Bus registration (platform, PCI)

原创 The little black house is closed Posted at 2020-07-06 16:30:05

Category column: Linux kernel

Linux kernel The column includes this content

There may be different types of buses in the kernel, and these different memories need to be stored in the kernel for management. Here, the platform bus and the pci bus are taken as examples for illustration.

The platform bus is a virtual bus. When a device is directly connected to the processor, the virtual bus can be used to simulate the structure that the device is connected to the processor through the platform bus.

In the kernel, the registration of the platform bus is located in the drivers/base/platform.c file.

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

        early_platform_cleanup();
        //该函数的实现与CPU架构相关，当前分析的架构中其实现为空

        error = device_register(&platform_bus);
        //注册总线设备，关于总线设备的定义如下：
        //struct device platform_bus = {
        //      .init_name = "platform",
        //};

        if (error) {
                put_device(&platform_bus);
                return error;
        }
        error = bus_register(&platform_bus_type);
        if (error)
                device_unregister(&platform_bus);
        of_platform_register_reconfig_notifier();
        return error;
}
```

About device registration, as follows:

```
int device_register(struct device *dev)
{
        device_initialize(dev);
        return device_add(dev);
}
//可见，设备注册主要完成两个任务，分别是：初始化设备，添加设备

void device_initialize(struct device *dev)
{
        dev->kobj.kset = device_kset;   //设置该设备所属的集合
        kobject_init(&dev->kobj, &device_ktype);        //设置该设备所属类型
        //上边的操作与内核所提出的设备模型机制相关，利用该机制，内核可对所有的设备进行管理
        INIT_LIST_HEAD(&dev->dma_pools);
        mutex_init(&dev->mutex);
#ifdef CONFIG_PROVE_LOCKING
        mutex_init(&dev->lockdep_mutex);
#endif
        lockdep_set_novalidate_class(&dev->mutex);
        spin_lock_init(&dev->devres_lock);
        INIT_LIST_HEAD(&dev->devres_head);
        device_pm_init(dev);
        set_dev_node(dev, -1);   //设置该设备关联的NUMA节点，此时设置为-1
#ifdef CONFIG_GENERIC_MSI_IRQ
        INIT_LIST_HEAD(&dev->msi_list);
#endif
        INIT_LIST_HEAD(&dev->links.consumers);
        INIT_LIST_HEAD(&dev->links.suppliers);
```

```
        INIT_LIST_HEAD(&dev->links.needs_suppliers);
        INIT_LIST_HEAD(&dev->links.defer_sync);
        dev->links.status = DL_DEV_NO_DRIVER;
}

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

        dev = get_device(dev);
        //该函数的主要目的为对dev->kref->refcount变量进行自增
        if (!dev)
                goto done;

        if (!dev->p) {
                error = device_private_init(dev);
                //该函数主要用来初始设备的私有数据
                if (error)
                        goto done;
        }

        if (dev->init_name) {
                dev_set_name(dev, "%s", dev->init_name);
                //设置dev->kobj->name为dev->init_name
                dev->init_name = NULL;
        }

        if (!dev_name(dev) && dev->bus && dev->bus->dev_name)
                dev_set_name(dev, "%s%u", dev->bus->dev_name, dev->id);
        //注册总线设备时，该判断条件不成立

        if (!dev_name(dev)) {
                error = -EINVAL;
                goto = name_error;
        }

        pr_debug("device: '%s': %s\n", dev_name(dev), __func__);

        parent = get_device(dev->parent);        //获取当前设备的父节点设备
        kobj = get_device_parent(dev, parent);   //获取父节点设备的kobject对象
        if (IS_ERR(kobj)) {
                error = PTR_ERR(kobj);
                goto parent_error;
        }
        if (kobj)
                dev->kobj.parent = kobj;        //如果父节点设备的kobject对象存在，则设置当前设备的kobject对象的父节点

        if (parent && (dev_to_node(dev) == NUMA_NO_NODE))
                set_dev_node(dev, dev_to_node(parent)); //如果设备的父节点存在，且当前设备未设置NUMA节点，则将当前设备的NUMA节

        error = kobject_add(&dev->kobj, dev->kobj.parent, NULL);
        if (error) {
                glue_dir = get_glue_dir(dev);
                goto Error;
        }

        error = device_platform_notify(dev, KOBJ_ADD);
        if (error)
                goto platform_error;

        error = device_create_file(dev, &dev_attr_uevent);
        //根据属性dev_attr_uevent创建相关文件
        if (error)
                goto attrError;

        error = device_add_class_symlinks(dev);
        if (error)
```

```
                goto SymlinkError;

        error = device_add_attrs(dev);
        if (error)
                goto AttrsError;

        error = bus_add_device(dev);
        //向总线中添加设备，如果当前设备为总线，则该函数不执行，直接返回0
        if (error)
                goto BusError;
        ...
        bus_probe_device(dev);
        //如果当前设备是实际存在的设备时，会检测该设备所挂载的总线上是否使能了设备驱动自动注册，如果已经使能，则调用device_initial_pro
        ...
}
```

After the bus is registered in the form of a device according to the above process, the actual bus object is registered next.

```
int bus_register(struct bus_type *bus)
{
        //这里以platform总线为例，因此传参对象为platform_bus_type，定义如下：
        //struct bus_type platform_bus_type = {
        //      .name = "platform",
        //      .dev_groups = platform_dev_groups,
        //      .match = platform_match,
        //      .uevent = plaform_uevent,
        //      .dma_configure = platform_dma_configure,
        //      .pm      = &platform_dev_pm_ops,
        //};

        int retval;
        struct subsys_private *priv;
        struct lock_class_key *key = &bus->lock_key;

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

        priv->bus = bus;
        bus->p = priv;

        BLOCKING_INIT_NOTIFIER_HEAD(&priv->bus_notifier);

        retval = kobject_set_name(&priv->subsys.kobj, "%s", bus->name);
        if (retval)
                goto out;

        priv->subsys.kobj.kset = bus_kset;
        priv->subsys.kobj.ktype = &bus_ktype;
        priv->drivers_autoprobe = 1;

        retval = kset_register(&priv->subsys);
        ...
        retval = bus_create_file(bus, &bus_attr_uevent);
        ...
        priv->devices_kset = kset_create_and_add("devices", NULL, &priv->subsys.kobj);
        ...
        priv->drivers_kset = kset_create_and_add("drivers", NULL, &priv->subsys.kobj);
        ...
        klist_init(&priv->klist_devices, klist_devices_get, klist_devices_put);
        klist_init(&priv->klist_drivers, NULL, NULL);
        ...
        retval = add_probe_files(bus);
        ...
        retval = bus_add_groups(bus, bus->bus_groups);
}
```

Through the analysis of the above code, it can be seen that there are no substantial device-related operations in the process of bus registration, but mainly based on the concept of the device model proposed by the kernel to create related bus files.

Regarding the PCI bus, its implementation form is different from that of the platform bus:
first, the initialization of the platform bus is in the early stage of kernel initialization, that is, start_kernel->arch_call_rest_init->rest_init->kernel_init->kernel_init_freeable->driver_init->platform_bus_init. The initialization of the PCI bus is done through the driver form.
Second, when initializing the platform bus, it is necessary to abstract the bus into a device, register the device, and then register the bus. The PCI bus does not need it, just register the bus as the former.

The PCI bus registration prototype is as follows:

```
static int __init pci_driver_init(void)
{
        int ret;

        ret = bus_register(&pci_bus_type);
        //关于pci_bus_type，其原型定义如下:
        //struct bus_type pci_bus_type {
        //      .name = "pci",
        //      .match = pci_bus_match,
        //      .uevent = pci_uevent,
        //      .probe = pci_device_probe,
        //      ...
        //};
        if (ret)
                return ret;

#ifdef CONFIG_PCIEPORTBUS
        ret = bus_register(&pcie_port_bus_type);
        if (ret)
                return ret;
#endif
        dma_debug_add_bus(&pci_bus_type);
        return 0;
}
```

As can be seen from the above code, there is indeed no operation related to registering devices during the registration process of the PCI bus. In the process of bus registration, both call the same registration function, so the execution process of the two is roughly the same.

Regarding the difference between the two, it may be related to the following two factors:

1. platform is a virtual bus, pci is not a virtual bus;
2. The platform does not have an actual device corresponding to it, but pci has a device corresponding to it (that is, the PCI controller).

In summary, it is the understanding of the registration process of the bus.