PCI 设备详解三

上篇文章已经分析了探测 PCI 总线的部分代码，碍于篇幅，这里另启一篇。重点分析下 **pci_scan_root_bus** 函数

2016-10-24

pci_scan_root_bus 函数

```
struct pci_bus *pci_scan_root_bus(struct device *parent, int
        struct pci_ops *ops, void *sysdata, struct list_head
{
    struct pci_host_bridge_window *window;
    bool found = false;
    struct pci_bus *b;
    int max;
    /*寻找bus的资源*/
    list_for_each_entry(window, resources, list)
        if (window->res->flags & IORESOURCE_BUS) {
            found = true;
            break;
        }
    /*创建bus对应的结构*/
    b = pci_create_root_bus(parent, bus, ops, sysdata, re
    if (!b)
        return NULL;

    if (!found) {
        dev_info(&b->dev,
         "No busn resource found for root bus, will use [bus
            bus);
        pci_bus_insert_busn_res(b, bus, 255);
    }
    /*遍历子总线*/
    max = pci_scan_child_bus(b);
```

```
    if (!found)
        pci_bus_update_busn_res_end(b, max);

    pci_bus_add_devices(b);
    return b;
}
```

这里首先寻找 bus 总线号资源，前面在 x86_pci_root_bus_resources 函数中已经分配了，所以这里理论上是已经分配好了，不过还是验证下！！内核中总是精益求精。接着调用了 pci_create_root_bus 函数创建了对应的 bus 结构，然后调用 pci_scan_child_bus 函数遍历该总线下所有的子总线。最后就调用 pci_bus_add_devices 添加设备。总体上就是这么几步，但是要弄清楚，还真是不小的工作量。我们一步步来：

1、pci_create_root_bus 函数

```
1 struct pci_bus *pci_create_root_bus(struct device *parent,
2          struct pci_ops *ops, void *sysdata, struct list_
3 {
4     int error;
5     struct pci_host_bridge *bridge;
6     struct pci_bus *b, *b2;
7     struct pci_host_bridge_window *window, *n;
8     struct resource *res;
9     resource_size_t offset;
10    char bus_addr[64];
11    char *fmt;
12    /*创建一个pci_bus结构*/
13    b = pci_alloc_bus();
14    if (!b)
```

```
15          return NULL;
16      /*基本的初始化*/
17      b->sysdata = sysdata;
18      b->ops = ops;
19      /*0号总线的总线号正是该条根总线下的总线号资源的起始号*/
20      b->number = b->busn_res.start = bus;
21      /**/
22      b2 = pci_find_bus(pci_domain_nr(b), bus);
23      if (b2) {
24          /* If we already got to this bus through a diffe
25          dev_dbg(&b2->dev, "bus already known\n");
26          goto err_out;
27      }
29      bridge = pci_alloc_host_bridge(b);
30      if (!bridge)
31          goto err_out;
33      bridge->dev.parent = parent;
34      bridge->dev.release = pci_release_host_bridge_dev;
35      dev_set_name(&bridge->dev, "pci%04x:%02x", pci_domai
36      error = pcibios_root_bridge_prepare(bridge);
37      if (error) {
38          kfree(bridge);
39          goto err_out;
40      }
41      /*桥也是作为一个设备存在*/
42      error = device_register(&bridge->dev);
43      if (error) {
44          put_device(&bridge->dev);
45          goto err_out;
46      }
47      /*建立总线到桥的指向*/
48      b->bridge = get_device(&bridge->dev);
49      device_enable_async_suspend(b->bridge);
50      pci_set_bus_of_node(b);
52      if (!parent)
53          set_dev_node(b->bridge, pcibus_to_node(b));
55      b->dev.class = &pcibus_class;
56      b->dev.parent = b->bridge;
57      dev_set_name(&b->dev, "%04x:%02x", pci_domain_nr(b),
58      error = device_register(&b->dev);
59      if (error)
60          goto class_dev_reg_err;
```

```
62      pcibios_add_bus(b);
64      /* Create legacy_io and legacy_mem files for this bus
65      pci_create_legacy_files(b);
67      if (parent)
68          dev_info(parent, "PCI host bridge to bus %s\n",
69      else
70          printk(KERN_INFO "PCI host bridge to bus %s\n",
72      /* Add initial resources to the bus */
73      list_for_each_entry_safe(window, n, resources, list)
74          /*从全局的资源链表摘下，加入到特定桥的windows链表中*/
75          list_move_tail(&window->list, &bridge->windows);
77          res = window->res;
78          offset = window->offset;
79          /*如果资源是总线号资源*/
80          if (res->flags & IORESOURCE_BUS)
81              pci_bus_insert_busn_res(b, bus, res->end);
82          else
83              pci_bus_add_resource(b, res, 0);
84          /*看总线地址到物理地址的偏移*/
85          if (offset) {
86              if (resource_type(res) == IORESOURCE_IO)
87                  fmt = " (bus address [%#06llx-%#06llx])"
88              else
89                  fmt = " (bus address [%#010llx-%#010llx]
90              snprintf(bus_addr, sizeof(bus_addr), fmt,
91                   (unsigned long long) (res->start - offs
92                   (unsigned long long) (res->end - offset
93          } else
94              bus_addr[0] = '\0';
95          dev_info(&b->dev, "root bus resource %pR%s\n", r
96      }
98      down_write(&pci_bus_sem);
99      /*加入根总线链表*/
100     list_add_tail(&b->node, &pci_root_buses);
101     up_write(&pci_bus_sem);
103     return b;
105 class_dev_reg_err:
106     put_device(&bridge->dev);
107     device_unregister(&bridge->dev);
108 err_out:
109     kfree(b);
```

```
110     return NULL;
111 }
```

该函数和之前的相比就略显庞大了。不过也难怪，到了最后的阶段一般都挺复杂。哈哈！这里调用 **pci_alloc_bus** 函数分配了一个 **pci_bus** 结构，然后做基本的初始化。注意一个就是

```
1 b->number = b->busn_res.start = bus;
```

总线号资源时预分配好的，且一个总线的总线号就是其对应总线号区间的起始号。

然后调用 **pci_find_bus** 检测下本次总线号是否已经存在对应的总线结构，如果存在，则表明有错误，当然一般是不会存在的。

然后调用 **pci_alloc_host_bridge** 函数分配了一个 **pci_host_bridge** 结构作为主桥。然后在主桥和总线之间建立关系。因为桥也是一种设备，所以需要注册。

所以一直到这里，代码虽然繁琐却不难理解。

到下面需要给总线分配资源了，之前我们是初始化了资源，并没有在总线和资源之间建立关系，需要分清楚。看下面的 **list_for_each_entry_safe**

这里实现的功能就是把 **window** 从 **resources** 链表中取下，然后加入到刚才创建 host-bridge 的 window 链表中，这样就算把资源分配给了主桥，回想下前面提到桥设

备的窗口就可以明白了。只是这里的意思貌似只考虑了一个主桥，虽然大部分都是一个主桥。然后把资源一个个资源都和总线相关联。这样总线的资源是有了。

最后调用 **list_add_tail** 把总线加入到全局的根总线链表。

下面看第二个函数 **pci_scan_child_bus**，总线的递归遍历就是在这里做的。

```
unsigned int pci_scan_child_bus(struct pci_bus *bus)
{
    unsigned int devfn, pass, max = bus->busn_res.start;
    struct pci_dev *dev;
    dev_dbg(&bus->dev, "scanning bus\n");
    /* Go find them, Rover! 遍历一条总线上的所有子总线，一条总线
    for (devfn = 0; devfn < 0x100; devfn += 8)
        /*在遍历每一个接口，这里一个接口最多有八个function*/
    /*在这里，就把总线上的每一个设备都探测过了并加入到了bus对应的设
        pci_scan_slot(bus, devfn);
    /* Reserve buses for SR-IOV capability. 加上预留的总线号
    max += pci_iov_bus_range(bus);
    /*
     * After performing arch-dependent fixup of the bus,
     * all PCI-to-PCI bridges on this bus.
     */
     /*查找PCI桥*/
    if (!bus->is_added) {
        dev_dbg(&bus->dev, "fixups for bus\n");
        pcibios_fixup_bus(bus);
        bus->is_added = 1;
    }
    /*据说是需要调用两次pci_scan_bridge，第一次配置，第二次遍历*/
    for (pass=0; pass < 2; pass++)
        list_for_each_entry(dev, &bus->devices, bus_list)
            if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE |
                dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
```

```
32                  /*遍历PCI桥*/
33                  max = pci_scan_bridge(bus, dev, max, pass
34          }
36      /*
37       * We've scanned the bus and so we know all about what
38       * the other side of any bridges that may be on this b
39       * any devices.
40       *
41       * Return how far we've got finding sub-buses.
42       */
43      dev_dbg(&bus->dev, "bus scan returning with max=%02x\r
44      return max;
45  }
```

这里做的工作也不难理解，先注意有个 max 变量，初始值是当前总线的总线号，表示已经探测的总线的数量，后续会用到。

一条总线上有 32 个插槽，而每一个插槽都可以包含八个功能即逻辑设备，所以这里以 8 递进。在循环中每次调用一下 pci_scan_slot 函数探测下具体的插槽。

```
1  int pci_scan_slot(struct pci_bus *bus, int devfn)
 2  {
 3      unsigned fn, nr = 0;
 4      struct pci_dev *dev;
 6      if (only_one_child(bus) && (devfn > 0))
 7          return 0; /* Already scanned the entire slot */
 8      /*遍历了第一个功能号，即fn=0*/
 9      dev = pci_scan_single_device(bus, devfn);
10      if (!dev)
11          return 0;
12      if (!dev->is_added)
```

```
13          nr++;
14      /*fn=1开始，遍历其他的功能*/
15      for (fn = next_fn(bus, dev, 0); fn > 0; fn = next_fn(b
16          dev = pci_scan_single_device(bus, devfn + fn);
17          if (dev) {
18              /**/
19              if (!dev->is_added)
20                  nr++;
21              /*如果找到第二个设备就说明这是个多功能的设备*/
22              dev->multifunction = 1;
23          }
24      }
26      /* only one slot has pcie device */
27      if (bus->self && nr)
28          pcie_aspm_init_link_state(bus->self);
30      return nr;
31  }
```

最先开始仍然是判断，如果这里该插槽只有一个逻辑设备即不是多功能的，且 devfn=0，那么就表示在寻找一个不存在的设备，直接 return 0，否则就调用 **pci_scan_single_device** 函数探测该插槽各个逻辑设备。接着调动了 **pci_scan_single_device** 函数，该函数检查下对应设备号的设备是否已经存在于总线的设备链表中，不存在才会往下调用 pci_scan_device 函数探测。

```
1  static struct pci_dev *pci_scan_device(struct pci_bus *bus,
 2  {
 3      struct pci_dev *dev;
 4      u32 l;
 5      /*获取设备厂商*/
 6      if (!pci_bus_read_dev_vendor_id(bus, devfn, &l, 60*100
```

```
 7          return NULL;
 8      /*分配一个dev结构*/
 9      dev = pci_alloc_dev(bus);
10      if (!dev)
11          return NULL;
13      dev->devfn = devfn;
14      dev->vendor = l & 0xffff;
15      dev->device = (l >> 16) & 0xffff;
17      pci_set_of_node(dev);
18      /*初始化设备*/
19      if (pci_setup_device(dev)) {
20          pci_bus_put(dev->bus);
21          kfree(dev);
22          return NULL;
23      }
25      return dev;
26  }
```

这里就要做实质性的工作了，创建了一个设备结构并设置相关的信息如设备号，厂商等，然后调用 **pci_setup_device** 函数对设备进行全面的初始化，比较重要是地址空间的映射。这里先不说这些，后面再提。最后会调用 **pci_device_add** 函数把设备注册进系统，主要还是在设备和总线之间建立联系。回到 **pci_scan_child_bus** 函数中，经过这一步就把当前总线上的各个逻辑设备遍历了一遍，也就是都凡是存在的逻辑设备都有了对应的结构，且都存在于总线的设备链表中。然后开始组个检测这些设备，其目的在于寻找 PCI-PCI 桥的存在也即 1 型设备。这里如果找到一个桥设备就会调用 **pci_scan_bridge** 函数遍历桥设备：

```
int pci_scan_bridge(struct pci_bus *bus, struct pci_dev *de
{
    struct pci_bus *child;
    int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CA
    u32 buses, i, j = 0;
    u16 bctl;
    u8 primary, secondary, subordinate;
    int broken = 0;
    /*这里是先读设备配置空间的总线号*/
    pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
    primary = buses & 0xFF;//父总线号
    secondary = (buses >> 8) & 0xFF;//子总线号
    subordinate = (buses >> 16) & 0xFF;//桥下最大的总线号
    dev_dbg(&dev->dev, "scanning [bus %02x-%02x] behind b
        secondary, subordinate, pass);
    /*!primary为真两种情况，1为空 2为0(代表根总线)，加上后面的&&
    if (!primary && (primary != bus->number) && secondary
        /*Primary bus硬件实现为0，当是root总线时，正好总线号也
        dev_warn(&dev->dev, "Primary bus is hard wired t
        /*手动设置*/
        primary = bus->number;
    }
    /* Check if setup is sensible at all 监测配置是否合法*/
    if (!pass &&
        (primary != bus->number || secondary <= bus->num
         secondary > subordinate)) {
        dev_info(&dev->dev, "bridge configuration invali
             secondary, subordinate);
        broken = 1;
    }
    /* Disable MasterAbortMode during probing to avoid re
       of bus errors (in some architectures) */
    pci_read_config_word(dev, PCI_BRIDGE_CONTROL, &bctl);
    pci_write_config_word(dev, PCI_BRIDGE_CONTROL,
                bctl & ~PCI_BRIDGE_CTL_MASTER_ABORT);
    if ((secondary || subordinate) && !pcibios_assign_all
        !is_cardbus && !broken) {
        unsigned int cmax;
        /*
         * Bus already configured by firmware, process i
         * pass and just note the configuration.
```

```
46           */
47          if (pass)
48              goto out;
50          /*
51           * If we already got to this bus through a diffe
52           * don't re-add it. This can happen with the i45
53           *
54           * However, we continue to descend down the hier
55           * scan remaining child buses.
56           */
57           /*得到子总线结构*/
58          child = pci_find_bus(pci_domain_nr(bus), seconda
59          if (!child) {
60              child = pci_add_new_bus(bus, dev, secondary)
61              if (!child)
62                  goto out;
63              /*设置子总线的primary指针*/
64              child->primary = primary;
65              /*给子总线也分配总线号资源*/
66              pci_bus_insert_busn_res(child, secondary, su
67              child->bridge_ctl = bctl;
68          }
69          /*递归遍历子总线*/
70          cmax = pci_scan_child_bus(child);
71          if (cmax > max)
72              max = cmax;
73          if (child->busn_res.end > max)
74              max = child->busn_res.end;
75      } else {
76          /*
77           * We need to assign a number to this bus which
78           * do in the second pass.
79           */
80          if (!pass) {
81              if (pcibios_assign_all_busses() || broken)
82                  /* Temporarily disable forwarding of the
83                     configuration cycles on all bridges i
84                     this bus segment to avoid possible
85                     conflicts in the second pass between
86                     bridges programmed with overlapping
87                     bus ranges. */
88                  pci_write_config_dword(dev, PCI_PRIMARY_
```

```
89                                  buses & ~0xffffff);
90              goto out;
91          }
93          /* Clear errors */
94          pci_write_config_word(dev, PCI_STATUS, 0xffff);
96          /* Prevent assigning a bus number that already e
97           * This can happen when a bridge is hot-plugged,
98           * this case we only re-scan this bus. */
99          child = pci_find_bus(pci_domain_nr(bus), max+1);
100         if (!child) {
101             child = pci_add_new_bus(bus, dev, ++max);
102             if (!child)
103                 goto out;
104             pci_bus_insert_busn_res(child, max, 0xff);
105         }
106         buses = (buses & 0xff000000)
107                | ((unsigned int)(child->primary)     <<
108                | ((unsigned int)(child->busn_res.start)
109                | ((unsigned int)(child->busn_res.end) <<
111         /*
112          * yenta.c forces a secondary latency timer of 1
113          * Copy that behaviour here.
114          */
115         if (is_cardbus) {
116             buses &= ~0xff000000;
117             buses |= CARDBUS_LATENCY_TIMER << 24;
118         }
120         /*
121          * We need to blast all three values with a sing
122          */
123         pci_write_config_dword(dev, PCI_PRIMARY_BUS, bus
125         if (!is_cardbus) {
126             child->bridge_ctl = bctl;
127             /*
128              * Adjust subordinate busnr in parent buses.
129              * We do this before scanning for children b
130              * some devices may not be detected if the b
131              * was lazy.
132              */
133              /*修正父总线的总线号资源范围*/
134             pci_fixup_parent_subordinate_busnr(child, ma
135             /* Now we can scan all subordinate buses...
```

```
136             max = pci_scan_child_bus(child);
137             /*
138              * now fix it up again since we have found
139              * the real value of max.
140              */
141             pci_fixup_parent_subordinate_busnr(child, ma
142         } else {
143             /*
144              * For CardBus bridges, we leave 4 bus numbe
145              * as cards with a PCI-to-PCI bridge can be
146              * inserted later.
147              */
148             for (i=0; i<CARDBUS_RESERVE_BUSNR; i++) {
149                 struct pci_bus *parent = bus;
150                 if (pci_find_bus(pci_domain_nr(bus),
151                                  max+i+1))
152                     break;
153                 while (parent->parent) {
154                     if ((!pcibios_assign_all_busses()) &
155                         (parent->busn_res.end > max) &&
156                         (parent->busn_res.end <= max+i)
157                         j = 1;
158                     }
159                     parent = parent->parent;
160                 }
161                 if (j) {
162                     /*
163                      * Often, there are two cardbus bri
164                      * -- try to leave one valid bus nu
165                      * for each one.
166                      */
167                     i /= 2;
168                     break;
169                 }
170             }
171             max += i;
172             pci_fixup_parent_subordinate_busnr(child, ma
173         }
174         /*
175          * Set the subordinate bus number to its real va
176          */
177         pci_bus_update_busn_res_end(child, max);
```

```
178          pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, r
179      }
181      sprintf(child->name,
182          (is_cardbus ? "PCI CardBus %04x:%02x" : "PCI Bus
183          pci_domain_nr(bus), child->number);
185      /* Has only triggered on CardBus, fixup is in yenta_s
186      while (bus->parent) {
187          if ((child->busn_res.end > bus->busn_res.end) ||
188              (child->number > bus->busn_res.end) ||
189              (child->number < bus->number) ||
190              (child->busn_res.end < bus->number)) {
191              dev_info(&child->dev, "%pR %s "
192                  "hidden behind%s bridge %s %pR\n",
193                  &child->busn_res,
194                  (bus->number > child->busn_res.end &&
195                   bus->busn_res.end < child->number) ?
196                      "wholly" : "partially",
197                  bus->self->transparent ? " transparent"
198                  dev_name(&bus->dev),
199                  &bus->busn_res);
200          }
201          bus = bus->parent;
202      }
204  out:
205      pci_write_config_word(dev, PCI_BRIDGE_CONTROL, bctl);
207      return max;
208  }
```

该函数通过递归的方式完成了所有总线以及设备的遍历。每一递归都执行两次该函数，第一次探测是否被 BIOS 处理，第二次才做真正的探测工作。

首先是先读取桥设备的配置空间，获得桥设备的 primary bus,secondary bus,subordinate bus 号，然后进行判断，如果 secondary bus 和 subordinate bus 均不为 0 则说明

配置有效，因为初始 **primary bus** 号被硬件初始化为 0，所以这里如果传递进来的 **bus number** 不是 0，就需要重新设置。

然后检查这些号码是否合法。合法情况下就在首次执行 **pci_scan_bridge** 函数的时候进行子总线的遍历。可以看到这里同样先是调用 **pci_find_bus** 函数查找下 secondary 号总线是否已经存在，不存在才调用 **pci_add_new_bus** 函数 new 一个新的 bus 结构，同时在该函数中也对总线的部分变量做了初始化。接着设置总线的 **primary** 指针。随后需要给总线分配总线号资源了。根据已有的配置，这里 secondary 是子总线的号，而 subordinate 就是总线下最大的总线号，所以这正是总线的总线号区间。然后继续调用 **pci_scan_child_bus** 函数继续遍历当前子总线。就这么层一层的递归下去。知道最后没有桥了，就从 pci_scan_child_bus 函数返回探测到的总线的数量即 max. 而如果配置空间没有被配置，那么就需要重新配置，这里首次执行 **pci_scan_bridge** 函数就只是把配置空间总线号区域清零。到了第二次，大题上根前面类似，不过这里因为没有 secondary 号，所以只能按照 max+1 来寻找或者创建子总线结构，同时对于子总线的总线区间设置成 0xff 即 255 最大值。然后写入到桥配置空间中。这个时候已经探测了一个新的总线，那么需要对父总线的总线号区间进行更新，然后执行 pci_scan_child_bus 函数探测当前子总线的其他总线，在递归返回的时候，需要再次执行更新。并且需要把总线的总线号资源设置成正确的区间。因为开始分配的时候设置默认总线号区间最大为 255.

整个递归流程完毕，就知道了一共存在多少总线，且总线上的设备都已经正确配置并都已经加入到了设备链表中。

总结：

本次分析可谓是困难重重，对于很多大牛来说，这或许根本不是事，但是笔者平时的研究没哟涉及到 PCI 设备这一层面，仅仅是为了分析 qemu 中的 virtIO 才着手分析 PCI 设备。其中可能不乏错误之处，还望老师们看到多多指正。笔者也正是发现只记录不分享，久而久之就越发懒散，好的东西信手沾来虽然容易，然是后续基本不会再看。而写下来给别人分享就不同了，因为担心写错，好多模糊的地方自己需要再三斟酌，同时也是对自己基础的强化，利人利己！

全文完