Linux那些事儿

系列丛书

之

我是PCI

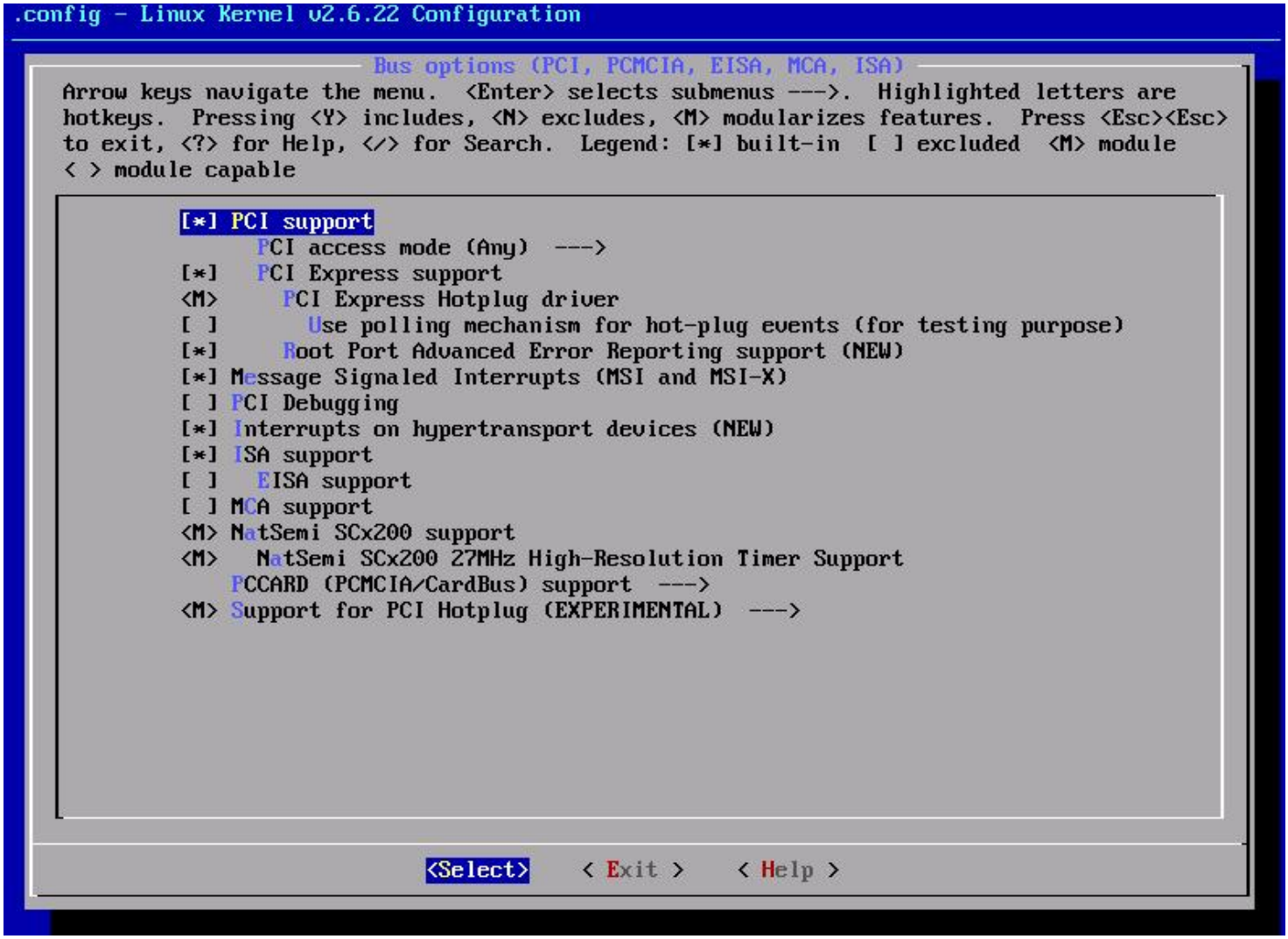
[1] 原文为blog.csdn.net/fudan_abc 上的《linux 那些事儿之我是PCI》，有闲情逸致的或者有批评建议的可以到上面做客，也可以email 到ilttv.cn@gmail.com

目录

引子

老夫子们痛心疾首的总结说，现代青年的写照是——自负太高，反对太多，商议太久，行动太迟，后悔太早。上天戏弄，俺不幸的混进了 80 后的革命队伍里，成了一名现代青年，前有老夫子的忧心忡忡，后有 90 后的轻蔑嘲弄，终日在迷失与老土这样的两极词汇里徘徊。为了说明俺也是有主义有信仰的，也是经历过楼市股市狂风暴雨考验的，这里就讲讲 PCI，而且不再做过多的罗唆铺垫，直接开门见山，让他们看看 80 后不仅仅知道什么是网恋什么是异性同居，知道怎么靠上半身上位怎么用下半身写作，还知道什么叫 PCI，什么叫雷厉风行。

Linux 里的 PCI 是一个系统工程。现在各行各业都爱和工程这个词儿沾个边儿，勤工助学是爱心工程，交通治理是畅通工程，只要加上个“工程”，那股热火朝天无私奉献的劲儿就出来了，不过很让人惭愧的是，提到工程俺首先反应出来的还是豆腐渣工程，实在是给 80 后摸黑。PCI 这个系统工程当然不是豆腐渣工程，否则不可能结实牢固的运转这么多年，我相信与它一番亲密接触之后，就会和国家“畅通工程”专家组组长、东南大学交通学院院长王炜教授的研究成果“中国城市环境污染不是由汽车造成的，而是由自行车造成的。自行车的污染比汽车更大。”一样，收获会是巨大的、空前绝后的。为了高屋建瓴的审视一下 PCI 这个系统工程，咱们先去看看使用 make menuconfig 配置内核时显示的那个界面



很严肃的告诉你，这里所说的工程所依托的内核版本仍然是 2.6.22，这话我只在这儿告诉你一次，好话不会再说第二遍。为什么这里要采用与 USB 那块儿不同的手法，从一张图片儿开始咱们的故事？作为一个勤于分析善于思考的 80 后，俺经过复杂的推理和多方面的论证，得出了这么一个结论：图片要比文字形象的多，要更得人心一些。不然为啥你看到有关石靖的文字报道和评论后还不满足，非要漫天的去搜索她的那些图片儿那？为啥你看到那个越南美女黄垂玲的消息后，就非要到处找那几段儿视频小录像那？

PCI，我们来了

现在这段时间最火的工程是什么？当然不会是 PCI 这个系统工程了，你即使不是党员也总归是个中国人，是中国人都要毫不犹豫的回答“探月工程”。不过，如果你在两年前就这么问我的话，俺会面带羞涩的回答你，是“中国芯工程”，谁让汉芯偏偏就是俺们交大的那，谁让汉芯又偏偏是假的那，俺无法回答你，俺陈进手下的哥们儿也无法回答你。到现在俺还依稀记得，在 2002 年的那个秋天，俺刚到交大就遇到陈进时的情景，旁边儿一见多识广的哥们儿指着一个瘦高瘦高文质彬彬的帅哥说：“看，那小伙儿就是陈进，搞汉芯的，博导，大牛！”俺无比激动的望过去，差点崇拜的华丽丽的摔倒，就没诚想到了 05 年就什么都成假的了，这要放到宪哥的节目里，肯定是最难的一期真的假不了。不过搞计算机的再怎么敢玩儿虚的，

搞航空航天的都不敢这么玩儿，所以说一会儿航天飞机一会儿探月工程，和那个中国芯工程比起来是冰火两重天，特提神儿。

因此，现在俺决定，下次望见月亮时，俺要自豪的在心里默默的呼喊一下：月亮，我们来了！作为热身，借这地儿俺也要呼喊一下：PCI，我们来了！虽说俺都不记得上次望见月亮是哪年哪月的事儿了，但是 PCI 总归是就在眼前的。

还是看看前面的那张图，第一项，“PCI support”，只有选上它，咱们的故事才能够继续得下去，所以说为公为私，为你为我，这一项都必须得选上。

选上了第一项，才有第二项“PCI access mode”的存在，它是一个单项选择题，有四个答案，分别是 BIOS、MMConfig、Direct、Any，记不记得青春年少的时候，考试时涂的那些答题卡？亲爱的老师们改的时候一般都是拿个挖了很多洞洞的纸片儿往答题卡上套，这样一目了然就知道你哪些答对了哪些答错了，有些天资聪慧的同学很容易的就想出了个对策，在不怎么肯定的题目里将模棱两可的答案都给涂上，这样老师们拿纸片儿往上套的时候就总会是对的，不过可惜的是高考不是这么个改法，也就不能这么搞。这里“PCI access mode”的四个选项也不能这么搞，你是只能选上一个的，其实你也不用这么搞，Any 选项本身就表示了你告诉内核说前面三个答案随便一个都可以，你只要选上了它，就相当于所有的四个答案都给选上了，可惜啊可惜，为啥咱们考试时咋就没有这么一个选项捏？

这么一说，你就明白了，即使咱们什么都不懂，随便蒙上一个 Any 也错不了，咱们都是挤了多少次独木桥挤过来的，这点技巧还是有的，但是现在不是考试，所以咱们不但要会答题还要明白题目的意思。祭出咱们早已很娴熟的猜题大法，可以很快的就从名字里就猜到 PCI access mode 是说 PCI 的访问方式的，访问什么？当然是访问 PCI 设备，那么几个答案的意思也就是，你可以通过 BIOS 去访问，也可以抛开 BIOS，直接（Direct）去访问，或者通过一个叫 MMConfig 的东东去访问，Any 就是表示你如果拿不准的话可以让内核去选择一种访问方式，当然，内核一向都是很严谨刻板的，绝不会真的抓阄儿抓到哪个就使用哪种方式，它会按照一定的优先级，首先尝试 MMConfig，然后是 Direct，如果这两种方式都不起效，最后再使用 BIOS。

那现在你可能会感到疑惑的是，这里所谓的 access 到底是怎么回事，为啥还搞出这么多名堂来。说起来话长，俺在讲 usb core 的时候就提到了，每个 PCI 设备都有那么一张表，就是传说中的配置寄存器，有了它，写 PCI 驱动的身体甭儿好，吃饭甭儿香，中断号什么的也不用再去申请了，直接从表里拿出来就是，这就少了很多冗长的步骤手续和推诿扯皮，省事又省心。关于这张表，现在还不是详细去讲它的最佳时机，咱们虽然是在开门见山，但视力范围毕竟有限还看不了那么远。目前来说，俺只想问的一个问题是，它的内容都从哪儿来的？其中一部分，包括设备 ID，厂商 ID 等等自然都是厂商固化在设备里边儿的，它们的内容咱们不需要去改变也改变不了，而还有一部分则是需要咱们的内核去酌情修改设置的，具体在什么时候去设置，就牵涉到了另外一个无比重要的概念——总线枚举。

随着剧情的深入发展，人物总是会越来越多，为了今后能够更好的理清他们之间的关系，在讲总线枚举之前有必要先窥探一下 PCI 系统的全貌。引用一下 LDD3 里的一张图

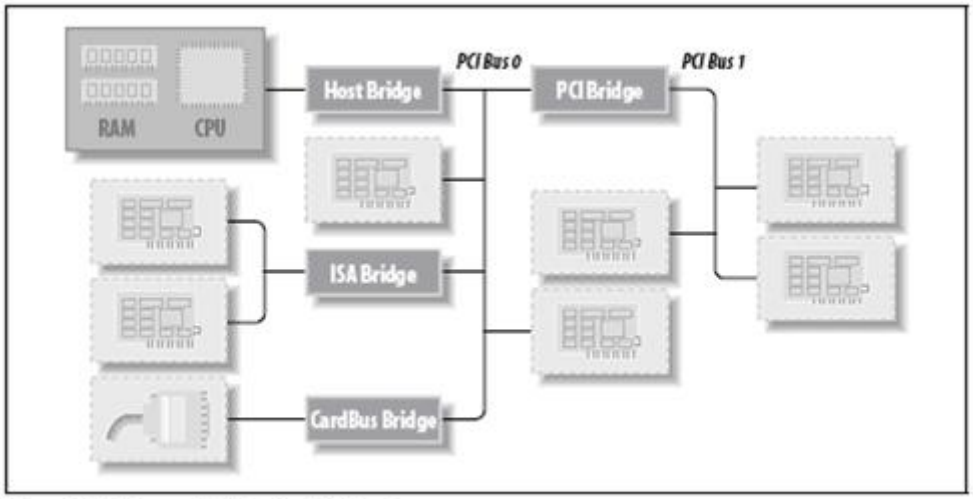


Figure 12-1. Layout of a typical PCI system

Greg 大侠没那么媚俗，对这种小图片儿提不起兴趣，在 LDD3 里也就将它给一笔带过了，俺不同，本就是大俗人一个，所以这里替他展开来说一下。其实这张图是非常偷工减料的，并不足以揭示整个 PCI 系统的细节，但说明问题还是足够了。里面的几个 Bridge，也就是桥，是用来将多个 PCI 总线，或者 PCI 总线与 ISA 等其它总线连接起来的东东。各种桥里比较特殊的一个是 Host Bridge，它连接的是 CPU 和 PCI 总线 0，说 Host Bridge 你可能丈二摸不着头脑，但是说北桥你就明白了，不然你都不好意思说你会玩儿电脑，更别说会玩儿 linux 了，有北桥自然就有南桥，大名鼎鼎的南北桥谁都知道，就像在江湖上行走的都得知道南北少林一样。而南北桥合起来就叫做芯片组，芯片组里距 CPU 最近的一个就是北桥，它是位于天子脚下呼风唤雨的重臣。北桥里通常还集成着内存控制器，所以 CPU 和内存之间的交流也要依靠它来完成，如果是那种集成显卡的板子，显示芯片也要呆在它里边儿。CPU 和北桥之间的连接靠的是 FSB（前端总线——Front Side Bus），就是俗称前端总线的那个，FSB 的速度即是咱们通常所说的外频，外频的高低直接影响了 CPU 对内存的存取。基于北桥的这种显赫的江湖地位，江湖里一般尊称它为 Host Bridge，也就是主桥。实际上，芯片组的名称就是以北桥的名称来命名的，像 intel 875P 芯片组的北桥芯片是 82875P 等等。那么上边儿图里的哪个桥对应着南桥那？左瞅瞅右看看，还就 ISA Bridge 有点像，一般来说，PCI-ISA 桥就称为南桥，但它并不是仅仅是为了连接 PCI 总线和 ISA 总线这么简单，它里边儿会集成很多东东，比如中断控制器、IDE 控制器、DMA 控制器等，咱们在 USB 那块大书特书的 USB 控制器一般也集成在它里边儿。

明白了 Host Bridge，再看看 PCI bus0，也就是 primary PCI bus，主 PCI 总线。至于有没有 PCI bus1，PCI bus2 等等俺不好说，但是只要你的系统里有 PCI 总线，这个 PCI bus0 总是得存在的。一个系统自然是可以拥有多个 PCI 总线的，这多个 PCI 总线之间的连接要靠另外一种桥，PCI-PCI 桥。通过 PCI-PCI 桥，整个 PCI 系统就构成了一个层次的、树状的结构，类似的树咱们在讲 USB Core 时就遇到过了，不过不同的是 USB 那棵大树是靠 HUB 搭建的，而这里的 PCI 树是靠 PCI-PCI 桥搭建的，日后再有人问你“这是什么树？”，你就不能仅仅回答“大树”了，要拍拍胸脯慷慨激昂的回答“USB 树”或者“PCI 树”。就像 HUB 同时也是 USB 设备一样，PCI-PCI 桥、CardBus 桥等等这些五花八门的桥也都是 PCI 设备，当然，不用我说你也知道，PCI 设备不仅仅是指这些桥。每个 PCI 总线上都可以支持 32 个 PCI 设备，每个设备又可以支持 8 种功能，不要被设备和功能这两个词儿给绕晕了，这里所谓的设备一般指那种 PCI 接口卡，更确切的说是指 PCI 总线的接口芯片，每块 PCI 接口卡上可以有若干个功能模块，它们共用了同一个 PCI 接口芯片，这样可以降低成本，满足节约型社

会的要求。从逻辑的角度说，每个功能模块都是一个逻辑设备，既然都说是逻辑了，自然就只能意会不能言传了。江湖上将只拥有一个功能模块（逻辑设备）的设备称为单功能设备，拥有多个的就称为多功能设备，当然再多不能多过 8 个。

这么多的 PCI 设备，不管它是桥，还是其它的，都必须得有那张表示配置寄存器的表，表当然是要拿来用的，要想将它给拿出来，就得去访问设备，可事实是系统一开始只有那条 PCI bus0 能够访问，其它的 PCI 总线和设备一切都还是未知，那么如何才能使未知变成已知？这就又回归到前面的问题“总线枚举”了。系统引导时，对于 PCI 子系统来说，会首先有个总线枚举的阶段，从 PCI bus0 开始扫描，遇到一个设备是 PCI-PCI 桥，就指定一个新的总线号，比如 1，这样 PCI bus1 就有了，也就可以访问了，如果遇到一个其它 PCI 设备，也要将它记录在案，直到将所有的 PCI-PCI 桥和 PCI 设备给晾出来，组成一棵 PCI 树，不过现在这个树是软件意义上的，前面刚提到的那棵 PCI 树是硬件意义上的。

经过这么一个漫长曲折的总线枚举过程，PCI 树就从只有 PCI bus0 这么一个小牙儿长成了枝枝丫丫的参天大树，树里每个设备的那张配置寄存器的内容也明明白白的晾在那里了，而且，里面的一些内容，比如中断线什么的也给酌情设置好了，该映射的地址也都给映射了，总之，往白了说，总线枚举的过程就是内核里的 PCI 树成长的过程。大树底下好乘凉，有了这么一棵大树，写 PCI 驱动的才能吃的香睡的沉。

已经绕了大老远的了，也该绕回去了，这个总线枚举和 PCI access mode 又有嘛关系？这还要再说说 PCI BIOS。基于 PCI 总线在整个计算机世界里特殊的江湖地位，BIOS 中也专门提供了针对 PCI 总线的操作，这些操作里就包括了总线枚举的整个过程。在系统加电以后自检时，就会完成对 PCI 总线的枚举，之后对 PCI 设备的访问就都是通过 BIOS 调用的形式进行，提供有这些功能和服务的 BIOS 就称之为 PCI BIOS。其实这就是 PCI access mode 里的那个 BIOS 选项所表达的意思，但是，一些旧的主板上，BIOS 并不买 PCI 的账，不支持这么做，还有一些嵌入式系统里甚至于根本就没有 BIOS 的存在，为了适应各种革命形式的需要，linux 就自己实现了包括总线枚举在内的一整套 PCI 总线操作，而不再去依赖 BIOS，这就是那个 Direct 选项的由来。当然在 64 位的平台上，是没有什么 PCI BIOS 的，采用的总是 Direct 方式，你使用 make menuconfig 配置内核的时候也就根本看不到有 PCI access mode 这么一项给你选。提到这里，顺便说一下，俺的机子还不是 64 位的，所以最开始使用 make menuconfig 得到的那张图，还有以后会讲到的与架构有关的代码都是针对 32 位平台的，但是里面的道道儿都是一样的。

至于 MMConfig 方式，是 PCI Express 才用得上的。关于 PCI Express，俺这里的原则是，会提到，但不会过多的去关注它。不过这个 MMConfig 因为与 Direct 有着千丝万缕的关系，俺会在日后遇到合适的机会时去详细的说一下，这里暂时留下个悬念吧。

回到开篇的那张图，第三项是“PCI Express support”，它还有它下边儿紧接着的三项都是有关 PCI Express 的，选不选看你了，它们的细节就飘过了，但是 PCI 的进化历程还是有必要了解一下的。先看一看从 PCI 的老巢 PCI SIG 的主页上抓下来的一张图

这张图够婀娜多姿的，看来 PCI SIG 的那帮家伙和俺一样的媚俗，不过它到是很能形象的表示从 PCI 到 PCI-X 再到 PCI Express 这么一个 PCI 的发展历程，至于中间的那个 IOV，即 I/O VIRTUALIZATION，也就是江湖人称的那个 I/O 虚拟化技术，离咱们的主题差个十万八千里，我不用说，你也不用问，让它随风而去吧。至于这里所谓的 PCI，往狭义里讲就是最初的传统的 PCI 规范，每种事物都有它存在的目的和意义，PCI 就是为了替代 ISA 而生的，上世纪 90 年代初，在咱们这些人还在情窦初开暗恋小女生的时候，PCI 一出现就统一了 VESA、ISA 等当时并存的多种 I/O 总线，来势不可谓不凶，它能够得到江湖老大的地位，当然有它的独到之处，它使用了比 ISA 更高的时钟频率，可以达到 33MHz 或 66MHz，优势是显而易见的。但是随着历史车轮的向前推进，随着咱们从一个嘴上没毛的小屁孩儿长成一个嘴上有毛的小混混，仅仅暗恋已经不能满足内心的需求，这样的时钟频率也已经不能满足某些应用的需要，于是就衍生出来了 PCI-X 和 AGP 等这样的东东，但是 PCI-X 带来的复杂度和高成本并不符合节约型经济型社会的要求，所以 PCI Express 就顺应时代潮流，登上了历史舞台，至于 PCI Express 是靠什么上位的，八卦记者才会去关心这个，俺这里就飘过了。

接下来就是第四项，“Message Signaled Interrupts (MSI and MSI-X)”，关于 MSI 中断的。到了 21 世纪，什么都在爆炸，知识在爆炸，连满大街看到的发型都是爆炸式的，谁让咱们苦命那，反抗不了命运就只有接受这些无穷尽的名词儿吧。了解 MSI 之前，先普及一下中断。狭义的说，内核就是用来管理各种设备的，要管理就得需要交流，就得明白点儿管理的艺术掌握点儿管理的技巧，这个技巧有两种，一是轮询，定期的去探访设备，体恤设备的民情，看有没有什么需要处理的，就像逢年过节时要给低保户孤寡老人送温暖，不逢年过节时要给另外某些人送温暖一样。二是中断，设备需要的时候主动向内核发信号，通知内核说自己有需要了，快来满足我。对内核来说，这两种方式两种技巧，哪一种为优哪一种为劣？很明显，你说那么多孤苦伶仃的人要一个个定时定期的去嘘寒问暖，公仆们累不累，公仆也是人啊，而且这样也够低效，每次只能解决暂时的问题，局部的问题，同时也占用了公仆们大量的时间。而第二种方式就不一样了，哪个设备有需要了向上报告一声，内核就会明白设备的心声就会安排酌情处理。当然，现实是残酷的，形式是复杂的，公仆们可能更多的采用第一种方

式，第二种方式使用起来就不是那么高效了，不过我们要理解，第一种方式已经耗费了他们大量的时间和精力，他们也是血肉之躯，嗯，我们要理解。

上边儿是从世俗的角度说的，现在再从物理的角度上看一看，中断就是一种电信号，设备在希望获得处理器关注的时候就可以发送这么一个信号，并直接送入中断控制器（如 8259A）的输入引脚上，然后再由中断控制器向处理器发送相应的信号。处理器一经检测到该信号，便中断自己手头儿的工作，转而去处理中断。此后，处理器会通知内核已经产生中断了，这样，内核就可以对这个中断进行适当的处理。内核要处理一个中断，也不是说只要这个中断产生就可以了，还需要一个对应的中断处理函数，不然就会像你的很多呼声一样石沉大海了。这个所谓的中断处理函数是需要提前注册好放在内核里边儿的，和特定的中断信号线绑定在一起，中断信号线就是标识每个中断的一个值，是比较稀缺的资源，属于重点保护对象。那么究竟稀缺到什么程度？这要看使用的是什么中断控制器。

俺只说 X86 上面儿的，常见的中断控制器一个是 PIC，一个是 APIC，PIC 就是 Programmable Interrupt Controller，可编程中断控制器的意思，那 APIC 就是高级可编程中断控制器，这种文字游戏咱们见得多了。PIC 是由两片 8259A 级联在一起组成的，每个可以处理 8 个不同的 IRQ，两个加一块儿 8 加 8 等于几？答 16 就错了，是 15 个，因为级联时还要去掉用于连接的那个。这么多设备嗷嗷待哺的总共才有 15 个中断请求线，你说稀缺不稀缺？

APIC 就要好一点儿了，不然怎么敢称高级。PIC 只能用于一个处理器的系统，在多处理器系统上，处理器之间也可能会产生中断，这种情况 PIC 就不能应对了。很好理解，谁说公仆们互相之间就不能有需要了？不然那句官官相护是怎么得来的？而 APIC 就能够用在多处理器的系统上，解决多个处理器之间的互相需要，同时它支持的中断请求线的数目也有了很大幅度的提高，可以达到 255 个，不过，即使这样，中断请求线仍然还是属于稀缺的需要保护的资源。当然，单处理器系统也是可以使用 APIC 的了，这个时候它虽然没有了处理器之间互相需要的情况，但对中断请求线总是多多益善的。不过，这样就产生了一个兼容性的问题，就像经济的发展离不开房地产，计算机的发展也离不开兼容，APIC 就可以兼容 PIC，按照标准的 8259A 的工作方式来工作，这要归咎于 APIC 的组织结构。

对于 APIC 来说，能够将中断传递给系统中的每个 CPU 至关重要，所以它通常都要包含两个部分，Local APIC 和 I/O APIC。系统中的每个 CPU 都会有一个 Local APIC（那种超线程的 CPU 是不止包括一个 Local APIC 的），负责将中断信号传递到指定的处理器。而 I/O APIC 则负责收集中断信号并将它们转发给各个 Local APIC，它也可以有多个。APIC 就是这么一个多级的体系结构，如果要模拟 8259A，只需要将 Local APIC 给禁止掉，再进行适当的配置，I/O APIC 就可以按照 PIC 的工作方式工作了。你可以通过查看/proc/interrupts 文件来获悉系统有没有在使用 I/O APIC，

```
localhost:/usr/src/linux/ # cat /proc/interrupts
                    CPU0
  0:    20565961      IO-APIC-edge    timer
  1:        3763      IO-APIC-edge    i8042
  6:           4      IO-APIC-edge    floppy
  7:           0      IO-APIC-edge    parport0
  8:           1      IO-APIC-edge    rtc
  9:           0      IO-APIC-level   acpi
 12:        3095      IO-APIC-edge    i8042
 15:      737432      IO-APIC-edge    ide1
169:       11066      IO-APIC-level   eth0
177:           0      IO-APIC-level   uhci_hcd:usb1, es1371
```

185: 19441 IO-APIC-level ioc0
NMI: 0
LOC: 19332194
ERR: 0
MIS: 0
如果你在上面的结果里瞅见了 IO-APIC 这样的字眼，就说明你的系统正在使用 APIC。
社会总是要向前发展的，咱们经济发展的支柱是房地产，同时还会有其它杂七杂八的柱子，北大经济学院的院长刘伟就说了：“我把堵车看成是一个城市繁荣的标志，是一件值得欣喜的事情。如果一个城市没有堵车，那它的经济也可能凋零衰败。1998 年特大水灾刺激了需求，拉动增长，光水毁房屋就几百万间，所以水灾拉动中国经济增长 1.35%。”那技术发展的支柱是什么？这个可以往现实的地方想也可以往高尚的地方想，反正总归不会是水灾了，不过不管它是什么，都不影响 PCI Spec v2.2 开始提出一种全新的中断方式，也就是前面看到的 Message Signaled Interrupts，翻成中文咋看咋别扭，所以就原汁原味儿的使用英文名称了，还好它有个简短的简称 MSI。MSI 通过向一个预定义的内存地址写入一个已经预定义好的 Message 来提出中断请求，这个 Message 到达主桥时，主桥会将它转换为具体的中断，发送到处理器。对 PCI 设备来说，这就消除了对中断引脚电路的需要，但是 PCI Spec 里还是要求支持 MSI 的设备最好同时也要具有中断引脚。另外再友情提醒一下，你如果选上 MSI，还得同时选上 CONFIG_X86_LOCAL_APIC 才能正常工作，如果你的 CPU 的 datasheet 上已经写明了没有实现 APIC，那你也就不用再操心 MSI 了。至于 MSI-X 则是 MSI 的增强型，不予关心，飘过。
在 MSI 之后，紧接着的一项是“PCI Debugging”，调试用的。
下面一项，“Interrupts on hypertransport devices”，hypertransport 是 AMD 在 99 年提出的一种总线技术，细节就不说了，飘过。
接下来与咱们讲的主题有关的就是最后一项的那个 PCI Hotplug 支持，道儿上混的都知道热插拔是嘛意思，那么 PCI 热插拔的意思也就是明摆着的。

加菲猫说过，每个成功男人的背后，都有一个女人，每个不成功男人的背后，都有两个，每个 menuconfig 的背后都有无数个 Kconfig，按照习俗，开篇那张图背后的那些个 Kconfig 应该到 drivers/pci 目录下去找，

```
#
# PCI configuration
#
config ARCH_SUPPORTS_MSI
        bool
        default n

config PCI_MSI
        bool "Message Signaled Interrupts (MSI and MSI-X)"
         depends on PCI
         depends on ARCH_SUPPORTS_MSI
         help
            This allows device drivers to enable MSI (Message Signaled
            Interrupts).  Message Signaled Interrupts enable a device to
            generate an interrupt using an inbound Memory Write on its
```

```
16              PCI bus instead of asserting a device IRQ pin.
17
18              Use of PCI MSI interrupts can be disabled at kernel boot time
19              by using the 'pci=nomsi' option.    This disables MSI for the
20              entire system.
21
22              If you don't know what to do here, say N.
23
24 config PCI_DEBUG
25          bool "PCI Debugging"
26          depends on PCI && DEBUG_KERNEL
27          help
28            Say Y here if you want the PCI core to produce a bunch of debug
29            messages to the system log.    Select this if you are having a
30            problem with PCI support and want to see more of what is going on.
31
32            When in doubt, say N.
33
34 config HT_IRQ
35          bool "Interrupts on hypertransport devices"
36          default y
37          depends on PCI && X86_LOCAL_APIC && X86_IO_APIC
38          help
39              This allows native hypertransport devices to use interrupts.
40
41              If unsure say Y.
```

这么短短数十行就是 drivers/pci/Kconfig 文件的全部内容，它只包括了寥寥几项，远远不能涵盖那张图里的内容，像 PCI access mode 等项都没在这里边儿提到。咋办？到 arch/i386/Kconfig 文件里找，当然，如果你的平台不是 X86 32 位的，就要到 arch 目录下对应的子目录里找了。下面就看下它里边儿与前面所讲有关系的那些内容

```
1058 menu "Bus options (PCI, PCMCIA, EISA, MCA, ISA)"
1059
1060 config PCI
1061          bool "PCI support" if !X86_VISWS
1062          depends on !X86_VOYAGER
1063          default y if X86_VISWS
1064          select ARCH_SUPPORTS_MSI if (X86_LOCAL_APIC && X86_IO_APIC)
1065          help
1066            Find out whether you have a PCI motherboard. PCI is the name of a
1067            bus system, i.e. the way the CPU talks to the other stuff inside
1068            your box. Other bus systems are ISA, EISA, MicroChannel (MCA) or
1069            VESA. If you have PCI, say Y, otherwise N.
1070
1071            The PCI-HOWTO, available from
```

```
1072                <http://www.tldp.org/docs.html#howto>, contains valuable
1073                information about which PCI hardware does work under Linux and which
1074                doesn't.
1075
1076 choice
1077                prompt "PCI access mode"
1078                depends on PCI && !X86_VISWS
1079                default PCI_GOANY
1080                ---help---
1081                  On PCI systems, the BIOS can be used to detect the PCI devices and
1082                  determine their configuration. However, some old PCI motherboards
1083                  have BIOS bugs and may crash if this is done. Also, some embedded
1084                  PCI-based systems don't have any BIOS at all. Linux can also try to
1085                  detect the PCI hardware directly without using the BIOS.
1086
1087                  With this option, you can specify how Linux should detect the
1088                  PCI devices. If you choose "BIOS", the BIOS will be used,
1089                  if you choose "Direct", the BIOS won't be used, and if you
1090                  choose "MMConfig", then PCI Express MMCONFIG will be used.
1091                  If you choose "Any", the kernel will try MMCONFIG, then the
1092                  direct access method and falls back to the BIOS if that doesn't
1093                  work. If unsure, go with the default, which is "Any".
1094
1095 config PCI_GOBIOS
1096                bool "BIOS"
1097
1098 config PCI_GOMMCONFIG
1099                bool "MMConfig"
1100
1101 config PCI_GODIRECT
1102                bool "Direct"
1103
1104 config PCI_GOANY
1105                bool "Any"
1106
1107 endchoice
1108
1109 config PCI_BIOS
1110                bool
1111                depends on !X86_VISWS && PCI && (PCI_GOBIOS || PCI_GOANY)
1112                default y
1113
1114 config PCI_DIRECT
1115                bool
```

```
1116            depends on PCI && ((PCI_GODIRECT || PCI_GOANY) || X86_VISWS)
1117            default y
1118
1119 config PCI_MMCONFIG
1120            bool
1121            depends on PCI && ACPI && (PCI_GOMMCONFIG || PCI_GOANY)
1122            default y
1123
1124 source "drivers/pci/pcie/Kconfig"
1125
1126 source "drivers/pci/Kconfig"
1127
1128 config ISA_DMA_API
1129            bool
1130            default y
1131
1132 config ISA
1133            bool "ISA support"
1134            depends on !(X86_VOYAGER || X86_VISWS)
1135            help
1136              Find out whether you have ISA slots on your motherboard.  ISA is the
1137              name of a bus system, i.e. the way the CPU talks to the other stuff
1138              inside your box.  Other bus systems are PCI, EISA, MicroChannel
1139              (MCA) or VESA.  ISA is an older system, now being displaced by PCI;
1140             newer boards don't support it.  If you have ISA, say Y, otherwise N.
1141
1142 config EISA
1143            bool "EISA support"
1144            depends on ISA
1145            ---help---
1146              The Extended Industry Standard Architecture (EISA) bus was
1147              developed as an open alternative to the IBM MicroChannel bus.
1148
1149              The EISA bus provided some of the features of the IBM MicroChannel
1150              bus while maintaining backward compatibility with cards made for
1151              the older ISA bus.  The EISA bus saw limited use between 1988 and
1152              1995 when it was made obsolete by the PCI bus.
1153
1154              Say Y here if you are building a kernel for an EISA-based machine.
1155
1156              Otherwise, say N.
1157
1158 source "drivers/eisa/Kconfig"
1159
```

```
1160 config MCA
1161         bool "MCA support" if !(X86_VISWS || X86_VOYAGER)
1162         default y if X86_VOYAGER
1163         help
1164           MicroChannel Architecture is found in some IBM PS/2 machines and
1165           laptops.  It is a bus system similar to PCI or ISA. See
1166           <file:Documentation/mca.txt> (and especially the web page given
1167           there) before attempting to build an MCA bus kernel.
1168
1169 source "drivers/mca/Kconfig"
1170
1171 config SCx200
1172         tristate "NatSemi SCx200 support"
1173         depends on !X86_VOYAGER
1174         help
1175           This provides basic support for National Semiconductor's
1176           (now AMD's) Geode processors.  The driver probes for the
1177           PCI-IDs of several on-chip devices, so its a good dependency
1178           for other scx200_* drivers.
1179
1180           If compiled as a module, the driver is named scx200.
1181
1182 config SCx200HR_TIMER
1183         tristate "NatSemi SCx200 27MHz High-Resolution Timer Support"
1184         depends on SCx200 && GENERIC_TIME
1185         default y
1186         help
1187           This driver provides a clocksource built upon the on-chip
1188           27MHz high-resolution timer.  Its also a workaround for
1189           NSC Geode SC-1100's buggy TSC, which loses time when the
1190           processor goes idle (as is done by the scheduler).  The
1191           other workaround is idle=poll boot option.
1192
1193 config K8_NB
1194         def_bool y
1195         depends on AGP_AMD64
1196
1197 source "drivers/pcmcia/Kconfig"
1198
1199 source "drivers/pci/hotplug/Kconfig"
1200
1201 endmenu
```

Kconfig 的语法在 Documentation/kbuild/kconfig-language.txt 里，挺直白挺简单，也挺有意思的，1058 行的 menu 就表示生成一个下级菜单，这个菜单的内容直到 1201 行的 endmenu 结

束。1076 的 choice 到 1107 的 endchoice 就生成了一个“PCI access mode”的单项选择题。每个 config 都生成一个菜单项。1126 行的 source 就将前面的 drivers/pci/Kconfig 文件内容导入到这里边儿。

PCI 全接触

中新浙江网 11 月 6 日电 近日，备受关注的浙江湖州市南浔区三人围追堵截偷车贼，致使小偷跳河溺水身亡的不作为间接故意杀人案，终于尘埃落定。

昨天，湖州市南浔区法院对三人一审判决颜克于犯故意杀人罪判处有期徒刑 3 年 9 个月；廖红军犯故意杀人罪判处有期徒刑 3 年 3 个月；韩应龙犯故意杀人罪判处有期徒刑 3 年，缓刑 4 年。

知道这个南浔区，也就是南浔镇么？俺大学里同一宿舍的兄弟就是那旮沓的，他嘴里边儿号称的中国四大名镇之一，真的假的俺没有考证过，从他嘴里说相声般崩出的刘镛、张颂贤、徐迟、庞云曾、顾福昌等等等这一长串名字来看，所言应该非虚。不过上边儿的这则新闻说的不是南浔的旖旎风光和人文历史，而是告诉咱们，要用一颗平常心去看待偷车贼，要用一颗平常心去看代码。道儿上混的人都知道，变态的代码和偷车贼都是我们生活中不得不应对的很平常的一部分。说到这里你应该能够猜得出，接下来咱们就要看具体的代码了。

由 Kconfig 这张地图的分布来看，pci 这块儿的代码应该分布在两个地方，drivers/pci 和 arch/i386/pci，使用 wc –l 命令分别看一下多少行，具体数字俺就不说了，反正吓死人了，所以还是紧赶的看看另一张地图 Makefile，看能不能缩小一下目标，先看 drivers/pci 下边儿的

```
1 #
2 # Makefile for the PCI bus specific drivers.
3 #
4
5 obj-y          += access.o bus.o probe.o remove.o pci.o quirks.o \
6                    pci-driver.o search.o pci-sysfs.o rom.o setup-res.o
7 obj-$(CONFIG_PROC_FS) += proc.o
8
9 # Build PCI Express stuff if needed
10 obj-$(CONFIG_PCIEPORTBUS) += pcie/
11
12 obj-$(CONFIG_HOTPLUG) += hotplug.o
13
14 # Build the PCI Hotplug drivers if we were asked to
15 obj-$(CONFIG_HOTPLUG_PCI) += hotplug/
16
17 # Build the PCI MSI interrupt support
18 obj-$(CONFIG_PCI_MSI) += msi.o
19
20 # Build the Hypertransport interrupt support
21 obj-$(CONFIG_HT_IRQ) += htirq.o
22
```

```
23 #
24 # Some architectures use the generic PCI setup functions
25 #
26 obj-$(CONFIG_X86) += setup-bus.o
27 obj-$(CONFIG_ALPHA) += setup-bus.o setup-irq.o
28 obj-$(CONFIG_ARM) += setup-bus.o setup-irq.o
29 obj-$(CONFIG_PARISC) += setup-bus.o
30 obj-$(CONFIG_SUPERH) += setup-bus.o setup-irq.o
31 obj-$(CONFIG_PPC32) += setup-irq.o
32 obj-$(CONFIG_PPC64) += setup-bus.o
33 obj-$(CONFIG_MIPS) += setup-bus.o setup-irq.o
34 obj-$(CONFIG_X86_VISWS) += setup-irq.o
35
36 #
37 # ACPI Related PCI FW Functions
38 #
39 obj-$(CONFIG_ACPI)    += pci-acpi.o
40
41 # Cardbus & CompactPCI use setup-bus
42 obj-$(CONFIG_HOTPLUG) += setup-bus.o
43
44 ifndef CONFIG_X86
45 obj-y += syscall.o
46 endif
47
48 ifeq ($(CONFIG_PCI_DEBUG),y)
49 EXTRA_CFLAGS += -DDEBUG
50 endif
```

5 行，一看到 obj-y，后面儿那一堆文件就是必须得关注的，不容置疑。

7 行，关于 proc 文件系统的，咱们都得忙着发展自己口袋里的经济，精力有限，proc.c 文件就不用理睬了。

10 行，CONFIG_PCIEPORTBUS 有没有选要看你都对 Kconfig 做了些什么，不过不管你做了什么，这里都不打算过多的关注 PCI Express，所以整个 pcie/目录就可以飘过了。

12~15 行，热插拔有关的，关于 PCI 的热插拔，走到目前这一步时，俺还不清楚要对它讲到什么程度，hotplug 目录还是暂且飘过吧。

18 行，如果 Kconfig 那里你选上了 CONFIG_PCI_MSI，文件 msi.c 就必须得关注了。

21 行，没想过要聊到 Hypertransport 有关的东东，htirq.c 仍然飘过。

23~34 行，与体系架构相关，前面已经说过，与架构有关的代码都只会看 X86 上的，所以只用关注 setup-bus.c 文件，setup-irq.c 就不用理睬了。

39 行，现在电源管理一般都是用的 ACPI 了，为了响应十一五建设节约型社会的号召，还是应该关注一下。

44 行，既然要看的 X86 上的，syscall.c 就可以忽略不计了。

现在可以来统计一下咱们接下来需要关注的的文件，有 access.c，bus.c，probe.c，remove.c，pci.c，quirks.c，pci-driver.c，search.c，pci-sysfs.c，rom.c，setup-res.c，hotplug.c，msi.c，setup-bus.c，

pci-acpi.c，就这些了，说多不多说少不少，使用 wc –l 命令统计一下，7382 total，还处于一个勉强可以接受的范围。不过不要忘了，drivers/pci 下面的代码只是其中一部分，还有 arch/i386/pci，再看看它里面的 Makefile

```
1 obj-y                 := i386.o init.o
2
3 obj-$(CONFIG_PCI_BIOS)        += pcbios.o
4 obj-$(CONFIG_PCI_MMCONFIG)    += mmconfig.o direct.o mmconfig-shared.o
5 obj-$(CONFIG_PCI_DIRECT)      += direct.o
6
7 pci-y                 := fixup.o
8 pci-$(CONFIG_ACPI)            += acpi.o
9 pci-y                 += legacy.o irq.o
10
11 pci-$(CONFIG_X86_VISWS)       := visws.o fixup.o
12 pci-$(CONFIG_X86_NUMAQ)       := numa.o irq.o
13
14 obj-y                  += $(pci-y) common.o early.o
```

这个 Makefile 地图所描述的代码版图大小取决于 pci-y 是怎么定义的，这里玩了个花活儿，先在 7~9 这几行预先定义 pci-y 的值，然后在 11 行和 12 行判断 CONFIG_X86_VISWS 或者 CONFIG_X86_NUMAQ 有没有被定义，如果定义了这两个中的任何一个，就会重定义 pci-y 的值。一定要注意到 11~12 这两行里的是“:=”，而不是“+=”。至于 CONFIG_X86_VISWS 和 CONFIG_X86_NUMAQ，你使用 make menuconfig 的时候可以在“Processor type and features”下的子菜单“Subarchitecture Type”里看到它们，仍然是道单项选择题，我这里选的是第一项“PC-compatible”，所以它们两个就与俺没啥关系了，所以 visws.c、numa.c、irq.c 这几个文件就可以华丽丽的飘过了。现在统计一下剩下的那些需要咱们去关注的的文件，2753 total。两岸三地都属于一个中国，不管是 drivers/pci 那儿的，还是 arch/i386/pci 那儿的，也都只属于一个 PCI 子系统，本着一个中国的原则，咱们要统筹的全面的考察分析位于两个地方的代码，于是，7382 和 2753 这两个数字加起来就是咱们接下来需要关心的部分了，这个突破了五位数的数字左看右看横看竖看都显得那么的阴森恐怖，不过实话告诉你，已经比看到 Makefile 之前少了很多了，人家咋说也是整个一 PCI 子系统，就像走在 T 台上的芙蓉姐姐和杨二车那姆一样，看起来恐怖但也是很有内涵的，岂能够让人三眼两眼三言两语就给看透了说透了？

那现在咱们就高瞻远瞩统筹全面的扫视一下这两个地方的代码，根据在 USB Core 那里得来的经验，可以推测对于 USB、PCI 这样的子系统都应该有一个 subsys_initcall 这样的入口，咱们得先找到它。朱德庸在《关于上班这件事》里说了，要花前半生找入口，花后半生找出口。可见寻找入口对于咱们这一生，对于看内核代码这件事儿都是无比重要的，当然寻找 subsys_initcall 这个入口是不用花前半生那么久的，要是那么久，俺宁愿回家卖红薯也不在这儿看代码了，俺们豫剧里的七品芝麻官都知道当官不为民作主，不如回家卖红薯。下边儿俺就把找到的给列出来，为什么说“列”出来？难道还会有很多么？你猜对了，PCI 这边儿入口格外多，而且是有预谋有组织成系列的，不单单有 subsys_initcall，还有 arch_initcall、postcore_initcall 等等等等。

文件	函数	入口	内存位置
arch/i386/pci/ acpi.c	**pci_acpi_init**	**subsys_initcall**	**.initcall4.init**
arch/i386/pci/	**pcibios_init**	**subsys_initcall**	**.initcall4.init**

common.c			
arch/i386/pci/ i386.c	**pcibios_assign_resources**	**fs_initcall**	**.initcall5.init**
arch/i386/pci/ legacy.c	**pci_legacy_init**	**subsys_initcall**	**.initcall4.init**
drivers/pci/ pci-acpi.c	**acpi_pci_init**	**arch_initcall**	**.initcall3.init**
drivers/pci/ pci-driver.c	**pci_driver_init**	**postcore_initcall**	**.initcall2.init**
drivers/pci/ pci- sysfs.c	**pci_sysfs_init**	**late_initcall**	**.initcall7.init**
drivers/pci/ pci.c	**pci_init**	**device_initcall**	**.initcall6.init**
drivers/pci/ probe.c	**pcibus_class_init**	**postcore_initcall**	**.initcall2.init**
drivers/pci/ proc.c	**pci_proc_init**	**__initcall**	**.initcall6.init**
arch/i386/pci/ init.c	**pci_access_init**	**arch_initcall**	**.initcall3.init**

看看那一列入口，形尽而意不同的种种 xxx_initcall 让人眼花缭乱的，真不知道该从哪儿下手，应了 keso 那句话：所有的痛苦都来自选择，所谓幸福，就是没有选择。像 usb 子系统那样子简简单单一个 subsys_initcall，没得选择，傻强都知道怎么走。不过你迷惘一阵儿就可以了，可别真的被绕进去了。要知道“多少事，从来急；天地转，光阴迫。一万年太久，只争朝夕。四海翻腾云水怒，五洲震荡风雷激。要看清一切入口，全无敌。”咱们要只争朝夕看清一切入口的，因此到 include/linux/init.h 文件里碰碰运气

```
111 /*
112 * A "pure" initcall has no dependencies on anything else, and purely
113 * initializes variables that couldn't be statically initialized.
114 *
115 * This only exists for built-in code, not for modules.
116 */
117 #define pure_initcall(fn)             __define_initcall("0",fn,1)
118
119 #define core_initcall(fn)             __define_initcall("1",fn,1)
120 #define core_initcall_sync(fn)        __define_initcall("1s",fn,1s)
121 #define postcore_initcall(fn)         __define_initcall("2",fn,2)
122 #define postcore_initcall_sync(fn)    __define_initcall("2s",fn,2s)
123 #define arch_initcall(fn)             __define_initcall("3",fn,3)
124 #define arch_initcall_sync(fn)        __define_initcall("3s",fn,3s)
125 #define subsys_initcall(fn)           __define_initcall("4",fn,4)
126 #define subsys_initcall_sync(fn)      __define_initcall("4s",fn,4s)
127 #define fs_initcall(fn)               __define_initcall("5",fn,5)
128 #define fs_initcall_sync(fn)          __define_initcall("5s",fn,5s)
129 #define rootfs_initcall(fn)           __define_initcall("rootfs",fn,rootfs)
130 #define device_initcall(fn)           __define_initcall("6",fn,6)
131 #define device_initcall_sync(fn)      __define_initcall("6s",fn,6s)
132 #define late_initcall(fn)             __define_initcall("7",fn,7)
133 #define late_initcall_sync(fn)        __define_initcall("7s",fn,7s)
134
135 #define __initcall(fn) device_initcall(fn)
```

这些入口有个共同的特征，它们都是用__define_initcall 宏定义的，其实，如果你是从 USB

Core 那儿一路走过来的话，看清楚它们并不难，如果你实在忘记了或者说没见到过也没关系，不用你去蓦然回首了，俺会在这里再说一下，因为菩萨说俺今年要多做善事。
内核对子系统或者模块的初始化其实包括了两个方面，一是对各种参数的解析，一是调用上面的各种入口函数。先来说说参数的解析，这里所谓的参数包括了内核参数和模块参数，什么是内核参数？打开你的 grub 文件找找 kernel 打头儿的那些行，比如

```
kernel      /boot/vmlinuz-2.6.18-kdb root=/dev/sda1 ro splash=silent vga=0x314
```

这行里边儿的 root，splash，vga 等等就都是内核参数，USB 那边儿咱们已经见过一个类似的东东，就是 nousb，PCI 这边儿那？当然也有，具体的可以到 Documentation/kernel-parameters.txt 文件里找找

```
1302    pci=option[,option...] [PCI] various PCI subsystem options:
1303        off         [IA-32] don't probe for the PCI bus
1304        bios        [IA-32] force use of PCI BIOS, don't access
1305                    the hardware directly. Use this if your machine
1306                    has a non-standard PCI host bridge.
1307        nobios      [IA-32] disallow use of PCI BIOS, only direct
1308                    hardware access methods are allowed. Use this
1309                    if you experience crashes upon bootup and you
1310                    suspect they are caused by the BIOS.
1311        conf1       [IA-32] Force use of PCI Configuration
1312                    Mechanism 1.
1313        conf2       [IA-32] Force use of PCI Configuration
1314                    Mechanism 2.
1315        nommconf    [IA-32,X86_64] Disable use of MMCONFIG for PCI
1316                    Configuration
1317        nomsi       [MSI] If the PCI_MSI kernel config parameter is
1318                    enabled, this kernel boot option can be used to
1319                    disable the use of MSI interrupts system-wide.
1320        nosort      [IA-32] Don't sort PCI devices according to
1321                    order given by the PCI BIOS. This sorting is
1322                    done to get a device order compatible with
1323                    older kernels.
1324        biosirq     [IA-32] Use PCI BIOS calls to get the interrupt
1325                    routing table. These calls are known to be buggy
1326                    on several machines and they hang the machine
1327                    when used, but on other computers it's the only
1328                    way to get the interrupt routing table. Try
1329                    this option if the kernel is unable to allocate
1330                    IRQs or discover secondary PCI buses on your
1331                    motherboard.
1332        rom         [IA-32] Assign address space to expansion ROMs.
1333                    Use with caution as certain devices share
1334                    address decoders between ROMs and other
1335                    resources.
1336        irqmask=0xMMMM [IA-32] Set a bit mask of IRQs allowed to be
```

```
1337                    assigned automatically to PCI devices. You can
1338                    make the kernel exclude IRQs of your ISA cards
1339                    this way.
1340            pirqaddr=0xAAAAA        [IA-32] Specify the physical address
1341                    of the PIRQ table (normally generated
1342                    by the BIOS) if it is outside the
1343                    F0000h-100000h range.
1344            lastbus=N       [IA-32] Scan all buses thru bus #N. Can be
1345                    useful if the kernel is unable to find your
1346                    secondary buses and you want to tell it
1347                    explicitly which ones they are.
1348            assign-busses   [IA-32] Always assign all PCI bus
1349                    numbers ourselves, overriding
1350                    whatever the firmware may have done.
1351            usepirqmask     [IA-32] Honor the possible IRQ mask stored
1352                    in the BIOS $PIR table. This is needed on
1353                    some systems with broken BIOSes, notably
1354                    some HP Pavilion N5400 and Omnibook XE3
1355                    notebooks. This will have no effect if ACPI
1356                    IRQ routing is enabled.
1357            noacpi          [IA-32] Do not use ACPI for IRQ routing
1358                    or for PCI scanning.
1359            routeirq        Do IRQ routing for all PCI devices.
1360                    This is normally done in pci_enable_device(),
1361                    so this option is a temporary workaround
1362                    for broken drivers that don't call it.
1363            firmware        [ARM] Do not re-enumerate the bus but instead
1364                    just use the configuration from the
1365                    bootloader. This is currently used on
1366                    IXP2000 systems where the bus has to be
1367                    configured a certain way for adjunct CPUs.
1368            noearly         [X86] Don't do any early type 1 scanning.
1369                    This might help on some broken boards which
1370                    machine check when some devices' config space
1371                    is read. But various workarounds are disabled
1372                    and some IOMMU drivers will not work.
1373            bfsort          Sort PCI devices into breadth-first order.
1374                    This sorting is done to get a device
1375                    order compatible with older (<= 2.4) kernels.
1376            nobfsort        Don't sort PCI devices into breadth-first order.
1377            cbiosize=nn[KMG]        The fixed amount of bus space which is
1378                    reserved for the CardBus bridge's IO window.
1379                    The default value is 256 bytes.
1380            cbmemsize=nn[KMG]       The fixed amount of bus space which is
```

```
1381                    reserved for the CardBus bridge's memory
1382                    window. The default value is 64 megabytes.
```

这么将近 100 行的东东都是 PCI 这边儿可以设置的参数选项，与之相比，USB 那里也忒单纯了。咱们地球上都在搞军备竞争，内核也去跟风搞什么代码竞争，看谁的代码长谁的代码复杂，现在连内核参数也搞竞争了，看谁的多。当然这种竞争与 keso 说的中国互联网行业的“抄袭盛行，低水平竞争，恶性竞争。”不一样。

这么多选项有的前边儿已经提到了，没有提到的俺也不打算在这里讲，日后碰到再说了。明白了内核参数，再看看模块参数。其实俺所说的模块参数就是你使用 insmod 或者 modprobe 加载模块时在命令行里指定的参数，它们在驱动里使用 module_param 这样的宏来声明。不过千万要明白一点的是，俺意思并不是说内核参数就不能使用 module_param 这样的宏去定义了，USB 那边儿的 nousb 就是使用__module_param_call 宏定义的。

另外还要明白一点的是，当模块被编译进内核的时候，它的那些模块参数就需要在 grub 文件的 kernel 行里来指定了，比如，

```
modprobe usbcore autosuspend=2
```

放到 kernel 行里就是

```
usbcore.autosuspend=2
```

当然在 drivers/usb/usb.c 里 autosuspend 的默认值就是 2，所以这里再去指定为 2 纯粹就是春节申遗多此一举。

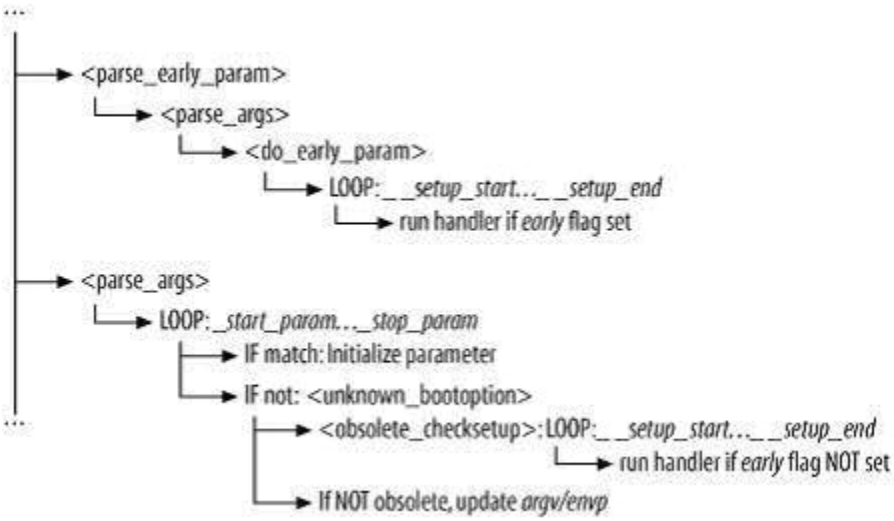
你如果很好奇想知道一个模块都有哪些参数可以使用，敲命令“modinfo -p ${modulename}”就可以了。对于已经加载到内核里的模块，它们的参数都会陈列在/sys/module/${modulename}/parameters/目录下面儿等着你去检阅，当然你也可以使用"echo -n ${value} > /sys/module/${modulename}/parameters/${parm}"这样的命令去修改它们。

现在看看内核是怎么进行参数解析的，这主要依靠一个名叫 parse_args 的函数，它的细节不用关心，只需要明白在内核启动时，或者加载模块结协模块命令行参数时它都会被调用就可以了。如果你是一个有心人，去扫了眼 init/main.c 文件里的 start_kernel 函数，你就会发现，内核启动时调用了一次 parse_args 还觉得不过瘾，又调用了第二次

```
545         parse_early_param();
546         parse_args("Booting kernel", static_command_line, __start___param,
547                 __stop___param - __start___param,
548                 &unknown_bootoption);
```

parse_args 的第一次调用就在 parse_early_param 函数里面，为什么会出现两次调用 parse_args 的情况？怪只怪内核参数又分成了两种，一种是普普通通的，一种是有特权，需要在普通选项之前处理的，就像银行的 VIP 客户，有专门开辟的窗口接待，不管咱们排队排的多长，他们一到就会优先服务，即使没有 VIP，那个窗口儿就是空着也不会给你用，就让你在那儿等。既然这个世界上人有两种，有特权的和没特权的，那内核参数同样也就有两种，有特权的和没特权的。

特权谁都想有，但是有了特权又都怕人说自己有特权，就像哈医大二院的纪委书记在接受央视的采访“老人住院费 550 万元”时评价自己单位“我们就是一所人民医院……就是一所贫下中农的医院，从来不用特权去索取自己身外的任何利益……我们不但没有多收钱还少收了。”那样，人就是这么的复杂和奇怪。内核参数就不一样了，有特权也是光明正大的用，从不藏着掖着，直接使用 early_param 宏去声明，让你一眼就看出它是有特权的，而普通的内核参数则是使用 module_param 系列的宏声明的。使用 early_param 声明的那些参数就会首先由 parse_early_param 去解析，然后才轮得着其它的参数。这里引用一下《Understanding Linux Network Internals》里的图 7-2



关于这张图，你只要注意到__setup_start、__setup_end、__start___param、__stop___param 这几个东东就可以了，它们很明确的告诉我们特权参数和普通参数在内存里存放的位置是不一样的，特权参数放在__setup_start 和__setup_end 之间的节里，普通参数放在__start___param 和__stop___param 之间的节里。

PCI 里边儿也出现了一个 early_param，在 drivers/pci/pci.c 里面

1425 early_param("pci", pci_setup);

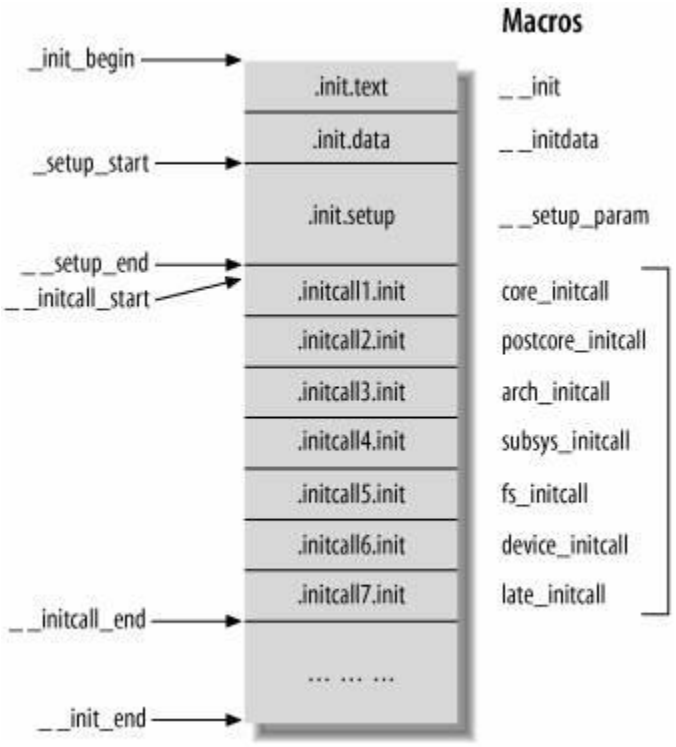
early_param 怎么定义的你可以去 include/linux/init.h 里面看，俺这里就不说了，反正这行的意思就是发现你 grub 文件的 kernel 行里有“pci=”这样的东东的时候，就会调用函数 pci_setup 进行处理，怎么处理？这都是以后的事儿了。

参数解析完之后，再经过一个漫长而曲折的过程，就会调用到各种各样的xxx_initcall入口函数，这些函数的调用不是随便的，而是按照一定顺序的，这个顺序就取决于__define_initcall宏。__define_initcall宏用来将指定的函数指针放到.initcall.init节里，这都是讲USB Core的时候说过的，再扼要重述一遍。

内核可执行文件由许多链接在一起的对象文件组成。对象文件有许多节，如文本、数据、init 数据、bass 等等。这些对象文件都是由一个称为链接器脚本的文件链接并装入的。这个链接器脚本的功能是将输入对象文件的各节映射到输出文件中；换句话说，它将所有输入对象文件都链接到单一的可执行文件中，将该可执行文件的各节装入到指定地址处。 vmlinux.lds 是存在于 arch/<target>/ 目录中的内核链接器脚本，它负责链接内核的各个节并将它们装入内存中特定偏移量处。在 vmlinux.lds 文件里查找 initcall.init 就可以看到下面的内容

```
__inicall_start = .;
.initcall.init : AT(ADDR(.initcall.init)  －  0xC0000000) {
*(.initcall1.init)
*(.initcall2.init)
*(.initcall3.init)
*(.initcall4.init)
*(.initcall5.init)
*(.initcall6.init)
*(.initcall7.init)
}
__initcall_end = .;
```

这就告诉我们.initcall.init节又分成了 7 个字节，而xxx_initcall入口函数指针具体放在哪一个子节里边儿是由xxx_initcall的定义中，__define_initcall宏的参数决定的，比如core_initcall将函数指针放在.initcall1.init子节，device_initcall将函数指针放在了.initcall6.init子节等等。各个子节的顺序是确定的，即先调用.initcall1.init中的函数指针再调用.initcall2.init中的函数指针，等等。不同的入口函数被放在不同的子节中，因此也就决定了它们的调用顺序。
关于内核里初始化有关的一些内存节，可以再参考下《Understanding Linux Network Internals》的图 7-3



现在你再回头去看看前面那张描述 PCI 里边儿各种 xxx_initcall 入口的表，应该不会再觉得迷惘了，对整个 PCI 子系统的运作流程也大致有个谱了。

PCI 的那些内核参数

经过上节的头脑风暴，咱们明白了，PCI 这边儿入口虽然多，但还是有规律可循有法可依的，内核启动时，得一个一个严格的按照顺序调用它们来完成 PCI 子系统的初始化，不能乱了章法。这点儿并不是所有人都会明白的，比如前段儿时间厦门那出儿卖房事件里宣称“我海关有人，谁敢动我”的那位海关老婆，她就觉得有法是不如有人的。
（背景知识：厦门网 11 月 14 日电，06 年 11 月 5 日，曾先生夫妻向被告购买了香秀里的这套房子，成交价为 98 万元。按照合同约定，房产交付时，被告应保证室内整洁，无结构性损坏，门窗及配套水、电、煤气等应当完好。2007 年 3 月 11 日，被告按照合同约定的时间把房子交了出来。曾先生去验房时，发现这套房子“就像被洗劫过一样”：房子内所有的开关及插座面板都被拆了，厨房、卫生间的水龙头、水槽都不见了，家具只要是有门的、有抽屉的，全都被拆卸一空，连滑轮、拉锁等小零配件也都不放过，甚至床靠背的装饰也被挖掉。法院认为，被告未按合同约定按时、完整地交付房屋，已构成违约。制造这起闻所未闻事件的房东夫妇，前天被思明区法院判定得赔偿修复费 7000 余元、违约金 2940 元、买主 5 个月

租房损失 1.4 万元。）

同时，咱们还明白，如果在 grub 文件 kernel 那一行添加有“pci=”这样的东东，在调用那些入口函数之前，就必须得先调用一个 pci_setup 函数来解析这部分内核参数。pci_setup 函数在 drivers/pci/pci.c 里有定义

```
1403 static int __devinit pci_setup(char *str)
1404 {
1405        while (str) {
1406              char *k = strchr(str, ',');
1407              if (k)
1408                    *k++ = 0;
1409              if (*str && (str = pcibios_setup(str)) && *str) {
1410                    if (!strcmp(str, "nomsi")) {
1411                          pci_no_msi();
1412                    } else if (!strncmp(str, "cbiosize=", 9)) {
1413                          pci_cardbus_io_size = memparse(str + 9, &str);
1414                    } else if (!strncmp(str, "cbmemsize=", 10)) {
1415                          pci_cardbus_mem_size = memparse(str + 10, &str);
1416                    } else {
1417                          printk(KERN_ERR "PCI: Unknown option `%s'\n",
1418                                      str);
1419                    }
1420              }
1421              str = k;
1422        }
1423        return 0;
1424 }
1425 early_param("pci", pci_setup);
```

1405 行，友情提醒一下，参数里的这个 str 并不包括“pci=”，它只包含了等号后面的那一串选项，选项之间使用逗号做间隔，而且它们之间不能有空格。这个 while 循环的目的就是以逗号为地标，层层推进找出每个选项进行处理。还是举个例子吧，显得形象一点儿，假设内核参数里设置了“pci=nobios,nomsi,conf1”，那么这里的 str 就是“nobios,nomsi,conf1”，while 循环要从 str 里面依次地解析出 nobios，nomsi，conf1，并进行相应的处理，至于它们都什么意思，暂时不用去关心。你可以想像一下冰糖葫芦，都那么成串儿的排列着，不过不一样的是，冰糖葫芦你可以从上边儿那个开始吃，也可以从下边儿那个开始吃，没有法律去无聊的规定你吃的顺序，而这里的 str 却只能从第一个选项开始，一个一个流水线式的解析。

1406 行，再假设这是第一次进入 while 循环，那么这个 k 就等于“,nomsi,conf1”。

1408 行，有关字符串结束符，俺在讲 USB Core 的时候都已经强调的不能再强调了。这里要提醒你注意的是“++”这个自增运算符，你可千万别以为这行的意思是先把 k 向前移动一个字符的位置，然后将这个新位置设置为 0，相反，它是先将 k 的位置，也就是“,nomsi,conf1”的第一个逗号那里设置为 0，然后再将 k 移动一个字符，此时 k 就变成了“nomsi,conf1”，而 str 就变成了“nobios”。用 C 里的行话来解释的话，就是 side effect 和 sequence point。

1409 行，这个函数的精华就在 if 这儿，更准确的说，就在 if 的条件里边儿。先是判断 str 是不是为空，因为上面假设了这是第一次进入 while 循环，str 现在还是“nobios”，显然不是为空的。如果 str 不为空，就会去调用 pcibios_setup 函数对 str 做处理，然后判断

pcibios_setup(str)的返回值 str 是不是为空，看来不与 pcibios_setup 函数亲密接触一下，咱们是不可能知道这个返回的 str 究竟是个什么东西的。你可以在 arch/i386/pci/common.c 文件里找到 pcibios_setup 函数

```
346 char * __devinit pcibios_setup(char *str)
347 {
348         if (!strcmp(str, "off")) {
349                 pci_probe = 0;
350                 return NULL;
351         } else if (!strcmp(str, "bfsort")) {
352                 pci_bf_sort = pci_force_bf;
353                 return NULL;
354         } else if (!strcmp(str, "nobfsort")) {
355                 pci_bf_sort = pci_force_nobf;
356                 return NULL;
357         }
358 #ifdef CONFIG_PCI_BIOS
359         else if (!strcmp(str, "bios")) {
360                 pci_probe = PCI_PROBE_BIOS;
361                 return NULL;
362         } else if (!strcmp(str, "nobios")) {
363                 pci_probe &= ~PCI_PROBE_BIOS;
364                 return NULL;
365         } else if (!strcmp(str, "nosort")) {
366                 pci_probe |= PCI_NO_SORT;
367                 return NULL;
368         } else if (!strcmp(str, "biosirq")) {
369                 pci_probe |= PCI_BIOS_IRQ_SCAN;
370                 return NULL;
371         } else if (!strncmp(str, "pirqaddr=", 9)) {
372                 pirq_table_addr = simple_strtoul(str+9, NULL, 0);
373                 return NULL;
374         }
375 #endif
376 #ifdef CONFIG_PCI_DIRECT
377         else if (!strcmp(str, "conf1")) {
378                 pci_probe = PCI_PROBE_CONF1 | PCI_NO_CHECKS;
379                 return NULL;
380         }
381         else if (!strcmp(str, "conf2")) {
382                 pci_probe = PCI_PROBE_CONF2 | PCI_NO_CHECKS;
383                 return NULL;
384         }
385 #endif
386 #ifdef CONFIG_PCI_MMCONFIG
```

```
387         else if (!strcmp(str, "nommconf")) {
388                 pci_probe &= ~PCI_PROBE_MMCONF;
389                 return NULL;
390         }
391 #endif
392         else if (!strcmp(str, "noacpi")) {
393                 acpi_noirq_set();
394                 return NULL;
395         }
396         else if (!strcmp(str, "noearly")) {
397                 pci_probe |= PCI_PROBE_NOEARLY;
398                 return NULL;
399         }
400 #ifndef CONFIG_X86_VISWS
401         else if (!strcmp(str, "usepirqmask")) {
402                 pci_probe |= PCI_USE_PIRQ_MASK;
403                 return NULL;
404         } else if (!strncmp(str, "irqmask=", 8)) {
405                 pcibios_irq_mask = simple_strtol(str+8, NULL, 0);
406                 return NULL;
407         } else if (!strncmp(str, "lastbus=", 8)) {
408                 pcibios_last_bus = simple_strtol(str+8, NULL, 0);
409                 return NULL;
410         }
411 #endif
412         else if (!strcmp(str, "rom")) {
413                 pci_probe |= PCI_ASSIGN_ROMS;
414                 return NULL;
415         } else if (!strcmp(str, "assign-busses")) {
416                 pci_probe |= PCI_ASSIGN_ALL_BUSSES;
417                 return NULL;
418         } else if (!strcmp(str, "routeirq")) {
419                 pci_routeirq = 1;
420                 return NULL;
421         }
422         return str;
423 }
```

这个函数是个练家子，会使排山倒海这一招儿，或者说它是张惠妹的 fans，喜欢她的《排山倒海》，这么一排排的 if-else 还真让人有点招架不住。其实有点 C 常识的人扫一眼就明白，这是在拿 str 逐个儿的和一些字符串做比较，而这些字符串应该不是第一次见了，就是前面贴出来的内核参数里“pci=”后边儿可能会有的那些选项。下边儿先挨个儿简单提一下，混个脸熟，日后还会遇到的。

348 行，off 选项意味着内核启动时就不会再有总线枚举这个过程了，也就是说你机子里挂在 PCI 总线上的那些设备就都游离于内核之外了，什么后果自己想想吧，所以使用起来要

慎重，没事儿的时候最好不要设置它。前面从 Documentation/kernel-parameters.txt 找出来的内容已经说明了它只对 X86 32 位平台起作用，要知道 pcibios_setup 函数内核里不是只有一个的，这里是因为早就声明了基于 32 位 X86 平台，所以就从 drivers/pci/pci.c 里的 pci_setup 函数直接跳到了 arch/i386/pci/common.c 里的这个 pcibios_setup，其实在 arch 目录里列出的那些个架构里，都有这么一个 pcibios_setup，不过位置不同，所看到的风景也就不同，它们有些是什么都没做就直接返回，有些是没有处理 off 这一项，这点儿你了解就可以了。
如果指定了 off，就会将变量 pci_probe 设置为 0，pci_probe 在 arch/i386/pci/pci.h 里声明并在 common.c 里定义

```
20 unsigned int pci_probe = PCI_PROBE_BIOS | PCI_PROBE_CONF1 | PCI_PROBE_CONF2 |
21                         PCI_PROBE_MMCONF;
```

pci_probe 的初值为几个宏的合作社形式，它们分别对应了 PCI access mode 的几种方式，在 arch/i386/pci/pci.h 里定义

```
15 #define PCI_PROBE_BIOS          0x0001
16 #define PCI_PROBE_CONF1         0x0002
17 #define PCI_PROBE_CONF2         0x0004
18 #define PCI_PROBE_MMCONF        0x0008
19 #define PCI_PROBE_MASK          0x000f
20 #define PCI_PROBE_NOEARLY       0x0010
21
22 #define PCI_NO_SORT             0x0100
23 #define PCI_BIOS_SORT           0x0200
24 #define PCI_NO_CHECKS           0x0400
25 #define PCI_USE_PIRQ_MASK       0x0800
26 #define PCI_ASSIGN_ROMS         0x1000
27 #define PCI_BIOS_IRQ_SCAN       0x2000
28 #define PCI_ASSIGN_ALL_BUSSES   0x4000
```

PCI_PROBE_BIOS 对应了 BIOS 方式，PCI_PROBE_MMCONF 对应了 MMConfig 方式，这好理解，看名字就知道了，不好理解的是 PCI_PROBE_CONF1 和 PCI_PROBE_CONF2 都对应了 Direct 方式，这是因为曾经有过两种 PCI Configuration Mechanism，用行话来说就是 Type1 和 Type2，内核要想不通过 BIOS 直接去访问设备的话，也必须得对应有两种访问方式，即这里的 conf1 和 conf2。Type2 主要是在 PCI 发展的少年时期，某些主桥用过，现在一般都不会再用了，但是为了兼容一些老的主板，conf2 还是保留了下来。至于剩下那些宏碰到再说了。
off 接下来是 bfsort 和 nobfsort，本来在几个月之前，是没有这两个角色的，自然也就不会有 351 到 357 这几行，俺也就不会在这儿辛苦的码下边儿的这些字，不过既然偶然选择了新内核而不是几个月前的次新内核，那这两个东东是必然躲不过的了。人生本来就是由一系列的偶然所组成，之前的偶然决定了现在的必然，出生、上学、工作所有你做过的事遇到的人都是偶然，当然也包括俺说的 linux 的这些事儿，这些偶然组合起来就决定了俺接下来必然得说说 bfsort 和 nobfsort 的那些风花雪月。
话说几个月前吧，一个名叫 Matt Domsch 的小伙子发现了一件奇怪的事情，在 DELL 的一款有两个网口的服务器上，进入 BIOS 时，显示两个网口分别是 NIC1 和 NIC2，而启动 linux 后，在 2.4 内核上显示的是 eth0 和 eth1，在 2.6 内核上显示的调了个个儿，变成了 eth1 和 eth0，接着他又在 HP 和 SUN 的一些服务器上也发现了类似的现象。经过了一番不屈不挠的论证分析，他发现了这个现象的根源，在 2.4 的内核里，是按照广度优先（breadth-first）算

法来进行总线枚举的，得出的那个全局的所有 PCI 设备的链表中设备的顺序自然也是广度优先的。而 2.6 内核里，这个枚举过程变成了深度优先（depth-first）。具体在出现这个现象的服务器上，由于物理上的原因，NIC1 出现在总线号更小的那个 PCI 总线上，但是使用深度优先算法时，NIC2 比 NIC1 更早被发现，于是 NIC2 成了 eth0，NIC1 却成了 eth1，如果换成了广度优先，NIC1 就会更早被发现，eth0 就不会给 NIC2 抢去了。显然正常情况下，是应该先找到 NIC1 后找到 NIC2，NIC1 对应 eth0，NIC2 对应 eth1。

做人不能做到足协那地步，光知道叙述问题而回避怎么去解决，即使这个问题再小可它也总归是问题，咱们的温总理就说了：中国财富再多，除以 13 亿人，就少得可怜了；中国问题再小，乘以 13 亿人，也就很大了。所以说不要小看这个小问题，俺曾经就在某个地方遇到过 eth0 神秘失踪 eth1 鸠占鹊巢的情况，友邦惊诧了好大一会儿。这道理谁都懂，Matt Domsch 小伙儿自然也懂，同时他也明白韩寒那句话：没有一个问题能在二十句话内解决，不论什么东西最后都要引到自己研究的领域中去，哪怕嫖娼之类的问题也是。于是他就为内核提交了一个有很多个 20 行长的 patch，添加了 bfsort 和 nobfsort，让用户去选择使用不使用广度优先，如果用户指定了 bfsort，则枚举最后得到的 PCI 设备链表里就是按照广度优先的顺序排列。

至于怎么个广度优先和深度优先，讲到具体的总线枚举过程时再去深入了解，这里就不多说了，再说，关于算法俺是个菜鸟，不然就不会在北京西郊宾馆和清华园 google 大楼之间的那条路上来回徘徊了好多次之后还是被 google 无情的拒绝了。还是看看下面的 pci_bf_sort，如果你指定了 bfsort，它就会被设置为 pci_force_bf，指定了 nobfsort，就会设置为 pci_force_nobf。pci_bf_sort、pci_force_bf、pci_force_nobf 这几个生面孔同样也是 Matt Domsch 提交的那个 patch 里添加的，pci_bf_sort 在 arch/i386/pci/common.c 的开头儿有定义，而 pci_force_bf 和 pci_force_nobf 出现在 arch/i386/pci/pci.h 里

```
33 enum pci_bf_sort_state {
34        pci_bf_sort_default,
35        pci_force_nobf,
36        pci_force_bf,
37        pci_dmi_bf,
38 };
```

358~375 行的这些选项是针对 BIOS 访问方式的，如果你明确指定了 bios，那说明你希望使用 PCI BIOS 而不是直接访问硬件，这表示内核，或者说是你完全信任 BIOS，不过要永远记住，信任的代价可能是非常昂贵的，所以一般情况下咱们在 PCI access mode 那儿都是选择 Any，让内核去决定，而不是在这儿明确的指明。接下来的 nobios 当然是相反的意思了。从 nosoft 的长相看，又是与排序相关的，前边儿讲 Matt Domsch 的那些风花雪月的时候有提到，在 2.4 的内核里，是按照广度优先（breadth-first）算法来进行总线枚举的，其实这种说法是不准确的，没有使用 PCI BIOS 时，确实是这样，但是使用了 PCI BIOS 时就不一定了，虽说通常情况下 PCI BIOS 也是按照广度优先这么一个顺序去枚举的，但是毕竟 PCI BIOS 的 Spec 里并没有明确规定一个 PCI BIOS 应该遵守的顺序，这么一来就有了变数，而这个 nosoft 就是用来减少这个变数的。如果你指定了 nosoft，就是告诉内核，即使使用了 PCI BIOS 的方式，也不按照 PCI BIOS 枚举的顺序去排序设备。

接下来的 biosirq 牵涉到 PCI 的中断机制。咱们已经知道了每个 PCI 总线上都可以挂上很多个功能设备，也知道了设备是可以主动向 CPU 发出中断请求表明自己需要的。同时，咱们也应该能够感同身受，就好像上头儿发往下边的体恤通常会被层层和谐掉一样，凡是下边发往上头儿的请求通常也是要经过那么几层来传达的，只是如果谁比较特殊，或者和上头儿关系好，会少几层麻烦而已（但中断控制器那一层总归是躲不开的），显然 PCI 设备并不属于

这类特殊品种，和上头儿的关系也没那么铁，它们要是请求 CPU 干点啥事儿，得一级一级一层一层的向上转达。首先，他们得连接到 PCI 总线的中断请求线上，这东西每个 PCI 总线也就只有 4 个，对应了每个 PCI 插槽（slot）上的 4 个中断引脚（pin）INTA#、INTB#、INTC#和 INTD，所以很多 PCI 设备得发扬开源共享的精神一起用。然后，PCI 的这几条中断请求线并不是就可以直接接到中断控制器上了，之间还需要通过一个所谓的可编程中断路由器（PCI Interrupt Router），至于这个 router 是如何将 PCI 的各个中断请求线给接到中断控制器上的，又是分别接到中断控制器的哪个引脚上的，就依赖于具体的系统了。对于内核来说，这种中断的路由拓扑状况自然是必须得了然于胸的，怎么去做到了然于胸？可以通过 PCI BIOS（如果板子上有 BIOS 并且支持的话），也可以自己搞定。那么这个 biosirq 就是告诉内核通过 PCI BIOS 去获取一个名叫中断路由表（PCI Interrupt routing table）的东东，从而达到对中断路由情况的了然于胸。不得不说的是，中断路由器只有在中断控制器为 PIC 的时候才用得着，如果为 APIC，就不用麻烦他老人家了。

作为主板的至亲好友，BIOS 自然是知道，也应该知道在 PCI 设备、PCI 插槽、中断路由器等之间的中断线路是如何连接的，它也有义务和责任在自己的 BIOS ROM 里存储相关的一些信息，事实上 PCI BIOS Spec 里也是这么规定的，内核可以在 0xF0000h~0xFFFFF 这段儿地址上查找得到中断路由表。但是不幸的是，在 05 年的阳春三月，俺们正张罗着一顿又一顿的散伙饭的时候，一个小伙儿发现在自己的硬件环境里，BIOS 不能在 0xF0000h~0xFFFFF 之间生成和保存中断路由表，于是他就提交了一个 patch 来解决这个问题，于是就有了 pirqaddr。pirqaddr 允许你去告诉内核中断路由表保存在哪个地方，这样的话，内核可以直接检测这个地址来获得中断路由表，而不用再去在 0xF0000h~0xFFFFF 之间挖地三尺进行大范围的搜索。这么做还不经意间带来了一个好处，对于那些每次都使用一样的地址保存中断路由表的 BIOS，即使他们老老实实确实使用的是 0xF0000h~0xFFFFF 之间的某个地址，你也可以使用 pirqaddr 明确的将它指定出来，从而减少内核在这个范围内查找时带来的消耗。

pirq_table_addr 是在 arch/i386/pci/common.c 里定义的一个 unsigned long 数，simple_strtoul 函数能够将字符串转换为 unsigned long，在这里就是将 pirqaddr 所指定的地址转化为数字赋给 pirq_table_addr。

376~385 行之间的这两个 conf1 和 conf2 对应的就是前面提到的两种 PCI Configuration Mechanism。

386 行，nommconf 用来告诉内核不要使用 MMConfig 方式访问设备。

392 行，noacpi，用来禁止使用 ACPI 处理任何 PCI 相关的内容，包括 PCI 总线的枚举和 PCI 设备中断路由。咱们已经习惯于将 ACPI 和电源管理牢牢的联系在一起，所以很难会理解它和 PCI 这边儿的关系。其实仔细瞅瞅 ACPI 的全称 the Advanced Configuration & Power Interface，里边儿不仅有 Power，还有 Configuration，你就应该能够悟出点儿道了。ACPI 出现的目的是在咱们的 OS 和硬件平台之间隔出个抽象层，这样 OS 和平台就可以各自发展各自的，新的 OS 可以控制老的平台，老的 OS 也可以控制新的平台而不需要额外的修改。ACPI 这个抽象层里包含了很多寄存器和配置信息，绝大部分 OS 需要从 BIOS 得到的信息都可以从 ACPI 得到，并且趋势是未来的任何新的特性相关的信息都只能从 ACPI 得到，这些信息里当然也包括 PCI 设备的中断路由情况等。

acpi_noirq_set 在 include/asm-i386/acpi.h 里定义，如果编译内核的时候配置上了 ACPI，它就将全局变量 acpi_noirq 设置为 1，意思就是上边儿说的，否则，它就是一个空函数。

396 行，在内核启动过程的开始阶段，会对 PCI 设备进行一次早期的扫描，这时会使用前面提到的 type1 方式尝试访问每个可能存在的 PCI 设备的配置空间，如果设备本身就不存在，那么尝试去访问它的配置空间的话，在一些有 bug 的板子上就会发生自检。为了避免这种情况，你可以使用 noearly 选项来禁止这个早期的扫描，当然，如果 type1 方式已经被禁止的

话这次扫描自然也就不会发生。
400 行，一看到 CONFIG_X86_VISWS 俺就高兴，说明有那么一大段不用去看了。
一路飘到 412 行，rom，PCI Spec 规定，PCI 设备可以携带一个扩展 ROM，并将与自己有关的初始化代码放到它里边儿，内核通执行这些代码来完成与设备有关的初始化。同时，PCI Spec 还规定了，这些代码不能在设备的 ROM 里执行，必须得拷贝到系统的 RAM 里再执行，于是 ROM 与 RAM 这两个本来不搭嘎的东东就产生了联系，这个联系就是行话所谓的映射。这里的选项 rom 就是告诉内核将设备的 ROM 映射到系统的 RAM 里。
415 行，assign-busses，表示内核将无视 PCI BIOS 分配的总线号，自己重新分配。
418 行，routeirq，在 ldd3 里，Greg 告诉我们，在驱动程序访问 PCI 设备的任何资源之前都要先调用 pci_enable_device，这个函数会为设备完成中断路由，也就是分配中断请求线等工作。但是 Greg 语重心长的这句话并不是人人都能记住，于是为了防止某些 PCI 驱动没有调用 pci_enable_device 就去访问设备的资源，routeirq 就粉墨登场了，它清楚明白的告诉内核不要信任那些写 PCI 驱动的，得自己为所有的 PCI 设备做中断路由。
到此算是呕心沥血的对 pcibios_setup()里边儿排列的所有选项检阅了一遍，台下走的人不容易，台上检阅的人也不容易，所以要理解前年河南新密市的那些搞阅兵式的领导们，他们在凛冽寒风中喊出那么一句句“同志们好！”“同志们辛苦了！”也是很不容易的。
回到 pci_setup 函数 1409 行的那个 if，pcibios_setup()的返回值已经很清楚了，不是 NULL，就是将 str 原封不动的返回，如果返回值 str 为 NULL，则说明这个选项已经在 pcibios_setup()里面处理过了，这个 if 就不用再进去了，接着处理下一个吧，如果将 str 完好无损的返回了，就说明 pcibios_setup()里陈列的那些选项里没有找到你指定的这个选项，需要再进到 if 里面碰碰运气。
扫一眼这个 if 里边儿的那些行，我们发现，仍然是一些 if-else 排列，不过比 pcibios_setup()那里少多了，只涉及了三个选项，可以抱着一颗平常心去看一下。如果指定了 nomsi，就会调用 drivers/pci/msi.c 里的 pci_no_msi 函数

```
671 void pci_no_msi(void)
672 {
673         pci_msi_enable = 0;
674 }
```

这个世界上像 pci_no_msi()这么单纯的角色已经不多了，它只是将 msi.c 文件里的一个 static 变量设置为 0，表示禁用 MSI 中断，如果你在 menconfig 的时候选上了“Message Signaled Interrupts (MSI and MSI-X)”，只要在内核启动的时候指定了 nomsi 同样也可以强行将它禁止掉。
cbiosize 和 cbmemsize 都是 CardBus 桥专用的，就此飘过。
在 if 里碰完了运气，接下来要将 k 赋给 str，根据前面的假设，此时 str 为 nobios，k 为“nomsi,conf1”，将 k 赋给 str 之后就会开始下一次的 while 循环。当然 nobios 是肯定能够得到处理的，不过即使你哪天心情不好随便指定了一个字符串，从而找不到它对应的那个选项，内核也只会幽怨的打印一句“PCI: Unknown option …”，同时将它给忽略掉。

初始化(一)

解析完了 PCI 的那些内核参数，再翻过多少座山跨过多少条河，内核就会遇到 init/main.c 里一个名叫 do_initcalls 的函数。do_initcalls 对内核来说只不过是漫长冒险旅程中的一个驿站，对 PCI 这个故事来说却是命运转轮的开始，内核在它里边完成了对.initcall.init 节里各种 xxx_initcall 函数的执行，PCI 的那些自然也包括在内。你不用像新东方老罗“我走来走去，为中国的命运苦苦思索。”那样走来走去为 PCI 的命运思索，因为决定 PCI 命运的那些 xxx_initcall 早列在之前的那张表里了，也不用你再去蓦然回首，这里会再贴一遍。

文件	函数	入口	内存位置
arch/i386/pci/acpi.c	pci_acpi_init	subsys_initcall	.initcall4.init
arch/i386/pci/common.c	pcibios_init	subsys_initcall	.initcall4.init
arch/i386/pci/i386.c	pcibios_assign_resources	fs_initcall	.initcall5.init
arch/i386/pci/legacy.c	pci_legacy_init	subsys_initcall	.initcall4.init
drivers/pci/pci-acpi.c	acpi_pci_init	arch_initcall	.initcall3.init
drivers/pci/pci- driver.c	pci_driver_init	postcore_initcall	.initcall2.init
drivers/pci/pci- sysfs.c	pci_sysfs_init	late_initcall	.initcall7.init
drivers/pci/pci.c	pci_init	device_initcall	.initcall6.init
drivers/pci/probe.c	pcibus_class_init	postcore_initcall	.initcall2.init
drivers/pci/proc.c	pci_proc_init	__initcall	.initcall6.init
arch/i386/pci/init.c	pci_access_init	arch_initcall	.initcall3.init

咱们从讲 USB Core 时就已经知道对这些 xxx_initcall 函数的调用是必须按照一定顺序的，先调用.initcall1.init 中的再调用.initcall2.init 中的，很明显，表里列出来的应该最先被调用的是.initcall2.init 子节中的两个函数 pcibus_class_init 和 pci_driver_init。现在问题出现了，对于处于同一子节中的那些函数，比如 pcibus_class_init 和 pci_driver_init 这两个函数来说又是哪个会最先被调用？当然，你可以说处在前边儿地址的会最先被调用，这是大实话，因为 do_initcalls 函数的实现就是在.initcall.init 所处的地址上来回的 for 循环。可你怎么知道同一子节的函数哪个在前边儿哪个在后边儿？

别的不多说，先看看 gcc 的 Using the GNU Compiler Collection 中的一段话：

the linker searches and processes libraries and object files in the order they are specified. Thus, ‘foo.o -lz bar.o’ searches library ‘z’ after file ‘foo.o’ but before ‘bar.o’.

看完这段话，希望会听到你说：我悟道了！更希望会看到你翻出来 drivers/pci/Makefile 文件，瞅到下边儿这两行

```
5 obj-y         += access.o bus.o probe.o remove.o pci.o quirks.o \
6                  pci-driver.o search.o pci-sysfs.o rom.o setup-res.o
```

probe.o 在 pci-driver.o 的前面，那么 probe.c 里的 pcibus_class_init 函数也会在 pci- driver.c 里的 pci_driver_init 函数之前被调用。再给你看一句话，Documents/kbuild/makefile.txt 的 3.2 中的：

The order of files in $(obj-y) is significant.

既然 pcibus_class_init 会首先被调用，那咱们就先窥视一下它的庐山真面目

```
100 static struct class pcibus_class = {
101         .name           = "pci_bus",
```

```
102         .release        = &release_pcibus_dev,
103 };
104
105 static int __init pcibus_class_init(void)
106 {
107         return class_register(&pcibus_class);
108 }
109 postcore_initcall(pcibus_class_init);
```

class_register 是设备模型中一个很基础的函数，在这里它的目的就是注册一个名叫“pci_bus”的 class，关于 class，你应该不会感到陌生了，usb 那里就已经注册过一个 usb_host，不过不同的是那时使用的是 class_create，而现在使用的是 class_register，咱不陪写代码的哥们儿玩这些文字游戏了，不管它是 create 还是 register，咱只看它们能够带来啥后果，当然所谓的后果要体现在 sysfs 上，所以去 look 一下/sys/class 目录

```
atm         dma graphics  hwmon          i2c-adapter     input
mem         misc net   pci_bus          scsi_device     scsi_disk
scsi_host sound      spi_host   spi_master      spi_transport   tty
usb_device      usb_endpoint  usb_host  vc          vtconsole
```

想当初，usb 子系统初始化的时候，调用了一次 class_create(THIS_MODULE, "usb_host")，然后上边儿就多了一个 usb_host 目录，那么现在调用这个 class_register，上边儿又会多出什么？这个大家一眼就能看出来，即使一眼看不出来两眼也能看出来了，赚钱买猪肉的本事没有，寻找这种敏感地带的本事还都是有的，凭空多出来的就是那个 pci_bus。从这点儿看，create 还是 register 对咱们来说都差不多，都是在/sys/class 下边儿创建了一个类，usb_host 类的目录里是各个具体的主机控制器，pci_bus 类的目录里对应的就是各个 pci 总线了。本来难得糊涂一下明白这些就成了，不过如果真想稍微不那么糊涂一点儿，可以去扫两眼 class_create 的定义，你就会发现它里面最终也会调用一个 class_register，这两个的差别就是 class_create 要更傻瓜一些，你指定个类的名称就可以调用它了，它里面会帮你创建一个 struct class 结构体，而 class_register 则更费事一些，你需要自己亲自动手创建一个 struct class 结构体。如果你觉得自己挺特殊，需要指定自己的 release 函数等，那就必须得使用 class_register 了，PCI 就属于这种情况，至于它怎么个特殊，就是后话了。

pcibus_class_init 之后，接着就应该是 pci_driver_init

```
542 struct bus_type pci_bus_type = {
543             .name               = "pci",
544             .match              = pci_bus_match,
545             .uevent             = pci_uevent,
546             .probe              = pci_device_probe,
547             .remove             = pci_device_remove,
548             .suspend            = pci_device_suspend,
549             .suspend_late       = pci_device_suspend_late,
550             .resume_early       = pci_device_resume_early,
551             .resume             = pci_device_resume,
552             .shutdown           = pci_device_shutdown,
553             .dev_attrs          = pci_dev_attrs,
554 };
555
```

```
556 static int __init pci_driver_init(void)
557 {
558             return bus_register(&pci_bus_type);
559 }
560
561 postcore_initcall(pci_driver_init);
```

还记得 linux 设备模型里存在于总线、设备、驱动之间的那个著名的三角关系么？如果不记得，那就先听俺讲个小故事：

话说多少年以前有个人非常的健忘，他老婆很无奈，就对他说：“听说南村的谁谁谁专治女性不孕男性健忘，你还是去找他医一下吧！”好男人准则第一条就是要听老婆的话，于是这个人就背上弓箭，骑上马出发了。人不是都有三急么，半道儿上他想大便，就把马拴在一棵大树上，躲在树后，顺手把箭插在地上。方便过后，正在顺爽，顺爽，一顺再顺，顺出新自我，忽然看见了地上的箭，惊出了一身冷汗，“好险，不知谁射来的箭，差点要了我的小命！”紧赶的往外跑，一脚踩在大便上，不禁连皱眉头，大骂不已：“谁这么缺德，不讲公共卫生，在这里随地大小便……”。等到看到拴在树上的马，又高兴起来，心想：虽然吃了点儿苦头，捡到一匹马着实得美，就像金帝美滋滋巧克力，全新的色彩，全新的味道。于是他骑上马晕晕乎乎不知所之，沿着原路折了回去。一边想着我这是在哪儿呢，一边瞧见了一座房子。“咦，这房子好生面熟？……”这个时候，他老婆正从屋里看见他糊里糊涂的样子，气不打一处来，出门儿来责备他。只见他不卑不亢的作了一个揖，说：“这位大嫂，你我素昧平生，何苦出言不逊？”

这个故事的教育意义就在于告诉我们健忘是一种病态，善忘是一种境界，做人不能健忘到如此地步。那个三角关系中的总线落实在 USB 就是 usb_bus_type，落实在 PCI 就是上面的 pci_bus_type，pci_driver_init 函数的目的就是注册 PCI 总线，只有总线存在了，才会有设备的那条链表和驱动的那条链表，才会有设备和驱动之间的 match。

初始化(二)

.initcall2.init 子节中的两个函数已经见识过了，该轮到.initcall3.init 子节里的了，就是上边儿表中的 acpi_pci_init 和 pci_access_init，这两个又是谁先谁后那？acpi_pci_init 在 drivers/pci/pci-acpi.c 文件里，而 pci_access_init 在 arch/i386/pci/init.c 文件里，它俩根本就不在同一个目录下面，所以前边儿判断 pcibus_class_init 和 pci_driver_init 的顺序的技巧并不适用，那有什么方法可以让咱们找出它们的顺序？看看王冉怎么说：“昨天是五一劳动节，可是全国都在放大假绝大多数人不劳动。可见，庆祝一件事的最好的方法就是不去做这件事。譬如，庆祝世界杯的最好的方式就是不去参加世界杯——中国队几乎一直都是这么做的。再譬如，庆祝情人节的最好的方式就是不去找情人——于是，很多中国的男人把情人节的前一天（2 月 13 日）过成了情人节。”按他这说法，认清这俩函数之间顺序的最好方法就是不去管它们的顺序，俺可以点兵点将的随便点一个出来先说，不过作为一个很清楚自己责任和使命的 80 后，俺还是决定去发掘一下它们的顺序。

其实这个问题可以转化为 arch/i386/pci 下面的 Makefile 和 drivers/pci 下面的 Makefile 谁先谁后的问题，往大的方面说，就是内核是怎么构建的，也就是 kbuild 的问题。内核里的 Makefile 主要有三种：第一种是根目录里的 Makefile，它虽然只有一个，但地位远远凌驾于其它 Makefile 之上，里面定义了所有与体系结构无关的变量和目标；第二种是 arch/*/Makefile，

看到 arch 就知道它是与特定体系结构相关的，它包含在根目录下的 Makefile 中，为 kbuild 提供体系结构的特定信息，而它里面又包含了 arch/*/下面各级子目录的那些 Makefile；第三种就是密密麻麻躲在 drivers/等各个子目录下边儿的那些 Makefile 了。而 kbuild 构建内核的过程中，是首先从根目录 Makefile 开始执行，从中获得与体系结构无关的变量和依赖关系，并同时从 arch/*/Makefile 中获得体系结构特定的变量等信息，用来扩展根目录 Makefile 所提供的变量。此时 kbuild 已经拥有了构建内核需要的所有变量和目标，然后，Make 进入各个子目录，把部分变量传递给子目录里的 Makefile，子目录 Makefile 根据配置信息决定编译哪些源文件，从而构建出一个需要编译的文件列表。然后，然后还有很漫长的路，你编译内核要耗多久，它就有多漫长，不过说到这儿前面问题的答案就已经浮出水面了，很明显，arch/i386/pci 下面的 Makefile 是处在 drivers/pci 下面的 Makefile 前面的，也就是说，pci_access_init 处在 acpi_pci_init 前面，所以接下来先看看 pci_access_init。

```
5 /* arch_initcall has too random ordering, so call the initializers
6    in the right sequence from here. */
7 static __init int pci_access_init(void)
8 {
9       int type __maybe_unused = 0;
10
11 #ifdef CONFIG_PCI_DIRECT
12      type = pci_direct_probe();
13 #endif
14 #ifdef CONFIG_PCI_MMCONFIG
15      pci_mmcfg_init(type);
16 #endif
17      if (raw_pci_ops)
18          return 0;
19 #ifdef CONFIG_PCI_BIOS
20      pci_pcbios_init();
21 #endif
22      /*
23       * don't check for raw_pci_ops here because we want pcbios as last
24       * fallback, yet it's needed to run first to set pcibios_last_bus
25       * in case legacy PCI probing is used. otherwise detecting peer busses
26       * fails.
27       */
28 #ifdef CONFIG_PCI_DIRECT
29      pci_direct_init(type);
30 #endif
31      if (!raw_pci_ops)
32          printk(KERN_ERR
33          "PCI: Fatal: No config space access function found\n");
34
35      return 0;
36 }
37 arch_initcall(pci_access_init);
```

这个函数的主要工作是根据你配置内核时做的那道单项选择题“PCI access mode”来进行一些初始化，那几对儿#ifdef…#endif 就是专为你的答案准备的，你可能注意到它们里面只有 DIRECT、MMCONFIG、BIOS，没有 ANY 啊，那选 ANY 的时候咋办？如果仔细看了下前面贴的 arch/i386/Kconfig 里的与 PCI access mode 有关的那一段你就知道，选 ANY 就相当于其它三个一块儿选了。

9 行，定义一个变量 type，问题不在 type，在__maybe_unused。__maybe_unused 在内核里定义为__attribute__((unused))，而__attribute__也不是遇到一次两次了。Linux 内核代码使用了大量的 GNU C 扩展，以至于 GNU C 成为能够编译内核的唯一编译器，GNU C 的这些扩展对代码优化、目标代码布局、安全检查等方面也提供了很强的支持。__attribute__就是这些扩展中的一个，它主要被用来声明一些特殊的属性，这些属性被用来指示编译器进行特定方面的优化和更仔细的代码检查。GNU C 支持十几个属性，讲 USB Core 时已经遇到的有 section、packed、aligned 等，这里的 unused 也是这些属性中的一个，它用于函数和变量，表示该函数或变量可能不使用，可以在编译时防止编译器产生“变量未使用”这样的警告信息。这里用在 type 身上，就是说 type 虽然是在这儿定义了，但未必会用到。什么时候用不到它？你单选 BIOS 的时候，内核在 20 行调用了 pci_pcbios_init 之后就跑到 31 行去了，这个过程没 type 什么事儿。

11 行，如果选了 DIRECT，就调用 arch/i386/pci/direct.c 里的 pci_direct_probe 函数

```
268 int __init pci_direct_probe(void)
269 {
270     struct resource *region, *region2;
271
272     if ((pci_probe & PCI_PROBE_CONF1) == 0)
273         goto type2;
274     region = request_region(0xCF8, 8, "PCI conf1");
275     if (!region)
276         goto type2;
277
278     if (pci_check_type1())
279         return 1;
280     release_resource(region);
281
282 type2:
283     if ((pci_probe & PCI_PROBE_CONF2) == 0)
284         return 0;
285     region = request_region(0xCF8, 4, "PCI conf2");
286     if (!region)
287         return 0;
288     region2 = request_region(0xC000, 0x1000, "PCI conf2");
289     if (!region2)
290         goto fail2;
291
292     if (pci_check_type2()) {
293         printk(KERN_INFO "PCI: Using configuration type 2\n");
294         raw_pci_ops = &pci_direct_conf2;
```

```
295         return 2;
296     }
297
298     release_resource(region2);
299 fail2:
300     release_resource(region);
301     return 0;
302 }
```

因为 Direct 方式又分为了 conf1 和 conf2 两种，所以 pci_direct_probe 函数要进行一下检测，判断要使用哪一种，如果你在内核参数里没有强行指定是 conf1 还是 conf2 的话，它会首先检测一下 conf1，如果成了就直接返回不再检测 conf2 了，如果 conf1 不成，再去检测 conf2。conf2 现在一般都不会再用了，从 PCI Spec v2.2 那时候开始就已经将它给扔到高粱地里去了，这里还留着它主要是为了兼容很久很久以前的一些老主板，所以下面只会针对 conf1 去说事儿。

要搞清楚 pci_direct_probe 函数的检测过程，得明白这么几点。第一个就是 struct resource。处理器和设备进行交流主要是通过两个空间，I/O 空间或者内存空间，某些处理器两个空间都有，某些却只有一个内存空间，这两个空间的差别引出了设备那些寄存器映射方式的差别，映射到处理器的 I/O 空间时，就成了 I/O 端口，映射到内存空间时，就成 I/O 内存，不过不管是 I/O 端口还是 I/O 内存，在 linux 里都属于宝贵的 I/O 资源，都要使用 include/linux/ioport.h 文件里的 struct resource 结构来描述。

```
13 /*
14  * Resources are tree-like, allowing
15  * nesting etc..
16  */
17 struct resource {
18     resource_size_t start;
19     resource_size_t end;
20     const char *name;
21     unsigned long flags;
22     struct resource *parent, *sibling, *child;
23 };
```

每一个 struct resource 结构体都表示了一段独立的连续地址区间，name 代表了这段区间也就是这个资源的名称，start 和 end 描述了它的地址范围，parent、sibling 和 child 用来维持一个资源的树形结构，child 指向它的第一个子区间，同一资源的所有子区间通过 sibling 形成一个单向链表，且都通过 parent 来指向它们的父区间。flags 描述了一个资源的种类和属性，同样都在 ioport.h 里定义，比如

```
39 #define IORESOURCE_IO       0x00000100      /* Resource type */
40 #define IORESOURCE_MEM      0x00000200
41 #define IORESOURCE_IRQ      0x00000400
42 #define IORESOURCE_DMA      0x00000800
```

分别代表了 IO，Memory，中断，DMA 四种资源类型，对应了/proc 下面的 ioports，iomem，interrupt，dma 四个文件。不过不管是哪一种资源，只要是资源，在内核里就都是要先提出申请，得到批准后才能使用的，这点儿不能拿咱们的生活经验往上套，就说前档子上海某区的那桩高压线事件吧，22 万伏高压线能不能入地，占不占你地盘儿不是你小区平头小百姓

说了算的，是由 xx 公司和防暴警察说了算的，他们不用去获得你的批准就照样占用你小区的资源，将高压线铁塔矗立到你旁边儿，管你什么绿地不绿地辐射不辐射，当然，他们虽然没有得到老百姓的批准，但有没有得到其它什么批准就谁也不知道了。内核里就不一样，什么都得放到明处，你想使用哪种资源就要明确的提出申请，申请的函数对于 I/O 端口来说是 request_region，对于 I/O 内存来说就是 request_mem_region，如果你的申请得到批准了，它们就会返回一个 struct resource 结构体，如果没有得到批准，返回的就是一个冷冰冰的 NULL。就比如说已经说过的 UHCI 吧，UHCI 的 spec 规定，UHCI 的那些寄存器智能映射到 I/O 空间，使用 request_region 函数申请成功之后，look 一下/proc/ioports 文件

```
localhost:~ # cat /proc/ioports
0000-001f : dma1
0020-0021 : pic1
0cf8-0cff : PCI conf1
(此处省略若干行)
bca0-bcbf : 0000:00:1d.2
 bca0-bcbf : uhci_hcd
bcc0-bcdf : 0000:00:1d.1
 bcc0-bcdf : uhci_hcd
bce0-bcff : 0000:00:1d.0
 bce0-bcff : uhci_hcd
c000-cfff : PCI Bus #10
 cc00-ccff : 0000:10:0d.0
d000-dfff : PCI Bus #0e
 dcc0-dcdf : 0000:0e:00.1
    dcc0-dcdf : e1000
 dce0-dcff : 0000:0e:00.0
    dce0-dcff : e1000
e000-efff : PCI Bus #0c
 e800-e8ff : 0000:0c:00.1
    e800-e8ff : qla2xxx
 ec00-ecff : 0000:0c:00.0
    ec00-ecff : qla2xxx
fc00-fc0f : 0000:00:1f.1
 fc00-fc07 : ide0
```

就可以看到里面的“uhci_hcd”。对于 pci_direct_probe 函数来说，在 274 行也使用了 request_region(0xCF8, 8, "PCI conf1")来申请一个起始地址为 0cf8-0cff 的 I/O 端口资源，上面/proc/ioports 文件里的“PCI conf1”也反映出了这一点。

关于 struct resource，剩下的一点疑问是，它描述的是物理地址？还是逻辑地址？还是其它的什么地址？绕来绕去的地址种类有很多，有些可能就根本没什么大的区别，就像一块儿走在 T 台上的芙蓉姐姐和杨二车那姆一样，俺看了很久愣是没看出谁是谁又有啥区别，但你不能就这么说她们是一个样儿，所以还是挨个儿看一下，希望不会被绕进去。

与咱们最贴近的是一个“用户虚拟地址”，是用户空间所能看到的地址，每个进程都有这么一个虚拟地址空间。然后是耳熟能详的“物理地址”，在处理器和系统内存之间使用的地址。接着是“总线地址”，是处理器在总线上所看到的地址。第四个是“内核逻辑地址”，这些地址组成了常规的内核地址空间，映射了大部分乃至全部的系统主内存，被视为物理地址使用，在

大多数的体系结构中，逻辑地址及其所关联的物理地址之间的区别，仅仅在于一个常数的偏移量，在拥有大量内存的 32 位机上，仅通过逻辑地址未必能够寻址所有的物理内存。最后一个是“内核虚拟地址”，和逻辑地址的区别在于前者不一定会直接映射到物理地址，所有的逻辑地址都可看成是内核虚拟地址。

对内核来说，总线地址、逻辑地址都可以当物理地址来用，其实对于 CPU 来说，总线地址就相当于物理地址，为外设分配地址就是来分配总线地址，分配得到的地址一般都不能直接使用，还要通过 ioremap 映射为内核虚拟地址之后使用专门的接口来访问。所以你说 struct resource 描述的是逻辑地址也对，物理地址也没什么不妥。

关于 pci_direct_probe 函数，要明白的第二点就是调用 request_region 时使用的参数 0xCF8。凡是 PCI 设备不都是有个配置寄存器组么，要访问这些寄存器，不是通过 I/O 端口就是通过 I/O 内存。如果通过 I/O 内存的方式，将它们统统映射到内存空间，那一千种设备需要多大空间？一万种那？大量的内存将被占用，从而不能用在革命最需要的地方，所以说这很不符合节约型社会的要求。那就只有通过 I/O 端口了，不过 I/O 端口是非常稀缺的，X86 上只有 64K 这么多，将每个配置寄存器都映射到 I/O 空间显然也是不现实的，现实一点的做法是为所有设备的配置寄存器组都采用相同的 I/O 端口，CPU 则通过这个统一的 I/O 端口向主桥发出命令，再由各个 PCI 桥添加一些附加条件来间接的完成具体的读写操作。对于 32 位的 X86 架构来说，有两个 32 位的 I/O 端口用于这个目的，就是地址端口 0cf8 和数据端口 0cfc。访问某个设备的配置寄存器时，CPU 向地址端口写入目标的地址，然后通过数据端口读写数据。写入地址端口 0cf8 里的是一个包括总线号、设备号、功能号以及配置寄存器地址在内的综合地址。

最高位为 1	保留不用（7 位）	总线号（8 位）	设备号（5 位）	功能号（3 位）	寄存器地址（8 位）最低两位为 0

这里的总线号、设备号、功能号就是上面说的所谓的附加条件，寄存器地址指的就是该寄存器在配置寄存器组里的偏移，最低两位为 0 那是因为配置寄存器组里的那些寄存器都是双字为单位的，内核在 include/linux/pci_regs.h 里定义了大量的宏来表示这个偏移地址。

知道了 0cf8 和 0cff 这两个统一的 I/O 端口，也使用 request_region 成功的申请到了端口资源，然后就可以使用这两个 I/O 端口与各种 PCI 设备眉来眼去了么？事情没这么简单，你是申请到了 0cf8 和 0cff，但这并不说明系统里就有 PCI 总线了，所以还要不可避免地做一下验证，对于 conf1 这个验证使用的是 arch/i386/pci/direct.c 里的 pci_check_type1，这也是咱们要明白的第三点。

```
217 static int __init pci_check_type1(void)
218 {
219     unsigned long flags;
220     unsigned int tmp;
221     int works = 0;
222
223     local_irq_save(flags);
224
225     outb(0x01, 0xCFB);
226     tmp = inl(0xCF8);
227     outl(0x80000000, 0xCF8);
228     if (inl(0xCF8) == 0x80000000 && pci_sanity_check(&pci_direct_conf1)) {
229         works = 1;
230     }
```

```
231     outl(tmp, 0xCF8);
232     local_irq_restore(flags);
233
234     return works;
235 }
```

不过这个要明白的第三点还真是让人不怎么明白，在＃￥％×……的做了一番验证后（＃￥％×……所代表的意义对俺来说至今还是个秘密，不过有一点是明确的，如果 0cf8 确实是主桥的地址端口，则读时候，返回的肯定是先前写到里边儿的值，所以 inl(0xCF8) == 0x80000000 就是判断 0cf8 是不是真的是当作主桥的地址端口用的。如果不相等，则肯定不是，conf1 将不起作用，如果相等了，才说明 0cf8 极有可能就是主桥的地址端口，但也有可能只是凑巧，所以要接着调用 pci_sanity_check 做更深入的验证），又调用了 pci_sanity_check 函数，使用 pci_direct_conf1 做进一步的验证。

```
179 /*
180  * Before we decide to use direct hardware access mechanisms, we try to do some
181  * trivial checks to ensure it at least _seems_ to be working -- we just test
182  * whether bus 00 contains a host bridge (this is similar to checking
183  * techniques used in XFree86, but ours should be more reliable since we
184  * attempt to make use of direct access hints provided by the PCI BIOS).
185  *
186  * This should be close to trivial, but it isn't, because there are buggy
187  * chipsets (yes, you guessed it, by Intel and Compaq) that have no class ID.
188  */
189 static int __init pci_sanity_check(struct pci_raw_ops *o)
190 {
191     u32 x = 0;
192     int devfn;
193
194     if (pci_probe & PCI_NO_CHECKS)
195         return 1;
196     /* Assume Type 1 works for newer systems.
197        This handles machines that don't have anything on PCI Bus 0. */
198     if (dmi_get_year(DMI_BIOS_DATE) >= 2001)
199         return 1;
200
201     for (devfn = 0; devfn < 0x100; devfn++) {
202         if (o->read(0, 0, devfn, PCI_CLASS_DEVICE, 2, &x))
203                 continue;
204         if (x == PCI_CLASS_BRIDGE_HOST || x == PCI_CLASS_DISPLAY_VGA)
205                 return 1;
206
207         if (o->read(0, 0, devfn, PCI_VENDOR_ID, 2, &x))
208                 continue;
209         if (x == PCI_VENDOR_ID_INTEL || x == PCI_VENDOR_ID_COMPAQ)
210                 return 1;
```

```
211         }
212
213         DBG(KERN_WARNING "PCI: Sanity check failed\n");
214         return 0;
215 }
```

参数里的 struct pci_raw_ops 结构体定义在 include/linux/pci.h 里

```
294 struct pci_raw_ops {
295         int (*read)(unsigned int domain, unsigned int bus, unsigned int devfn,
296                     int reg, int len, u32 *val);
297         int (*write)(unsigned int domain, unsigned int bus, unsigned int devfn,
298                      int reg, int len, u32 val);
299 };
```

和那个著名的 file_operations 属于近亲关系，采用类似的手法，封装了对 PCI 设备配置寄存器的读写操作。PCI 设备的访问方式有 BIOS、DIRECT 的 conf1 和 conf2、MMConfig 多种，因此对应的 struct pci_raw_ops 结构体也就有多个，与 conf1 相依相生的就是 direct.c 中的 pci_direct_conf1

```
10 /*
11 * Functions for accessing PCI configuration space with type 1 accesses
12 */
13
14 #define PCI_CONF1_ADDRESS(bus, devfn, reg) \
15         (0x80000000 | (bus << 16) | (devfn << 8) | (reg & ~3))
16
17 int pci_conf1_read(unsigned int seg, unsigned int bus,
18                            unsigned int devfn, int reg, int len, u32 *value)
19 {
20         unsigned long flags;
21
22         if ((bus > 255) || (devfn > 255) || (reg > 255)) {
23             *value = -1;
24             return -EINVAL;
25         }
26
27         spin_lock_irqsave(&pci_config_lock, flags);
28
29         outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);
30
31         switch (len) {
32         case 1:
33             *value = inb(0xCFC + (reg & 3));
34             break;
35         case 2:
36             *value = inw(0xCFC + (reg & 2));
37             break;
```

```
38        case 4:
39            *value = inl(0xCFC);
40            break;
41        }
42
43        spin_unlock_irqrestore(&pci_config_lock, flags);
44
45        return 0;
46 }
47
48 int pci_conf1_write(unsigned int seg, unsigned int bus,
49                             unsigned int devfn, int reg, int len, u32 value)
50 {
51        unsigned long flags;
52
53        if ((bus > 255) || (devfn > 255) || (reg > 255))
54            return -EINVAL;
55
56        spin_lock_irqsave(&pci_config_lock, flags);
57
58        outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);
59
60        switch (len) {
61        case 1:
62            outb((u8)value, 0xCFC + (reg & 3));
63            break;
64        case 2:
65            outw((u16)value, 0xCFC + (reg & 2));
66            break;
67        case 4:
68            outl((u32)value, 0xCFC);
69            break;
70        }
71
72        spin_unlock_irqrestore(&pci_config_lock, flags);
73
74        return 0;
75 }
76
77 #undef PCI_CONF1_ADDRESS
78
79 struct pci_raw_ops pci_direct_conf1 = {
80        .read =                 pci_conf1_read,
81        .write =        pci_conf1_write,
```

82 };
14 行，PCI_CONF1_ADDRESS 宏用来计算向地址端口 0cf8 中写入的目标地址值，要注意，devfn 表示的不仅仅是设备号，还有功能号，前面说从逻辑的角度讲，功能又可以称之为逻辑设备，那么设备号和功能号合起来也可以叫做逻辑设备号。另外，虽然各个配置寄存器都是以双字为单位，但它们都可以按照字节、16 位字、32 位双字来读写，所以 reg 的最低两位并不一定为 0，为了满足向地址端口 0cf8 的构成，得通过 reg & ~3 将最低两位置为 0。
22 行，几乎每个函数都有的例行检查，总线号只有 8 位，逻辑设备号 5 加 3 也有 8 位，寄存器地址同样也只有 8 位，所以它们都不能够大于 255。
27 行，要操作公用的端口 0cf8 和 0cfc 了，所以得加把锁。社会经验不成文法则第 2008 条：占用公用资源时请加把锁。别让别人瞅见了。
29 行，计算出目标设备和寄存器地址，写入端口 0cf8。
31 行，len 为读取的字节数，因为对配置寄存器可以按照字节、16 位字、32 位双字来读写，所以 switch 有三个 case，因为配置寄存器都是 little-endian 的，所以 case 1 时 0xCFC + (reg & 3)，case 2 时 0xCFC + (reg & 2)。
48 行，write 和 read 的区别只是 in 和 out 的区别，飘过。
现在回过头看看 pci_sanity_check 都利用 pci_direct_conf1 干了些什么，首先判断是不是设置了 PCI_NO_CHECKS。PCI_NO_CHECKS 意思是说不再用去验证了， conf1 是一个值得信赖的好同志，是可以为革命服务的。在解析内核参数的时候，如果发现用户指明了使用 conf1 或者 conf2，就会同时设置 PCI_NO_CHECKS。
然后会使用 for 循环遍历所有可能的功能，也就是逻辑设备，去寻觅那个唯一的主桥。先是分别使用 conf1 去读它们的 PCI_CLASS_DEVICE 寄存器，如果读到的值为 PCI_CLASS_BRIDGE_HOST 和 PCI_CLASS_DISPLAY_VGA 之一，就说明主桥是存在的，如果两个都不是，需要再进一步去读 PCI_VENDOR_ID 寄存器，如果读到的值为 PCI_VENDOR_ID_INTEL 或者 PCI_VENDOR_ID_COMPAQ，同样可以说明主桥是存在的，并且是 Intel 或者 Compaq 制造的，因为这两家的一些产品没有 class ID，自然也读出来的 PCI_CLASS_DEVICE 寄存器值是 trivial 的。由此看来，for 的过程就是一个寻寻觅觅的过程，如果众里寻它千百度，主桥仍然不在灯火阑珊处，并不就说明主桥不存在，还有种可能是 conf1 根本就没起作用。
本来 pci_sanity_check 这样就已经 perfect 了，但是在一年多以前，一个小伙儿提出了抗议，他发现在 Horus 系统上（采用了 Newisys 为 AMD 设计的 Horus 芯片，从而能够在服务器中使用高达 32 个 AMD Opteron 处理器），使用 pci_sanity_check 验证 conf1 验证总是失败，因为 PCI 总线 0 上啥也没有，于是他就添了 196~199 这么几行专门针对这种情况进行处理。原话是这样的：

Horus systems don’t have anything on bus 0 which makes the Type 1 sanity checks fail. Use the DMI BIOS year to check for newer systems and always assume Type 1 works on them. I used 2001 as an pretty arbitary cutoff year.

这里涉及到一个新概念，DMI（Desktop Management Interface），而看到 DMI，你就不能不想到 SMBIOS（System Management BIOS）。SMBIOS 是主板厂商显示序列号、电池型号、网卡型号等等系统管理信息时所必须遵守的一套规范，DMI 就是用来访问、收集这些信息的接口，简单点说，SMBIOS 和 DMI 之间的关系就是规范和访问接口的关系。DMI 信息存储在 BIOS ROM 的一段数据区里，江湖人称 MIFD（Management Information Format Database），启动的时候，BIOS 会将它们拷贝到内存里，这样，查询 CPU、内存等在内的系统配置信息时就不用再进入 BIOS。
这么说吧，你在网上凭借自己的三寸不烂之舌获得了一个 mm 的信任，偷偷远程登陆到她

的机子上，想查看一下她电脑的一些系统信息然后再神通广大的样子告诉她从而显示自己是多么的 powerful，这时咋办？一是打电话或者网上直接问她，这太弱了点，她会直接给你说“有病啊你”，二就是通过 DMI 去查，在 linux 上，可以通过 dmidecode，执行 dmidecode –t type 就可以得尝所愿了，type 代表的是获取信息的类型，不过这年头儿好像 mm 用 linux 的不多，linux 社区里可是严重的男女比例失衡，搞 linux 搞内核很难会博得 ppmm 的崇拜，可要有心理准备啊同志们。