 platform_driver 实例，如下：

```
static struct platform_driver ehci_platform_driver = {
    .id_table  = ehci_platform_table,
    .probe          = ehci_platform_probe,        /*驱动探测函数*/
    .remove         = ehci_platform_remove,
    .shutdown       = usb_hcd_platform_shutdown,
    .driver         = {
        .name       = "ehci-platform",          /*名称，用于匹配设备*/
        .pm  = &ehci_platform_pm_ops,
       .of_match_table = vt8500_ehci_ids,
    }
};
```

在驱动探测函数 ehci_platform_probe()中将创建、设置和添加表示主机控制器的 usb_hcd 实例，执行结果如下图所示，源代码请读者自行阅读：



usb_hcd 关联的 hc_driver 实例为 **ehci_platform_hc_driver**，它复制于通用 ehci_hc_driver 实例。usb_hcd 后面的私有数据结构为 ehci_hcd，其最后一个成员是 platform 总线 EHCI 主机控制器的 ehci_platform_priv 私有数据结构。

ehci_hcd 结构体定义在/drivers/usb/host/ehci.h 头文件。

ehci_platform_priv 结构体定义在/drivers/usb/host/ehci-platform.c 文件内。

内核在/drivers/usb/host/ehci-hcd.c 文件内定义了 ECHI 主机控制器通用的 hc_driver 实例 ehci_hc_driver，如下所示：

```
static const struct hc_driver ehci_hc_driver = {
    .description =      hcd_name,
    .product_desc =     "EHCI Host Controller",
    .hcd_priv_size =    sizeof(struct ehci_hcd),        /*私有数据*/

    .irq =              ehci_irq,           /*中断处理函数*/
    .flags =            HCD_MEMORY | HCD_USB2 | HCD_BH,

    .reset =            ehci_setup,
    .start =            ehci_run,
    .stop =             ehci_stop,
    .shutdown =         ehci_shutdown,

    .urb_enqueue =      ehci_urb_enqueue,       /*urb 入队，还要调度 urb 的传输*/
    .urb_dequeue =      ehci_urb_dequeue,
    .endpoint_disable = ehci_endpoint_disable,
    .endpoint_reset =   ehci_endpoint_reset,
    .clear_tt_buffer_complete =   ehci_clear_tt_buffer_complete,

    .get_frame_number =     ehci_get_frame,

    .hub_status_data =  ehci_hub_status_data,       /*中断轮询定时器中调用*/
    .hub_control =      ehci_hub_control,           /*实现控制传输*/
    .bus_suspend =      ehci_bus_suspend,
    .bus_resume =       ehci_bus_resume,
    .relinquish_port =  ehci_relinquish_port,
    .port_handed_over =     ehci_port_handed_over,
    .free_dev =         ehci_remove_device,
};
```

ehci_hc_driver 实例中操作函数定义在 ehci-hcd.c、ehci-hub.c、ehci-sched.c、ehci-mem.c 等文件内，请读者自行阅读源代码。

在/drivers/usb/host/ehci-platform.c 文件内定义了以下初始化（模块加载）函数：

```
static int __init ehci_platform_init(void)
{
    if (usb_disabled())
        return -ENODEV;

    pr_info("%s: " DRIVER_DESC "\n", hcd_name);
```

```
    ehci_init_driver(&ehci_platform_hc_driver, &platform_overrides);
                                    /*由 ehci_hc_driver 生成 ehci_platform_hc_driver 实例*/
    return platform_driver_register(&ehci_platform_driver);        /*注册 ehci_platform_driver 实例*/
}
module_init(ehci_platform_init);        /*模块加载（初始化）函数*/
```

ehci_init_driver()用于复制 ehci_hc_driver 实例数据至 ehci_platform_hc_driver 实例，并修改私有数据大小等部分成员。ehci_platform_init()函数最后注册 ehci_platform_driver 实例，与 ls1x_ehci_pdev 实例匹配后，将调用 probe()函数，创建、添加表示 EHCI 主机控制器的 usb_hcd 实例及其私有数据。

8.9.4 设备与驱动

内核在/drivers/usb/core/driver.c 文件内定义了 USB 总线实例：

```
struct bus_type usb_bus_type = {
    .name =   "usb",
    .match =  usb_device_match,     /*匹配函数，后面再介绍*/
    .uevent = usb_uevent,
};
```

USB 总线设备链表下挂载的有设备 usb_device 实例、接口 usb_interface 实例等，驱动链表下挂载有设备驱动 usb_device_driver 实例、接口驱动 usb_driver 实例等。

USB 设备 usb_device 实例和接口 usb_interface 实例都是在探测设备时，由 USB 驱动核心层自动创建的。USB 设备匹配的通用设备驱动 usb_generic_driver 实例也由内核定义和注册，因此实际需要用户实现和注册的就是接口驱动 usb_driver 实例。

1 设备

USB 设备是表现出一种或多种功能的逻辑或物理实体。HUB 与周边设备都属于设备。主机会为每个总线上的设备指定一个专属地址。复合设备含有 HUB 以及一个或多个永久相连的设备。主机以（与对待单一设备）相同的方式对待复合设备，就如同这个集线器和其功能是单独的物理设备一样。每个 HUB 和嵌入式设备都拥有独特的地址。

复合设备只有 1 个总线地址，但有多个独立接口，每个接口提供一种功能。例如，复合设备可同时含有用于大容量存储设备和键盘的接口。不同接口可使用主机上不同的驱动程序，通常说的 USB 设备驱动程序是匹配 USB 设备接口的，而不是 USB 设备。

USB 驱动框架中设备由 usb_device 结构体表示，接口由 usb_interface 结构体表示。USB 设备下接口 usb_interface 实例由设备 usb_device 实例管理。

■usb_device

USB 设备在内核中由 usb_device 结构体表示，定义如下（/include/linux/usb.h）：

```
struct usb_device {
    int         devnum;          /*设备编号，设备地址*/
    char        devpath[16];     /*设备 ID*/
    u32         route;           /*xHCI（USB3.0）*/
    enum usb_device_state  state;       /*设备状态：配置、地址等*/
```

```
enum usb_device_speed speed;        /*速度模式，高速、全速、低速*/

struct usb_tt    *tt;              /*事务转换器，适用于高速 HUB*/
int          ttport;               /*事务转换器端口数*/
unsigned int toggle[2];       /*位图，标记端点是输入（0）还是输出（1）*/

struct usb_device *parent;       /*父设备，即集线器*/
struct usb_bus *bus;              /*主机控制器结构中嵌入的 usb_bus 结构体实例*/
struct usb_host_endpoint ep0;       /*端点 0（控制端点）信息（描述字），/include/linux/usb.h*/

struct device dev;              /*内嵌的 device 实例*/

struct usb_device_descriptor descriptor;       /*设备描述符，/include/uapi/linux/usb/ch9.h*/
struct usb_host_bos *bos;              /*二进制设备目标描述字，/include/linux/usb.h*/
struct usb_host_config *config;       /*所有的设备配置（含配置描述字和接口），/include/linux/usb.h*/

struct usb_host_config *actconfig;          /*当前活跃的设备配置*/
struct usb_host_endpoint *ep_in[16];       /*输入端点列表，/include/linux/usb.h*/
struct usb_host_endpoint *ep_out[16];      /*输出端点列表，/include/linux/usb.h*/

char **rawdescriptors;         /*各配置的原始描述字*/

unsigned short bus_mA;      /*从总线上可取的电流 mA 数*/
u8 portnum;           /*父接口编号，初始为 1*/
u8 level;             /*在 USB 总线拓扑结构中的层级数*/

unsigned can_submit:1;         /*可以提交 URB*/
unsigned persist_enabled:1;       /*USB_PERSIST 使能*/
unsigned have_langid:1;
unsigned authorized:1;
unsigned authenticated:1;
unsigned wusb:1;                  /*无线 USB*/
unsigned lpm_capable:1;
unsigned usb2_hw_lpm_capable:1;
unsigned usb2_hw_lpm_besl_capable:1;
unsigned usb2_hw_lpm_enabled:1;
unsigned usb2_hw_lpm_allowed:1;
unsigned usb3_lpm_enabled:1;
int string_langid;
/*静态字符串，厂商信息，序列号等*/
char *product;
char *manufacturer;
char *serial;
```

```
    struct list_head filelist;      /*usbfs*/
    int maxchild;       /*如果是 HUB，表示端口数量*/

    u32 quirks;
    atomic_t urbnum;        /*提交 URB 的数量*/
    unsigned long active_duration;      /*设备活跃时长*/
#ifdef CONFIG_PM
    unsigned long connect_time;
    unsigned do_remote_wakeup:1;
    unsigned reset_resume:1;
    unsigned port_is_suspended:1;
#endif
    struct wusb_dev *wusb_dev;      /*无线 USB*/
    int slot_id;
    enum usb_device_removable removable;
    struct usb2_lpm_parameters  l1_params;
    struct usb3_lpm_parameters  u1_params;
    struct usb3_lpm_parameters  u2_params;
    unsigned lpm_disable_count;
};
```

usb_device 结构体主要成员简介如下：

●**devnum**：设备地址。

●**state**：设备状态，由 usb_device_state 枚举类型表示，定义如下（/include/uapi/linux/usb/ch9.h）：

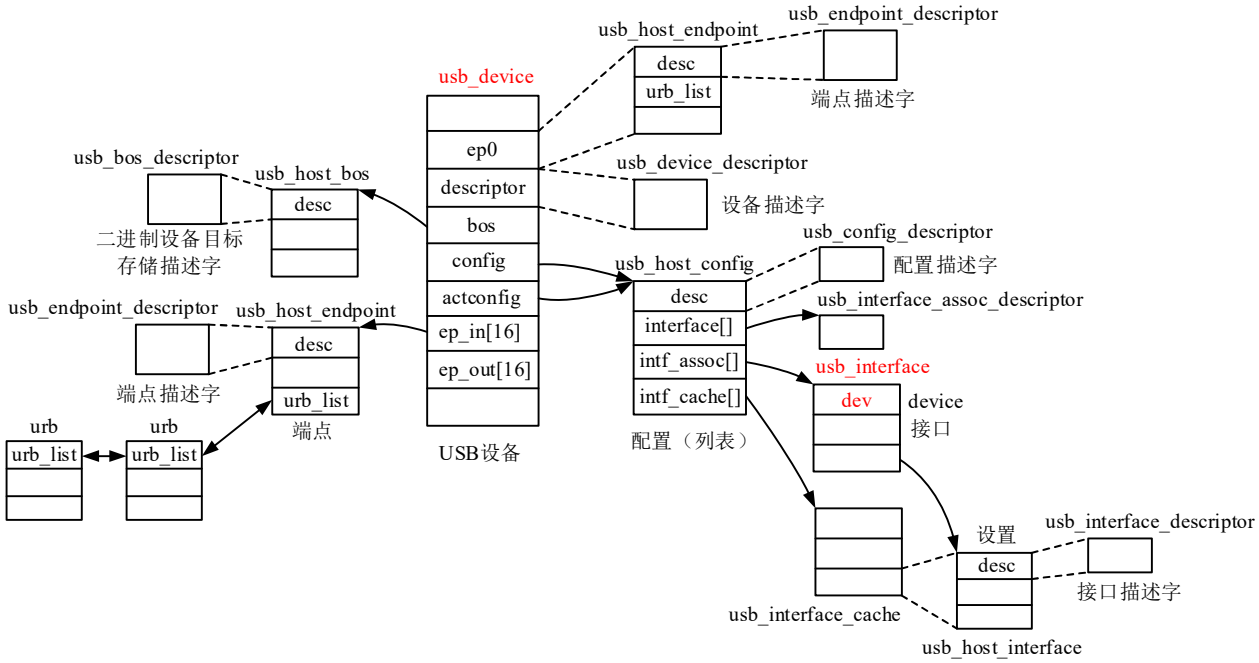
```
enum usb_device_state {
    USB_STATE_NOTATTACHED = 0,
    USB_STATE_ATTACHED,
    USB_STATE_POWERED,              /* wired */
    USB_STATE_RECONNECTING,         /* auth */
    USB_STATE_UNAUTHENTICATED,      /* auth */
    USB_STATE_DEFAULT,              /*默认*/
    USB_STATE_ADDRESS,              /*地址*/
    USB_STATE_CONFIGURED,           /*配置*/
    USB_STATE_SUSPENDED
};
```

●**speed**：速度模式，由 usb_device_speed 枚举类型表示，定义如下（/include/uapi/linux/usb/ch9.h）：

```
enum usb_device_speed {
    USB_SPEED_UNKNOWN = 0,          /* enumerating */
    USB_SPEED_LOW, USB_SPEED_FULL,      /* usb 1.1 */
    USB_SPEED_HIGH,             /* usb 2.0 */
    USB_SPEED_WIRELESS,         /* wireless (usb 2.5) */
    USB_SPEED_SUPER,            /* usb 3.0 */
};
```

●**dev**：通用设备 device 实例，挂接到 USB 总线 usb_bus_type。

usb_device 结构体中包含设备的各类描述字信息，各描述字数据结构定义在/include/uapi/linux/usb/ch9.h 头文件，这里就不一一列出各数据结构的定义了，只用图示的方式示意一下 usb_device 结构体中包含的描述字信息。



在上图中指针数组只画出了一个指针项，例如 ep_in[]是一个 16 项的指针数组，数组项指向包含端点描述字的 usb_host_endpoint 结构体实例。

■usb_interface

usb_interface 结构体表示 USB 设备接口，定义在/include/linux/usb.h 头文件：

```
struct usb_interface {
    struct usb_host_interface *altsetting;   /*指向替代设置 usb_host_interface 实例列表*/

    struct usb_host_interface *cur_altsetting;     /*当前激活的替代设置*/
    unsigned num_altsetting;                       /*替代设置数量*/

    struct usb_interface_assoc_descriptor *intf_assoc;

    int minor;              /*接口绑定的从设备号，如果使用 USB 设备主设备号有效*/
    enum usb_interface_condition condition;          /*接口绑定状态*/
    unsigned sysfs_files_created:1;     /* the sysfs attributes exist */
    unsigned ep_devs_created:1;         /* endpoint "devices" exist */
    unsigned unregistering:1;           /*接口未注册时设置*/
    unsigned needs_remote_wakeup:1;     /*设置表示驱动要求具有远程唤醒能力*/
    unsigned needs_altsetting0:1;       /* switch to altsetting 0 is pending */
    unsigned needs_binding:1;    /* needs delayed unbind/rebind */
    unsigned resetting_device:1;  /* true: bandwidth alloc after reset */

    struct device dev;          /*表示接口的 device 实例*/
```

```
    struct device *usb_dev;         /**/
    atomic_t pm_usage_cnt;          /*电源管理使用计数*/
    struct work_struct reset_ws;    /* for resets in atomic context */
};
```

在 USB 设备插入到 HUB 端口时，HUB 端口状态变化后，主机控制器将会检测到，并动态创建、设置和添加表示 USB 设备 usb_device 实例。

创建 usb_device 实例的函数为 usb_alloc_dev()（/drivers/usb/core/usb.c），添加 usb_device 实例函数为 usb_new_device()（/drivers/usb/core/hub.c）。添加设备时会选择设备配置，获取接口信息，创建并注册接口 usb_interface 实例（后面将介绍函数实现）。

2 驱动

USB 设备和接口匹配的驱动分别为设备驱动 usb_device_driver 和接口驱动 usb_driver。

■设备驱动

USB 设备驱动由 usb_device_driver 结构体表示，定义如下（/include/linux/usb.h）：

```
struct usb_device_driver {
    const char *name;           /*名称*/

    int (*probe) (struct usb_device *udev);         /*探测函数*/
    void (*disconnect) (struct usb_device *udev);       /*断开连接*/

    int (*suspend) (struct usb_device *udev, pm_message_t message);
    int (*resume) (struct usb_device *udev, pm_message_t message);
    struct usbdrv_wrap drvwrap;         /*内嵌驱动 device_driver 实例*/
    unsigned int supports_autosuspend:1;
};
```

usb_device_driver 结构体中 drvwrap 成员为 usbdrv_wrap 结构体，定义如下（/include/linux/usb.h）：

```
struct usbdrv_wrap {
    struct device_driver driver;
    int for_devices;        /*0 表示接口驱动，非 0 表示设备驱动*/
};
```

注册 usb_device_driver 实例的函数声明如下（定义在/drivers/usb/core/driver.c）：

```
int usb_register_device_driver(struct usb_device_driver *,struct module *);     /*/include/linux/usb.h*/
void usb_register(driver);          /*当前模块*/
```

以上两个函数比较简单，主要是注册 usb_device_driver.drvwrap.driver 表示的 device_driver 实例，挂接到 USB 总线 usb_bus_type，usb_device_driver.drvwrap.driver.probe()函数设为 **usb_probe_device()**，函数内调用 usb_device_driver.probe()函数。

注意：设备驱动 usb_device_driver 实例，可匹配所有的 usb_device 实例。

注销 usb_device_driver 实例的函数声明如下（/include/linux/usb.h）：

void usb_deregister_device_driver(struct usb_device_driver *);

内核在/drivers/usb/core/generic.c 定义了 usb_device_driver 实例 **usb_generic_driver**，它匹配所有的设备，在其 prob()函数将选择设备配置，获取配置下的接口描述字，为接口创建并注册 usb_interface 实例等。

■接口驱动

USB 接口驱动由 usb_driver 结构体表示，这才是我们通常在驱动程序中说的 USB 设备驱动，结构体定义如下（/include/linux/usb.h）：

```
struct usb_driver {
    const char *name;           /*名称*/

    int (*probe) (struct usb_interface *intf,const struct usb_device_id *id);     /*探测函数，必须*/
    void (*disconnect) (struct usb_interface *intf);        /*断开接口函数，必须*/

    int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,void *buf);       /*设备控制*/
    int (*suspend) (struct usb_interface *intf, pm_message_t message);       /*睡眠*/
    int (*resume) (struct usb_interface *intf);             /*恢复*/
    int (*reset_resume)(struct usb_interface *intf);     /*复位*/

    int (*pre_reset)(struct usb_interface *intf);       /*由 usb_reset_device()函数在设备复位前调用*/
    int (*post_reset)(struct usb_interface *intf);     /*由 usb_reset_device()函数在设备复位后调用*/

    const struct usb_device_id *id_table;       /*比对列表（标识列表），必须定义*/

    struct usb_dynids dynids;               /*动态 usb_device_id 实例列表*/
    struct usbdrv_wrap drvwrap;         /*内嵌 device_driver 实例，for_devices 为 0，见上文*/
    unsigned int no_dynamic_id:1;
    unsigned int supports_autosuspend:1;       /*1 表示允许自动睡眠，0 表示不允许自动睡眠*/
    unsigned int disable_hub_initiated_lpm:1;
    unsigned int soft_unbind:1;
};
```

usb_driver 结构体主要成员简介如下：

●**probe()**：探测函数，匹配接口时调用，必须定义。

●**disconnect()**：断开接口函数，必须定义。

●**id_table**：指向 usb_device_id 结构体数组（必须定义），用于匹配接口，结构体定义如下：

```
struct usb_device_id {      /*/include/linux/mod_devicetable.h*/
    __u16           match_flags;    /*匹配标记，匹配下面哪类信息*/

    /*厂商（设备）信息匹配*/
    __u16           idVendor;
    __u16           idProduct;
    __u16           bcdDevice_lo;
    __u16           bcdDevice_hi;
```

```
    /*设备类信息匹配*/
    __u8            bDeviceClass;
    __u8            bDeviceSubClass;
    __u8            bDeviceProtocol;

    /*接口类匹配*/
    __u8            bInterfaceClass;
    __u8            bInterfaceSubClass;
    __u8            bInterfaceProtocol;

    /* Used for vendor-specific interface matches */
    __u8            bInterfaceNumber;

    /* not matched against */
    kernel_ulong_t      driver_info
        __attribute__((aligned(sizeof(kernel_ulong_t))));
};
```

在/include/linux/usb.h 头文件中定义了匹配标记 match_flags 的取值，以及初始化 usb_device_id 实例的宏，例如：

```
#define USB_DEVICE_ID_MATCH_DEVICE \          /*匹配厂商、设备信息*/
        (USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH_PRODUCT)
...
#define USB_DEVICE(vend, prod) \
    .match_flags = USB_DEVICE_ID_MATCH_DEVICE, \
    .idVendor = (vend), \
    .idProduct = (prod)
```

●**dynids**：usb_dynids 结构体成员，定义如下（/include/linux/usb.h）：

```
struct usb_dynids {
    spinlock_t lock;
    struct list_head list;      /*双链表，管理动态 usb_device_id 实例*/
};
```

usb_dynids 结构体中 list 双链表管理的是 usb_dynid 结构体实例，定义如下（/include/linux/usb.h）：

```
struct usb_dynid {
    struct list_head node;
    struct usb_device_id id;        /*动态 usb_device_id 实例*/
};
```

在 USB 总线匹配函数中，接口与接口驱动检查是否匹配时，先将接口信息（描述字）与 id_table 指向 usb_device_id 实例列表中各项匹配，如果不匹配再与 dynids 链表管理的动态 usb_device_id 实例逐个匹配，以上有一个匹配成功，匹配函数返回 1，否则返回 0。

●**注册接口驱动**

注册 USB 接口驱动的接口函数声明如下（/include/linux/usb.h）：

int **usb_register_driver**(struct usb_driver *, struct module *,const char *);

```
#define usb_register(driver) \
        usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME)

#define module_usb_driver(__usb_driver) \          /*在模块中定义初始化函数，见下节*/
        module_driver(__usb_driver, usb_register, usb_deregister)
```

void usb_deregister(struct usb_driver *)：注销 usb_driver 实例。

以上注册函数最终都是调用 usb_register_driver()函数，定义如下（/drivers/usb/core/driver.c）：

```
int usb_register_driver(struct usb_driver *new_driver, struct module *owner,const char *mod_name)
/*owner：模块指针，mod_name：模块名称*/
{
    int retval = 0;

    if (usb_disabled())      /*受 nousb 变量控制，默认为 0，/drivers/usb/core/usb.c*/
        return -ENODEV;

    new_driver->drvwrap.for_devices = 0;          /*匹配接口，如果为 1 表示匹配 USB 设备*/
    new_driver->drvwrap.driver.name = new_driver->name;      /*驱动名称*/
    new_driver->drvwrap.driver.bus = &usb_bus_type;          /*USB 总线实例*/
    new_driver->drvwrap.driver.probe = usb_probe_interface;  /*调用 usb_driver->probe()函数*/
    new_driver->drvwrap.driver.remove = usb_unbind_interface;
    new_driver->drvwrap.driver.owner = owner;
    new_driver->drvwrap.driver.mod_name = mod_name;     /*模块名称*/
    spin_lock_init(&new_driver->dynids.lock);
    INIT_LIST_HEAD(&new_driver->dynids.list);

    retval = driver_register(&new_driver->drvwrap.driver);   /*注册驱动*/
    ...
    retval = usb_create_newid_files(new_driver);      /*在驱动对应的 sysfs 目录项下添加属性文件*/
    ...
out:
    return retval;
  ...
}
```

3 总线匹配函数

USB 设备、接口都挂载到 USB 总线上的设备链表，USB 设备驱动、接口驱动都挂到 USB 总线的驱动链表。设备驱动只能匹配设备，不能匹配接口，接口驱动只能匹配接口不能匹配设备。

USB 总线匹配函数定义如下（/drivers/usb/core/driver.c）：

```
static int usb_device_match(struct device *dev, struct device_driver *drv)
{
    if (is_usb_device(dev)) {       /*USB 设备，dev->type == &usb_device_type，/drivers/usb/core/usb.h*/

        if (!is_usb_device_driver(drv))     /*只有 USB 设备驱动才能匹配 USB 设备*/
            return 0;

        return 1;       /*USB 设备与任意 usb_device_driver 实例匹配*/

    } else if (is_usb_interface(dev)) {     /*接口，dev->type == &usb_if_device_type*/
        struct usb_interface *intf;
        struct usb_driver *usb_drv;
        const struct usb_device_id *id;

        if (is_usb_device_driver(drv))        /*只有 usb_driver 才能与接口匹配*/
            return 0;

        intf = to_usb_interface(dev);         /*usb_interface*/
        usb_drv = to_usb_driver(drv);         /*usb_driver*/

        id = usb_match_id(intf, usb_drv->id_table);     /*标识列表匹配，/drivers/usb/core/driver.c*/
        if (id)
            return 1;

        id = usb_match_dynamic_id(intf, usb_drv);       /*/drivers/usb/core/driver.c*/
        if (id)
            return 1;
    }
    return 0;
}
```

usb_device_match()函数只为表示 USB 设备和接口的 device 实例检查是否与驱动匹配，匹配流程如下：

（1）如果是 USB 设备 usb_device 实例且驱动是 usb_device_driver 实例则匹配，任意 usb_device_driver 实例可与任意 usb_device 实例匹配。usb_device 与所有的接口驱动 usb_driver 都不匹配。

（2）如果是 USB 接口 usb_interface 实例且驱动是 usb_driver 实例，则调用 usb_match_id()函数检查接口/设备中描述字信息，看其是否与 usb_driver->id_table 列表中某个列表项匹配（源代码请读者自行阅读），若匹配则函数返回 1，否则进行下一步。

（3）对于 USB 接口，如果 usb_match_id()函数检查不匹配，再调用 usb_match_dynamic_id()函数将接口/设备描述字中信息与 usb_driver->dynids.list 动态 usb_device_id 实例双链表中实例匹配，若匹配则返回 1，否则返回 0。

4 初始化

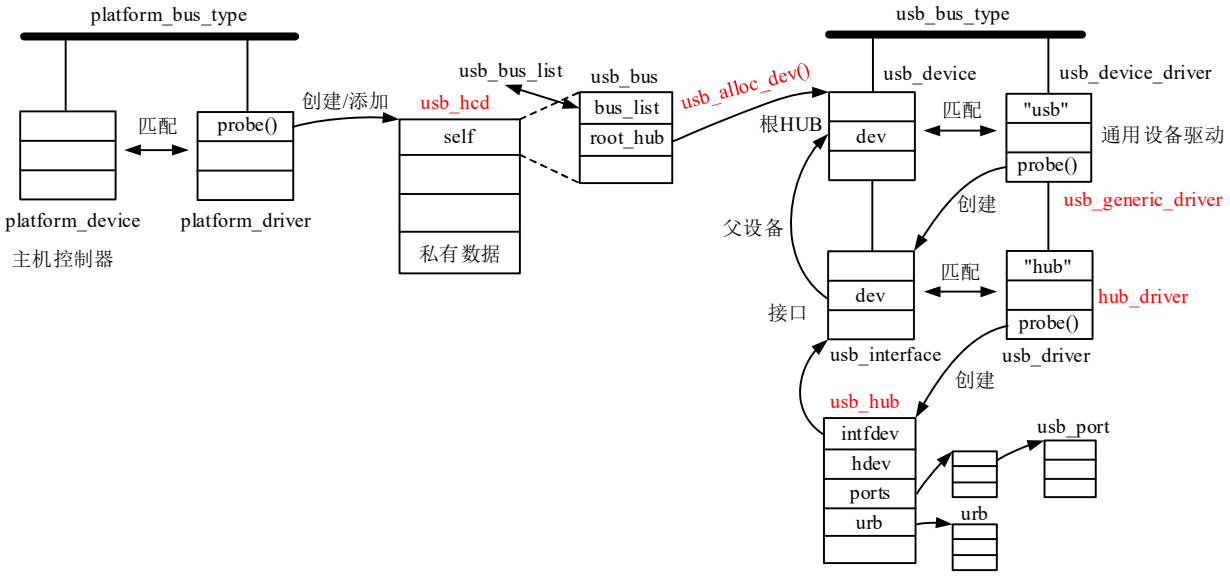
USB 总线驱动初始化函数 usb_init()定义在/drivers/usb/core/usb.c 文件内，代码如下：

```
static int __init usb_init(void)
{
    int retval;
    if (usb_disabled()) {
        ...
    }
    usb_init_pool_max();          /*初始化 pool_max[0]数组项，/drivers/usb/core/buffer.c*/

    retval = usb_debugfs_init();
                    /*在 debugfs 中创建"usb"目录项和"devices"文件，/drivers/usb/core/usb.c*/
    ...
    usb_acpi_register();      /*注册 acpi_bus_type 结构体 usb_acpi_bus 实例，/drivers/usb/core/usb-acpi.c*/
    retval = bus_register(&usb_bus_type);      /*注册 USB 总线实例*/
    ...
    retval = bus_register_notifier(&usb_bus_type, &usb_bus_nb); /*总线通知链上添加 usb_bus_nb 通知*/
    ...
    retval = usb_major_init();    /*注册 cdev 实例，主设备号为 USB_MAJOR，/drivers/usb/core/file.c*/
    ...
    retval = usb_register(&usbfs_driver);  /*注册 usb_driver 实例 usbfs_driver，/drivers/usb/core/devio.c*/
    ...
    retval = usb_devio_init();
                /*注册 cdev 实例，主设备号为 USB_DEVICE_MAJOR，/drivers/usb/core/devio.c*/
    ...
    retval = usb_hub_init();        /*HUB 驱动初始化（接口驱动），/drivers/usb/core/hub.c*/
                  /*注册 usb_driver 实例 hub_driver，创建 HUB 工作队列等，见本节下文*/
    ...
    retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
              /*注册 usb_device_driver 实例 usb_generic_driver，匹配所有 usb_device 实例*/
              /*/drivers/usb/core/generic.c*/
    ...
    return retval;
}
subsys_initcall(usb_init);          /*内核启动阶段调用此函数*/
```

8.9.5 HUB

USB 主机控制器中内嵌根 HUB，它被当作 USB 设备。在前面介绍的添加主机控制器 usb_hcd 实例时，将为主机控制器中根 HUB 创建表示 USB 设备的 usb_device 实例并注册。

注册 usb_device 实例时，将匹配通用的 USB 设备驱动 usb_generic_driver 实例，在其 probe()函数将为根 HUB 设备选择配置，获取接口信息，创建并注册接口 usb_interface 实例。usb_interface 实例将匹配 HUB 接口驱动的 usb_driver 结构体实例 **hub_driver**，其 probe()函数将创建并注册表示 HUB 的 usb_hub 实例等，如下图所示。



1 HUB 设备

在添加主机控制器 usb_hcd 实例的 usb_add_hcd()函数中，将创建和注册表示根 HUB 的 usb_device 实例，函数调用关系如下：

```
usb_add_hcd()      /*添加usb_hcd实例*/
    ├─► usb_alloc_dev()    /*创建表示根HUB的usb_device实例*/
    └─► register_root_hub()   /*注册表示根HUB的usb_device实例*/
```

■创建 USB 设备

创建 usb_device 实例的 **usb_alloc_dev**()函数定义如下（/drivers/usb/core/usb.c）：

```
struct usb_device *usb_alloc_dev(struct usb_device *parent,
                                 struct usb_bus *bus, unsigned port1)
/*parent：指向连接到的 HUB 的 usb_device 实例，此处为 NULL，bus：hcd->self，port1：此处为 0*/
{
    struct usb_device *dev;
    struct usb_hcd *usb_hcd = bus_to_hcd(bus);
    unsigned root_hub = 0;

    dev = kzalloc(sizeof(*dev), GFP_KERNEL);          /*分配 usb_device 实例*/
    ...
    if (!usb_get_hcd(usb_hcd)) {        /*引用计数加 1*/
        ...
    }
    /* Root hubs aren't true devices, so don't allocate HCD resources ，根 HUB 不需要分配 HCD 资源*/
    if (usb_hcd->driver->alloc_dev && parent &&!usb_hcd->driver->alloc_dev(usb_hcd, dev)) {
        ...
    }
```

```
    device_initialize(&dev->dev);              /*初始化设备*/
    dev->dev.bus = &usb_bus_type;              /*总线*/
    dev->dev.type = &usb_device_type;          /*设备类型*/
    dev->dev.groups = usb_device_groups;       /*属性组*/
    dev->dev.dma_mask = bus->controller->dma_mask;
    set_dev_node(&dev->dev, dev_to_node(bus->controller));
    dev->state = USB_STATE_ATTACHED;
    dev->lpm_disable_count = 1;
    atomic_set(&dev->urbnum, 0);

    INIT_LIST_HEAD(&dev->ep0.urb_list);
    dev->ep0.desc.bLength = USB_DT_ENDPOINT_SIZE;       /*端点 0 描述字长度*/
    dev->ep0.desc.bDescriptorType = USB_DT_ENDPOINT;    /*端点 0 描述字类型*/
    usb_enable_endpoint(dev, &dev->ep0, false);
                    /*dev->ep0 关联到 dev->ep_out[0]，端点 0 为输出，/drivers/usb/core/message.c*/
    dev->can_submit = 1;

    if (unlikely(!parent)) {          /*创建根 HUB 设备，没有父设备*/
        dev->devpath[0] = '0';
        dev->route = 0;
        dev->dev.parent = bus->controller;   /*表示主机控制器设备的 device 实例，嵌入到 xxx_device*/
        dev_set_name(&dev->dev, "usb%d", bus->busnum);
        root_hub = 1;         /*根 HUB*/
    } else {          /*普通设备*/
        ...
    }
    dev->portnum = port1;          /*根 HUB 为 0*/
    dev->bus = bus;                /*usb_hcd.usb_bus*/
    dev->parent = parent;          /*根 HUB 为 NULL*/
    INIT_LIST_HEAD(&dev->filelist);

#ifdef     CONFIG_PM
    pm_runtime_set_autosuspend_delay(&dev->dev,usb_autosuspend_delay * 1000);
    dev->connect_time = jiffies;
    dev->active_duration = -jiffies;
#endif
    if (root_hub)   /*根 HUB */
        dev->authorized = 1;
    else {
        dev->authorized = usb_hcd->authorized_default;
        dev->wusb = usb_bus_is_wusb(bus) ? 1 : 0;
    }
    return dev;         /*返回 usb_device 实例指针*/
}
```

■**注册根 HUB**

在添加 usb_hcd 实例的 usb_add_hcd()函数中将随后会调用 **register_root_hub()**函数注册表示根 HUB 设备的 usb_device 实例，函数定义如下（/drivers/usb/core/hcd.c）：

```
static int register_root_hub(struct usb_hcd *hcd)
{
    struct device *parent_dev = hcd->self.controller;          /*表示主机控制器的 device 实例*/
    struct usb_device *usb_dev = hcd->self.root_hub;           /*表示根 HUB 的 usb_device 实例*/
    const int devnum = 1;          /*根 HUB 设备地址为 1*/
    int retval;

    usb_dev->devnum = devnum;
    usb_dev->bus->devnum_next = devnum + 1;
    memset (&usb_dev->bus->devmap.devicemap, 0,sizeof usb_dev->bus->devmap.devicemap);
                                                               /*分配设备地址的位图清零*/
    set_bit (devnum, usb_dev->bus->devmap.devicemap);      /*设置根 HUB 地址 1 在位图中相应位*/
    usb_set_device_state(usb_dev, USB_STATE_ADDRESS);      /*设置设备状态*/

    mutex_lock(&usb_bus_list_lock);

    usb_dev->ep0.desc.wMaxPacketSize = cpu_to_le16(64);
                                        /*端点 0 在一个事务中可传输的最大数据长度（64 字节）*/
    retval = usb_get_device_descriptor(usb_dev, USB_DT_DEVICE_SIZE);
                /*读取设备描述字，写入 usb_dev.descriptor 成员，/drivers/usb/core/message.c*/
    ...    /*错误处理*/

    if (le16_to_cpu(usb_dev->descriptor.bcdUSB) >= 0x0201) {
                                                /*设备描述字中 BCD 形式的设备版本号*/
        retval = usb_get_bos_descriptor(usb_dev);
        if (!retval) {
            usb_dev->lpm_capable = usb_device_supports_lpm(usb_dev);
        } else if (usb_dev->speed == USB_SPEED_SUPER) {
            mutex_unlock(&usb_bus_list_lock);
            dev_dbg(parent_dev, "can't read %s bos descriptor %d\n",dev_name(&usb_dev->dev), retval);
            return retval;
        }
    }

    retval = usb_new_device (usb_dev);          /*注册 usb_device 实例，/drivers/usb/core/hub.c*/
    if (retval) {
        ...          /*注册失败*/
    } else {          /*注册成功*/
        spin_lock_irq (&hcd_root_hub_lock);
        hcd->rh_registered = 1;
```

```
        spin_unlock_irq (&hcd_root_hub_lock);

        if (HCD_DEAD(hcd))
            usb_hc_died (hcd); /* This time clean up */
    }
    mutex_unlock(&usb_bus_list_lock);
    return retval;          /*成功返回 0*/
}
```

usb_device 实例 ep0.desc 成员保存的是端点 0 描述字，descriptor 成员保存的是设备描述字。

register_root_hub()函数主要工作如下：

（1）将根 HUB 的设备地址设为 1，并设置地址位图中的位。

（2）设置 ep0.desc 成员，即端点 0 描述字。

（3）调用 usb_get_device_descriptor()函数读取设备描述字，写入 usb_dev.**descriptor** 成员，函数内最终调用 hc_driver 实例中的 **hub_control()**函数，实现控制传输（后面再介绍传输的实现）。

（4）调用 usb_new_device (usb_dev)函数注册表示根 HUB 的 usb_device 实例。

下面将单独介绍注册 usb_device 实例的 usb_new_device (usb_dev)函数实例，因为表示其它 USB 设备的 usb_device 实例也需要调用此函数注册。

2 注册 USB 设备

usb_new_device ()函数用于注册表示 USB 设备的 usb_device 实例，定义如下（/drivers/usb/core/hub.c）：

```
int usb_new_device(struct usb_device *udev)
{
    int err;

    if (udev->parent) {
        device_init_wakeup(&udev->dev, 0);
    }

    /* Tell the runtime-PM framework the device is active */
    pm_runtime_set_active(&udev->dev);
    pm_runtime_get_noresume(&udev->dev);
    pm_runtime_use_autosuspend(&udev->dev);
    pm_runtime_enable(&udev->dev);

    usb_disable_autosuspend(udev);          /*默认禁止所有设备自动睡眠*/

    err = usb_enumerate_device(udev);   /*主要是读取设备配置，/drivers/usb/core/hub.c*/
    ...
    udev->dev.devt = MKDEV(USB_DEVICE_MAJOR,
                    (((udev->bus->busnum-1) * 128) + (udev->devnum-1)));        /*生成设备号*/

    announce_device(udev);      /*输出信息，/drivers/usb/core/hub.c*/
```

```
        if (udev->serial)          /*输出设备信息*/
            add_device_randomness(udev->serial, strlen(udev->serial));
        if (udev->product)
            add_device_randomness(udev->product, strlen(udev->product));
        if (udev->manufacturer)
            add_device_randomness(udev->manufacturer,strlen(udev->manufacturer));

        device_enable_async_suspend(&udev->dev);          /*/include/linux/device.h*/

        /* check whether the hub or firmware marks this port as non-removable */
        if (udev->parent)
            set_usb_port_removable(udev);        /*设置设备接口移除*/

        err = device_add(&udev->dev);          /*添加设备，将创建设备文件*/
        ...
        /* Create link files between child device and usb port device.  创建符号链接文件*/
        if (udev->parent) {        /*存在父设备*/
            struct usb_hub *hub = usb_hub_to_struct_hub(udev->parent);     /*上层 HUB 的 usb_hub 实例*/
            int port1 = udev->portnum;
            struct usb_port*port_dev = hub->ports[port1 - 1];

            err = sysfs_create_link(&udev->dev.kobj,&port_dev->dev.kobj, "port");
            ...
            err = sysfs_create_link(&port_dev->dev.kobj,&udev->dev.kobj, "device");
            ...
            if (!test_and_set_bit(port1, hub->child_usage_bits))
                pm_runtime_get_sync(&port_dev->dev);
        }

        (void) usb_create_ep_devs(&udev->dev, &udev->ep0, udev);
                        /*创建并注册表示端点的 ep_device 实例，/drivers/usb/core/endpoint.c*/
        usb_mark_last_busy(udev);
        pm_runtime_put_sync_autosuspend(&udev->dev);
        return err;
        ...
    }
```

在调用 usb_new_device ()函数注册 usb_device 实例前，已经读取了设备描述字，函数内执行的主要工作如下：

（1）调用 usb_enumerate_device()函数读取设备配置描述字。

（2）添加 usb_device 实例内嵌 device 实例，添加时将会与通用 USB 设备驱动 usb_device_driver 实例 usb_generic_driver 匹配，并调用其 probe()函数，probe()函数将会选择配置，读取接口描述字，创建并注册接口 usb_interface 实例。

（3）调用 usb_create_ep_devs()函数创建表示端点 0 的 ep_device 实例。

下面分别对以上三个步骤做简要介绍。

■获取设备配置

usb_enumerate_device()函数主要用于获取设备配置，定义如下（/drivers/usb/core/hub.c）：

```
static int usb_enumerate_device(struct usb_device *udev)
{
    int err;
    struct usb_hcd *hcd = bus_to_hcd(udev->bus);

    if (udev->config == NULL) {
        err = usb_get_configuration(udev);
          /*获取设备配置，存入 usb_device 实例，/drivers/usb/core/config.c*/
        ...
    }

    /* read the standard strings and cache them if present */
    udev->product = usb_cache_string(udev, udev->descriptor.iProduct);
    udev->manufacturer = usb_cache_string(udev,udev->descriptor.iManufacturer);
    udev->serial = usb_cache_string(udev, udev->descriptor.iSerialNumber);
    ...
    return 0;
}
```

■通用 USB 设备驱动

usb_new_device ()函数读取设备配置后，将添加 udev->dev（device）实例，它将与 usb_generic_driver 通用 USB 设备驱动匹配，并调用其 probe()函数，实例定义如下（/drivers/usb/core/generic.c）：

```
struct usb_device_driver usb_generic_driver = {
    .name =   "usb",
    .probe = generic_probe,          /*探测函数*/
    .disconnect = generic_disconnect,
#ifdef     CONFIG_PM
    .suspend = generic_suspend,
    .resume = generic_resume,
#endif
    .supports_autosuspend = 1,
};
```

probe()探测函数 generic_probe()将选择设备配置，读取接口描述字，创建并注册接口 usb_interface 实例，注册 usb_interface 实例时将查找匹配的 usb_driver 驱动，并调用其 probe()函数，完成驱动程序的加载。

generic_probe()函数代码简列如下（/drivers/usb/core/generic.c）：

```
static int generic_probe(struct usb_device *udev)
{
```

```
    int err, c;

    if (udev->authorized == 0)
        dev_err(&udev->dev, "Device is not authorized for usage\n");
    else {
        c = usb_choose_configuration(udev);   /*选择配置，返回配置编号，/drivers/usb/core/generic.c*/
        if (c >= 0) {
            err = usb_set_configuration(udev, c);       /*/drivers/usb/core/message.c*/
             /*读取配置下的接口描述字，创建并注册接口 usb_interface 实例*/
            ...
        }
    }
    /* USB device state == configured ... usable */
    usb_notify_add_device(udev);
    return 0;
}
```

usb_choose_configuration()函数用于选择设备配置，usb_set_configuration()函数读取配置下的接口描述字，创建并注册接口 usb_interface 实例，其设备类型为 usb_if_device_type，源代码请读者自行阅读。

■创建端点实例

usb_new_device()函数最后调用 usb_create_ep_devs()函数创建表示端点 0 的 ep_device 实例，函数定义如下（/drivers/usb/core/endpoint.c）：

```
int usb_create_ep_devs(struct device *parent,struct usb_host_endpoint *endpoint,struct usb_device *udev)
{
    struct ep_device *ep_dev;       /*/drivers/usb/core/endpoint.c*/
    int retval;

    ep_dev = kzalloc(sizeof(*ep_dev), GFP_KERNEL);
    ...
    ep_dev->desc = &endpoint->desc;
    ep_dev->udev = udev;
    ep_dev->dev.groups = ep_dev_groups;
    ep_dev->dev.type = &usb_ep_device_type;
    ep_dev->dev.parent = parent;        /*父设备为 usb_device 实例*/
    dev_set_name(&ep_dev->dev, "ep_%02x", endpoint->desc.bEndpointAddress);

    retval = device_register(&ep_dev->dev);
    ...
    device_enable_async_suspend(&ep_dev->dev);
    endpoint->ep_dev = ep_dev;
    return retval;
    ...
}
```

创建端点 ep_device 实例的主要用途是将端点信息（属性）导出到用户空间（sysfs 文件系统）。

3 HUB 接口驱动

注册 HUB 设备 usb_device 实例时，会匹配通用的 usb 设备驱动 usb_generic_driver，在其 prob()函数中将选择设备配置，读取接口描述字，创建并注册接口 usb_interface 实例。在注册 usb_interface 实例时会查找匹配的接口驱动 usb_driver 实例，若匹配则调用 usb_driver 实例中的 probe()函数。

HUB 接口匹配的 usb_driver 实例为 hub_driver，这里的接口并不是 HUB 上的物理 USB 插口，而是其作为 USB 设备的逻辑接口（功能）。

hub_driver 实例定义如下（/drivers/usb/core/hub.c）：

```
static struct usb_driver hub_driver = {
    .name =   "hub",
    .probe =   hub_probe,          /*探测函数*/
    .disconnect =   hub_disconnect,
    .suspend =      hub_suspend,
    .resume =hub_resume,
    .reset_resume =      hub_reset_resume,
    .pre_reset =    hub_pre_reset,
    .post_reset =   hub_post_reset,
    .unlocked_ioctl = hub_ioctl,
    .id_table =     hub_id_table,       /*匹配列表*/
    .supports_autosuspend = 1,
};
```

hub_driver 驱动中匹配列表 hub_id_table 定义如下：

```
static const struct usb_device_id hub_id_table[] = {
    { .match_flags = USB_DEVICE_ID_MATCH_VENDOR
            | USB_DEVICE_ID_MATCH_INT_CLASS,
      .idVendor = USB_VENDOR_GENESYS_LOGIC,
      .bInterfaceClass = USB_CLASS_HUB,          /*HUB 类*/
      .driver_info = HUB_QUIRK_CHECK_PORT_AUTOSUSPEND},
    { .match_flags = USB_DEVICE_ID_MATCH_DEV_CLASS,
      .bDeviceClass = USB_CLASS_HUB},            /*HUB 类*/
    { .match_flags = USB_DEVICE_ID_MATCH_INT_CLASS,
      .bInterfaceClass = USB_CLASS_HUB},         /*HUB 类*/
    { }
};
```

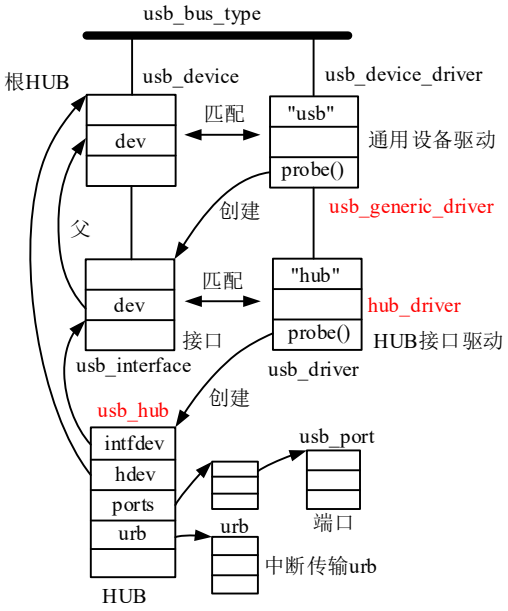
在初始化函数 usb_hub_init()中将注册 hub_driver 实例，并创建工作队列，代码简列如下：

```
int usb_hub_init(void)          /*/drivers/usb/core/hub.c*/
{
    if (usb_register(&hub_driver) < 0) {          /*注册 hub_driver 实例*/
        ...
    }
```

```
    hub_wq = alloc_workqueue("usb_hub_wq", WQ_FREEZABLE, 0);        /*创建工作队列*/
    if (hub_wq)
        return 0;
    ...
}
```

■usb_hub

在 hub_driver 实例的 prob()函数 hub_probe()中将创建并设置 usb_hub 结构体实例，如下图所示：



usb_hub 结构体定义如下（/drivers/usb/core/hub.h）：

```
struct usb_hub {
    struct device       *intfdev;       /*指向 usb_interface 中 device 实例*/
    struct usb_device   *hdev;          /*指向表示 HUB 的 usb_device 实例*/
    struct kref         kref;           /*引用计数*/
    struct urb      *urb;               /*用于中断轮询的 urb*/

    /* buffer for urb ... with extra space in case of babble */
    u8          (*buffer)[8];           /*传输数据缓存，8 个缓存指针*/
    union {
        struct usb_hub_status   hub;
        struct usb_port_status  port;
    }*status;       /* buffer for status reports */
    struct mutex        status_mutex;       /* for the status buffer */

    int         error;          /* last reported error */
    int         nerrors;        /* track consecutive errors */
    /*端口位标记*/
    unsigned long       event_bits[1];      /* status change bitmask */
    unsigned long       change_bits[1];     /* ports with logical connectstatus change */
    unsigned long       removed_bits[1];    /* ports with a "removed"device present */
```

```
        unsigned long       wakeup_bits[1];     /* ports that have signaledremote wakeup */
        unsigned long       power_bits[1];      /* ports that are powered */
        unsigned long       child_usage_bits[1]; /* ports powered on forchildren */
        unsigned long       warm_reset_bits[1]; /* ports requesting warmreset recovery */
        ...
        struct usb_hub_descriptor *descriptor;      /*指向 HUB 类描述符*/
        struct usb_tt       tt;         /*事务转换器*/

        unsigned        mA_per_port;        /* current for each child */
        #ifdef      CONFIG_PM
        unsigned        wakeup_enabled_descendants;
        #endif

        unsigned        limited_power:1;
        unsigned        quiescing:1;
        unsigned        disconnected:1;
        unsigned        in_reset:1;
        unsigned        quirk_check_port_auto_suspend:1;
        unsigned        has_indicators:1;
        u8              indicator[USB_MAXCHILDREN];
        struct delayed_work     leds;
        struct delayed_work     init_work;
        struct work_struct      events;         /*探测接口连接，创建设备*/
        struct usb_port     **ports;        /*usb_port 结构体指针数组*/
    };
```

usb_hub 结构体中 ports 成员指向指针数组，数组项指向 usb_port 实例，表示 HUB 中端口，usb_port 结构体定义如下（/drivers/usb/core/hub.h）：

```
    struct usb_port {
        struct usb_device *child;
        struct device dev;
        struct usb_dev_state *port_owner;
        struct usb_port *peer;
        struct dev_pm_qos_request *req;
        enum usb_port_connect_type connect_type;
        usb_port_location_t location;
        struct mutex status_lock;
        u8 portnum;                     /*端口编号*/
        unsigned int is_superspeed:1;
    };
```

■探测函数

HUB 接口驱动 hub_driver 实例探测函数 hub_probe()代码简列如下（/drivers/usb/core/hub.c）：

```
static int hub_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
    struct usb_host_interface *desc;        /*接口设置（接口描述字）*/
    struct usb_endpoint_descriptor *endpoint;        /*端点描述字*/
    struct usb_device *hdev;
    struct usb_hub *hub;

    desc = intf->cur_altsetting;        /*当前接口设置*/
    hdev = interface_to_usbdev(intf);        /*表示 HUB 的 usb_device 实例*/

#ifdef CONFIG_PM
    if (hdev->dev.power.autosuspend_delay >= 0)
        pm_runtime_set_autosuspend_delay(&hdev->dev, 0);
#endif

    if (hdev->parent) {        /*非根 HUB*/
        usb_enable_autosuspend(hdev);
    } else {                /*根 HUB*/
        const struct hc_driver *drv = bus_to_hcd(hdev->bus)->driver;        /*hc_driver 实例*/
        if (drv->bus_suspend && drv->bus_resume)
            usb_enable_autosuspend(hdev);
    }

    if (hdev->level == MAX_TOPO_LEVEL) {
        ...        /*错误处理*/
    }

#ifdef    CONFIG_USB_OTG_BLACKLIST_HUB
    ...
#endif
    ...    /*检测描述字错误*/

    /*只能有一个端点？？*/
    if (desc->desc.bNumEndpoints != 1)
        goto descriptor_error;

    endpoint = &desc->endpoint[0].desc;        /*端点 0 描述字*/

    if (!usb_endpoint_is_int_in(endpoint))        /*端点 0 必须是中断输入，/include/uapi/linux/usb/ch9.h*/
        goto descriptor_error;

    dev_info (&intf->dev, "USB hub found\n");
```

```
    hub = kzalloc(sizeof(*hub), GFP_KERNEL);          /*分配 usb_hub 实例*/
    ...
    /*初始化 usb_hub 实例*/
    kref_init(&hub->kref);
    hub->intfdev = &intf->dev;
    hub->hdev = hdev;       /*指向表示 HUB 设备的 usb_device 实例*/
    INIT_DELAYED_WORK(&hub->leds, led_work);       /*延时工作，/drivers/usb/core/hub.c*/
    INIT_DELAYED_WORK(&hub->init_work, NULL);
    INIT_WORK(&hub->events, hub_event);          /*工作执行函数为 hub_event()*/
    usb_get_intf(intf);
    usb_get_dev(hdev);

    usb_set_intfdata (intf, hub);         /*接口的驱动数据指向 usb_hub 实例*/
    intf->needs_remote_wakeup = 1;
    pm_suspend_ignore_children(&intf->dev, true);

    if (hdev->speed == USB_SPEED_HIGH)
        highspeed_hubs++;

    if (id->driver_info & HUB_QUIRK_CHECK_PORT_AUTOSUSPEND)
        hub->quirk_check_port_auto_suspend = 1;

    if (hub_configure(hub, endpoint) >= 0)          /*HUB 配置，/drivers/usb/core/hub.c*/
        return 0;
    ...
}
```

探测函数的主要工作是创建并设置 usb_hub 实例，初始化 events 工作执行函数为 **hub_event()**（用于处理端口事件，如插入设备），获取 hub 端点 0 描述字，调用 hub_configure()函数配置 HUB。

hub_configure()函数主要工作是读取 HUB 描述字（类描述字），探测 HUB 端口，创建 usb_port 实例，创建用于中断传输的 urb 实例等。下面将简要介绍 hub_configure()函数的实现。

●配置 HUB

hub_configure()函数代码简列如下（/drivers/usb/core/hub.c）：

```
static int hub_configure(struct usb_hub *hub,struct usb_endpoint_descriptor *endpoint)
/*hub：指向 usb_hub 实例，endpoint：指向端点 0 描述字*/
{
    struct usb_hcd *hcd;
    struct usb_device *hdev = hub->hdev;
    struct device *hub_dev = hub->intfdev;
    ...
    hub->buffer = kmalloc(sizeof(*hub->buffer), GFP_KERNEL);     /*分配缓存指针数组*/
    ...
    hub->status = kmalloc(sizeof(*hub->status), GFP_KERNEL);        /*HUB 状态*/
    ...
```

```
hub->descriptor = kmalloc(sizeof(*hub->descriptor), GFP_KERNEL);   /*为 HUB 描述字分配空间*/
....
ret = get_hub_descriptor(hdev, hub->descriptor);        /*读取 HUB 描述字*/
...
maxchild = hub->descriptor->bNbrPorts;        /*端口数量*/
...
hub->ports = kzalloc(maxchild * sizeof(struct usb_port *), GFP_KERNEL);
                                                        /*分配端口 usb_port  实例指针数组*/
...
spin_lock_init (&hub->tt.lock);
INIT_LIST_HEAD (&hub->tt.clear_list);
INIT_WORK(&hub->tt.clear_work, hub_tt_work);
...
ret = usb_get_status(hdev, USB_RECIP_DEVICE, 0, &hubstatus);
...
hcd = bus_to_hcd(hdev->bus);          /*usb_hcd*/
...             /*设置电流*/
ret = hub_hub_status(hub, &hubstatus, &hubchange);        /*HUB 状态*/
...
pipe = usb_rcvintpipe(hdev, endpoint->bEndpointAddress);
maxp = usb_maxpacket(hdev, pipe, usb_pipeout(pipe));

if (maxp > sizeof(*hub->buffer))
        maxp = sizeof(*hub->buffer);

hub->urb = usb_alloc_urb(0, GFP_KERNEL);              /*分配 urb 实例，用于中断传输，见下文*/
...
usb_fill_int_urb(hub->urb, hdev, pipe, *hub->buffer, maxp, hub_irq,hub, endpoint->bInterval);
                                                        /*填充 urb 实例*/
if (hub->has_indicators && blinkenlights)
        hub->indicator[0] = INDICATOR_CYCLE;

mutex_lock(&usb_port_peer_mutex);
for (i = 0; i < maxchild; i++) {
        ret = usb_hub_create_port_device(hub, i + 1);     /*分配/设置/注册端口 usb_port  实例*/
                                                        /*/drivers/usb/core/port.c*/
        ...
}
hdev->maxchild = i;
for (i = 0; i < hdev->maxchild; i++) {
        struct usb_port *port_dev = hub->ports[i];
        pm_runtime_put(&port_dev->dev);
}
...
```

```
    if (hcd->driver->update_hub_device) {
        ret = hcd->driver->update_hub_device(hcd, hdev,&hub->tt, GFP_KERNEL);
        ...
    }
    usb_hub_adjust_deviceremovable(hdev, hub->descriptor);

    hub_activate(hub, HUB_INIT);        /*激活 HUB，/drivers/usb/core/hub.c*/
    return 0;
    ...
}
```

hub_configure()函数读取 HUB 描述字，依此设置 usb_hub 实例，创建并设置用于端点 0 中断传输的 urb 实例，最后调用 hub_activate()函数激活 HUB 及其端口。hub_activate()函数用于分步骤激活 HUB，函数源代码请读者自行阅读。

8.9.6 数据传输

由前面的介绍我们知道，读取设备配置、描述字等信息时，就已经需要数据传输了，这里使用的是控制传输，数据传输由主机控制器实现。

在 USB 驱动程序中，传输的数据由 **urb** 结构体封装。USB 驱动核心层提供了实现 urb 实例填充和提交的接口函数。

USB 支持四种基本的数据传输模式：控制传输、同步传输、中断传输、批量传输。

每个数据传输都由 urb 结构体表示，发起传输的主要流程是：

（1）创建 urb 实例。

（2）填充 urb 实例。

（3）提交 urb 实例。

执行完以上 3 个步骤后，USB 主机控制器会完成传输，完成后调用 urb 实例中的 complete()回调函数。通常按照上面 3 个步骤执行的数据传输是异步传输，即调用者提交 urb 实例后，提交函数即返回，在传输完成后由 USB 驱动核心层调用 urb 实例中的 complete()回调函数。

同步传输是指在提交 urb 实例后等待，等待传输完成后提交操作才返回。

下面先介绍 urb 结构体定义，然后介绍创建、填充、提交 urb 实例函数的实现，以及同步传输的接口函数。

1 urb

USB 总线每个数据传输由 urb 结构体表示（USB Request Block），结构体定义在/include/linux/usb.h 头文件：

```
struct urb {
    /*USB 驱动核心层和主机控制器私有数据*/
    struct kref kref;           /*引用计数*/
    void *hcpriv;               /*主机控制器私有数据*/
    atomic_t use_count;         /* concurrent submissions counter */
    atomic_t reject;            /* submissions will fail */
    int unlinked;               /* unlink error code */
```

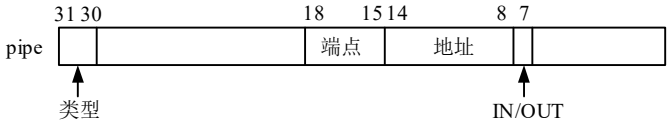
```
    /*公共数据，驱动可使用的数据*/
    struct list_head urb_list;      /*双链表成员，将 urb 实例添加到管理双链表*/
    struct list_head anchor_list;   /* the URB may be anchored */
    struct usb_anchor *anchor;
    struct usb_device *dev;         /*所属 USB 设备的 usb_device 实例*/
    struct usb_host_endpoint *ep;       /*传输端点*/
    unsigned int pipe;              /*传输类型（管道）*/
    unsigned int stream_id;         /*输入，流 ID*/
    int status;                     /* (return) non-ISO status */
    unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
    void *transfer_buffer;          /*指向数据缓存区*/
    dma_addr_t transfer_dma;        /* (in) dma addr for transfer_buffer */
    struct scatterlist *sg;         /* (in) scatter gather buffer list */
    int num_mapped_sgs;             /* (internal) mapped sg entries */
    int num_sgs;                    /*sg 链表中实例数量*/
    u32 transfer_buffer_length;     /*数据缓存区长度*/
    u32 actual_length;          /*返回值，表示实际传输的数据长度*/
    unsigned char *setup_packet;    /* (in)  控制传输设置阶段数据包*/
    dma_addr_t setup_dma;           /* (in) dma addr for setup_packet */
    int start_frame;                /* (modify) start frame (ISO) */
    int number_of_packets;          /* (in) number of ISO packets */
    int interval;               /* (modify) transfer interval* (INT/ISO) ，传输间隔，中断或等时传输*/
    int error_count;            /* (return) number of ISO errors */
    void *context;              /* (in) context for completion */
    usb_complete_t complete;    /*urb 传输完成后的回调函数*/
    struct usb_iso_packet_descriptor iso_frame_desc[0];
};
```

urb 结构体主要成员简介如下：

●**urb_list**：双链表成员，将 urb 实例添加到管理双链表。

●**ep**：指向表示传输端点的 usb_host_endpoint 实例。

●**pipe**：表示传输端点、传输类型等参数，布局如下：



bit7：表示输出还是输入，0 为输出，1 为输入（主机视角）。

bit[8,14]：表示设备地址，bit[15,18]：端点编号。

bit[30,31]：表示传输类型，00 为等时传输，01 为中断传输，10 为控制传输，11 为批量传输。

在/include/linux/usb.h 头文件定义了生成 pipe 值的函数，例如：

usb_sndctrlpipe(dev, endpoint)：生成向设备 dev，端点 endpoint 发起控制传输的 pipe 值。

●**transfer_buffer**：数据缓存区指针，缓存区必须是动态分配的。

●**transfer_flags**：传输标记，取值定义如下（/include/linux/usb.h）：

```
#define    URB_SHORT_NOT_OK      0x0001    /* report short reads as errors */
```

```
#define   URB_ISO_ASAP              0x0002   /*适用于等时传输*/
#define   URB_NO_TRANSFER_DMA_MAP   0x0004   /* urb->transfer_dma valid on submit */
#define   URB_NO_FSBR               0x0020   /* UHCI-specific */
#define   URB_ZERO_PACKET           0x0040   /* Finish bulk OUT with short packet */
#define   URB_NO_INTERRUPT          0x0080   /**/
#define   URB_FREE_BUFFER           0x0100   /* Free transfer buffer with the URB */
...
```

●**complete**：urb 传输完成的回调函数指针，函数原型为（/include/linux/usb.h）：

```
typedef void (*usb_complete_t)(struct urb *);
```

2 分配 urb 实例

内核在/drivers/usb/core/urb.c 文件内定义了创建 urb 实例的接口函数 **usb_alloc_urb**()，定义如下：

```
struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
/*iso_packets：等时数据包数量，中断、控制、批量传输时为 0，mem_flags：内存分配标记*/
{
    struct urb *urb;

    urb = kmalloc(sizeof(struct urb) +
        iso_packets * sizeof(struct usb_iso_packet_descriptor),mem_flags);      /*分配 urb 实例*/
    ...
    usb_init_urb(urb);      /*初始化 urb 实例，清零、初始化双链表成员等*/
    return urb;         /*返回 urb 实例指针*/
}
```

void **usb_free_urb**(struct urb *urb)：释放 urb 实例。

3 填充 urb 实例

内核在/include/linux/usb.h 头文件定义了填充 urb 实例的接口函数。

填充控制传输 urb 实例的函数如下所示：

```
static inline void usb_fill_control_urb(struct urb *urb,struct usb_device *dev,unsigned int pipe,
                    unsigned char *setup_packet,void *transfer_buffer,int buffer_length,
                    usb_complete_t complete_fn,void *context)
/*setup_packet：指向设置阶段数据缓存，transfer_buffer：数据阶段缓存， complete_fn：回调函数*/
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->setup_packet = setup_packet;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
}
```

填充批量传输 urb 实例的函数如下：

```
static inline void usb_fill_bulk_urb(struct urb *urb,struct usb_device *dev,unsigned int pipe,
                        void *transfer_buffer,int buffer_length,usb_complete_t complete_fn,void *context)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;
}
```

填充中断传输 urb 实例的函数如下：

```
static inline void usb_fill_int_urb(struct urb *urb,struct usb_device *dev,unsigned int pipe,
                        void *transfer_buffer,int buffer_length,usb_complete_t complete_fn,
                        void *context,int interval)
{
    urb->dev = dev;
    urb->pipe = pipe;
    urb->transfer_buffer = transfer_buffer;
    urb->transfer_buffer_length = buffer_length;
    urb->complete = complete_fn;
    urb->context = context;

    if (dev->speed == USB_SPEED_HIGH || dev->speed == USB_SPEED_SUPER) {
        interval = clamp(interval, 1, 16);
        urb->interval = 1 << (interval - 1);
    } else {
        urb->interval = interval;          /*传输间隔*/
    }

    urb->start_frame = -1;
}
```

4 提交 urb 实例

usb_submit_urb()函数用于提交 urb 实例，函数代码简列如下（/drivers/usb/core/urb.c）：

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
{
    static int  pipetypes[4] = {            /*传输（管道类型）*/
        PIPE_CONTROL, PIPE_ISOCHRONOUS, PIPE_BULK, PIPE_INTERRUPT
    };
```

```
int    xfertype, max;
struct usb_device          *dev;              /*USB 设备*/
struct usb_host_endpoint       *ep;           /*端点*/
int                        is_out;
unsigned int               allowed;

if (!urb || !urb->complete)          /*urb 必须定义 complete()函数*/
      return -EINVAL;
...
dev = urb->dev;          /*usb_device 实例*/
if ((!dev) || (dev->state < USB_STATE_UNAUTHENTICATED))
      return -ENODEV;

ep = usb_pipe_endpoint(dev, urb->pipe);        /*获取传输端点信息，/include/linux/usb.h*/
...                /*错误处理*/

urb->ep = ep;
urb->status = -EINPROGRESS;
urb->actual_length = 0;

/*安全检查*/
xfertype = usb_endpoint_type(&ep->desc);
if (xfertype == USB_ENDPOINT_XFER_CONTROL) {          /*控制端点*/
      struct usb_ctrlrequest *setup =(struct usb_ctrlrequest *) urb->setup_packet;

      if (!setup)
            return -ENOEXEC;
      is_out = !(setup->bRequestType & USB_DIR_IN) ||!setup->wLength;
} else {
      is_out = usb_endpoint_dir_out(&ep->desc);
}

/* Clear the internal flags and cache the direction for later use */
urb->transfer_flags &= ~(URB_DIR_MASK | URB_DMA_MAP_SINGLE |
      URB_DMA_MAP_PAGE | URB_DMA_MAP_SG | URB_MAP_LOCAL |
      URB_SETUP_MAP_SINGLE | URB_SETUP_MAP_LOCAL |
      URB_DMA_SG_COMBINED);
urb->transfer_flags |= (is_out ? URB_DIR_OUT : URB_DIR_IN);        /*传输标记*/

if (xfertype != USB_ENDPOINT_XFER_CONTROL &&dev->state < USB_STATE_CONFIGURED)
      return -ENODEV;

max = usb_endpoint_maxp(&ep->desc);        /*端点最大传输数据大小*/
...
```

```
/*周期传输（中断、等时）中一个帧或微帧内的数据限制/
if (xfertype == USB_ENDPOINT_XFER_ISOC) {
        int    n, len;

        /* SuperSpeed isoc endpoints have up to 16 bursts of up to
         * 3 packets each
         */
        if (dev->speed == USB_SPEED_SUPER) {
                int         burst = 1 + ep->ss_ep_comp.bMaxBurst;
                int         mult = USB_SS_MULT(ep->ss_ep_comp.bmAttributes);
                max *= burst;
                max *= mult;
        }

        /* "high bandwidth" mode, 1-3 packets/uframe? */
        if (dev->speed == USB_SPEED_HIGH) {
                int    mult = 1 + ((max >> 11) & 0x03);
                max &= 0x07ff;
                max *= mult;
        }

        if (urb->number_of_packets <= 0)
                return -EINVAL;
        for (n = 0; n < urb->number_of_packets; n++) {
                len = urb->iso_frame_desc[n].length;
                if (len < 0 || len > max)
                        return -EMSGSIZE;
                urb->iso_frame_desc[n].status = -EXDEV;
                urb->iso_frame_desc[n].actual_length = 0;
        }
} else if (urb->num_sgs && !urb->dev->bus->no_sg_constraint &&
                dev->speed != USB_SPEED_WIRELESS) {
        struct scatterlist *sg;
        int i;

        for_each_sg(urb->sg, sg, urb->num_sgs - 1, i)
                if (sg->length % max)
                        return -EINVAL;
}

/* the I/O buffer must be mapped/unmapped, except when length=0 */
if (urb->transfer_buffer_length > INT_MAX) /*缓存区长度不能超过限制值，/include/linux/kernel.h*/
        return -EMSGSIZE;
```

```
/*类型检查*/
if (usb_pipetype(urb->pipe) != pipetypes[xfertype])
        dev_WARN(&dev->dev, "BOGUS urb xfer, pipe %x != type %x\n",
                usb_pipetype(urb->pipe), pipetypes[xfertype]);

/* Check against a simple/standard policy */
allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_INTERRUPT | URB_DIR_MASK |
                URB_FREE_BUFFER);
switch (xfertype) {             /*传输类型*/
case USB_ENDPOINT_XFER_BULK:            /*批量和中断*/
case USB_ENDPOINT_XFER_INT:
        if (is_out)
                allowed |= URB_ZERO_PACKET;
/* FALLTHROUGH */
case USB_ENDPOINT_XFER_CONTROL:         /*控制传输*/
        allowed |= URB_NO_FSBR;         /* only affects UHCI */
/* FALLTHROUGH */
default:                        /* all non-iso endpoints */
        if (!is_out)
                allowed |= URB_SHORT_NOT_OK;
        break;
case USB_ENDPOINT_XFER_ISOC:
        allowed |= URB_ISO_ASAP;
        break;
}
allowed &= urb->transfer_flags;

if (allowed != urb->transfer_flags)
        dev_WARN(&dev->dev, "BOGUS urb flags, %x --> %x\n",urb->transfer_flags, allowed);

/*使周期传输的间隔合法*/
switch (xfertype) {
case USB_ENDPOINT_XFER_ISOC:
case USB_ENDPOINT_XFER_INT:
        /* too small? */
        switch (dev->speed) {
        case USB_SPEED_WIRELESS:
                if ((urb->interval < 6)&& (xfertype == USB_ENDPOINT_XFER_INT))
                        return -EINVAL;
        default:
                if (urb->interval <= 0)
                        return -EINVAL;
                break;
```

```
        }
        /* too big? */
        switch (dev->speed) {
        case USB_SPEED_SUPER:  /* units are 125us */
        /* Handle up to 2^(16-1) microframes */
            if (urb->interval > (1 << 15))
                return -EINVAL;
            max = 1 << 15;
            break;
        case USB_SPEED_WIRELESS:
            if (urb->interval > 16)
                return -EINVAL;
            break;
        case USB_SPEED_HIGH:    /* units are microframes */
            /* NOTE usb handles 2^15 */
            if (urb->interval > (1024 * 8))
                urb->interval = 1024 * 8;
            max = 1024 * 8;
            break;
        case USB_SPEED_FULL:    /* units are frames/msec */
        case USB_SPEED_LOW:
            if (xfertype == USB_ENDPOINT_XFER_INT) {
                if (urb->interval > 255)
                    return -EINVAL;
                /* NOTE ohci only handles up to 32 */
                max = 128;
            } else {
                if (urb->interval > 1024)
                    urb->interval = 1024;
                /* NOTE usb and ohci handle up to 2^15 */
                max = 1024;
            }
            break;
        default:
            return -EINVAL;
        }
        if (dev->speed != USB_SPEED_WIRELESS) {
            /* Round down to a power of 2, no more than max */
            urb->interval = min(max, 1 << ilog2(urb->interval));
        }
    }
    return usb_hcd_submit_urb(urb, mem_flags);      /*向 HCD 提交 urb，/drivers/usb/core/hcd.c*/
}
```

简而言之，usb_submit_urb()函数的主要工作就是对urb实例进行检查，合格后调用usb_hcd_submit_urb()函数将 urb 实例提交给主机控制器。

■urb 提交给主机控制器

usb_hcd_submit_urb()函数定义在/drivers/usb/core/hcd.c 文件内，代码简列如下：

```
int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
{
    int    status;
    struct usb_hcd   *hcd = bus_to_hcd(urb->dev->bus);       /*主机控制器*/

    /*增加引用计数*/
    usb_get_urb(urb);
    atomic_inc(&urb->use_count);
    atomic_inc(&urb->dev->urbnum);
    usbmon_urb_submit(&hcd->self, urb);       /*可能为空，/include/linux/usb/hcd.h*/

    if (is_root_hub(urb->dev)) {          /*与根 HUB 传输数据*/
        status = rh_urb_enqueue(hcd, urb);              /*实例控制传输等，/drivers/usb/core/hcd.c*/
    } else {                /*不是与根 HUB 传输数据*/
        status = map_urb_for_dma(hcd, urb, mem_flags);
        if (likely(status == 0)) {
            status = hcd->driver->urb_enqueue(hcd, urb, mem_flags);
            if (unlikely(status))
                unmap_urb_for_dma(hcd, urb);
        }
    }

    if (unlikely(status)) {
        usbmon_urb_submit_error(&hcd->self, urb, status);
        urb->hcpriv = NULL;
        INIT_LIST_HEAD(&urb->urb_list);
        atomic_dec(&urb->use_count);
        atomic_dec(&urb->dev->urbnum);
        if (atomic_read(&urb->reject))
            wake_up(&usb_kill_urb_queue);
        usb_put_urb(urb);
    }
    return status;
}
```

usb_hcd_submit_urb()函数对 urb 的提交要分情况而定，如下所示（如下图所示）：

1、与根 HUB 的数据传输

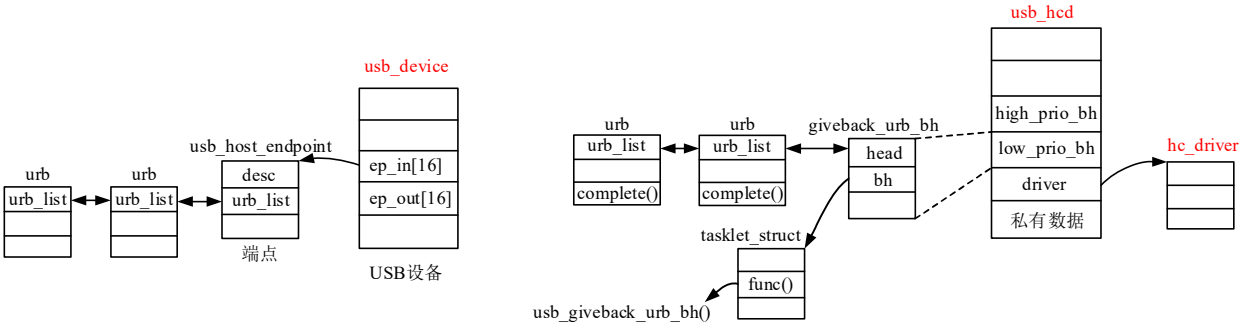
（1）中断传输：提交到端点 0 的 urb 双链表，激活 usb_hcd 轮询定时器 rh_timer，以探测根 HUB 状

态变化。

（2）控制传输：提交到端点 0 的 urb 双链表，并立即调用 hcd->driver->hub_control()函数执行控制传输，并处理传输结果。

2、与非根 HUB 的数据传输

调用 hcd->driver->urb_enqueue()函数将 urb 添加到端点的 urb 双链表。



urb 添加到端点 urb 实例双链表后，由主机控制器驱动程序负责调度，完成数据的传输。

完成传输的 urb 实例将从端点 urb 双链表中移出，添加到 usb_hcd 实例中的 giveback_urb_bh 结构体中的双链表，随后将通过 tasklet 机制，调用 urb 实例中的 complete()函数，执行传输完成后的工作。

在添加 usb_hcd 实例的 usb_add_hcd()函数中，将初始化其 high_prio_bh 和 low_prio_bh 成员，两个成员中内嵌 tasklet 实例的执行函数为 usb_giveback_urb_bh()，此函数扫描 giveback_urb_bh 实例中 urb 双链表，释放 urb 实例，调用其 complete()函数等。

5 同步传输

下面几个函数用于发送同步型数据，函数内将创建 urb 实例，并将其发送出去，等待传输完成，函数返回。这些函数可能用在中断上下文中（/drivers/usb/core/message.c）。

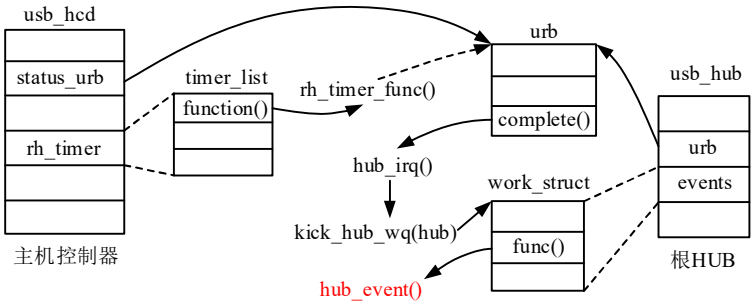
- int **usb_control_msg**(struct usb_device *dev, unsigned int pipe, \
  __u8 request, __u8 requesttype, __u16 value, __u16 index,void *data, __u16 size, int timeout);
- int **usb_interrupt_msg**(struct usb_device *usb_dev, unsigned int pipe,void *data, int len, \
  int *actual_length, int timeout);
- int **usb_bulk_msg**(struct usb_device *usb_dev, unsigned int pipe, \
  void *data, int len, int *actual_length,int timeout);

8.9.7 探测新设备

至此，似乎 USB 总线驱动和数据传输都介绍完了，但是还有一个重要的问题还没弄明白。那就是当向 USB 端口插入新设备时，主机控制器是如何检测到的？

外部设备插入 HUB 端口时，HUB 会检测到电压变化，感知到有设备插入，然后通过端点 0 的中断传输将端口的变化传输给主机控制器。这个传输不是 HUB 主动发起的，而是在主机控制器轮询时发起的。

下面用图示的方法简要说明探测新设备的过程，如下图所示：



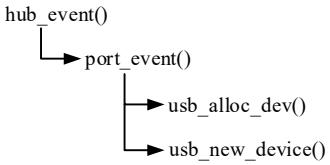
在创建根 HUB 的 usb_hub 实例时，创建了用于中断传输的 urb 实例，其 complete()函数为 **hub_irq()**，工作成员 events 的执行函数设为 hub_event()，这是探测设备的函数。

在创建主机控制器 usb_hcd 时，其轮询定时器 rh_timer 执行函数设为 **rh_timer_func()**，在添加 usb_hcd 实例时，会激活此定时器。

轮询定时器到期执行函数 rh_timer_func()将检测根 HUB 的状态变化，如果有状态变化则将根 HUB 的中断传输 urb 实例关联到 usb_hcd->status_urb 成员，并添加到 usb_hcd 实例的 giveback_urb_bh 结构体成员的双链表中，即将其视为传输完成的 urb，随后将调用 urb 的 complete()函数 **hub_irq()**。

hub_irq()函数将调用 kick_hub_wq()函数，kick_hub_wq()函数将激活 usb_hub 实例中 events 工作，其执行函数为 hub_event()。

hub_event()函数检测 HUB 端口的变化，为端口连接的设备创建并注册 usb_device 实例，函数调用关系简列如下图所示：



普通设备 usb_device 实例的创建和注册与前面介绍的根 HUB 设备的处理相似，usb_device 实例将匹配通用的 USB 设备驱动 usb_generic_driver，在其 probe()函数中读取设备配置和接口信息，创建并注册表示接口的 usb_interface 实例，usb_interface 实例将匹配用户注册的接口驱动程序。用户实际要编写的是接口驱动程序。

由于作者水平有限，只是简单介绍了一下探测设备的流程，具体函数源代码请读者自行阅读。

8.9.8 接口驱动示例

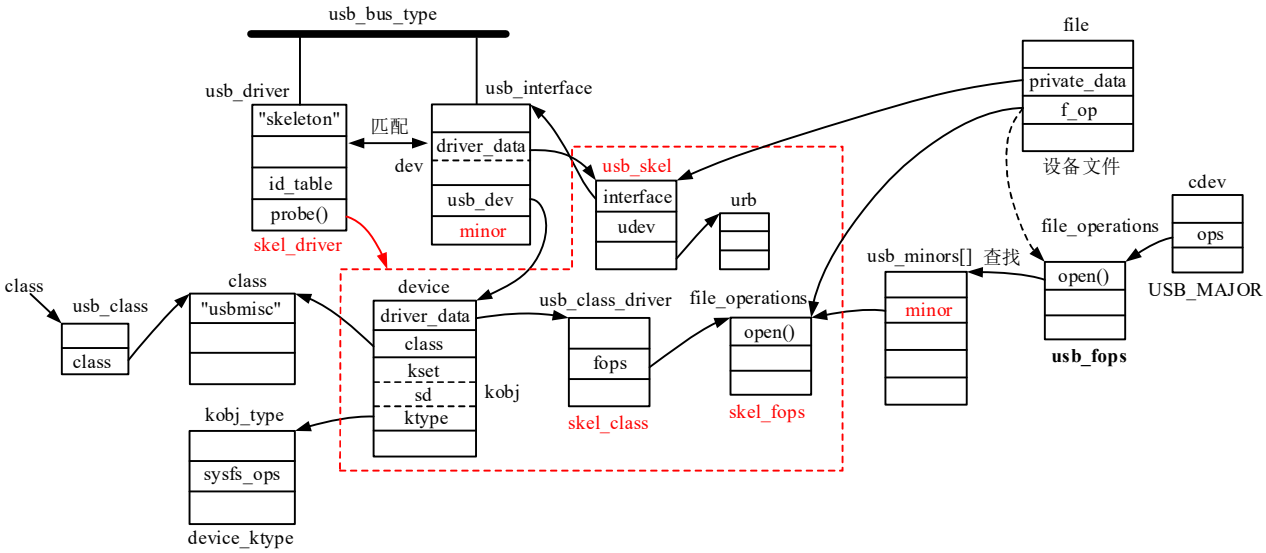
内核在/drivers/usb/usb-skeleton.c 文件内实现了一个 USB 接口驱动的示例程序，用户可依此模板编写 USB 接口驱动程序。下面对此示例程序做简要介绍。

1 驱动框架

在/drivers/usb/usb-skeleton.c 文件内实现的 USB 接口驱动框架如下图所示。

内核提供了一个将连接到 USB 总线上的设备视为字符设备的驱动程序框架，即所有 USB 设备共用一个主设备号 USB_MAJOR（180），不同的设备分配不同的从设备号。

在/drivers/usb/core/file.c 文件内定义的初始化函数 usb_major_init()（由 usb_init()函数调用），将创建并注册了对应的 cdev 实例，关联的 file_operations 结构体实例为 **usb_fops**。usb_fops 实例中只定义了 open()函数，函数内依据从设备号查找 file_operations 结构体指针数组 usb_minors[]，将其指向的 file_operations 实例赋予 file 实例。接口驱动程序在定义 file_operations 实例后，需要依从设备号关联到对应的 usb_minors[]数组项，从设备号是动态分配的。



接口驱动程序需定义 usb_class_driver 结构体实例，以此来注册定义的 file_operations 实例，结构体定义如下（/include/linux/usb.h）：

```
struct usb_class_driver {
    char *name;
    char *(*devnode)(struct device *dev, umode_t *mode);        /*设备文件名称*/
    const struct file_operations *fops;              /*文件操作结构*/
    int minor_base;          /*动态分配从设备号的起点*/
};
```

接口函数 int **usb_register_dev**(struct usb_interface *intf,struct usb_class_driver *class_driver)定义在文件/drivers/usb/core/file.c，函数内为 USB 设备创建设备类 class 实例，为接口分配从设备号，将 class_driver->fops 指向实例关联到 usb_minors[]数组中从设备号对应项，为接口创建并添加 device 实例，关联到其 usb_dev 成员，源代码请读者自行阅读。

在添加 device 实例时，将为接口创建设备文件，文件名称来源的顺序依次为：

（1）usb_class_driver->devnode()。

（2）dev->init_name 或 dev->kobject 名称。

在 usb_register_dev()函数中将设备 dev->kobject 名称为“class_driver->name”后接从设备号相对于 minor_base 的偏移量，如果没有定义 usb_class_driver->devnode()函数，这将做为接口的设备文件名称。

在 usb-skeleton.c 文件内接口驱动的 probe()函数中，将为接口创建了 usb_skel 结构体实例，用来搜集接口的信息，其中包含一个用于批量传输的 urb 实例；调用 usb_register_dev()接口函数创建并添加 device 实例，注册定义的 file_operations 实例 skel_fops 等。

usb_class_driver 结构体实例 skel_class 定义如下：

```
static struct usb_class_driver skel_class = {
    .name =         "skel%d",          /*设备文件名称*/
    .fops =         &skel_fops,
    .minor_base = USB_SKEL_MINOR_BASE,
};
```

文件操作结构 skel_fops 实例中的函数通过此批量传输 urb 实现数据的传输。

2 探测函数

接口驱动定义如下：

```
static struct usb_driver skel_driver = {
        .name =    "skeleton",
        .probe =   skel_probe,
        .disconnect =   skel_disconnect,
        .suspend =      skel_suspend,
        .resume =skel_resume,
        .pre_reset =    skel_pre_reset,
        .post_reset =   skel_post_reset,
        .id_table =     skel_table,          /*匹配厂商和产品 ID*/
        .supports_autosuspend = 1,
};
module_usb_driver(skel_driver);
```

probe()探测函数 skel_probe()代码简列如下：

```
static int skel_probe(struct usb_interface *interface,const struct usb_device_id *id)
{
        struct usb_skel *dev;
        struct usb_host_interface *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        size_t buffer_size;
        int i;
        int retval = -ENOMEM;

        dev = kzalloc(sizeof(*dev), GFP_KERNEL);        /*分配 usb_skel 实例*/
        ...
        kref_init(&dev->kref);
        sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
        mutex_init(&dev->io_mutex);
        spin_lock_init(&dev->err_lock);
        init_usb_anchor(&dev->submitted);
        init_waitqueue_head(&dev->bulk_in_wait);

        dev->udev = usb_get_dev(interface_to_usbdev(interface));        /*usb_device 实例*/
        dev->interface = interface;        /*接口*/

        iface_desc = interface->cur_altsetting;
        for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {        /*查找第一个批理输入、输出端点*/
                endpoint = &iface_desc->endpoint[i].desc;          /*端点描述字*/
```

```
            if (!dev->bulk_in_endpointAddr &&usb_endpoint_is_bulk_in(endpoint)) {
                /* we found a bulk in endpoint */
                buffer_size = usb_endpoint_maxp(endpoint);
                dev->bulk_in_size = buffer_size;
                dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
                dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);      /*分配缓存区*/
                ...
                dev->bulk_in_urb = usb_alloc_urb(0, GFP_KERNEL);        /*分配批量传输 urb*/
                ...
            }

            if (!dev->bulk_out_endpointAddr &&usb_endpoint_is_bulk_out(endpoint)) {
                /* we found a bulk out endpoint */
                dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
            }
        }
        ...     /*错误处理*/
        usb_set_intfdata(interface, dev);          /*interface->dev.driver_data=usb_skel*/
        retval = usb_register_dev(interface, &skel_class);          /*前面介绍过的接口函数*/
        ...
        return 0;
        ...
    }
```

3 文件操作函数

usb_class_driver 结构体实例 skel_class 中关联的 file_operations 实例 **skel_fops** 定义如下：

```
static const struct file_operations skel_fops = {          /*操作设备文件*/
    .owner =  THIS_MODULE,
    .read =    skel_read,
    .write =   skel_write,
    .open =    skel_open,
    .release = skel_release,
    .flush =   skel_flush,
    .llseek =  noop_llseek,
};
```

skel_fops 实例中的函数是用户操作接口设备文件的接口，下面简单介绍一下其中的 open()函数和 read()函数，其它函数请读者自行阅读源代码。

■打开函数

打开函数 skel_open()代码简列如下：

```
static int skel_open(struct inode *inode, struct file *file)
{
```

```
    struct usb_skel *dev;
    struct usb_interface *interface;
    int subminor;
    int retval = 0;

    subminor = iminor(inode);          /*从设备号*/
    interface = usb_find_interface(&skel_driver, subminor);          /*接口*/
    ...
    dev = usb_get_intfdata(interface);        /*usb_skel 实例*/
    ...
    retval = usb_autopm_get_interface(interface);          /*/drivers/usb/core/driver.c*/
    ...
    /* increment our usage count for the device */
    kref_get(&dev->kref);
    /* save our object in the file's private structure */
    file->private_data = dev;              /*file 私有数据指向 usb_skel 实例*/
exit:
    return retval;
}
```

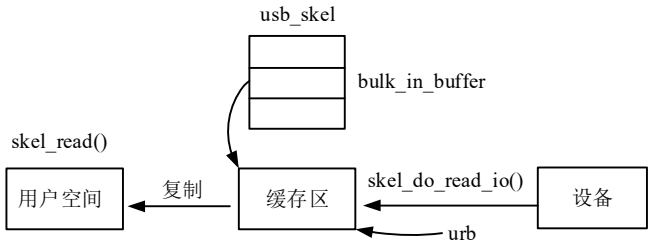
■读函数

在看 skel_read()函数前先看一下 usb_skel 结构体的定义：

```
struct usb_skel {
    struct usb_device    *udev;              /*指向 USB 设备 usb_device 实例*/
    struct usb_interface *interface;         /*接口*/
    struct semaphore     limit_sem;          /* limiting the number of writes in progress */
    struct usb_anchor    submitted;          /* in case we need to retract our submissions */
    struct urb       *bulk_in_urb;           /*指向读操作 urb，写操作另外创建 urb*/
    unsigned char        *bulk_in_buffer;    /*接收（读）数据缓存区（urb 中缓存区）*/
    size_t               bulk_in_size;       /*缓存区长度*/
    size_t               bulk_in_filled;     /*缓存区中已填充长度*/
    size_t               bulk_in_copied;          /*已复制到用户空间数据长度*/
    __u8                 bulk_in_endpointAddr;    /*批量输入端点编号*/
    __u8                 bulk_out_endpointAddr;   /*批量输出端点编号*/
    int              errors;             /* the last request tanked */
    bool             ongoing_read;       /*正在读*/
    spinlock_t           err_lock;       /* lock for errors */
    struct kref          kref;
    struct mutex         io_mutex;           /* synchronize I/O with disconnect */
    wait_queue_head_t bulk_in_wait;          /*读等待队列*/
};
```

读函数流程如下图所示，usb_skel 实例中 bulk_in_buffer 成员关联到读 urb 的缓存区，若缓存区有数据且长度够，则直接将缓存区数据复制到用户空间。若缓存区数据不够（或没有数据），则调用 skel_do_read_io()

函数从设备读数据，操作函数睡眠等待。skel_do_read_io()函数读到数据后，urb 实例回调函数将唤醒读操作睡眠的进程，唤醒后从缓存区复制数据到用户空间。如果有其它进程在进行读操作，当前进程也将睡眠等待。



读函数 skel_read()代码简列如下：

```
static ssize_t skel_read(struct file *file, char *buffer, size_t count,loff_t *ppos)
{
    struct usb_skel *dev;
    int rv;
    bool ongoing_io;

    dev = file->private_data;          /*usb_skel 实例*/

    /*不能读*/
    if (!dev->bulk_in_urb || !count)
        return 0;

    /* no concurrent readers */
    rv = mutex_lock_interruptible(&dev->io_mutex);
    if (rv < 0)
        return rv;

    if (!dev->interface) {            /* disconnect() was called */
        rv = -ENODEV;
        goto exit;
    }

    /* if IO is under way, we must not touch things */
retry:
    spin_lock_irq(&dev->err_lock);
    ongoing_io = dev->ongoing_read;
    spin_unlock_irq(&dev->err_lock);

    if (ongoing_io) {          /*正在读，需要睡眠等待*/
        /* nonblocking IO shall not wait */
        if (file->f_flags & O_NONBLOCK) {
            rv = -EAGAIN;
            goto exit;
        }
```

```
        /*
         * IO may take forever
         * hence wait in an interruptible state
         */
        rv = wait_event_interruptible(dev->bulk_in_wait, (!dev->ongoing_read));
        if (rv < 0)
            goto exit;
    }

    /* errors must be reported */
    ...     /*报告错误*/

    if (dev->bulk_in_filled) {          /*缓存区有数据，直接复制*/
        /* we had read data */
        size_t available = dev->bulk_in_filled - dev->bulk_in_copied;
        size_t chunk = min(available, count);

        if (!available) {
            rv = skel_do_read_io(dev, count);       /*读数据到缓存区*/
            if (rv < 0)
                goto exit;
            else
                goto retry;
        }
        /*缓存区有数据，复制到用户空间*/
        if (copy_to_user(buffer,dev->bulk_in_buffer + dev->bulk_in_copied,chunk))
            rv = -EFAULT;
        else
            rv = chunk;

        dev->bulk_in_copied += chunk;

        /*数据不够，再读*/
        if (available < count)
            skel_do_read_io(dev, count - chunk);
    } else {        /*缓存区没有数据，读数据*/
        rv = skel_do_read_io(dev, count);
        if (rv < 0)
            goto exit;
        else
            goto retry;
    }
exit:
    mutex_unlock(&dev->io_mutex);
```

```
    return rv;
}

skel_do_read_io()函数用于通过 USB 总线从设备读数据，函数定义如下：
static int skel_do_read_io(struct usb_skel *dev, size_t count)
{
    int rv;

    /*填充 urb*/
    usb_fill_bulk_urb(dev->bulk_in_urb,
            dev->udev,
            usb_rcvbulkpipe(dev->udev,dev->bulk_in_endpointAddr),
            dev->bulk_in_buffer,
            min(dev->bulk_in_size, count),
            skel_read_bulk_callback,        /*回调函数，唤醒睡眠等待读进程等*/
            dev);
    /* tell everybody to leave the URB alone */
    spin_lock_irq(&dev->err_lock);
    dev->ongoing_read = 1;
    spin_unlock_irq(&dev->err_lock);

    /* submit bulk in urb, which means no data to deliver */
    dev->bulk_in_filled = 0;
    dev->bulk_in_copied = 0;

    /*提交 urb，执行数据传输*/
    rv = usb_submit_urb(dev->bulk_in_urb, GFP_KERNEL);
    ...
    return rv;
}
skel_fops 实例中其它操作函数请读者自行阅读源代码。
```

至此，USB 总线及设备驱动都介绍完了，对于通用（标准）的 USB 设备，内核都已经实现了其驱动，用户只需要在配置内核时选择即可。移植内核时主要是要移植 USB 主机控制器驱动。

8.10 模块

模块是内核编译源文件产生的一种目标文件，用于向内核动态添加驱动程序、文件系统类型等组件，内核在运行过程中加载模块后，无需重启即可使用模块中的代码，模块也可以被动态卸载。模块加载到内核地址空间的 VMLLOC 区域，模块的动态加/卸载特性使内核具有了扩展性，内核映象可保持一个较小的尺寸，在运行时根据需要加卸载相关的功能模块。

在编译内核的过程中可以配置选择驱动程序或文件系统等代码是永久编译入内核还是以模块的形式编译，编译成模块时，产生的是.ko 形式的可重定位目标文件。用户进程通过系统调用加载模块时，内核将模块代码加载到内核 VMALLOC 区，并解决引用问题。所谓引用就是模块中引用内核或其它模块内的

函数或全局变量，模块在编译时不知道它们的地址，在加载模块时内核需要将地址告知模块。

模块的加卸载由用户进程通过系统调用完成，内核自身触发模块加卸载时，将信息传递给用户态守护进程，由用户进程完成加卸载工作。

本节首先简要介绍内核镜像、模块等文件格式，然后介绍模块的编译，最后介绍内核对模块的管理加卸载操作。

8.10.1 目标文件格式

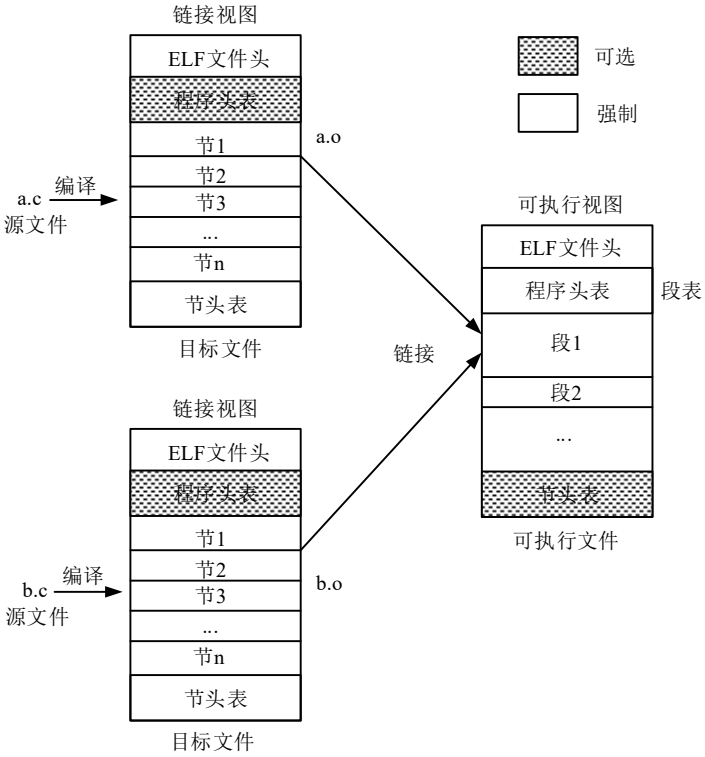
ELF(Executable and Linkable Format)是一种可执行文件、目标文件和库文件使用的标准文件格式，在 Linux 下成为标准格式已经很长时间了。内核映像文件、可重定位文件、模块文件都使用的是 ELF 格式。

1 ELF 文件格式

内核源代码本质上是一个大的程序，各源代码文件经过选择，编译生成目标文件，所有的目标文件链接成单一的可执行文件。

模块文件只编译成目标文件，不参与链接，在运行时动态加载到内核地址空间，解决与内核代码之间的引用关系。

目标文件（包括模块）和内核可执行文件都采用 ELF 文件格式，ELF 文件各个部分组成如下图所示（第 5 章有更详细的介绍）：



源文件编译生成的目标文件结构通常是链接视图，文件内容包括 ELF 文件头、节、节头表等。

ELF 文件头包含文件类型和大小的有关信息、节头表位置和长度等信息。节（section）就是源代码中定义的.text 节（代码）、.rodata 节（只读数据）中的内容，即指令和数据。节头表中每一个表项描述了一个节的信息。

各目标文件通过链接，拼接在一起，生成可执行文件。各目标文件中相同类型的节在可执行文件中放在一起，如所有目标文件中的.text 放在一起，形成段。段可以理解成是对相同属性节的归类和整理。

在可执行视图中，ELF 文件头中包含了段表（程序头表）的信息，段表中每一个表项描述了一个段的

信息。段的类型有 INTERP、LOAD、DYNAMIC 等，其中 LOAD 是表示必须映射到虚拟地址空间的段，其它段保存的是其它信息，不需要映射到虚拟地址空间。LOAD 段中包含.text 和.rodata 等节，它们都是要映射到虚拟地址空间中的节。

内核源代码文件选择编译成模块时，只编译生成可重定位目标文件，不参与内核可执行文件的链接。可重定位文件中没有使用绝对地址，使用的都是相对地址，因此目标文件可加载到任意地址运行。但是，模块会引用内核和其它模块定义的函数和变量，因此模块加载后需要解决引用问题。

2 内核导出符号

ELF 文件内的符号表节，保存了本程序定义和引用的所有全局变量和函数地址。如果程序引用了一个自身代码没有定义的符号，则称之为未定义符号。

用户空间工具 nm 可生成目标文件中定义和使用的所有符号列表，如：

```
00000000    T    add
            U    exit
0000001a    T    main
            U    printf
```

左侧第一列表示符号定义在文件中的位置（地址），对程序而言符号就是地址值。T 表示符号定义在.text 节，U 表示未定义符号，即需要从外部获取符号值（地址值）的符号。

ELF 文件内通过 3 个节来实现符号表机制，它们是.symtab、.strtab 和.hash。.strtab 节保存了符号字符串，可视为符号字符串数组，数组项中以 0 字节分隔。.symtab 节保存符号表数据结构实例，每个符号由固定大小的数据结构实例表示，结构中包含符号值（地址值）、类型等信息，而符号名称字符串由.strtab 节中字符串索引值表示，也就是说字符串保存在.strtab 节，.symtab 节中只保存字符串在.strtab 节内的索引值。.hash 节保存散列表，用于帮助索引值到符号字符串的快速查找。

在内核代码中，能被其它模块引用的符号（变量和函数）需要将其导出。在加载模块解决引用时需要扫描模块目标文件中的符号表，对未定义的符号，扫描内核和其依赖模块的导出符号列表，获取符号地址解决引用。 内核（模块）为了导出符号，在可执行文件（目标文件中）增加了额外的节，用于收集导出符号信息。

内核提供了声明导出符号的宏：EXPORT_SYMBOL 和 EXPORT_SYMBOL_GPL 等。二者分别用于导出符号和导出只用于 GPL 兼容代码的符号，它们的目的是将相应的符号信息放置到目标文件指定节当中。

以上两个宏定义在/include/linux/export.h 头文件内：

```
#define EXPORT_SYMBOL(sym)                  \       /*sym 表示变量名或函数名*/
    __EXPORT_SYMBOL(sym, "")

#define EXPORT_SYMBOL_GPL(sym)              \
    __EXPORT_SYMBOL(sym, "_gpl")

#define EXPORT_SYMBOL_GPL_FUTURE(sym)       \
    __EXPORT_SYMBOL(sym, "_gpl_future")

#define __EXPORT_SYMBOL(sym, sec)           \
    extern typeof(sym) sym;                 \
    __CRC_SYMBOL(sym, sec)                  \           /*符号校验和，保存在指定节*/
    static const char __kstrtab_##sym[]     \           /*定义字符串，保存在指定节中*/
    __attribute__((section("__ksymtab_strings"), aligned(1))) \
```

```
= MODULE_SYMBOL_PREFIX #sym;                    \
static const struct kernel_symbol __ksymtab_##sym \   /*定义并初始化 kernel_symbol 实例*/
__used                                          \
__attribute__((section("___ksymtab" sec "+" #sym), unused)) \   /*保存在指定节中*/
= { (unsigned long)&sym, __kstrtab_##sym }              /*‘##’表示字符串拼接*/
```

__EXPORT_SYMBOL(sym, sec)宏的主要功能有（还导出检验和保存在指定节）：

（1）定义名称为_kstrtab_*sym* 的字符数组，其内容为 MODULE_SYMBOL_PREFIX #sym，并保存在指定__ksymtab_strings 节中。

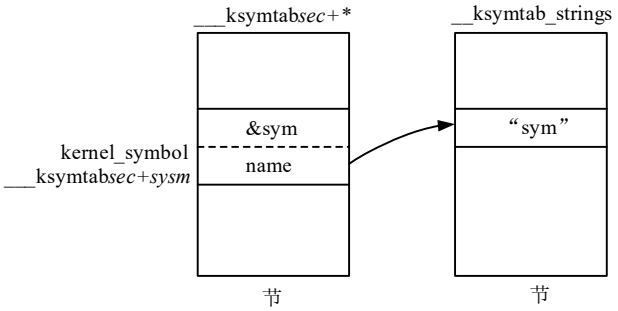
（2）定义 kernel_symbol 结构体实例保存在___ksymtab*sec+sym* 节中。

简单地说就是定义一个表示名称的字符数组位于保存到指定节，定义一个 kernel_symbol 结构体实例保存到指定节。

kernel_symbol 结构体定义在/include/linux/export.h 头文件：

```
struct kernel_symbol
{
    unsigned long value;        /*符号值，地址值*/
    const char *name;           /*符号名称字符串指针，指向_kstrtab_sym 字符数组*/
};
```

kernel_symbol 结构体包含两个成员，分别表示符号的值（地址值）以及符号名称字符串指针。在宏定义中 value 成员赋值为符号地址值，name 成员赋值为定义字符数组地址。下图示意了宏定义创建的数据结构实例关系：



在内核和模块代码都会使用以上两个宏，因此在内核可执行文件和模块目标文件内会生成以下节：

- ___ksymtab+*：保存 EXPORT_SYMBOL(sym)宏中创建的 kernel_symbol 实例。
- ___ksymtabgpl+*：保存 EXPORT_SYMBOL_GPL(sym)宏中创建的 kernel_symbol 实例。
- ___ksymtabgpl_future+*：保存 EXPORT_SYMBOL_GPL_FUTURE(sym)宏中创建的 kernel_symbol 实例。
- __ksymtab_strings：保存所有导出符号名称字符串。

内核在/include/asm-generic/vmlinux.lds.h 头文件中定义了标识只读数据段的宏：

```
#define RO_DATA_SECTION(align)                  \
    . = ALIGN((align));                         \
    ...                                         \
    __ksymtab               : AT(ADDR(__ksymtab) - LOAD_OFFSET) {       \
        VMLINUX_SYMBOL(__start___ksymtab) = .;  \           /*节起始地址*/
        *(SORT(___ksymtab+*))                   \           /*___ksymtab+*节*/
```

```
        VMLINUX_SYMBOL(__stop___ksymtab) = .;   \          /*节结束地址，下同*/
    }                                           \
                                                \
    __ksymtab_gpl       : AT(ADDR(__ksymtab_gpl) - LOAD_OFFSET) {   \
        VMLINUX_SYMBOL(__start___ksymtab_gpl) = .;          \
        *(SORT(___ksymtab_gpl+*))                   \       /*___ksymtab_gpl+*节*/
        VMLINUX_SYMBOL(__stop___ksymtab_gpl) = .;           \
    }                                           \
    ...
    __ksymtab_gpl_future : AT(ADDR(__ksymtab_gpl_future) - LOAD_OFFSET) { \
        VMLINUX_SYMBOL(__start___ksymtab_gpl_future) = .;\
        *(SORT(___ksymtab_gpl_future+*))            \       /*___ksymtab_gpl_future+*节*/
        VMLINUX_SYMBOL(__stop___ksymtab_gpl_future) = .;\
    }                                           \
    ...                                         \           /*保存校验和的节*/
                                                \
    /*__ksymtab_strings 节*/                    \
        __ksymtab_strings : AT(ADDR(__ksymtab_strings) - LOAD_OFFSET) { \
        *(__ksymtab_strings)                    \
    }                                           \
    ...                     \
    . = ALIGN((align));

#define   RODATA        RO_DATA_SECTION(4096)       /*嵌入链接脚本*/
```

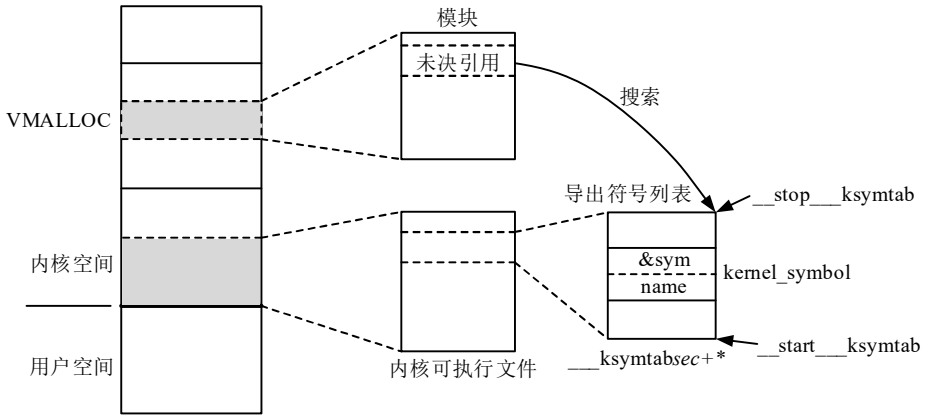
Linux 系统在/proc/kallsyms 文件中列出了导出符号的地址和名称（只读）。

模块链接文件为/scripts/module-common.lds，如下所示：

```
SECTIONS {
    /DISCARD/ : { *(.discard) }

    __ksymtab           0 : { *(SORT(___ksymtab+*)) }
    __ksymtab_gpl           0 : { *(SORT(___ksymtab_gpl+*)) }
    __ksymtab_unused 0 : { *(SORT(___ksymtab_unused+*)) }
    __ksymtab_unused_gpl  0 : { *(SORT(___ksymtab_unused_gpl+*)) }
    __ksymtab_gpl_future   0 : { *(SORT(___ksymtab_gpl_future+*)) }
    __kcrctab       0 : { *(SORT(___kcrctab+*)) }
    __kcrctab_gpl       0 : { *(SORT(___kcrctab_gpl+*)) }
    __kcrctab_unused  0 : { *(SORT(___kcrctab_unused+*)) }
    __kcrctab_unused_gpl   0 : { *(SORT(___kcrctab_unused_gpl+*)) }
    __kcrctab_gpl_future    0 : { *(SORT(___kcrctab_gpl_future+*)) }

    . = ALIGN(8);
    .init_array         0 : { *(SORT(.init_array.*)) *(.init_array) }
```

}

下图简单示意了模块加载后，解决符号引用的过程。模块被加载到内核地址空间 VMALLOC 区，扫描其符号表中的未决符号，搜索内核导出符号列表，获取符号地址，解决符号引用。如果模块引用了其它模块中的导出符号，还将搜索其它模块的导出符号列表。



8.10.2 生成模块

模块由内核中源代码文件生成。在配置内核时，可选择将某个子系统、驱动程序等代码永久编译入内核（即直接链接到内核可执行文件）或生成为模块。模块是可重定位的目标文件，模块独立于内核可执行文件，在运行时加载到内存中后再与内核链接，这类似于用户空间的动态库。模块作为特殊的目标文件，增加了一些额外的节，表示模块的信息。

1 额外信息

模块目标文件内除了增加导出符号的节之外，还有其它的节。例如：表示模块参数的节、表示模块信息的节以及在内核中表示模块数据结构的节等。

模块的额外参数通过宏来定义。模块源代码可永久编译入内核，也可编译成模块。永久编译入内核时这些宏定义有不同的实现（更为简单），内核在将源文件编译成模块时，在编译选项中会添加 MODULE 宏的定义，也就是说在将源代码永久编译入内核时，不会定义 MODULE 宏，编译成模块时会定义 MODULE 宏，详见/Makefile 文件中对 KBUILD_AFLAGS_MODULE 和 KBUILD_CFLAGS_MODULE 的定义。

■模块参数

模块参数是模块内的全局变量，可以在加载模块时对其赋值，形式如“参数名称=值”，这与引导加载程序传递命令行参数十分相似。模块中声明模块参数的宏如下（/include/linux/moduleparam.h）：

```
#define module_param(name, type, perm)                \
    module_param_named(name, name, type, perm)        /*/include/linux/moduleparam.h*/
```

name：表示模数的名称，

type：表示变量类型，如 byte, short, ushort, int, uint, long, ulong,charp,bool 等。

perm：表示访问权限，与文件权限控制相同。

module_param_named()宏定义如下：

```
#define module_param_named(name, value, type, perm)       \
```

```
    param_check_##type(name, &(value));                \    /*/include/linux/moduleparam.h*/
    module_param_cb(name, &param_ops_##type, &value, perm);    \    /*定义 kernel_param 实例*/
    __MODULE_PARM_TYPE(name, #type)        /*在.modinfo 节中导出参数信息*/
```

module_param_named()宏的最终效果是创建并设置 kernel_param 结构体实例，并将实例链接到名称为"__param"的节，并将参数信息导出到.modinfo 节，这些节都是位于模块目标文件中的节。

kernel_param 实例关联的 param_ops_##type 实例为 kernel_param_ops 结构体实例，根据变量类型的不同，实例也不相同，实例中主要包含读写变量值的函数，详见第 2 章（/kernel/params.c）。

在加载模块时，系统调用会将传递的模块参数与"__param"节内的 kernel_param 实例比较，相同则调用关联 kernel_param_ops 实例中的设置函数将变量值写入模块中变量。

■模块信息

模块还有许多其它信息通常通过 MODULE_INFO(tag, info)宏定义（/include/linux/module.h）：

```
#define   MODULE_INFO(tag, info)   __MODULE_INFO(tag, tag, info)
```

__MODULE_INFO()宏定义在/include/linux/moduleparam.h 头文件：

```
#define __MODULE_INFO(tag, name, info)                \
  static const char __UNIQUE_ID(name)[]                \    /*定义字符数组*/
  __used __attribute__((section(".modinfo"), unused, aligned(1)))    \    /*保存在.modinfo 节内*/
  = __stringify(tag) "=" info                                /*字符数组内容*/
```

__MODULE_INFO()宏的功能就是定义字符数组，内容为“tag=info”，并将其保存到**.modinfo** 节内。

在/include/linux/module.h 头文件中定义了声明模块信息宏，例如：

MODULE_SOFTDEP(_softdep)：必须在本模块之前加载的模块。

MODULE_ALIAS(_alias)：模块别名（用户空间可用），“alias=*_alias*”。

MODULE_LICENSE(_license)：模块版权，“license=*_license*”。

MODULE_AUTHOR(_author)：模块作者，“author=*_author*”。

MODULE_DESCRIPTION(_description)：模块描述，“description=*_description*”。

■加/卸载函数

加/卸载函数是指在模块被加/卸载时调用的模块内部函数。

声明模块加载和卸载函数的宏分别为 **module_init(initfn)**和 **module_exit(exitfn)**。

这两个宏定义在/include/linux/module.h 头文件：

```
#define module_init(initfn)                    \
    static inline initcall_t __inittest(void)   \
    { return initfn; }                    \
    int init_module(void) __attribute__((alias(#initfn)));    /*initfn 函数取别名 init_module*/

#define module_exit(exitfn)                    \
    static inline exitcall_t __exittest(void)        \
```

```
{ return exitfn; }                              \
void cleanup_module(void) __attribute__((alias(#exitfn)));    /*exitfn 函数取别名 cleanup_module*/
```

initfn 为加载函数名称，exitfn 为卸载函数名称。module_init()和 module_exit()宏分别为这两个函数取了别名 init_module 和 cleanup_module，这两个别名在所有模块中都是一样的。

2 编译模块

生成模块需要执行以下 3 个步骤：

（1）将模块包含的所有源文件（.c）编译成普通的.o 目标文件。

（2）生成目标文件后，分析模块，找到附加信息（如模块依赖关系），保存到一个独立的附加文件。

（3）将普通目标文件与附加文件链接，生成模块（.ko）。

生成第（2）步附加文件的程序（脚本）位于/scripts/mod/目录下，/scripts/mod/modpost.c 文件内的 main()函数用于向附加文件写入内容。

在第（2）步生成的附加文件中，在.gun.linkonce.module 节保存了 module 结构体实例，实例中保存了模块信息，内核可直接使用，后面将详细介绍。

在.modinfo 节将写入"**depends=*****"模块信息，其值是本模块依赖的模块名称，在.modinfo 节中还将写入其它模块信息，这里的.modinfo 节将与第（1）步生成的目标文件中的.modinfo 节合并。

在附加文件中还将创建其它的节，用于保存模块信息，这里就不一一介绍了（作者能力有限！）。

在执行 make 命令构建内核时将构建内核模块，而后也可以使用 **make modules** 重新编译所有的模块。

模块也可以单独编译，但需要已构建好的内核源码树，单独编译模块的命令如下：

make -C $KDIR M=$PWD

$KDIR 表示内核源代码所在目录（绝对路径），$PWD 表示模块代码所在目录（绝对路径），执行此命令后，Kbuild 机制将读取模块代码所在目录下的 Makefile 文件，编译模块。

在 Makefile 文件中，obj-m 列表下的目标文件将编译成模块，模块目标文件后缀为.ko。如果模块由单个文件组成，则 Makefile 文件内容如下（drivers/isdn/i4l/Makefile）：

```
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o     /*配置选项值为 m*/
```

如果模块由多个文件组成，则 Makefile 文件格式如下（/drivers/isdn/i4l/Makefile）：

```
obj-$(CONFIG_ISDN_I4L) += isdn.o
isdn-y := isdn_net_lib.o isdn_v110.o isdn_common.o
```

isdn.o 表示模块名称，在构建内核时 Kbuild 将编译、链接 isdn-y 内包含的目标文件，以生成模块。

内核、模块更详细的配置、构建信息请参考第 14 章。

8.10.3 加载模块

内核中通过 module 结构体管理模块，每个加载的模块在内核中一个实例表示。用户进程可通过系统调用向内核添加模块，也可通过用户空间工具添加模块。内核添加模块的方式也是通过激活用户进程实现的。

1 管理模块

在内核中每个加载的模块由 module 结构体表示。在构建模块时，在自动生成的附加文件中包含此结构体实例并初始化，保存在.gnu.linkonce.this_module 节（/scripts/mod/modpost.c），内核可直接使用此实例。

module 结构体定义在/include/linux/module.h 头文件，其成员简列如下：

```
struct module {
    enum module_state state;     /*模块状态，格举类型*/
    struct list_head list;       /*链表成员，将实例链接到全局管理链表*/
    char name[MODULE_NAME_LEN];  /*模块名称，用于查找模块*/
    ...                          /*导出 sysfs 相关成员*/
    const struct kernel_symbol *syms;      /*模块导出符号列表*/
    const unsigned long *crcs;
    unsigned int num_syms;

    /*内核参数*/
#ifdef CONFIG_SYSFS
    struct mutex param_lock;
#endif
    struct kernel_param *kp;         /*模块参数列表*/
    unsigned int num_kp;

    /*GPL 兼容导出符号*/
    unsigned int num_gpl_syms;
    const struct kernel_symbol *gpl_syms;
    const unsigned long *gpl_crcs;
    ...          /*未使用符号*/
    bool async_probe_requested;

    /*将来只兼容 GPL 导出符号*/
    const struct kernel_symbol *gpl_future_syms;
    const unsigned long *gpl_future_crcs;
    unsigned int num_gpl_future_syms;
    /* Exception table */
    unsigned int num_exentries;
    struct exception_table_entry *extable;

    int (*init)(void);      /*加载函数指针，赋予 init_module()指针*/

    void *module_init  ____cacheline_aligned;
    void *module_core;
    unsigned int init_size, core_size;
    unsigned int init_text_size, core_text_size;
    ...
```

```
    char *args;          /*加载模块时的命令行参数指针*/
    ...
#ifdef CONFIG_MODULE_UNLOAD      /*模块可卸载*/
    struct list_head source_list;          /*依赖本模块的模块链表*/
    struct list_head target_list;          /*本模块依赖的模块链表*/
    void (*exit)(void);                    /*卸载函数指针，赋予 cleanup_module()指针*/
    atomic_t refcnt;
#endif
    ...
} ____cacheline_aligned;
```

module 结构体部分成员简介如下：

●**state**：模块状态，枚举类型 module_state 定义如下（/include/linux/module.h）：

```
enum module_state {
    MODULE_STATE_LIVE,        /*正常状态*/
    MODULE_STATE_COMING,      /*装载状态*/
    MODULE_STATE_GOING,       /* Going away. */
    MODULE_STATE_UNFORMED,    /* Still setting it up. */
};
```

●**list**：内核定义了全局链表 modules（/kernel/module.c），用于管理内核所有的 module 实例，list 为双链表成员。

●**name**：模块名称名符数组，必须是唯一的，内核由名称来标识模块，一般用模块二进制文件名去掉.ko 后缀用于该成员。

●**init**：模块加载时调用的初始化函数指针，赋值为 init_module()函数指针。

●**exit**：模块卸载时调用的函数指针，赋值为 cleanup_module()函数指针。

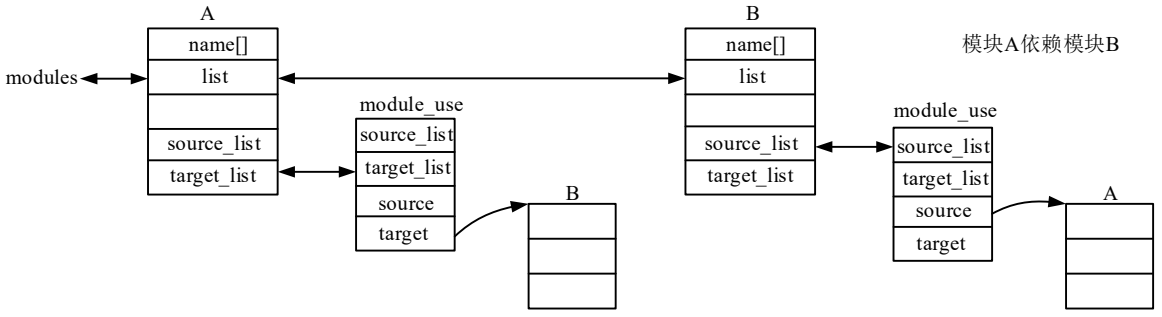
●**syms、gpl_syms、gpl_future_syms**：模块导出符号的 kernel_symbol 结构体列表。

●**source_list，target_list**：双链表成员。模块内部可能调用其它模块内定义的函数或变量，因此模块之间会形成依赖关系。例如：模块 A 内调用了模块 B 内的函数，则 A 依赖 B，在装载 A 之前需先装载 B，在卸载 B 之前必须保证依赖它的模块 A 已经卸装。

以上两个双链表成员链接的是 module_use 结构体实例（/include/linux/module.h）：

```
struct module_use {
    struct list_head source_list;      /*依赖本模块的模块链表*/
    struct list_head target_list;      /*本模块依赖的模块链表*/
    struct module *source, *target;    /*模块指针*/
};
```

source_list 成员用于链接依赖本模块的模块，如以上例子中模块 A 位于 B 模块此链表内，target_list 表示本模块依赖的模块链表，以上例子中模块 B 位于模块 A 此链表内，如下图所示。



加载的模块信息还将通过 sysfs 文件系统导到/proc/modules 文件内。

2 系统调用

模块通常安装到系统/lib/modules/$(KERNELRELEASE)/目录下， $(KERNELRELEASE)表示内核版本号，如 4.2.4（目录名称）。

内核和用户都可以发起模块的加载（需要特权），最终加载操作都通过 init_module()/finit_module()系统调用实现。这里先简要介绍 init_module()系统调用的实现，后面再介绍内核和用户如何发起模块的加载。

在调用 init_module()系统调用前，用户进程需打开模块（.ko 文件），并将模块内容起始地址及长度传递给系统调用。

init_module()系统调用实现代码位于/kernel/module.c 文件内，代码简列如下：

```
SYSCALL_DEFINE3(init_module, void __user *, umod,unsigned long, len, const char __user *, uargs)
/*umod：指向用户空间映射模块文件内容的基地址（指向文件内容开头）
*len：模块长度，uargs：参数。
*/
{
    int err;
    struct load_info info = { };          /*暂存加载信息，/kernel/module.c*/

    err = may_init_module();       /*需 CAP_SYS_MODULE 能力，且内核允许加载模块*/
    ...

    err = copy_module_from_user(umod, len, &info);         /*复制模块文件内容至内核 VMALLOC 区*/
    ...
    return load_module(&info, uargs, 0);     /*解决引用，调用模块初始化（加载）函数等*/
}
```

init_module()系统调用内将模块文件内容复制到内核 VMALLOC 区，然后调用 load_module()函数解决引用，调用模块初始化函数等。

load_module()函数代码简列如下（/kernel/module.c）：

```
static int load_module(struct load_info *info, const char __user *uargs,int flags)
{
    struct module *mod;
    long err;
    char *after_dashes;

    ...       /*在效性检查*/

    /* Figure out module layout, and allocate all the memory. */
    mod = layout_and_allocate(info, flags);         /*获取模块文件更详细信息，如各节信息等*/
    ...
    /* Reserve our place in the list. */
    err = add_unformed_module(mod);   /*mod 指向 module 实例，添加到内核管理结构等*/
    ...
```

```
/* To avoid stressing percpu allocator, do this once we're unique. */
err = percpu_modalloc(mod, info);       /*SMP，分配保留内存*/
...

/* Now module is in final location, initialize linked lists, etc. */
err = module_unload_init(mod);          /*初始化依赖链表*/
...
init_param_lock(mod);

err = find_module_sections(mod, info);     /*获取各导出符号节等信息，写入 module 实例*/
...
err = check_module_license_and_versions(mod);
...

/* Set up MODINFO_ATTR fields */
setup_modinfo(mod, info);          /*处理模块属性*/

/* Fix up syms, so that st_value is a pointer to location. */
err = simplify_symbols(mod, info);        /*解决未定义符号引用等*/
...

err = apply_relocations(mod, info);        /*模块内符号重定位*/
...

err = post_relocation(mod, info);     /*处理内核导出符号，体系结构相关工作等*/
...
flush_module_icache(mod);

/*复制命令行参数*/
mod->args = strndup_user(uargs, ~0UL >> 1);
...
dynamic_debug_setup(info->debug, info->num_debug);

/* Ftrace init must be called in the MODULE_STATE_UNFORMED state */
ftrace_module_init(mod);

/* Finally it's fully formed, ready to start executing. */
err = complete_formation(mod, info);          /*收尾工作*/
...

/* Module is ready to execute: parsing args may do that. */
after_dashes = parse_args(mod->name, mod->args, mod->kp, mod->num_kp,
                          -32768, 32767, NULL,unknown_module_param_cb);       /*处理命令行参数*/
...
```

```
    /*导出信息至 sysfs*/
    err = mod_sysfs_setup(mod, info, mod->kp, mod->num_kp);
    ...
    trace_module_load(mod);
    return do_init_module(mod);        /*调用模块初始化函数 mod->init()等*/
    ...
}
```

加载模块操作简单地说就是将模块文件内容复制到内核 VMALLOC 区，设置表示模块的 module 实例，添加到内核管理结构，解决未定义符号引用和模块自身的重定位，最后调用模块初始化函数。

在解决未定义符号引用时，对于引用的内核导出符号，搜索内核导出符号列表解决引用，对于引用其它模块导出的符号，则搜索内核 modules 链表中的 module 实例，实例中包括模块导出符号，从而解决引用。

init_module()系统调用需要用户进程先打开模块文件，映射到用户空间，然后将映射文件内容基址作为参数，调用 init_module()系统调用。

finit_module()系统调用也可以用来加载模块，且只需要先打开模块文件获取文件描述符即可，将文件描述符传递给系统调用，而不需要将文件内容映射到用户空间，它直接复制文件内容到内核 VMALLOC 区，然后调用 load_module()函数完成加载操作，实现函数请读者自行阅读。

移除模块的系统调用为 delete_module()，请读者自行阅读实现代码。

3 加载操作

用户和内核都可以发起模块的加载。

从用户角度可通过系统工具 insmod 和 modprobe 命令加载模块，insmod 用于加载单一的模块，不考虑依赖关系，系统调用内先将模块文件内容映射到用户空间，或打开模块文件，然后调用 init_module()或 finit_module()系统调用完成模块加载。

modprobe 会分析模块之间的依赖关系，在识别依赖关系后调用 init_module()系统调用依次加载模块。

内核态模块的加载由 **request_module(mod...)**函数（/include/linux/kmod.h）发起，参数 mod 表示模块名称，函数内调用/kernel/kmod.c 内的__request_module()函数，函数内启动用户工具"/sbin/modprobe"完成模块的加载。

在向内核注册 device 实例时，将向用户空间发送 uevent 事件信息，用户工具“/sbin/udevd”（守护进程）接收 uevent 事件信息，然后加载以模块编译的设备驱动程序（代替"/sbin/modprobe"）。

8.11 小结

本章介绍了内核用来管理设备和设备驱动程序的通用驱动模型。通用驱动模型中主要包括总线、设备类、设备、驱动等概念。

总线用于管理挂接在其上的设备和驱动，设备类用来管理同一类型的设备。设备和驱动挂接到某一总线上，在向总线注册设备或驱动时，会查找总线上匹配的驱动或设备，调用总线定义的匹配函数，若匹配成功将调用总线或驱动中的探测函数，在探测函数中将加载设备的驱动程序。

本章还介绍了 platform、SPI、I2C、USB 等具体总线（主机控制器）及设备驱动的实现，以及设备树和模块等相关内容。