Linux 设备驱动开发

PCI 驱动框架梳理

杜博睿

2022-8-19

PCI 驱动框架梳理

目录

1. PCI 总线驱动

1.1. PCI 与 PCIe 介绍

1.1.1 PCI 总线

PCI（外围部件互联 Peripheral Component Interconnect）是 Intel1991 年推出的一种局部总线，作为一种通用的总线接口标准，他在目前的计算机系统中得到了广泛的应用。从结构上看，PCI 是在 CPU 的供应商和原来的系统总线之间插入的一级总线，具体由一个桥接电路实现对这一层的管理，并实现上下之间的接口以协调数据的传送。PCI 总线结构如图 1.1 所示。

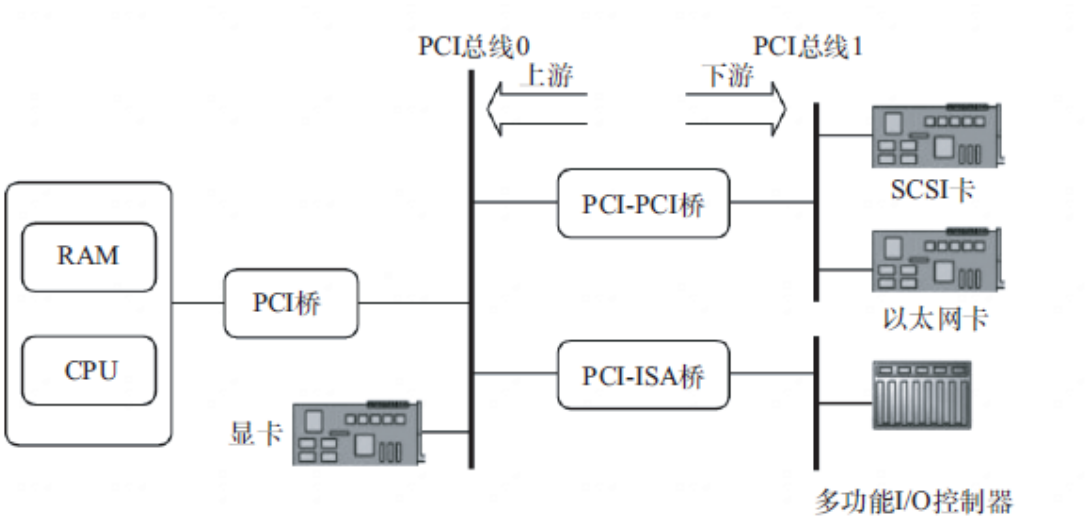

图 1.1 PCI 总线结构示意图

一条 PCI 总线一般有 32 个接口，每一个接口卡对应一个外部设备，每个设备可以有八个功能，每个功能称为逻辑设备。

1.1.2 PCIe 总线

PCI Express 是 INTEL 提出的新一代的高速串行计算机扩展总线标准,PCIe 采用了目前业内流行的点对点串行连接，比起 PCI 以及更早期的计算机总线的共享并行架构，每个设备都有自己的专用连接，独享通道带宽，不共享总线带宽。PCI 是总线结构，而 PCIe 则是点对点结构，PCIe 结构如图 1.2 所示。

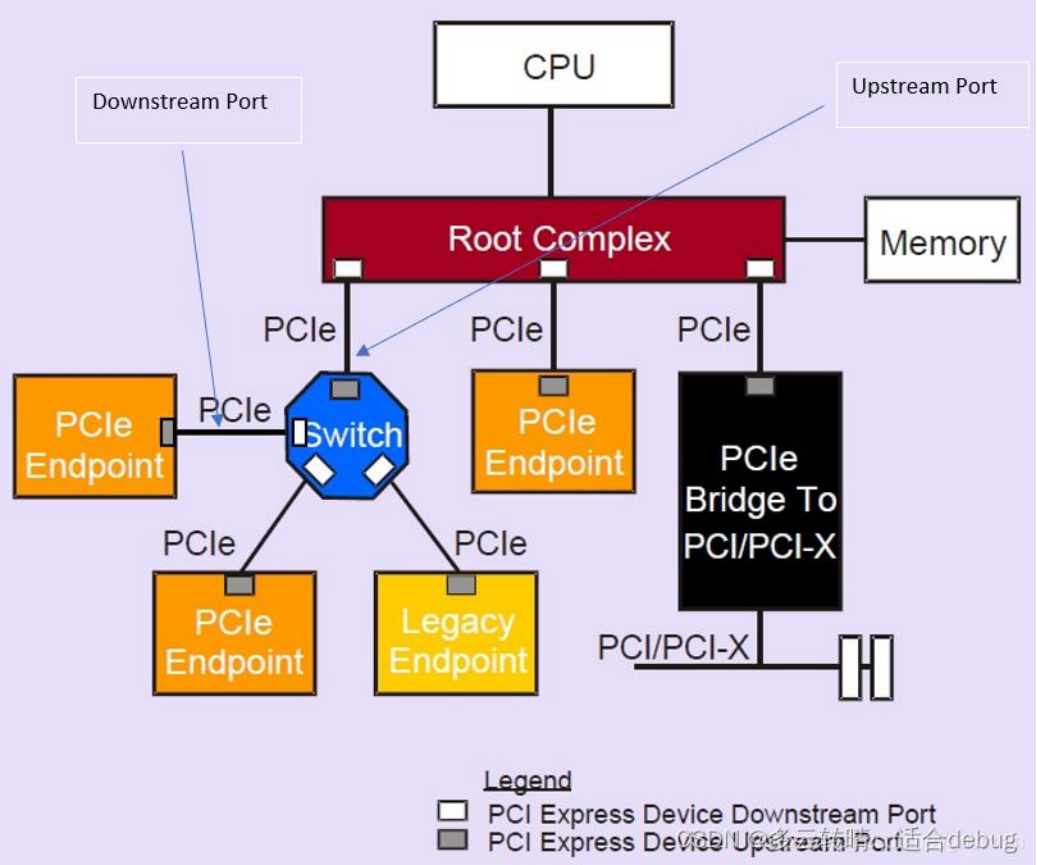

图 1.2 PCIe 结构示意图

1.1.3 PCI 设备配置空间

每个逻辑设备都对应有一个 PCI 配置空间，其中记录了此设备的详细信息。PCI 配置空间最大 256 个自己，其中开头的 64 个字节的格式是定义好了的，前 16 个字节的格式是一定的，包含了头部的类型、设备的种类、设备的性质以及制造商等等基本信息。PCI 配置空间前 64 个字节格式如图 1.3 所示。

HeaderType 表明了 PCI 设备的类型，Type 0 表示 EndPoint 设备，即具有各种功能的逻辑设备，Type 1 表示 Bridge 或者 Switch。

之后的 6 个 BAR 空间为基地址（Base Address），每个 BAR 记录了该 PCI 设备映射物理内存的地址，BAR 的最后一位区分 I/O 端口和 I/O 内存。如图 1.4 和图 1.5 所示。这些 BAR 映射的物理地址是连续的，即这些 BAR 共同描述设备的地址空间范围。

接下来值得注意的是中断，大部分的 PCI 设备与系统的交互就是通过中断进行的。IRQ Pin 记录设备是否支持中断，1 表示支持，0 表示不支持。IRQ Line 中记录了中断号。

当 PCI 设备为 PCI 桥时，值得注意的是 PCI 桥配置空间的三个窗口：IP 地址区间窗口、存储器区间窗口、可预取存储器窗口；这三个窗口使得从北桥出来的地址若在区间内则可以过桥来寻找设备，而从南桥出来的地址不在范围内才可以通过桥。

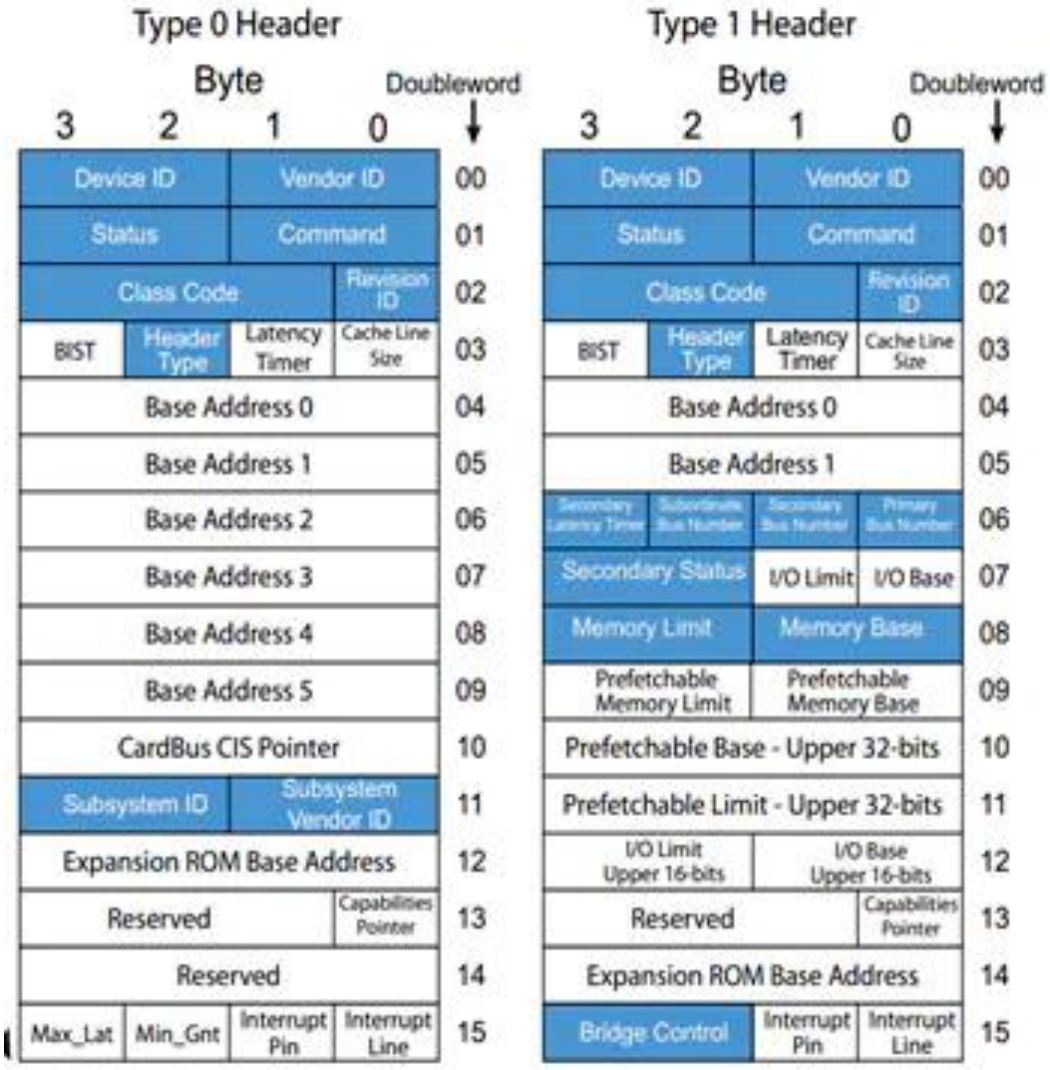

图 1.3 配置空间头部 64 字节

图 1.4 I/O 内存（MMIO）

图 1.5 I/O 端口（PIO）

1.1.4 PCI 驱动框架

PCI 设备驱动实际上包括 PCI 设备驱动（总线/控驱动）和设备本身的驱动（设备驱动）。前者主要由内核在系统初始化时完成。

1. 总线驱动程序：负责枚举、配置和控制设备，主要实现 PCI 控制器的注册与 PCI 设备的枚举。
2. 设备驱动程序：保存和还原设备上下文，针对 PCI 接口具体设备实现其指定功能。

PCI 驱动的工作流程和设备驱动模型如图 1.6、1.7 所示，

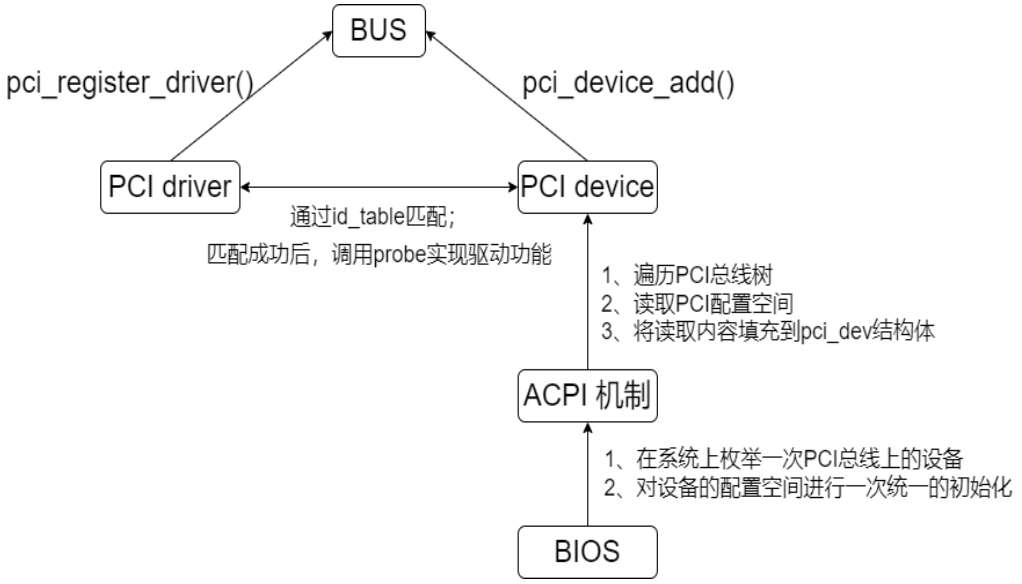

图 1.6 PCI 驱动整体工作流程

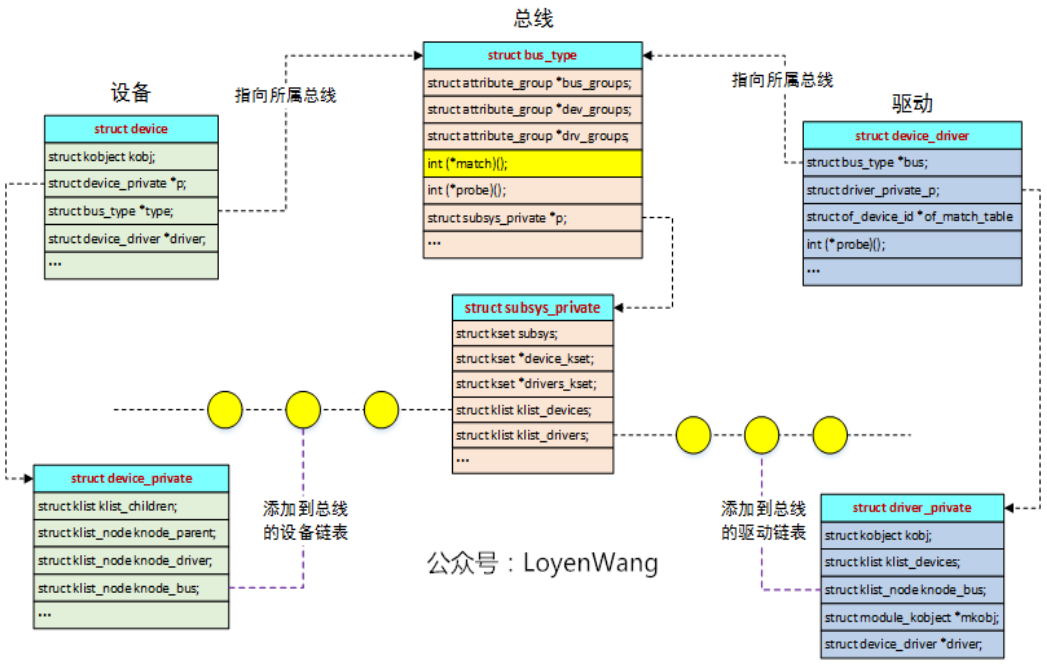

图 1.7 PCI 设备驱动模型

从上图可以看出 Linux 内核建立了一个统一的设备模型，分别用总线、设备、驱动进行抽象，其中设备和驱动都挂在总线上。PCI 设备模型是由总线驱动在系统初始化时建立的，BIOS 通过 ACPI 机制遍历枚举 PCI 总线树，读取 PCI 设备的配置空间填充入 pci_dev 结构体中；当新的设备注册或者新的驱动通过 pci_device_add()、pci_register_drive()注册到总线上是时，总线会进行匹配操作（match 函数），pci_dev 和 pci_driver 通过 id_table 查询比较 VendorID、Device ID 等进行匹配；匹配成功后就会进行 probe 函数操作实现驱动功能。

1.2. PCI 总线驱动的数据结构

在上节中我们已经了解到 PCI 总线的基本任务就是完成 PCI 控制器的注册以及 PCI 设备的枚举和配置。为了完成这几个任务，我们先了解一下总线驱动相关的数据结构，如图 2.1 所示。

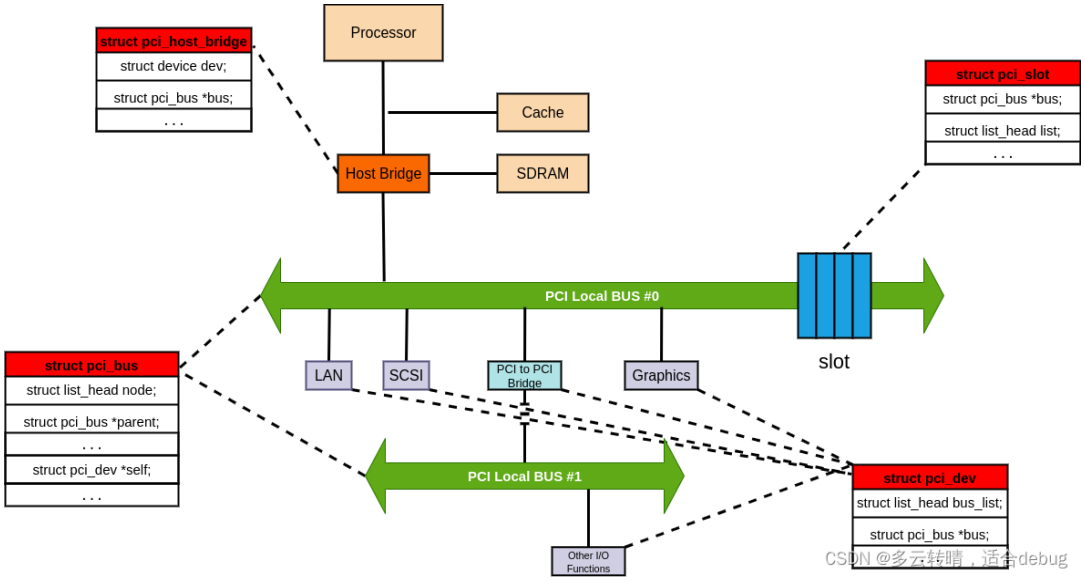

图 2.1 PCIe 的数据结构

这些结构体的描述如表 2.1 所示

数据结构	描述
`struct pci_host_bridge`	主桥数据结构，用来描述连接 CPU 和 PCIE 设备的主桥，该结构有 `Root bus0`成员，它也是一个设备，需要注册。
`struct pci_dev`	该结构体用来描述 PCI 设备，包括 EP 和 `pci 桥`等设备。
`struct pci_bus`	该结构体用来描述 PCI 的总线。
`struct pci_slot`	用来描述 bus 下的物理插槽。
`struct pci_bus_type`	这个是 `pci` 的总线模型，和之前的 `pci_bus` 是两码事，`pci_driver` 和 `pci_dev` 就是通过该总线关联起来。

表 2.1 PCI 总线描述

PCI 驱动框架梳理

1.2.1. PCI 总线描述：pci_bus

```
1.  struct pci_bus {
2.      /* 链表元素 node：对于 PCI 根总线而言，其 pci_bus 结构通过 node 成员链接
    到本节一开始所述的根总线链表中，根总线链表
3.      的表头由一个 list_head 类型的全局变量 pci_root_buses 所描述。而对于非根
    pci 总线，其 pci_bus 结构通过 node 成员链接
4.      到其父总线的子总线链表 children 中*/
5.      struct list_head node;
6.      /*parent 指针：指向该 pci 总线的父总线，即 pci 桥所在的那条总线*/
7.      struct pci_bus *parent;
8.      /*children 指针：描述了这条 PCI 总线的子总线链表的表头。这条 PCI 总线的所
    有子总线都通过上述的 node 链表元素链接成一
9.      条子总线链表，而该链表的表头就由父总线的 children 指针所描述*/
10.     struct list_head children;
11.     /* list of devices on this bus */
12.     struct list_head devices;
13.     /* devices 链表头：描述了这条 PCI 总线的逻辑设备链表的表头。除了链接在全
    局 PCI 设备链表中之外，每一个 PCI 逻辑设备也
14.     通过其 pci_dev 结构中的 bus_list 成员链入其所在 PCI 总线的局部设备链表
    中，而这个局部的总线设备链表的表头就由
15.     pci_bus 结构中的 devices 成员所描述*/
16.     struct pci_dev *self;
17.     /* 资源指针数组：指向应路由到这条 pci 总线的地址空间资源，通常是指向对应
    桥设备的 pci_dev 结构中的资源数组 resource［10：7］*/
18.     struct resource *resource[PCI_BUS_NUM_RESOURCES];
19.     /* 指针 ops：指向一个 pci_ops 结构，表示这条 pci 总线所使用的配置空间访问
    函数*/
20.     struct pci_ops *ops;
21.     /* 无类型指针 sysdata：指向系统特定的扩展数据*/
22.     void *sysdata;
23.     /* 指针 procdir：指向该 PCI 总线在 / proc 文件系统中对应的目录项*/
24.     struct proc_dir_entry *procdir;
25.     /* number：这条 PCI 总线的总线编号（bus number），取值范围 0—
    255 */
26.      unsigned char number;
27.     /* primary：表示引出这条 PCI 总线的“桥设备的主总线”（也即桥设备所在的
    PCI 总线）编号，取值范围 0—255*/
28.     unsigned char primary;/* secondary：表示引出这条 PCI 总线的桥设备的次
    总线号，因此 secondary 成员总是等于 number 成员的值。取值范围 0—255*/
29.     unsigned char secondary;
30.     /* subordinate：这条 PCI 总线的下属 PCI 总线（Subordinate pci bus）的
    总线编号最大值，它应该等于引出这条 PCI 总线的桥设备的 subordinate 值*/
31.     unsigned char subordinate;
32.     /* name［48］：这条 PCI 总线的名字字符串*/
33.     char name[48];
```

系统中所有根总线都通过其 pci_bus 结构体中的 node 成员链接成一条全局的根总线链表，其表头由 pci_root_buses 来描述。而根总线下面的所有下级总线则都通过 pci_bus 结构体中的 node 成员链接到其父总线的 children 链表中。这样，通过这两种 PCI 总线链表，Linux 内核就将所有的 pci_bus 结构体以一种倒置树的方式组织起来。例如，对于图 2.2 所示的 PCI 总线结构，它对应的总线链表结构应为图 2.3 所示。

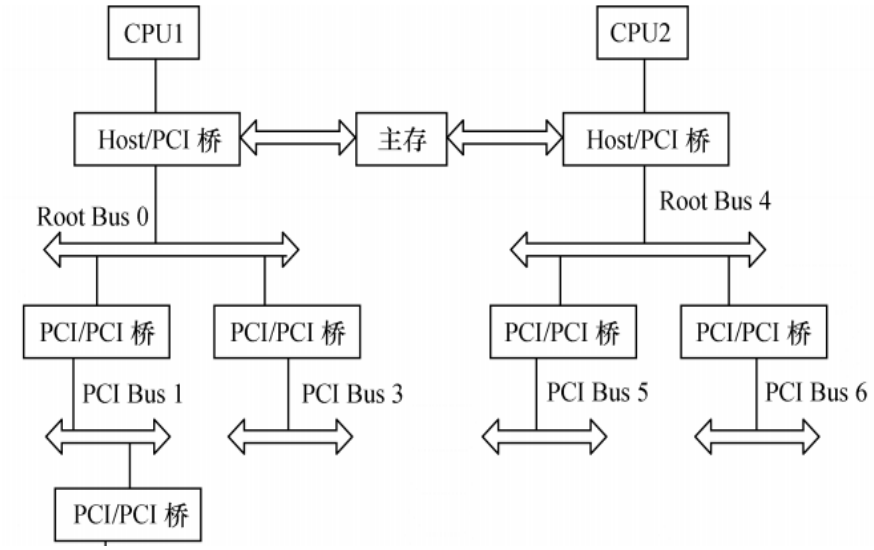

图 2.2 多跟 PCI 总线体系结构

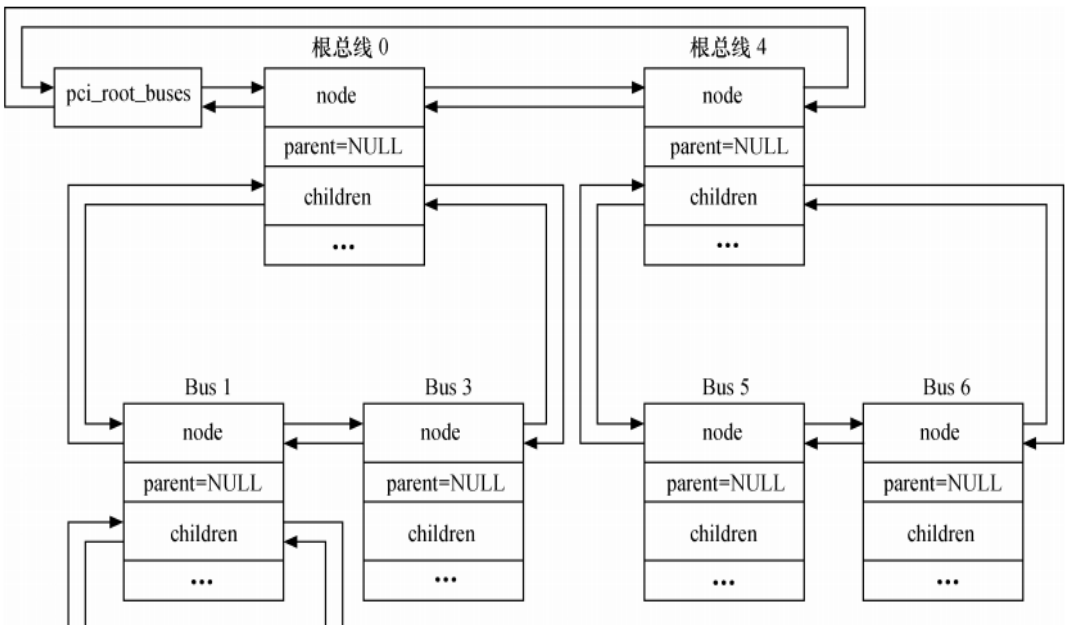

图 2.3 总线链表结构

1.2.2. PCI 设备描述：pci_dev

Linux 系统中，所有种类的 PCI 设备都可以用 pci_dev 结构体来描述（包括 EP 和 Bridge），每一个 PCI 逻辑设备都唯一的对应一个 pci_dev。

```
1.  struct pci_dev {
2.  /* 总线设备链表元素 bus_list：每一个 pci_dev 结构除了链接到全局设备链表中
    外，还会通过这个成员连接到 其所属 PCI 总线的设备链表中。每一条 PCI 总线都维护
    一条它自己的设备链表视图，以便描述所有连接在该 PCI 总线上的设备，其表头由 PCI
    总线的 pci_bus 结构中的 devices 成员所描述 t */
3.  struct list_head bus_list;
4.  /* 总线指针 bus：指向这个 PCI 设备所在的 PCI 总线的 pci_bus 结构。因此，对于桥
    设备而言，bus 指针将指向 桥设备的主总线（primary bus），也即指向桥设备所在
    的 PCI 总线 */
5.  struct pci_bus *bus; /* 这个 PCI 设备所在的 PCI 总线的 pci_bus 结
    构 */
6.  /* 指针 subordinate：指向这个 PCI 设备所桥接的下级总线。这个指针成员仅对桥设
    备才有意义，而对于一般的非桥 PCI 设备而言，该指针成员总是为 NULL */
7.  struct pci_bus *subordinate; /* 指向这个 PCI 设备所桥接的下级总
    线 */
8.  /* 无类型指针 sysdata：指向一片特定于系统的扩展数据*/
9.  void *sysdata; /* 指向一片特定于系统的扩展数据 */
10. /* 指针 procent：指向该 PCI 设备在 / proc 文件系统中对应的目录项*/
11. struct proc_dir_entry *procent; /* 该 PCI 设备在/proc/bus/pci 中对应的
    目录项 */
12. struct pci_slot *slot; /* 设备位于的物理插槽 */
13. /* devfn：这个 PCI 设备的设备功能号，也成为 PCI 逻辑设备号（0—255）。其中
    bit[7:3]是物理设备号（取值范围 0—31），bit[2:0]是功能号（取值范围 0—
    7）。 */
14. unsigned int devfn; /* 这个 PCI 设备的设备功能号 */
15. /* vendor：这是一个 16 无符号整数，表示 PCI 设备的厂商 ID*/
16. unsigned short vendor;
17. /*device：这是一个 16 无符号整数，表示 PCI 设备的设备 ID */
18. unsigned short device;
19. /* subsystem_vendor：这是一个 16 无符号整数，表示 PCI 设备的子系统厂商
    ID*/
20. unsigned short subsystem_vendor;
21. /* subsystem_device：这是一个 16 无符号整数，表示 PCI 设备的子系统设备
    ID*/
22. unsigned short subsystem_device;
23. /* class：32 位的无符号整数，表示该 PCI 设备的类别，其中，bit［7：0］为编程
    接口，bit［15：8］为子类别代码，bit ［23：16］为基类别代码，bit［31：24］
    无意义。显然，class 成员的低 3 字节刚好对应与 PCI 配置空间中的类代码
    */
24. unsigned int class;
25. /* hdr_type：8 位符号整数，表示 PCI 配置空间头部的类型。其中，bit［7］=1 表
    示这是一个多功能设备，bit［7］=0 表示这是一个单功能设备。Bit［6：0］则表示
    PCI 配置空间头部的布局类型，值 00h 表示这是一个一般 PCI 设备的配置空间头部，值
```

同一条 PCI 总线上的 PCI 设备通过 `struct list_head bus_list` 组成总线的设备链表；表头为 `pci_bus` 结构体中 `struct list_head devices` 成员。

1.2.3. PCI 驱动描述：pci_driver

```
1.  1.  struct pci_driver {
2.  2.      struct list_head    node;
3.  3.      const char      *name;
4.  4.      const struct pci_device_id *id_table;   /* Must be non-
    NULL for probe to be called */ /*不能为 NULL，以便 probe 函数调用*/
5.  5.      /* 新设备添加 */
6.  6.      int  (*probe)(struct pci_dev *dev, const struct pci_de-
    vice_id *id); /* New device inserted */
7.  7.      void (*remove)(struct pci_dev *dev);    /* Device re-
    moved (NULL if not a hot-plug capable driver) */ /* 设备移出 */
8.  8.      int  (*suspend)(struct pci_dev *dev, pm_mes-
    sage_t state);   /* Device suspended *//* 设备挂起 */
9.  9.      int  (*suspend_late)(struct pci_dev *dev, pm_mes-
    sage_t state);
10. 10.     int  (*resume_early)(struct pci_dev *dev);
11. 11.     int  (*resume) (struct pci_dev *dev);   /* De-
    vice woken up */ /* 设备唤醒 */
12. 12.     void (*shutdown) (struct pci_dev *dev);
13. 13.     int  (*sriov_config-
    ure) (struct pci_dev *dev, int num_vfs); /* On PF */
14. 14.     const struct pci_error_handlers *err_handler;
15. 15.     const struct attribute_group **groups;
16. 16.     struct device_driver    driver;
17. 17.     struct pci_dynids   dynids;
18. 18. };
```

1.3. PCI 总线驱动的流程框架

Linux 的总线驱动基本流程如下：

第一步：Linux 分配数据结构 pci_contoller，并初始化，包括 PCI 的 mem/io 空间范围和访问 PCI 配置空间所需的 handler。

第二步：PCI 设备的枚举：扫描系统上所有 PCI 设备，初始化它们的配置空间，用数据结构将 PCI 设备信息联系起来，构建 PCI 树。

第三步： 加载 PCI 设备驱动。

1.3.1. 初始化 PCI 控制器

pci_controller 是内核描述 PCI 子系统信息的数据结构，里面定义了可供 PCI 设备使用的 mem 资源和 io 资源的范围，访问 pci 设备配置空间的 handler 等等。PCI 控制器的注册与具体的开发板相关。

```
1.  //龙芯 1A 开发板 2.6.33 内核
2.  static struct resource loongson_pci_mem_resource = {
3.      .name   = "pci memory space",
4.      .start  = 0x14000000UL,
5.      .end    = 0x17ffffffUL,
6.      .flags  = IORESOURCE_MEM,
7.  };
8.
9.  static struct resource loongson_pci_io_resource = {
10.     .name   = "pci io space",
11.     .start  = 0x00004000UL,
12.     .end    = IO_SPACE_LIMIT,
13.     .flags  = IORESOURCE_IO,
14. };
```

```
1.  static struct pci_controller  loongson_pci_controller = {
2.      .pci_ops        = &sb2f_pci_pci_ops,
3.      .io_resource    = &loongson_pci_io_resource,
4.      .mem_resource   = &loongson_pci_mem_resource,
5.      .mem_offset     = 0x00000000UL,
6.      .io_offset      = 0x00000000UL,
7.  };
8.
9.  static void __init setup_pcimap(void)
10. {
11.     /*
12.      * local to PCI mapping for CPU accessing PCI space
13.      * CPU address space [256M,448M] is window for access-
    ing pci space
14.      * we set pcimap_lo[0,1,2] to map it to pci space[0M,64M], [320M,
    448M]
15.      *
16.      * pcimap: PCI_MAP2  PCI_Mem_Lo2 PCI_Mem_Lo1 PCI_Mem_Lo0
17.      *       [<2G]   [384M,448M] [320M,384M] [0M,64M]
18.      */
19.     SB2F_PCIMAP = 0x46140;
20. }
21.
22. static int __init pcibios_init(void)
23. {
24.     setup_pcimap();
25.
26.     //设置 pci 总线的 3 个 mem space 空间的高 6 位
27.
28.     if(!disablepci)
29.     {
30.     //将 pci 控制器添加到内核的 pci 控制器链表中
31.         register_pci_conroller(&loongson_pci_controller);
32.     }
33.
34.     return 0;
35. }
```

如此完成了 PCI 控制器的初始化，其中控制器结构体的 pci_ops 成员提供的读写函数就是对 PCI 设备配置寄存器的读写。

1.3.2. 基于 ACPI 的 PCI 设备枚举

ACPI(Advanced Configuration and Power Interface)， ACPI 提供了电源、硬件和固件的接口。这里只关注软件角度的 ACPI 的结构，在屏蔽了硬件细节的同时，提供了一系列系统资源，包括：ACPI 寄存、ACPI BIOS、ACPI Tables，枚举过程可以看作两步，

BIOS 的枚举：初始化所有 PCI 设备

系统的枚举：根据 BIOS 枚举生成 pci_dev 结构体。

下面来看枚举的具体过程。首先需要在内核中创建一个 PCI 总线（pci_bus_type）来挂载 PCI 设备和 PCI 驱动。此处 pcibus_class_init 是调用了一个 class_register 函数组测了一个 pcibus_class 结构体。接着 pci_driver_init 调用 bus_register 完成了 pci_bus_type 的注册。

```
1.  static struct class pcibus_class = {
2.          .name       = "pci_bus",
3.          .dev_release    = &release_pcibus_dev,  //资源
4.          .dev_groups = pcibus_groups,
5.      };
6.
7.      static int __init pcibus_class_init(void)
8.      {
9.          return class_register(&pcibus_class);
10.     }
11.     postcore_initcall(pcibus_class_init);
```

```
1.  struct bus_type pci_bus_type = {
2.      .name       = "pci",
3.      .match      = pci_bus_match,        //匹配函数
4.      .uevent     = pci_uevent,           //用户事件
5.      .probe      = pci_device_probe,     //匹配后的执行函数
6.      .remove     = pci_device_remove,    //退出函数
7.      .shutdown   = pci_device_shutdown,
8.      .dev_groups = pci_dev_groups,
9.      .bus_groups = pci_bus_groups,
10.     .drv_groups = pci_drv_groups,
11.     .pm     = PCI_PM_OPS_PTR,           //电源管理相关
12. };
13. EXPORT_SYMBOL(pci_bus_type);
14.
15. static int __init pci_driver_init(void)
16. {
17.     return bus_register(&pci_bus_type);
18. }
19. postcore_initcall(pci_driver_init);
```

接着是 ACPI 机制初始化 PCI 的相关操作，主要分为两个部分：

第一部分：acpi_pci_root_init 完场 PCI 设备的相关操作（包括 PCI 主桥，PCI 桥、PCI 设备的枚举，配置空间的设置，总线号的分配等）；

第二部分：acpi_pci_link_init 完成 PCI 中断的相关操作。

整个函数流程如图 2.4 所示。

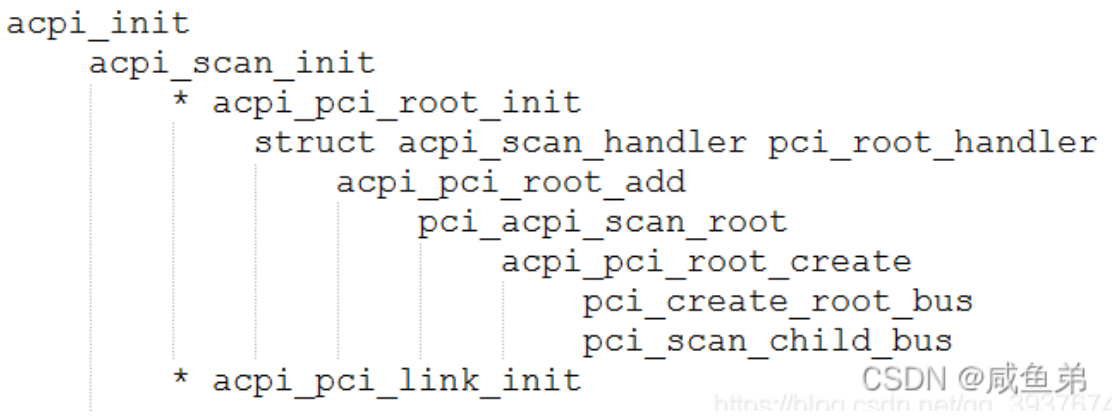

图 2.4 枚举过程的函数流程

acpi_pci_root_add 函数：通过 ACPI 表中的_SEG 和_BBN 参数获得 HOST 主桥的 Segment 和 Bus 号，创建 acpi_pci_root 结构（表示 HOST 主桥信息）。

```
static int acpi_pci_root_add(struct acpi_device *device,
                 const struct acpi_device_id *not_used)
{
    ......
    negotiate_os_control(root, &no_aspm);
    root->bus = pci_acpi_scan_root(root);
    //错误检验
    if (!root->bus) {
        dev_err(&device->dev,
            "Bus %04x:%02x not present in PCI namespace\n",
            root->segment, (unsigned int)root->secondary.start);
        device->driver_data = NULL;
        result = -ENODEV;
        goto remove_dmar;
    }
    ......
}
```

pci_acpi_scan_root 函数：开始枚举，先调用 pci_find_bus 去判断当前总线号是否已经存在，如果存在就退出，如果不存在就调用 acpi_pci_root_create 函数去对这条 PCI 总线进行遍历，在这里我们需注意一个结构就是 acpi_pci_root_ops，它是新版本的内核（4.0）提供给我们对于 PCI 信息的一些接口函数的集合（这其中就包括对配置空间的读写方法）

```
1.  struct pci_bus *pci_acpi_scan_root(struct acpi_pci_root *root)
2.  {
3.      int domain = root->segment;
4.      int busnum = root->secondary.start;
5.      int node = pci_acpi_root_get_node(root);
6.      struct pci_bus *bus;
7.
8.      ......
9.
10.     bus = pci_find_bus(domain, busnum);
11.     if (bus) {                                  //如果当前总线存在就不会进行
    create，直接就返回
12.         struct pci_sysdata sd = {
13.             .domain = domain,
14.             .node = node,
15.             .companion = root->device
16.         };
17.
18.         memcpy(bus->sysdata, &sd, sizeof(sd));
19.     } else {
20.         struct pci_root_info *info;
21.
22.         info = kzalloc_node(sizeof(*info), GFP_KERNEL, node);  //数据
    结构实体化
23.         if (!info)
24.             dev_err(&root->device->dev,
25.                 "pci_bus %04x:%02x: ignored (out of memory)\n",
26.                 domain, busnum);
27.         else {
28.             info->sd.domain = domain;
29.             info->sd.node = node;
30.             info->sd.companion = root->device;
31.             bus = acpi_pci_root_create(root, &acpi_pci_root_ops,
32.                                &info->common, &info->sd);
33.         }
34.     }
35.
36.     ......
37.
38.     return bus;
39. }
```

acpi_pci_root_create 函数：将 resources、读写方法、总线号等数据传入 pci_create_root_bus 这个函数，通过这个函数返回一个总线结构 pci_bus 中。

pci_scan_child_bus 函数：

PCI 驱动框架梳理

```
1.  /*通过 for 循环先遍历当前总线上的每一个 PCI 设备（包括 PCI 桥设备），遍历后将
    其注册为 pci_dev 结构体，之后通过递归调用来进入当前总线的下一级子总线继续完成
    上述过程，直到某条 PCI 总线上无 PCI 桥设备，返回最大的总线号。*/
2.  unsigned int pci_scan_child_bus(struct pci_bus *bus)
3.  {
4.      unsigned int devfn, pass, max = bus->busn_res.start;
5.      struct pci_dev *dev;
6.
7.      dev_dbg(&bus->dev, "scanning bus\n");
8.
9.      /* Go find them, Rover! */
10.     for (devfn = 0; devfn < 0x100; devfn += 8)
11.         pci_scan_slot(bus, devfn);
12.
13.     /* Reserve buses for SR-IOV capability. */
14.     max += pci_iov_bus_range(bus);
15.
16.     if (!bus->is_added) {
17.         dev_dbg(&bus->dev, "fixups for bus\n");
18.         pcibios_fixup_bus(bus);
19.         bus->is_added = 1;
20.     }
21.
22.     //为了兼容 x86 和其他不使用 BIOS 的架构的 CPU，执行两次
23.     for (pass = 0; pass < 2; pass++)
24.         list_for_each_entry(dev, &bus->devices, bus_list) {
25.             if (pci_is_bridge(dev))    //如果是总线上有桥设备就递归调
    用
26.                 max = pci_scan_bridge(bus, dev, max, pass);
27.         }
28.
29.     dev_dbg(&bus->dev, "bus scan return-
    ing with max=%02x\n", max);
30.     return max;
31. }
```

pci_scan_slot 实质是调用 pci_scan_single_device，通过 pci_scan_device 函数根据 BIOS 遍历结果填充 pci_dev 结构，之后在用 pci_device_add 注册这个结构体。

pci_scan_bridge：

```
1.  int pci_scan_bridge(struct pci_bus *bus, struct pci_dev *dev,
2.                          int max, int pass)
3.  {
4.      struct pci_bus *child;
5.      int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);
6.      u32 buses, i, j = 0;
7.      u16 bctl;
8.      u8 primary, secondary, subordinate;
9.      int broken = 0;
10.
11.     ......
12.
13.     if ((secondary || subordinate) && !pcibios_assign_all_bus-
    ses() &&
14.         !is_cardbus && !broken) {
15.         unsigned int cmax;
16.
17.         if (pass)
18.             goto out;
19.
20.         child = pci_find_bus(pci_domain_nr(bus), secondary);     //检查
    该 bus 是否存在
21.         if (!child) {
22.             child = pci_add_new_bus(bus, dev, secondary);
23.             if (!child)
24.                 goto out;
25.             child->primary = primary;
26.             pci_bus_insert_busn_res(child, secondary, subordi-
    nate);
27.             child->bridge_ctl = bctl;
28.         }
29.
30.         cmax = pci_scan_child_bus(child);           //递归调用，进入到下
    一级 PCI BUS
31.         if (cmax > subordinate)
32.             dev_warn(&dev->dev, "bridge has subordi-
    nate %02x but max busn %02x\n",
33.                     subordinate, cmax);
34.         /* subordinate should equal child->busn_res.end */
35.         if (subordinate > max)
36.             max = subordinate;
37.     } else {
38.         ......
39.     }
```

本函数（通过 for 循环）调用了两次，原因是：在不同架构的对于 PCI 设备的枚举实现是不同的，例如在 x86 架构中有 BIOS 为我们提前去遍历一遍 PCI 设备，但是在 ARM 或者 powerPC 中 uboot 是没有这种操作的，所以为了兼容

这两种情况，这里就执行两次对应于两种不同的情况，当 pci_scan_slot 函数执行完了后，我们就得到了一个当前 PCI 总线的设备链表，在执行 pci_scan_bridge 函数前，会遍历这个设备链表，如果存在 PCI 桥设备，就调用 pci_scan_bridge 函数，而在本函数内部会再次调用 pci_scan_child_bus 函数，去遍历子 PCI 总线设备（注意：这时的 BUS 就已经不是 PCI BUS 0 了）就是通过这种一级一级的递归调用，在遍历总 PCI 总线下的每一条 PCI 子总线。直到某条 PCI 子总线下无 PCI 桥设备，就停止递归，并修改 subbordinate 参数，（最大 PCI 总线号）返回。

至此，已经完成了 PCI 总线的枚举过程，得到了 PCI 总线下所有设备的信息，并将它们注册成了各自的 pci_dev 结构体形成了设备树，接下来就是等待后续要注册的 PCI 驱动与他们匹配。

1. PCI 设备驱动（rtl8139）

上面已经分析了 PCI 总线驱动，从本质上讲，PCI 只是一种总线，前文中的总线驱动知识为了辅助设备本身的驱动，他不是目的而是手段，对于 PCI 设备，驱动部分除了要实现总线驱动以外，其主体仍然是设备本身的驱动。这一章以 rtl8139 网卡驱动为例分析梳理 PCI 设备的设备驱动代码的框架和流程。

2.1.初始化设备驱动

在 2.1.3 节中已经构建了 pci_driver 结构体，接下来定义网卡驱动的结构体 rtl_8139_pcidriver。

```
1.  static struct pci_driver rtl8139_pci_driver = {
2.        name: MODNAME,
3.        id_table: rtl8139_pci_tbl,
4.        probe: rtl8139_init_one,
5.        remove: rtl8139_remove_one,
6.    };
```

然后进行初始化：

```
1.  static int __init rtl8139_init_module (void)
2.  {
3.      return pci_module_init (&rtl8139_pci_driver);
4.  }
```

可以看到初始化中主要是调用了 pci_moduleinit 接口函数，在此接口中主要是调用了 pci_register_driver()。此函数位于 Linux/drivers/pci/pci.c 中，它主要实现了三件事：

1、把传递来的 rtl8139_pci_driver 在内核中进行了注册，将他挂载到内核中的 PCI 设备大链表中。

2、遍历总线上的所有 PCI 设备以及他们的配置空间与 rtl8139_pci_driver 中的 id_table 成员进行比对匹配。匹配成功后执行 probe 函数进行设备初始化。

3、将 rtl_8139_pci_driver 挂在网卡的数据结构（pci_dev）上，表示网卡有了自己的驱动，驱动也找到了他的服务对象。

在这三件事中的第二件事，调用 probe 函数（此驱动是 rtl8139_init_one）对整个设备进行了初始化并做了一些准备工作。他是这样实现的：

（1）利用 init_ethrdev 建立 net_device，让他在内核中代表这个网络设备。（同样是指代此网卡设备，pci_dev 负责网卡的 PCI 规范，net_device 则负责网络设备这个职责）。

```
dev = init_etherdev (NULL, sizeof (*tp));
  if (dev == NULL) {
     printk ("unable to alloc new ethernet\n");
     return -ENOMEM;
  }
  tp = dev->priv;
```

（2）开启设备（开启设备的寄存器与内存的映射功能）

```
rc = pci_enable_device (pdev);
  if (rc)
     goto err_out;
```

（3）获得各项资源

```
mmio_start = pci_resource_start (pdev, 1);
mmio_end = pci_resource_end (pdev, 1);
mmio_flags = pci_resource_flags (pdev, 1);
mmio_len = pci_resource_len (pdev, 1);
```

在总线驱动中的 BIOS 枚举阶段，为硬件设备分配了互不冲突的地址并将此地址写入了配置空间，然后在系统初始化时已经将设备配置空间的信息存到了他们的 pci_dev 结构体中的 resource 成员中，此处直接从 resource 中读取即可。

（4）将得到的地址进行映射

```
1.  ioaddr = ioremap (mmio_start, mmio_len);
2.
3.    if (ioaddr == NULL) {
4.       printk ("cannot remap MMIO, aborting\n");
5.       rc = -EIO;
6.       goto err_out_free_res;
7.    }
```

ioremap 是讷河提供的结构函数，用来映射外设寄存器到主存，将物理地址 mmio_start 映射为了虚拟地址 ioaddr。

（5）重启网卡设备（将寄存器相应位置置 1）

```
1.  writeb ((readb(ioaddr+ChipCmd) & ChipCmdClear) | CmdReset,
2.              ioaddr+ChipCmd);
```

（6）huode MAC 地址，并储存到 net_device 中

```
1.  for(i = 0; i < 6; i++) { /* Hardware Address */
2.       dev->dev_addr[i] = readb(ioaddr+i);
3.       dev->broadcast[i] = 0xff;
4.    }
```

MAC 地址是网络中标识网卡的物理地址，这个地址在以后的收发数据包时会用的上。

（7）向 net_device 中登记一些主要函数

```
1.  dev->open = rtl8139_open;
2.  dev->hard_start_xmit = rtl8139_start_xmit;
3.  dev->stop = rtl8139_close;
```

rtl8139_remove_one 则基本是 rtl8139_init_one 的逆过程。

2.2. 打开设备

rtl8139_open（）在此阶段完成了三件事：

1、注册设备的中断函数：

```
1.  retval = request_irq (dev->irq, rtl8139_interrupt, SA_SHIRQ,
2.                            dev->name, dev);
3.    if (retval) {
4.       return retval;
5.    }
```

网卡在发送数据完成后以及接收到数据时，使用中断的形式告知的，因此需要注册一个中断处理函数，其中断号是在总线驱动中分配的，此处直接从 pci_dev 中的 irq 成员中取出即可。

2、分配缓冲空间

网卡数据发送的过程：先从应用程序中把数据包拷贝到一段连续的内存中(这段内存就是我们这里要分配的缓存)；然后把这段内存的地址写进网卡的数据发送地址寄存器(TSAD)中；再把这个数据包的长度写进另一个寄存器(TSD)中；然后就把这段内存的数据发送到网卡内部的发送缓冲中(FIFO)；最后由这个发送缓冲区把数据发送到网线上。因此，需要创建数据发送和接受的内存缓冲区。

```
1.  tp->tx_bufs = pci_alloc_consistent(tp->pci_dev, TX_BUF_TOT_LEN,
2.                                        &tp->tx_bufs_dma);
3.  tp->rx_ring = pci_alloc_consistent(tp->pci_dev, RX_BUF_TOT_LEN,
4.                                        &tp->rx_ring_dma);
```

其中，tx_bufs_dma 和 rx_ring_dma 是这一段内存的物理地址供网卡访问使用，而 tx_bufs 和 rx_bufs 是虚拟地址供 CPU 访问使用。

3、缓存空间初始化

2.3. 数据收发

2.3.1. 发送数据

当一个网络应用程序要向网络发送数据时，它要利用 Linux 的网络协议栈来解决一系列问题，找到网卡设备的代表 net_device，由这个结构来找到并控制这个网卡设备来完成数据包的发送，具体是调用 net_device 的 hard_start_xmit 成员

函数，这是一个函数指针，在我们的驱动程序里它指向的是 rtl8139_start_xmit，正是由它来完成我们的发送工作的，下面我们就来剖析这个函数。它一共做了四件事。

（1）检查这个要发送的数据包的长度，如果它达不到以太网帧的长度，必须采取措施进行填充。

```
1.  if( skb->len < ETH_ZLEN ){//if data_len < 60
2.      if( (skb->data + ETH_ZLEN) <= skb->end ){
3.          mem-
    set( skb->data + skb->len, 0x20, (ETH_ZLEN - skb->len) );
4.          skb->len = (skb->len >= ETH_ZLEN) ? skb->len : ETH_ZLEN;}
5.      else{
6.          printk("%s:(skb->data+ETH_ZLEN) > skb->end\n",
7.                                                  __FUNCTION__);
8.      }
9.   }
```

（2）将数据拷贝到发送缓存

```
1.  memcpy (tp->tx_buf[entry], skb->data, skb->len);
```

（3）将数据包长度写入 TSD

```
1.  writel(tp->tx_flag | (skb->len >= ETH_ZLEN ? skb->len : ETH_ZLEN),
2.                          ioaddr+TxStatus0+(entry * 4));
```

（4）若缓存已满则停发 queue

```
1.  if ((tp->cur_tx - NUM_TX_DESC) == tp->dirty_tx)
2.    netif_stop_queue (dev);
```

2.3.2. 接收数据

当数据从网线过来时，网卡产生一个中断，调用终端服务函数 rtl8139_interrupt 从网卡的中断状态寄存器中读取值，有三种情形：

（1） 不做处理

```
1.  if ((status &(PCIErr | PCSTimeout | RxUnderrun | RxOverflow |RxFIFOO-
    ver | TxErr | TxOK | RxErr | RxOK)) == 0)
2.       goto out;
```

（2） 接收信号，调用 rtl8139_rx_interrupt

```
1.  if (status & (RxOK | RxUnderrun | RxOverflow | RxFIFOOver))
2.      rtl8139_rx_interrupt (dev, tp, ioaddr);
```

（3） 发送信号，调用 rtx8138_tx_interrupt

```
1.  if (status & (TxOK | TxErr)) {
2.    spin_lock (&tp->lock);
3.    rtl8139_tx_interrupt (dev, tp, ioaddr);
4.    spin_unlock (&tp->lock);
5.    }
```

在接收中断处理函数 rtl8139_rx_interrupt 中完成了四件事：（此函数是个大循环，只要接收缓存不为空就一直读取数据）

（1） 计算接受包的长度

```
1.  int ring_offset = cur_rx % RX_BUF_LEN;
2.    rx_status = le32_to_cpu (*(u32 *) (rx_ring + ring_offset));
3.    rx_size = rx_status >> 16;
```

（2） 分配数据结构

```
1.  skb = dev_alloc_skb (pkt_size + 2);
```

（3） 把数据从接收缓存拷到数据结构中

```
1.  eth_copy_and_sum (skb, &rx_ring[ring_offset + 4], pkt_size, 0);
```

（4） 把数据包发送到协议栈

```
1.  skb->protocol = eth_type_trans (skb, dev);
2.    netif_rx (skb);
```

至此完成了网卡驱动的框架梳理。

2.4. 总结

通过对 rtl8139 网卡驱动的代码分析我们可以以此总结 PCI 设备驱动的框架和流程，首先构建驱动对应的 pci_driver 结构体，并将此结构体注册到内核，挂载到 pci_bus 上；然后对驱动与设备通过设备号、厂商号等进行匹配，匹配成功后执行 probe 进行初始化，并将对应的 pci_dev 与 pci_driver 相关联；随后打开设备：注册设备驱动的中断函数分配好空间等；最后根据相应的驱动功能编写中断函数，这就是整个 PCI 设备驱动的流程。PCI 驱动还包含许多细节，如总线驱动对配置空间的访问，Linux 中断处理等，在本文中不一一叙述；只需把握 PCI 驱动的本质与框架即可，即 PCI 本质是一种总线，PCI 设备的驱动最终的实现仍是块设备驱动、网络设备驱动等。

参考资料

1、浅谈 Linux PCI 设备驱动（一）_linuxdrivers 的博客-CSDN 博客_linux pci 驱动

2、 Linux PCI 总线驱动-1_多云转晴，适合 debug 的博客-CSDN 博客_linux pcie 驱动分析

3、【原创】Linux PCI 驱动框架分析（一） - LoyenWang - 博客园 (cnblogs.com)

4、PCI 学习笔记-leonwang202-ChinaUnix 博客

5、Linux 源码阅读——PCI 总线驱动代码（三）PCI 设备枚举过程_咸鱼弟的博客-CSDN 博客_pci_read_config_word

6、linux 网络设备驱动（一）_wwwlyj123321 的博客-CSDN 博客_linux 网络设备驱动

7、DMA 之理解_书中倦客的博客-CSDN 博客_dma

8、Linux 中断之中断处理浅析_爱洋葱的博客-CSDN 博客_linux 中断