华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象
华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《Linux 那些事儿之我是 USB》

作者：华清远见

专业始于专注 卓识源于远见

第 1 章 Linux 那些事儿之我是 USB Core

1、引子

老夫子们痛心疾首地总结说，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，俺不幸地混进了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。

为了说明我也是有主义、有信仰的，也是经历过楼市股市狂风暴雨考验的，这里就讲一讲USB，让他们看一看80后不仅仅知道什么是网恋，什么是异性同居，怎么靠上半身上位，怎么用下半身写作，还知道什么叫USB。

还是要说在前面，在这里耗费二八青春码这些，并不是因为喜欢它，相反，对它是毫无感觉可言，虽然每天都必须和它相依为伴，不离不弃，不过那可是丝毫没有办法的事情，非我所愿。是不是特说到心坎里去了？人生不如意十之八九，感情都很无奈是吧，不过您别多想，咱这里只谈USB，至于风花雪月的那些事儿咱再找机会私下唠。

一句话总结：哥写的不是USB，是寂寞。

2、它从哪里来

“你从哪里来，我的朋友，好像一只蝴蝶，飞进我的窗口。”

在毛阿姨的嘹亮歌声中，USB好像一只蝴蝶飞进了千家万户。它从哪里来，它从Intel来。Intel不养蝴蝶，而是做CPU，它只是在蝴蝶的翅膀上烙上“Intel inside”，蝴蝶让咱们的同胞去养了，然后带着Intel飞进了千万家。

不过，与PCI、AGP属于Intel单独提出的硬件标准不同，Compaq、IBM、Microsoft等也一起参与了USB这个游戏。他们一起于1994年11月提出了USB，并于1995年11月制定了0.9版本，1996年制定了1.0版本。不过USB并没有因为有这些大佬的支持立即迎来它的春天，只怪它诞生在了冬季，生不逢时啊！

因为缺乏操作平台的良好支持和大量支持它的产品，这些标准都成了空谈。1998年USB1.1的出现，忽如一夜春风来，它就像春天里的一朵油菜花，终于涂上了浓重的一抹黄色。

为什么要开发USB？

不过咱们这里的问题没有那么复杂，同样无关政治民生，关乎的只是咱们的需要。USB出现以前，电脑的接口处于春秋战国时代，串口并口等多方割据，键盘、鼠标、Modem、打印机、扫描仪等都要连接在这些不同种类的接口上，一个接口只能连接一个设备。不过咱们的电脑不可能有那么多这种接口，所以扩展能力不足，而且它们的速度也确实很有限。还有关键的一点是，热插拔对它们来说也是比较危险的操作，不想用了都成黄脸婆了还不能立即换掉，不能满足众多男人们的内心潜在需要。

USB正是为了解决速度、扩展能力、易用性等问题应景而生的。

3、PK

在2006，最火的是超级女生，最流行的词是“PK”。“她的一生充满了PK”——从湖南卫视在《大长今》预告片中铿锵地说出了这句旁白时起，“PK”已经不仅仅是“PK”。

USB的一生也充满了PK，不过USB还不够老，说一生还太早了，发哥说的好：“我才刚上路呢！”

USB最初的设计目标就是替代串行、并行等各种低速总线，以达到以一种单一类型的总线连接各种不同的设备。它现在几乎可以支持所有连接到PC上的设备，1999年提出的USB 2.0理论上可以达到480 Mb/s的速度，2008年公布的USB 3.0标准更是提供了十倍于USB 2.0的传输速度。

因此，USB与串口、并口等的这场PK从一开始就是不平等的，这样的开始也注定了以什么样的结果结束，只能说命运选择了USB。我们很多人都说命运掌握在自己手里，但是从USB充满PK的一生中，可以知道，

华清远见

只有变得比别人更强，命运才能掌握在自己手里。

有了USB在这场PK中的大获全胜，才有了USB键盘、USB鼠标、USB打印机、USB摄像头、USB扫描仪、USB音箱等。就像有了李宇春在“超女”PK中的胜利，才有了李宇春的蒙牛绿色心情。至于将来，“PK自己的，让别人去说吧！”USB如是说。

4、漫漫辛酸路

USB的一生充满了PK，并在PK中发展，从1.0、1.1、2.0到3.0，漫漫辛酸路，一把辛酸泪。

USB 2.0的高速模式（High-Speed）最高已经达到了480 Mb/s，即60 Mb/s，也就是说，照这个速度，你将自己从网上下载的小短片备份到自己的移动硬盘上的时间长约一秒钟。而USB 3.0的Super-Speed模式比这个还要提高了几乎10倍，达到了4.8G b/s。

USB走过的这段辛酸路，对咱们来说最直观的结果也就是传输速度提高了，过程很艰辛，结果很简单，是不？

USB的各个版本都是兼容的。每个USB 2.0 控制器带有3个芯片，根据设备的识别方式将信号发送到正确的控制芯片。我们可以将USB 1.1设备连接到USB 2.0的控制器上使用，不过它只能达到USB 1.1的速度。同时也可以将USB 2.0的设备连接到USB 1.1的控制器上，不过不能指望它能以USB 2.0的速度运行。毕竟走过的路太辛酸了，没有那么快就能忘掉，好像我们不时地要去交大门口的老赵烤肉店忆苦思甜一样，我们不能忘本，USB也不能。

显然，Linux对USB1.1和USB 2.0都是支持的，并抢在Windows前面，在2.6.31内核中率先对USB 3.0进行了支持。

5、我型我秀

USB既然能一路PK走过来，也算是一个挺能秀的角色了，不然也不会有那么多的拥护者。

既然这里说的就是USB，也挑一些大家可能感兴趣的帮它秀一下。USB为所有的USB外设都提供了单一的标准的连接类型，这就简化了外设的设计，也让我们不用再去想哪个设备对应哪个插槽的问题，就像种萝卜，一个萝卜一个坑，但是哪个萝卜种到哪个坑里是不用我们关心的。

USB支持热插拔，而其他的比如SCSI设备等必须在关掉主机的情况下才能增加或移走外围设备。所以说，USB的一生不仅仅是PK的一生，也是丰富多彩的一生，不用实行一夫一妻制，可以不用关机就能更换不同种类的外设。

USB在设备供电方面提供了灵活性。USB设备可以通过USB电缆供电，不然咱们的移动硬盘、ipod什么的也用不了了。相对应，有的USB设备也可以使用普通的电源供电。

USB能够支持从几十KB到几十MB的传输速率，来适应不同种类的外设。它可以支持多个设备同时操作，也支持多功能的设备。多功能的设备当然指的就是一个设备同时有多个功能， 比如USB扬声器。这通过在一个设备中包含多个接口来支持，一个接口支持一个功能。

USB可以支持多达127个设备。

USB可以保证固定的带宽，这个对视频音频设备是利好。

最后，应该给USB的这场秀的一个评价：“我型我秀”造福了少数人还有电信、移动、联通，USB造福了全人类。

6、我是一棵树

华清远见

“我是一棵树，静静地站在田野里，风儿吹过，我不知它的去向，人儿走过，我不知谁会为我停留。”

如图3.6.1所示，USB子系统的拓扑也是一颗树，它并不以总线的方式来部署。

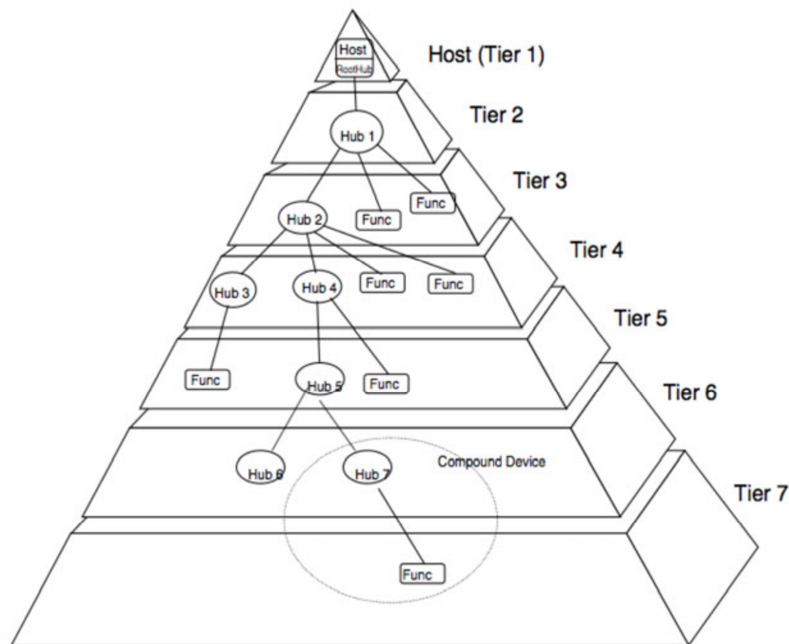


图 3.6.1 USB 子系统的树形结构

我曾经指着路边一棵老得奇形怪状的树问朋友：“这是什么树？”朋友的回答让我很晕：“大树。”那上面图里的是什么树？自然也是大树了，不过却是USB的大树。这棵大树主要包括了USB连接、USB Host Controller（USB主机控制器）和USB设备三个部分。而USB设备还包括了Hub和功能设备（也就是上图里的Func）。

什么是USB主机控制器？控制器，顾名思义，用于控制，控制什么？控制所有的USB设备的通信。通常计算机的CPU并不是直接和USB设备打交道，而是和控制器打交道。它要对设备做什么，它会告诉控制器，而不是直接把指令发给设备。然后控制器再去负责处理这件事情，它会去指挥设备执行命令，而CPU就不用管剩下的事情。控制器替他去完成剩下的事情，事情办完了再通知CPU。否则让CPU去盯着每一个设备做每一件事情，那是不现实的。

这就好比让一个学院的院长去盯着我们每一个本科生上课，去管理我们的出勤，这是不现实的。所以学生就被分成了几个系，通常院长有什么指示直接跟各系领导说就可以了，如果他要和三个系主任说事情，他即使不把三个人都召集起来开会，也可以给三个人各打一个电话，打完电话他就忙他自己的事情去了。而三个系主任就会去安排下面的人去执行具体的任务，完了之后他们就会向院长汇报。

那么Hub是什么？在大学里，有的宿舍里网口有限，但是我们这一代人上大学基本上是每人一台电脑，所以网口不够，于是有人会使用Hub，让多个人共用一个网口，这是以太网上的Hub。而USB的世界里同样有Hub，其实原理是一样的，任何支持USB的电脑不会说只允许你只能一个时刻使用一个USB设备，比如你插入了U盘，你同样还可以插入USB键盘，还可以再插一个USB鼠标，因为你会发现你的电脑里并不只是一个USB接口。这些接口实际上就是所谓的Hub口。

而现实中经常是让一个USB控制器和一个Hub绑定在一起，专业一点说叫集成，而这个Hub也被称作Root Hub。换而言之，和USB控制器绑定在一起的Hub就是系统中最根本的Hub，其他的Hub可以连接到它这里，然后可以延伸出去，外接别的设备，当然也可以不用别的Hub，让USB设备直接接到Root Hub上。

而USB连接指的就是连接USB设备和主机（或Hub）的四线电缆。电缆中包括VBUS（电源线）、GND（地线）还有两根信号线。USB系统就是通过VBUS和GND向USB设备提供电源的。主机对连接的USB设备提供电源供其使用，而每个USB设备也能够有自己的电源，如图3.6.2所示。

华清远见

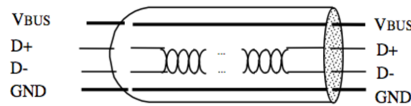


图 3.6.2　USB 四线电缆

现在，如图3.6.1所示的USB大树里只有Compound Device还没有说。那么，Compound Device又是什么样的设备？其实，在USB的世界里，不仅仅有Compound Device，还有Composite Device，简单的中文名字已经无法形象地表达它们的区别。正如图3.6.3所示，Compound Device是那些将Hub和连在Hub上的设备封装在一起所组成的设备。而Composite Device则是包含彼此独立的多个接口的设备。从主机的角度看，一个Compound Device和单独的一个Hub然后连接了多个USB设备是一样的，它里面包含的Hub和各个设备都会有自己独立的地址，而一个Composite Device里不管具有多少接口，它都只有一个地址。

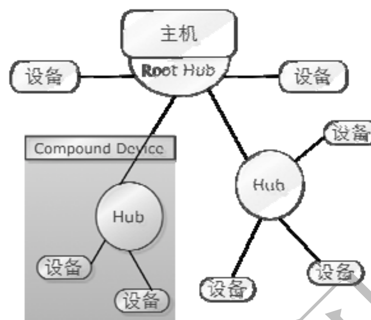


图 3.6.3　Compound Device

USB大树要想茁长成长离不开USB协议。USB总线是一种轮询方式的总线。协议规定所有的数据传输都必须由主机发起，由主机控制器初始化所有的数据传输，各种设备紧紧围绕在主机周围。

USB通信最基本的形式是通过USB设备中一个叫Endpoint（端点）的东西，而主机和端点之间的数据传输是通过Pipe（管道）。

端点就是通信的发送或者接收点，你要发送数据，那你只要把数据发送到正确的端点那里就可以了。而管道，实际上只是为了让我们能够找到端点，就相当于我们日常说的邮编地址。

比如一个国家，为了通信，我们必须给各个地方取名，然后给各条大大小小的路取名。严格来说，管道的另一端应该是USB主机，USB协议中边也是这么说的，协议中边说管道代表着一种能力，怎样一种能力呢，在主机和设备上的端点之间移动数据。

端点不但是有方向的，而且这个方向还是确定的，或者in，或者out，没有又是in又是out的，鱼与熊掌是不可兼得的，脚踩两只船虽然是每个男人的美好愿望，但不具可操作性，也不提倡。所以你到北京就叫上访，北京的下来就叫慰问，这都是生来就注定的。

有没有特殊的端点呢？看你怎么去理解0号端点了，协议中规定了，所有的USB设备必须具有端点0，它可以作为in端点，也可以作为out端点。USB系统软件利用它来实现默认的控制管道，从而控制设备。

端点也是限量供应的，不是想有多少就有多少的，除了端点0，低速设备最多只能拥有2个端点，高速设备也最多只能拥有15个in端点和15个out端点。这些端点在设备内部都有唯一的端点号，这个端点号是在设备设计时就已经指定的。

为什么端点0就那么有个性呢？这还是有内在原因的。管道的通信方式其实有两种，一种是stream的，一种是message的，message管道要求从它那儿过的数据必须具有一定的格式，不是随便传的，因为它主要就是用于主机向设备请求信息的，必须得让设备明白请求的是什么。而stream管道就没这么苛刻，随和多了，对数据没有特殊的要求。协议中说，message管道必须对应两个相同号码的端点，一个用来in，一个用来out，咱们的默认管道就是message管道，当然，与默认管道对应的端点0就必须是两个具有同样端点号0的端点。

华清远见

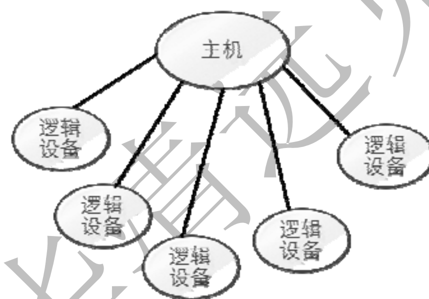
USB端点有四种类型，也就分别对应了四种不同的数据传输方式。它们是控制传输（Control Transfers），中断传输（Interrupt Data Transfers），批量传输(Bulk Data Transfers)，等时传输(Isochronous Data Transfers)。控制传输用来控制对USB设备不同部分的访问，通常用于配置设备，获取设备信息，发送命令到设备，或者获取设备的状态报告。总之就是用来传送控制信息的，每个USB设备都会有一个名为“端点0”的控制端点，内核中的USB Core使用它在设备插入时进行设备的配置。

中断传输用来以一个固定的速率传送少量的数据，USB键盘和USB鼠标使用的就是这种方式，USB的触摸屏也是，传输的数据包含了坐标信息。

批量传输用来传输大量的数据，确保没有数据丢失，但不保证在特定的时间内完成。U盘使用的就是批量传输，咱们用它备份数据时需要确保数据不能丢，而且也不能指望它能在一个固定的比较快的时间内复制完。

等时传输同样用来传输大量的数据，但并不保证数据是否到达，以稳定的速率发送和接收实时的信息，对传送延迟非常敏感。显然是用于音频和视频一类的设备，这类设备期望能够有一个比较稳定的数据流，比如你在用QQ视频聊天时，肯定希望每分钟传输的图像/声音速率是比较稳定的，不能说这一分钟前对方看到你在向她向你深情表白，可是下一分钟却看见画面停滞在那里，只能看到你那傻样一动不动，你说这不浪费感情吗！

如图3.6.1所示的树形结构描述的是实实在在的物理拓扑，对于内核中的实现来说，没有这么复杂，所有的Hub和设备都被看作是一个个的逻辑设备（Logical Device），如图3.6.4所示，好像它们本来就直接连接在Root Hub上一样。



华清远见

图 3.6.4 USB 逻辑拓扑结构

如图3.6.5所示，一个USB逻辑设备就是一系列端点的集合，它与主机之间的通信发生在主机上的一个缓冲区和设备上的一个端点之间，通过管道来传输数据。也就是说管道的一端是主机上的一个缓冲区，一端是设备上的端点。

那么图3.6.5中的接口又是指什么？简单地说，USB端点被捆绑为接口（Interface），一个接口代表一个基本功能。有的设备具有多个接口，像USB扬声器就包括一个键盘接口和一个音频流接口。在内核中一个接口要对应一个驱动程序，USB扬声器在Linux里就需要两个不同的驱动程序。到目前为止，可以这么说，一个设备可以包括多个接口，一个接口可以具有多个端点，当然以后我们会发现并不仅仅止于此。

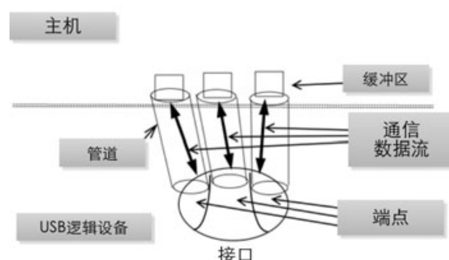


图 3.6.5 USB 数据通信

7、我是谁

我是谁？USB也一遍一遍问着自己，当然它不会真的是一颗树，它也不会是太阳，Linux里没有太阳，真要有的话也只能是Linus。USB子系统只是Linux庞大家族里的一个小部落，主机控制器是它们的族长，族里的每个USB设备都需要被系统识别，被我们识别，而sysfs就是它们对外的窗口，我们可以从sysfs里了解认识每一个USB设备。以一个仅包含一个USB接口的USB鼠标为例，如图3.7.1所示就是该设备对应得sysfs目录树。

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|   |-- bAlternateSetting
|   |-- bInterfaceClass
|   |-- bInterfaceNumber
|   |-- bInterfaceProtocol
|   |-- bInterfaceSubClass
|   |-- bNumEndpoints
|   |-- detach_state
|   |-- iInterface
|   `-- power
|       `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

图 3.7.1 USB 鼠标的 sysfs 目录树

其中：

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1

表示鼠标。下层目录：

/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0

表示了鼠标的USB接口。sysfs里USB设备都是类似的表示，设备的目录下包括了表示设备接口的目录。目录里的各个文件表示的设备或接口的描述，大都对应了设备描述符、接口描述符等的相应值，可以通过这些值获得感兴趣的信息。新名词就像任小强的理论一样层出不穷，什么是设备描述符还有接口描述符？我们这里要暂时忽略它的存在，先关心关心USB设备在sysfs里是如何命名的，弄清它是谁，也就是说上面路径的含义。

USB系统中的第一个USB设备是Root Hub，前面已经说了它是和主机控制器绑定在一起的。这个Root Hub通常包含在PCI设备中，是连接PCI总线和USB总线的bridge，控制着连接到其上的整个USB总线。所有的Root Hub，内核的USB Core都分配有独特的编号，在上面的例子里就是usb2。

USB总线上的每个设备都以Root Hub的编号作为其名字的第一个号码。这个号码后跟着一个“-”字符，以及设备所插入的端口号。因此，上面例子中的USB鼠标的设备名就是2-1。因为该USB鼠标具有一个接口，导致了另外一个USB设备被添加到sysfs路径中。因为物理USB设备和单独的USB接口在sysfs中都将表示为单独的设备。USB接口的命名是设备名直到该接口，上面就是2-1后面跟一个“:”和USB配置（Configuration）的编号，然后是一个“.”和该接口的编号。因此上面的鼠标USB接口就是2-1:1.0，表示使用的是第一个配置，接口编号为0。

sysfs并没有展示USB设备的所有部分，设备可能包含的可选配置都没有显示，不过这些可以通过usbfs找到，该文件系统被挂在到/proc/bus/usb目录，从/proc/bus/usb/device文件可以知道系统中存在的所有USB设备的可选配置。

这里既然提到了USB设备的配置，就还是先简要说一下。一个设备可以有一种或者几种配置，这能理解吧？没见过具体的USB设备？那么手机见过吧，每个手机都会有多种配置，或者说“设定”。比如，我的这款Nokia 6300手机，手机语言可以设定为English、繁体中文、简体中文，一旦选择了其中一种，那么手机里边所显示的所有的信息都是该种语言/字体。再举一个最简单的例子，操作模式也有好几种，标准、无声、会议等。如果我设为“会议”模式，那么就是只振动不发声，要是设为“无声”模式，那么就啥动静也不会有，只能凭感觉了，以前去公司面试的话通常就是设为“无声”模式，因为觉得振动也不好，让人家面试官听到了还是不合适。那么USB设备的配置也是如此，不同的USB设备当然有不同的配置了，或者说需要配置哪些东西也会不一样。

8、好戏开始了

首先要去drivers/usb目录下走一走看一看。

```
atm/  class/  core/  gadget/  host/  image/  misc/  mon/  serial/  storage/
Kconfig  Makefile  README usb-skeleton.c
```

ls命令的结果就是上面的10个目录和4个文件。usb-skeleton.c是一个简单的USB driver的框架，感兴趣的读者可以去看一看，目前来说，它还吸引不了我的眼球。那么首先应该关注什么？如果迎面走来一个美女，你会首先看脸、脚还是其他部位？当然答案依据每个人的癖好会有所不同。不过这里的问题应该只有一个答案，那就是Kconfig、Makefile、README。

README里有关于这个目录下内容的一般性描述，它不是关键，只是帮助你了解。再说了，面对“读我吧！读我吧！”这么热情奔放的呼唤，善良的我们是不可能无动于衷的，所以先来看一看里面都有些什么内容。

```
Here is a list of what each subdirectory here is, and what is contained in them.

core/        - This is for the core USB host code, including the
             usbfs files and the hub class driver ("khubd").

host/        - This is for USB host controller drivers.  This
             includes UHCI, OHCI, EHCI, and others that might
             be used with more specialized "embedded" syste ms.

gadget/      - This is for USB peripheral controller drivers and
             the various gadget drivers which talk to them.

Individual USB driver directories.  A new driver should be added to the
first subdirectory in the list below that it fits into.

image/       - This is for still image drivers, like scanners or
             digital cameras.
input/       - This is for any driver that uses the input subsystem,
             like keyboard, mice, touchscreens, tablets, etc.
media/       - This is for multimedia drivers, like video cameras,
             radios, and any other drivers that talk to the v4l
             subsystem.
net/     - This is for network drivers.
serial/      - This is for USB to serial drivers.
storage/- This is for USB mass-storage drivers.
class/       - This is for all USB device drivers that do not fit
             into any of the above categories, and work for a range
             of USB Class specified devices.
misc/        - This is for all USB device drivers that do not fit
             into any of the above categories.
```

drivers/usb/README文件描述了前边使用ls命令列出的那10个文件夹的用途。那么什么是USB Core? Linux内核开发人员们，专门写了一些代码，负责实现一些核心的功能，为别的设备驱动程序提供服务，比如申请内存，实现一些所有的设备都会需要的公共的函数，并美其名曰为“USB Core”。

时代总在发展，当年胖杨贵妃照样能迷死唐明皇，而如今人们欣赏的则是林志玲这样的魔鬼身材。同样，早期的Linux内核，其结构并不是如今天这般有层次感，远不像今天这般错落有致，那时候drivers/usb/这个目录下边放了很多文件，USB Core与其他各种设备的驱动程序的代码都堆砌在这里，后来，怎奈世间万千的变幻，总爱把有情人分开。于是在drivers/usb/目录下面出来了一个core目录，就专门放一些核心的代码，比如初始化整个USB系统，初始化Root Hub，初始化主机控制器的代码，再后来甚至把主机控制器相关的代码也单独建了一个目录，叫host目录，这是因为USB主机控制器随着时代的发展，也开始有了好几种，不再像刚开始那样只有一种。所以呢，设计者们把一些主机控制器公共的代码仍然留在core目录下，而一些各主机控制器单独的代码则移到host目录下面让负责各种主机控制器的人去维护。

那么USB gadget呢？gadget白了说就是配件的意思，主要就是一些内部运行Linux的嵌入式设备，比如PDA，设备本身有USB设备控制器（USB Device Controller），可以将PC，也就是我们的主机作为master端，将这样的设备作为slave端和主机通过USB进行通信。从主机的观点来看，主机系统的USB驱动程序控制插入其中的USB设备，而USB gadget的驱动程序控制外围设备如何作为一个USB设备和主机通信。比如，我们的嵌入式板子上支持SD卡，如果我们希望在将板子通过USB连接到PC之后，这个SD卡被模拟成U盘，那么就要通过USB gadget架构的驱动。

gadget目录下大概能够分为两个模块，一个是udc驱动，这个驱动是针对具体cpu平台的，如果找不到现成的，就要自己实现。另外一个就是gadget驱动，主要有file_storage、ether、serial等。另外还提供了USB gadget API，即USB设备控制器硬件和gadget驱动通信的接口。PC及服务器只有USB主机控制器硬件，它们并不能作为USB gadget存在，而对于嵌入式设备，USB设备控制器常被集成到处理器中，设备的各种功能，如U盘、网卡等，常依赖这种USB设备控制器来与主机连接，并且设备的各种功能之间可以切换，比如可以根据选择作为U盘或网卡等。

剩下的几个目录分门别类地放了各种USB设备的驱动，U盘的驱动在storage目录下，触摸屏和USB键盘鼠标的驱动在input目录下等。另外，在USB协议中，除了通用的软硬件电气接口规范等，还包含了各种各样的Class协议，用来为不同的功能定义各自的标准接口和具体的总线上的数据交互格式和内容。这些Class协议的数量非常多，最常见的比如支持U盘功能的Mass Storage Class，以及通用的数据交换协议CDC Class。此外还包括Audio Class, Print Class 等。理论上说，即使没有这些Class，通过专用驱动也能够实现各种各样的应用功能。但是，正是Mass Storage Class的使用，使得各个厂商生产的U盘都能通过操作系统自带的统一的驱动程序来使用，对U盘的普及使用起了极大的推动作用，制定其他这些Class也是为了同样的目的。

我们响应了README的呼唤，它便给予了我们想要的，通过它我们了解了USB目录里的那些文件夹都有着什么样的角色。到现在为止，就只剩下Kconfig、Makefile两个文件了，它们又扮演着什么样的角色？就好像我吃东西时总是喜欢把好吃的留在最后享受一样，我也习惯于将重要的内容留在最后去描述。对于一个希望能够在Linux内核的汪洋代码里看到一丝曙光的人来说，将它们放在多么重要的地位都不过分。我们在去香港通过海关时，总会有免费的地图和各种指南，有了它们在手里我们才不至于像无头苍蝇般迷惘地行走在陌生的街道上。即使在境内出去旅游时一般也总是会首先找一份地图，当然了，这时就是要去买了，拿是拿不到的，不同的地方有不同的特色，别人的特色是服务，咱们的特色是索取。Kconfig、Makefile就是Linux kernel迷宫里的地图，我们每次浏览kernel寻找属于自己的那一段代码时，都应该首先看一看目录下的这两个文件。

不过，这里很明显，要想了解USB协议在内核中的实现，USB Core就是我们需要关注的对象。

9、不一样的 core

我们来看core目录。关于USB，有一个很重要的模块，她的名字耐人追寻，usbcore。如果你的电脑是装了Linux操作系统，那么你用lsmod命令看一下，有一个模块叫做usbcore。当然，你要是玩嵌入式系统的高手，那么也许你的电脑里没有USB模块。不过我听说如今玩嵌入式的人也喜欢玩USB，因为USB设备很符合嵌入式的口味。查看我的lsmod命令的输出吧：

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

```
localhost:/usr/src/linux-2.6.22.1/drivers/usb/core # lsmod
Module                  Size  Used by
af_packet              55820  2
raw                    89504  0
nfs                   230840  2
lockd                  87536  2 nfs
nfs_acl                20352  1 nfs
sunrpc                172360  4 nfs,lockd,nfs_acl
ipv6                  329728  36
button                 24224  0
battery                27272  0
ac                     22152  0
apparmor               73760  0
aamatch_pcre           30720  1 apparmor
loop                   32784  0
usbhid                 60832  0
dm_mod                 77232  0
ide_cd                 57120  0
hw_random              22440  0
ehci_hcd               47624  0
cdrom                  52392  1 ide_cd
uhci_hcd               48544  0
shpchp                 61984  0
bnx2                  157296  0
usbcore               149288  4 usbhid,ehci_hcd,uhci_hcd
e1000                 130872  0
pci_hotplug            44800  1 shpchp
reiserfs              239616  2
edd                    26760  0
fan                    21896  0
thermal                32272  0
processor              50280  1 thermal
qla2xxx               149484  0
firmware_class         27904  1 qla2xxx
scsi_transport_fc      51460  1 qla2xxx
sg                     52136  0
megaraid_sas           47928  3
piix                   27652  0 [permanent]
sd_mod                 34176  4
scsi_mod          163760  5 qla2xxx,scsi_transport_fc,sg,megaraid_sas,sd_mod
ide_disk               32768  0
ide_core              164996  3 ide_cd,piix,ide_disk
```

找到了usbcore那一行吗？它就是咱们这里要说的USB子系统的核心，如果要在Linux里使用USB，这个模块是必不可少的，另外，你应该会在usbcore那一行的最后看到ehci_hcd或uhci_hcd这样的东西，它们就是前面说的USB主机控制器的驱动模块，你的USB设备要工作，合适的USB主机控制器模块也是必不可少的。

USB Core负责实现一些核心的功能，为别的设备驱动程序提供服务，提供一个用于访问和控制USB硬件的接口，而不用去考虑系统当前存在哪种主机控制器。至于USB Core、USB主机控制器和USB设备驱动三者之间的关系，如图7所示。

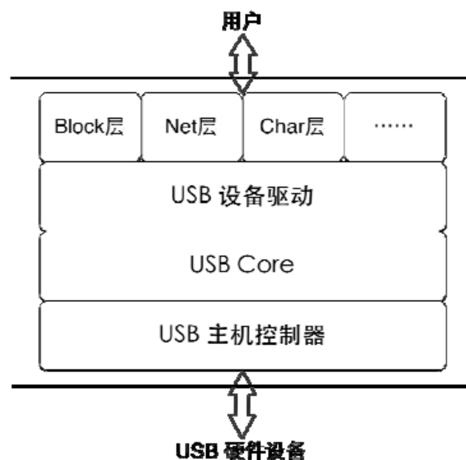


图 7　内核中 USB 子系统的结构

驱动和主机控制器像不像core的两个保镖？没办法，这可是core啊！协议中也说了，主机控制器的驱动（HCD）必须位于USB软件的最下一层。HCD提供主机控制器硬件的抽象，隐藏硬件的细节，在主机控制器之下是物理的USB及所有与之连接的USB设备。而HCD只有一个客户，对一个人负责，就是咱们的USB Core，USB Core将用户的请求映射到相关的HCD，用户不能直接访问HCD。

咱们写USB驱动时，只能调用core的接口，core会将咱们的请求发送给相应的HCD，用得着咱们操心的只有这么一亩三分地，core为咱们完成了大部分的工作，Linux的哲学是不是和咱们生活中不太一样？

走到drivers/usb/core目录里去，使用ls命令看一看，

```
Kconfig  Makefile  buffer.c  config.c  devices.c  devio.c  driver.c
endpoint.c  file.c  generic.c  hcd-pci.c  hcd.c  hcd.h  hub.c  hub.h
inode.c  message.c  notify.c  otg_whitelist.h  quirks.c  sysfs.c  urb.c
usb.c  usb.h
```

再使用wc命令统计一下，将近两万行的代码，core不愧是core，为大家默默地做这么多事，人民的好公仆鞠躬尽瘁，我们要用感恩的心去深刻理解你的内心回报你的付出。

不过这么多文件中不一定都是我们所需要关注的，先拿咱们的地图来看一看接下来该怎么走。先看一看Kconfig文件。

```
4 config USB_DEBUG
5   bool "USB verbose debug messages"
6   depends on USB
7   help
8     Say Y here if you want the USB core & hub drivers to produce a bunch
9     of debug messages to the system log. Select this if you are having a
10    problem with USB support and want to see more of what is going on.
```

这是USB的调试tag，如果你在写USB设备驱动的话，最好还是打开它吧，不过这里它就不是我们关注的重点了。

```
15 config USB_DEVICEFS
16        bool "USB device filesystem"
17        depends on USB
18        ---help---
19          If you say Y here (and to "/proc file system support" in the "File
20          syste ms" section, above), you will get a file /proc/bus/usb/devices
21          which lists the devices currently connected to your USB bus or
22          busses, and for every connected device a file named
23          "/proc/bus/usb/xxx/yyy", where xxx is the bus number and yyy the
24          device number; the latter files can be used by user space progra ms
25          to talk directly to the device. These files are "virtual", meaning
26          they are generated on the fly and not stored on the hard drive.
27
28          You may need to mount the usbfs file system to see the files, use
29          mount -t usbfs none /proc/bus/usb
30
31          For the format of the various /proc/bus/usb/ files, please read
32          <file:Documentation/usb/proc_usb_info.txt>.
33
34          Usbfs files can't handle Access Control Lists (ACL), which are the
35          default way to grant access to USB devices for untrusted users of a
36          desktop system. The usbfs functionality is replaced by real
37          device-nodes managed by udev. These nodes live in /dev/bus/usb and
38          are used by libusb.
```

这个选项是关于usbfs文件系统的。usbfs文件系统挂载在/proc/bus/usb上（mount -t usbfs none /proc/bus/usb），显示了当前连接的USB设备及总线的各种信息，每个连接的USB设备在其中都会有一个文件进行描述。比如文件/proc/bus/usb/xxx/yyy，xxx表示总线的序号，yyy表示设备在总线的地址，不过不能够依赖它们来稳定地访问设备，因为同一个设备两次连接对应的描述文件可能会不同，比如，第一次连接一个设备时，它可能是002/027，一段时间后再次连接，它可能就已经改变为002/048。

这就好比好不容易你暗恋的MM今天见你时对你抛了个媚眼，你心花怒放，赶快去买了100块彩票庆祝，到第二天再见到她时，她对你说："你是谁啊"，你悲痛欲绝地刮开那张100块彩票，上面清一色的都是"谢谢你"。usbfs与咱们探讨的主题关系不大，况且也已经足可以开个专题来讨论了，所以以后不会去过多提及它。

```
74 config USB_SUSPEND
75       bool "USB selective suspend/resume and wakeup (EXPERIMENTAL)"
76       depends on USB && PM && EXPERIMENTAL
77       help
78         If you say Y here, you can use driver calls or the sysfs
79         "power/state" file to suspend or resume individual USB
80         peripherals.
82         Also, USB "remote wakeup" signaling is supported, whereby some
83         USB devices (like keyboards and network adapters) can wake up
84         their parent hub.  That wakeup cascades up the USB tree, and
85         could wake the system from states like suspend-to-RAM.
86
87         If you are unsure about this, say N here.
```

这一项是有关USB设备的挂起和恢复。开发USB的人都是节电节能的好孩子，所以协议中就规定了，所有的设备都必须支持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内（3 ms），如果没有发生总线传输，就要进入挂起状态。当它收到一个non-idle的信号时，就会被唤醒。节约用电从USB做起。不过目前来说内核对挂起休眠的支持普遍都不太好，而且许多的USB设备也没有支持它，还是暂且不提了。

剩下的还有几项，不过似乎与咱们关系也不大，还是去看一看Makefile。

```
5 usbcore-objs    := usb.o hub.o hcd.o urb.o message.o driver.o \
6                       config.o file.o buffer.o sysfs.o endpoint.o \
7                       devio.o notify.o generic.o quirks.o
8
9 ifeq ($(CONFIG_PCI),y)
10        usbcore-objs    += hcd-pci.o
11 endif
12
13 ifeq ($(CONFIG_USB_DEVICEFS),y)
14        usbcore-objs    += inode.o devices.o
15 endif
16
17 obj-$(CONFIG_USB)       += usbcore.o
19 ifeq ($(CONFIG_USB_DEBUG),y)
20 EXTRA_CFLAGS += -DDEBUG
21 endif
```

Makefile可比Kconfig简略多了，所以看起来也更亲切点，咱们总是拿的钱越多越好，看的代码越少越好。这里之所以会出现CONFIG_PCI，是因为通常USB的Root Hub包含在一个PCI设备中，前面也已经聊过了。hcd-pci和hcd顾名而思义就是主机控制器，它们实现了主机控制器公共部分，按协议中的说法它们就是HCDI（HCD的公共接口），host目录下则实现了各种不同的主机控制器，咱们这里不怎么会聊到具体主机控制器的实现。CONFIG_USB_DEVICEFS在前面的Kconfig文件中也见到了，关于usbfs的，与咱们的主题无关，inode.c和devices.c两个文件也可以不用管了。

这么看来，好像大都需要关注的样子，没有减轻多少压力，不过这里本身就是USB Core部分，是要做很多的事为咱们分忧的，所以多点也是可以理解的。

10、从这里开始

USB Core从USB子系统的初始化开始，我们也需要从那里开始，它们位于文件drivers/usb/core/usb.c

```
938 subsys_initcall(usb_init);
939 module_exit(usb_exit);
```

我们看到一个subsys_initcall，它也是一个宏，我们可以把它理解为module_init，只不过因为这部分代码比较核心，开发人员们把它看作一个子系统，而不仅仅是一个模块。这也很好理解，usbcore这个模块代表的不是某一个设备，而是所有USB设备赖以生存的模块，在Linux中，像这样一个类别的设备驱动被归结为一个子系统。比如PCI子系统，比如SCSI子系统，基本上，drivers/目录下面第一层的每个目录都算一个子系统，因为它们代表了一类设备。

subsys_initcall(usb_init)的意思就是告诉我们usb_init是USB子系统真正的初始化函数，而usb_exit()将是整个USB子系统的结束时的清理函数，于是我们就从usb_init开始看起。

```
863 static int __init usb_init(void)
864 {
865     int retval;
866     if (nousb) {
867     pr_info("%s: USB support disabled\n", usbcore_name);
868         return 0;
869     }
870
871     retval = ksuspend_usb_init();
872     if (retval)
873         goto out;
874     retval = bus_register(&usb_bus_type);
875     if (retval)
876         goto bus_register_failed;
877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;
880     retval = usb_major_init();
881     if (retval)
882         goto major_init_failed;
883     retval = usb_register(&usbfs_driver);
884     if (retval)
885         goto driver_register_failed;
886     retval = usb_devio_init();
887     if (retval)
888         goto usb_devio_init_failed;
889     retval = usbfs_init();
890     if (retval)
891         goto fs_init_failed;
892     retval = usb_hub_init();
893     if (retval)
894         goto hub_init_failed;
895     retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
896     if (!retval)
897         goto out;
898
899     usb_hub_cleanup();
900 hub_init_failed:
901     usbfs_cleanup();
902 fs_init_failed:
903     usb_devio_cleanup();
904 usb_devio_init_failed:
905     usb_deregister(&usbfs_driver);
906 driver_register_failed:
907     usb_major_cleanup();
908 major_init_failed:
909     usb_host_cleanup();
910 host_init_failed:
911     bus_unregister(&usb_bus_type);
912 bus_register_failed:
913     ksuspend_usb_cleanup();
914 out:
915     return retval;
916 }
```

首先看863行的__init标记，写过驱动的应该不会陌生，它对内核来说就是一种暗示，表明这个函数仅在初始化期间使用，在模块被装载之后，它占用的资源就会释放掉用作它处。它的暗示你懂，可你的暗示，她却不懂或者懂装不懂，多么让人感伤。它在自己短暂的一生中一直从事繁重的工作，吃的是草吐出的是牛奶，留下的是整个USB子系统的繁荣。

受这种精神所感染，我觉得还是有必要为它说的更多些。__init的定义在include/linux/init.h文件中

```
43 #define __init          __attribute__ ((__section__ (".init.text")))
```

好像这里引出了更多的疑问，__attribute__是什么？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。而__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，这些属性主要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，section是其中的一个，我们查看GCC的手册可以看到下面的描述

```
'section ("section-name")'
Normally, the compiler places the code it generates in the `text'
    section.  Sometimes, however, you need additional sections, or you
need certain particular functions to appear in special sections.
The `section' attribute specifies that a function lives in a
particular section.  For example, the declaration:

extern void foobar (void) __attribute__ ((section ("bar")));
puts the function 'foobar' in the 'bar' section.
Some file formats do not support arbitrary sections so the
'section' attribute is not available on all platfor ms.  If you
need to map the entire contents of a module to a particular
section, consider using the facilities of the linker instead.
```

通常编译器将函数放在.text节，变量放在.data或.bss节，使用section属性，可以让编译器将函数或变量放在指定的节中。那么前面对__init的定义便表示将它修饰的代码放在.init.text节。连接器可以把相同节的代码或数据安排在一起，比如__init修饰的所有代码都会被放在.init.text节里，初始化结束后就可以释放这部分内存。

那内核又是如何调用到这些__init修饰的初始化函数？要回答这个问题，还需要回顾一下上面938行的代码，上面已经提到subsys_initcall也是一个宏，它也在include/linux/init.h中定义。

```
125 #define subsys_initcall(fn)            __define_initcall("4",fn,4)
```

这里又出现了一个宏__define_initcall，它用于将指定的函数指针fn放到initcall.init节里，而对于具体的subsys_initcall宏，则是把fn放到.initcall.init的子节.initcall4.init里。要弄清楚.initcall.init、.init.text和.initcall4.init，我们还需要了解一点内核可执行文件相关的概念。

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。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。

我可以负责任地告诉你，要看懂vmlinux.lds这个文件是需要一番工夫的，不过大家都是聪明人，聪明人做聪明事，所以你需要做的只是搜索initcall.init，然后便会看到似曾相识的内容

```
__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 = .;
```

这里的__initcall_start指向.initcall.init节的开始，__initcall_end指向它的结尾。而.initcall.init节又被分为了7个子节，分别如下所示。

```
.initcall1.init
.initcall2.init
.initcall3.init
.initcall4.init
.initcall5.init
.initcall6.init
.initcall7.init
```

我们的subsys_initcall宏便是将指定的函数指针放在了.initcall4.init子节。其他的比如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，都可以从include/linux/init.h文件找到它们的定义。各个字节的顺序是确定的，即先调用.initcall1.init中的函数指针再调用.initcall2.init中的函数指针等。__init修饰的初始化函数在内核初始化过程中调用的顺序和.initcall.init节里函数指针的顺序有关，不同的初始化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。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京|上海|深圳|成都|南京|武汉|西安|广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

至于实际执行函数调用的地方，就在/init/main.c文件中，内核的初始化，不在那里还能在哪里？里面的do_initcalls函数会直接用到这里的__initcall_start、__initcall_end来进行判断。不多说了，我的思想已经如滔滔江水泛滥成灾了，还是回到久违的usb_init函数吧。

11、面纱

前面说了那么多，才接触到usb_init，有点偷窥USB面纱下神秘容颜的味道。当然，我们并不需要去经历爱情、被判与死亡，所需要经历的只是忍受前面大段大段的唠叨。

人往往可以被高尚感动，但始终不能因为高尚而爱上它。因为被__init给盯上，usb_init在做牛做马的辛勤劳作之后便不得不灰飞烟灭，不可谓不高尚，但它始终只能是我们了解面纱后面内容的跳板，是起点，却不是终点，我们不会为它停留太久，有太多的精彩和苦恼在等着我们。

```
866     if (nousb) {
867     pr_info("%s: USB support disabled\n", usbcore_name);
868         return 0;
869     }
```

866行，知道C语言的人都会知道nousb是一个标志，只是不同的标志有不一样的精彩。这里的nousb是用来让我们在启动内核时通过内核参数去掉USB子系统的，Linux社会是一个很人性化的世界，它不会去逼迫我们接受USB，一切都只关乎我们自己的需要。不过我想我们一般来说是不会去指定nousb的吧，毕竟它那么地讨人喜爱。如果你真的指定了nousb，那它就只会幽怨的说一句“USB support disabled”，然后退出usb_init。

867行，pr_info只是一个打印信息的可辨参数宏，printk的变体，在include/linux/kernel.h中定义：

```
242 #define pr_info(fmt,arg...) \
243         printk(KERN_INFO fmt,##arg)
```

1999年的ISO C标准里规定了可变参数宏，与函数语法类似，比如：

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

里面的“…”就表示可变参数，调用时，它们就会替代宏体里的__VA_ARGS__。GCC总是会显得特立独行一些，它支持更复杂的形式，可以给可变参数取个名字，比如：

```
#define debug(format, args...) fprintf (stderr, format, args)
```

有了名字总是会容易交流一些。是不是与pr_info比较接近了？除了“##”，它主要是针对空参数的情况。既然说是可变参数，那传递空参数也总是可以的，空即是多，多即是空，股市里的哲理在这里同样也是适合的。如果没有“##”，传递空参数时，比如：

```
debug ("A message");
```

展开后，里面的字符串后面会多一个多余的逗号。这个逗号你应该不会喜欢，而“##”则会使预处理器去掉这个多余的逗号。

```
871     retval = ksuspend_usb_init();
872     if (retval)
873         goto out;
874     retval = bus_register(&usb_bus_type);
875     if (retval)
876         goto bus_register_failed;
877     retval = usb_host_init();
878     if (retval)
879         goto host_init_failed;
880     retval = usb_major_init();
881     if (retval)
882         goto major_init_failed;
883     retval = usb_register(&usbfs_driver);
884     if (retval)
885         goto driver_register_failed;
886     retval = usb_devio_init();
887     if (retval)
888         goto usb_devio_init_failed;
889     retval = usbfs_init();
```

华清远见

```
890        if (retval)
891            goto fs_init_failed;
892        retval = usb_hub_init();
893        if (retval)
894            goto hub_init_failed;
895        retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
896        if (!retval)
897            goto out;
```

871行到897行是代码里的排比句，相似的init，不相似的内容，很显然都是在完成一些初始化，也是usb_init任劳任怨所付出的全部。这里先简单的说一下。

871行，电源管理方面的。如果在编译内核时没有打开电源管理，也就是说没有定义CONFIG_PM，它就什么也不做。

874行，注册USB总线，只有成功地将USB总线子系统注册到系统中，我们才可以向这个总线添加USB设备。基于它显要的江湖地位，就拿它作为日后突破的方向了，擒贼先擒王，这个越老越青春的道理在Linux中也是同样适用的。

877行，执行主机控制器相关的初始化。

880行，一个总线同时也是一个设备，必须单独注册，因为USB是通过快速串行通信来读写数据，这里把它当作了字符设备来注册。

883行~891行，都是usbfs相关的初始化。

892行，Hub的初始化。

895行，注册USB设备驱动，戴好眼镜看清楚了，是USB device driver而不是USB driver，前面说过，一个设备可以有多个接口，每个接口对应不同的驱动程序，这里所谓的device driver对应的是整个设备，而不是某个接口。内核中结构到处有，只是USB这儿格外多。

剩下的几行代码都是有关资源清除的，usb_init这个短短的函数在承载着我们的希望时嘎然而止了，你的感觉是什么？我的感觉是：这哪是我能说的清楚的啊。它的每个分叉都更像是一个陷阱，黑黝黝看不到底，但是已经没有回头的路。

华清远见

12、模型，又见模型

上文说usb_init给我们留下了一些岔路口，每条都像是不归路，让人不知道从何处开始，也看不到路的尽头。趁着徘徊彷徨时，咱们还是先聊一下Linux的设备模型。各位看官听好了，这可不是任小强们的房模，也不是张导的炮模，不存在忽悠你们的可能。

顾名而思义就知道设备模型是关于设备的模型，对咱们写驱动的和不写驱动的人来说，设备的概念就是总线和与其相连的各种设备了。电脑城的IT工作者都会知道设备是通过总线连到计算机上的，而且还需要对应的驱动才能用，可是总线是如何发现设备的？设备又是如何和驱动对应起来的？它们经过怎样的艰辛才找到命里注定的那个它？它们的关系如何？白头偕老型的还是朝三暮四型的？这些问题就不是他们关心的了，而是咱们需要关心的。经历过高考千锤百炼的咱们还能够惊喜地发现，这些疑问的中心思想中心词汇就是总线、设备和驱动，没错，它们都是咱们这里要聊的Linux设备模型的名角。

总线、设备、驱动，也就是bus、device、driver，既然是名角，在内核中都会有它们自己专属的结构，在include/linux/device.h中定义。

```
52 struct bus_type {
53        const char              * name;
54        struct module            * owner;
55
56        struct kset             subsys;
57        struct kset             drivers;
58        struct kset             devices;
59        struct klist            klist_devices;
60        struct klist            klist_drivers;
```

```
61
62      struct blocking_notifier_head bus_notifier;
63
64      struct bus_attribute    * bus_attrs;
65      struct device_attribute * dev_attrs;
66      struct driver_attribute * drv_attrs;
67      struct bus_attribute drivers_autoprobe_attr;
68      struct bus_attribute drivers_probe_attr;
69
70      int (*match)(struct device * dev, struct device_driver * drv);
71      int (*uevent)(struct device *dev, char **envp,
72      int num_envp, char *buffer, int buffer_size);
73      int             (*probe)(struct device * dev);
74      int             (*remove)(struct device * dev);
75      void            (*shutdown)(struct device * dev);
76
77      int (*suspend)(struct device * dev, pm_message_t state);
78      int (*suspend_late)(struct device * dev, pm_message_t state);
79      int (*resume_early)(struct device * dev);
80      int (*resume)(struct device * dev);
81
82      unsigned int drivers_autoprobe:1;
83 };

124 struct device_driver {
125     const char              * name;
126     struct bus_type         * bus;
127
128     struct kobject          kobj;
129     struct klist            klist_devices;
130     struct klist_node       knode_bus;
131
132     struct module           * owner;
133     const char              * mod_name;  /* used for built-in modules */
134     struct module_kobject   * mkobj;
135
136     int     (*probe)        (struct device * dev);
137     int     (*remove)       (struct device * dev);
138     void    (*shutdown)     (struct device * dev);
139     int     (*suspend)      (struct device * dev, pm_message_t state);
140     int     (*resume)       (struct device * dev);
141 };

410 struct device {
411     struct klist            klist_children;
412     struct klist_node       knode_parent;   /* node in sibling list */
413     struct klist_node       knode_driver;
414     struct klist_node       knode_bus;
415     struct device           *parent;
416
417     struct kobject kobj;
418     char    bus_id[BUS_ID_SIZE];    /* position on parent bus */
419     struct device_type      *type;
420     unsigned                is_registered:1;
421     unsigned                uevent_suppress:1;
422     struct device_attribute uevent_attr;
423     struct device_attribute *devt_attr;
424
425     struct semaphore        sem; /* semaphore to synchronize calls to
426                                   * its driver.
427                                   */
428
429     struct bus_type * bus;          /* type of bus device is on */
430     struct device_driver *driver;   /* which driver has allocated this
431                                        device */
432     void            *driver_data;   /* data private to the driver */
433     void            *platform_data; /* Platform specific data, device
434                                        core doesn't touch it */
435     struct dev_pm_info      power;
436
437 #ifdef CONFIG_NUMA
438     int             numa_node;   /* NUMA node this device is close to */
439 #endif
440     u64             *dma_mask;      /* dma mask (if dma'able device) */
441     u64             coherent_dma_mask;/* Like dma_mask, but for
```

```
442                                          alloc_coherent mappings as
443                                          not all hardware supports
444                                          64 bit addresses for consistent
445                                         allocations such descriptors. */
446
447      struct list_head       dma_pools;     /* dma pools (if dma'ble) */
448
449      struct dma_coherent_mem *dma_mem; /* internal for coherent mem
450                                          override */
451      /* arch specific additions */
452      struct dev_archdata     archdata;
453
454      spinlock_t            devres_lock;
455      struct list_head        devres_head;
457      /* class_device migration path */
458      struct list_head        node;
459      struct class           *class;
460      dev_t              devt;    /* dev_t, creates the sysfs "dev" */
461      struct attribute_group  **groups;      /* optional groups */
463      void    (*release)(struct device * dev);
464 };
```

有没有发现它们的共性是什么？对，很长很复杂。不过不妨把它们看成艺术品，既然是艺术品，当然不会让你那么容易就看懂了，不然怎么称大师称名家。这么想一想咱们就会比较宽慰了，阿Q是鲁迅先生对咱们80后最大的贡献。

我知道进入了21世纪，最缺的就是耐性，房价股价都让咱们没有耐性，内核的代码也让人没有耐性。不过作为最没有耐性的一代人，还是要平心静气地扫一下上面的结构，我们会发现，struct bus_type中有成员struct kset drivers 和struct kset devices，同时struct device中有两个成员struct bus_type * bus和struct device_driver *driver，struct device_driver中有两个成员struct bus_type * bus和struct klist klist_devices。先不说什么是klist、kset，光从成员的名字看，它们就是一个完美的三角关系。我们每个人心中是不是都有两个她？一个梦中的她，一个现实中的她。

凭一个男人的直觉，我们可以知道，struct device中的bus表示这个设备连到哪个总线上，driver表示这个设备的驱动是什么。struct device_driver中的bus表示这个驱动属于哪个总线，klist_devices表示这个驱动都支持哪些设备，因为这里device是复数，又是list，更因为一个驱动可以支持多个设备，而一个设备只能绑定一个驱动。当然，struct bus_type中的drivers和devices分别表示了这个总线拥有哪些设备和哪些驱动。

我们还需要看一看什么是klist、kset。还有上面device和driver结构中出现的kobject结构是什么？作为在一个五星红旗下长大的孩子，我可以肯定地告诉你，kobject和kset都是Linux设备模型中最基本的元素，总线、设备、驱动是西瓜，kobjcet、klist是种瓜的人，没有幕后种瓜人的汗水不会有清爽解渴的西瓜。我们不能光知道西瓜是多么的甜，还要知道种瓜人的辛苦。kobject和kset不会在意自己的得失，它们存在的意义在于把总线、设备和驱动这样的对象连接到设备模型上。种瓜的人也不会在意自己的汗水，在意的只是能不能种出甜蜜的西瓜。

一般来说应该这么理解，整个Linux的设备模型是一个OO的体系结构，总线、设备和驱动都是其中鲜活存在的对象，kobject是它们的基类，所实现的只是一些公共的接口，kset是同种类型kobject对象的集合，也可以说是对象的容器。只是因为C里不可能会有C++里类的class继承、组合等的概念，只有通过kobject嵌入到对象结构中来实现。这样，内核使用kobject将各个对象连接起来组成了一个分层的结构体系，就好像马克思列宁主义将我们13亿人也连接成了一个分层的社会体系一样。kobject结构中包含了parent成员，指向了另一个kobject结构，也就是这个分层结构的上一层结点。而kset是通过链表来实现的，这样就可以明白，struct bus_type结构中的成员drivers和devices表示了一条总线拥有两条链表，一条是设备链表，一条是驱动链表。我们知道了总线对应的数据结构，就可以找到这条总线关联了多少设备，又有哪些驱动来支持这类设备。

那么klist呢？其实它就包含了一个链表和一个自旋锁，我们暂且把它看成链表也无妨。本来在2.6.11内核中，struct device_driver结构的devices成员就是一个链表类型。这么一说，咱们上面的直觉都是正确的，如果买股票、摸彩票时直觉都这么管用，就不会有咱们这被压扁的一代了。

21世纪的人都知道，三角关系很难处。那么总线、设备和驱动之间是如何和谐共处的呢？先说一说总线中的那两条链表是怎么形成的。内核要求每次出现一个设备就要向总线汇报，或者说注册，每次出现一个驱动，也要向总线汇报，或者说注册。比如系统初始化时，会扫描连接了哪些设备，并为每一个设备建立起一个struct device的变量，每一次有一个驱动程序，就要准备一个struct device_driver结构的变量。把这些变量统统加入相应的链表，device 插入devices 链表，driver插入drivers链表。这样通过总线就能找到每一个设备，每一个驱动。然而，假如计算机里只有设备却没有对应的驱动，那么设备无法工作。反过来，倘若只有驱动却没有设备，驱动也起不了任何作用。在他们遇见彼此之前，双方都如同路埂里的野草，一个飘啊飘，一个摇啊摇，谁也不知道未来在哪里，只能在生命的风里飘摇。于是总线上的两张表里就慢慢地就挂上了许多孤单的灵魂。devices开始多了，drivers开始多了，它们像是两个来自世界，devices们彼此取暖，drivers们一起狂欢，但它们有一点是相同的，都只是在等待属于自己的另一半。

现在，总线上的两条链表已经有了，这个三角关系的三个边已经有了两个，剩下的那个呢？链表里的设备和驱动又是如何联系的？先有设备还是先有驱动？很久很久以前，在那激情燃烧的岁月里，先有的是设备，每一个要用的设备在计算机启动之前就已经插好了，插放在它应该在的位置上，然后计算机启动，操作系统开始初始化，总线开始扫描设备，每找到一个设备，就为其申请一个struct device结构，并且挂入总线中的devices链表中来。然后每一个驱动程序开始初始化，开始注册其struct device_driver结构，然后它去总线的devices链表中去寻找(遍历)，去寻找每一个还没有绑定驱动的设备，即struct device中的struct device_driver指针仍为空的设备，然后它会去观察这种设备的特征，看是否是它所支持的设备，如果是，那么调用一个叫做device_bind_driver的函数，然后它们就结为了秦晋之好。换句话说，把struct device中的struct device_driver driver指向这个驱动，而struct device_driver driver把struct device加入它的那张struct klist klist_devices链表中来。就这样，bus、device和driver，这三者之间或者说他们中的两两之间，就给联系上了。知道其中之一，就能找到另外两个。一荣俱荣，一损俱损。

但现在情况变了，在这红莲绽放的日子里，在这樱花伤逝的日子里，出现了一种新的名词，叫热插拔。此时设备可以在计算机启动以后再插入或者拔出计算机了。因此，很难再说是先有设备还是先有驱动了，因为都有可能。设备可以在任何时刻出现，而驱动也可以在任何时刻被加载，所以，现在的情况就是，每当一个struct device诞生，它就会去bus的drivers链表中寻找自己的另一半。反之，每当一个一个struct device_driver诞生，它就去bus的devices链表中寻找它的那些设备。如果找到了合适的，那么和之前那种情况一样，调用device_bind_driver绑定好。如果找不到，没有关系。等待吧，等到昙花再开，等到将风景看透，心中相信，这世界上总有一个人是你所等的，只是还没有遇到而已。

13、繁华落尽

看过了Linux设备模型固的繁华似锦，该是体味境界之美时了。

Linux设备模型中的总线落实在USB子系统里就是usb_bus_type，它在usb_init函数的874行注册，在drivers/usb/core/driver.c文件中定义。

```
1523 struct bus_type usb_bus_type = {
1524     .name =         "usb",
1525     .match =        usb_device_match,
1526     .uevent =       usb_uevent,
1527     .suspend =      usb_suspend,
1528     .resume =       usb_resume,
1529 };
```

看来是要走向这个分叉了，既然没有回头的路，就放平心情，欣赏沿路美景吧！name自然就是USB总线的绰号了。match这个函数指针就比较有意思了，它充当了一个红娘的角色，在总线的设备和驱动之间牵线搭桥，类似于交通大学BBS上的鹊桥版，虽然上面的条件都琳琅满目的，但明显这里match的条件不是那么苛刻，要更为实际一些。match指向了函数usb_device_match

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
```

```
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545         /* interface drivers never match devices */
546         if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551
552     } else {
553         struct usb_interface *intf;
554         struct usb_driver *usb_drv;
555         const struct usb_device_id *id;
556
557         /* device drivers never match interfaces */
558         if (is_usb_device_driver(drv))
559         return 0;
560
561         intf = to_usb_interface(dev);
562         usb_drv = to_usb_driver(drv);
563
564         id = usb_match_id(intf, usb_drv->id_table);
565         if (id)
566             return 1;
567
568         id = usb_match_dynamic_id(intf, usb_drv);
569         if (id)
570         return 1;
571     }
572
573     return 0;
574 }
```

540行，经历了繁华的Linux设备模型，这两个参数我们都已经很熟悉了，对应的就是总线两条链表里的设备和驱动，也可以说是鹊桥版上的挂牌的和摘牌的。总线上有新设备或新的驱动添加时，这个函数总是会被调用，如果指定的驱动能够处理指定的设备，也就是匹配成功，函数返回0。梦想是美好的，现实是残酷的，匹配是未必成功的，红娘再努力，双方对不上眼也是实在没办法的事。

543行，一遇到if和else，我们就知道又处在两难境地了，代码里我们可选择的太多，生活里我们可选择的太少。出生，长大，死亡，好像一直身不由己地随着命运在走。这里的岔路口只有两条路，一条给USB设备走，一条给USB接口走，各走各的路，分开了，就不再相见。

14、接口是设备的接口

设备可以有多个接口，每个接口代表一个功能，每个接口对应着一个驱动。Linux设备模型中的device落实在USB子系统，成了两个结构，一个是struct usb_device，一个是struct usb_interface。一个石头砸了两个坑，一支箭射下来两只麻雀。一个USB键盘，上面带一个扬声器，因此有两个接口，那这样肯定得要两个驱动程序，一个是键盘驱动程序，一个是音频流驱动程序。道上的兄弟喜欢把这样两个整合在一起的东西叫做一个设备，那好，让他们去叫吧，我们用interface来区分这两者行了吧。于是有了这里提到的数据结构，struct usb_interface。

```
140 struct usb_interface {
141     /* array of alternate settings for this interface,
142      * stored in no particular order */
143     struct usb_host_interface *altsetting;
144
145     struct usb_host_interface *cur_altsetting;      /* the currently
146                                          * active alternate setting */
147     unsigned num_altsetting;        /* number of alternate settings */
148
149     int minor;                      /* minor number this interface is
150                                          * bound to */
151     enum usb_interface_condition condition;     /* state of binding */
152     unsigned is_active:1;           /* the interface is not suspended */
153     unsigned needs_remote_wakeup:1;/*driver requires remote wakeup */
154
```

```
155     struct device dev;           /* interface specific device info */
156     struct device *usb_dev; /* pointer to the usb class's device, if any*/
157     int pm_usage_cnt;             /* usage counter for autosuspend */
158 };
```

143行，这里有一个altsetting成员，只用耗费一个脑细胞就可以明白它的意思就是alternate setting，可选的设置。那么再耗费一个脑细胞就可以知道145行的cur_altsetting表示当前正在使用的设置，147行的num_altsetting表示这个接口具有可选设置的数量。前面提到过USB设备的配置，那这里的设置是什么意思？这可不是耗费一两个脑细胞就可以明白的了，不过不要怕，虽然说脑细胞是不可再生资源，但并不是多宝贵的东西，不然怎么黄金、煤炭的股票都在疯长，就不见人的工资长。

咱们是难得糊涂几千年了，不会去区分配置和设置，起码我平时即使是再无聊也不会去想这个。但老外不一样，他们不知道老子也不知道郑板桥，所以说他们挺较真儿这个，还分了两个词，配置是configuration，设置是setting。

先说配置，一个手机可以有多种配置，比如可以摄像，还可以接在电脑里当做一个U盘，那么这两种情况就属于不同的配置，在手机里面有相应的选择菜单，你选择了哪种它就按哪种配置进行工作，供你选择的这个选项就叫做配置。很显然，当你摄像时你不可以访问这块U盘，当你访问这块U盘时你不可以摄像，因为你做了选择。第二，既然一个配置代表一种不同的功能，那么很显然，不同的配置可能需要的接口就不一样，假设你的手机里从硬件上来说一共有5个接口，那么可能当你配置成U盘时它只需要用到某一个接口；当你配置成摄像时，它可能只需要用到另外两个接口；可能你还有别的配置，你可能就会用到剩下的那两个接口。

再说一说设置，一个手机可能各种配置都确定了，是振动还是铃声已经确定了，各种功能都确定了，但是声音的大小还可以变吧，通常手机的音量是一格一格地变动，大概也就5格至6格，那么这个可以算一个设置吧。

如果你还是不明白什么是配置什么是设置的话，那就直接用大小关系来理解好了，毕竟大家对互相之间的大小关系都更敏感一些，不要说不是。这么说吧，设备大于配置，配置大于接口，接口大于设置，更准确的说是设备可以有多个配置，配置里可以包含一个或更多的接口，而接口通常又具有一个或更多的设置。

149行，minor，分配给接口的次设备号。Linux下所有的硬件设备都是用文件来表示的，俗称设备文件，在/dev目录下面，为了显示自己并不是普通的文件，它们都会有一个主设备号和次设备号，如下：

```
brw-r----- 1 root disk 8,  0 Sep  26 09:17 /dev/sda
brw-r----- 1 root disk 8,  1 Sep  26 09:17 /dev/sda1
crw-r----- 1 root tty 4,  1 Sep  26 09:17 /dev/tty1
```

这是在/dev目录下执行ls命令后的部分显示结果。作为新一代的年轻人，咱们对数字都是比较敏感的，一眼就能看到在每一行的日期前面有两个逗号隔开的数字，对于普通文件而言这个位置显示的是文件的长度。而对于设备文件，这里显示的两个数字表示了该设备的主设备号和次设备号。一般来说，主设备号表明了设备的种类，也表明了设备对应着哪个驱动程序，而次设备号则是因为一个驱动程序要支持多个设备而为了让驱动程序区分它们而设置的。也就是说，主设备号用来帮你找到对应的驱动程序，次设备号给你的驱动用来决定对哪个设备进行操作。上面就显示了我的移动硬盘主设备号为8，系统里tty设备的主设备为4。

设备要想在Linux里分得一个主设备号，有一个立足之地，也并不是那么容易的。主设备号虽说不是什么特别稀缺的资源，但还是需要设备先在驱动里提出申请，获得系统的批准才能拥有一个。因为一部分的主设备号已经被静态地预先指定给了许多常见的设备，你申请时要避开它们，选择一个里面没有列出来的（也就是名花还没有主的），很严肃地说，挖墙角是很不道德的。这些已经被分配掉的主设备号都列在Documentation/devices.txt文件中。当然，如果你是用动态分配的形式，就可以不去理会这些，直接让系统为你作主，替你选择一个即可。

很显然，任何一个有理智、有感情的人都会认为USB设备是很常见的，linux理应为它预留了一个主设备号。下面看一看include/linux/usb.h文件。

华清远见

```
7 #define USB_MAJOR                    180
8 #define USB_DEVICE_MAJOR             189
```

苏格拉底说过，学的越多，知道的越多；知道的越多，发现需要知道的更多。当我们知道了主设备号，满怀激情与向往地来寻找USB的主设备号时，我们却发现这里在上演真假李逵。这两个哪个才是我们苦苦追寻的她？

你可以在内核中搜索它们都曾经出现什么地方，或者就跟随我回到usb_init函数。

```
880      retval = usb_major_init();
881      if (retval)
882      goto major_init_failed;
883      retval = usb_register(&usbfs_driver);
884      if (retval)
885          goto driver_register_failed;
886      retval = usb_devio_init();
887      if (retval)
888          goto usb_devio_init_failed;
889      retval = usbfs_init();
890      if (retval)
891          goto fs_init_failed;
```

前面只提了一句883~891行是与usbfs相关的就简单略过了，这里稍微多说一点。usbfs为咱们提供了在用户空间直接访问USB硬件设备的接口，但是世界上没有免费的午餐，它需要内核的大力支持，usbfs_driver就是用来完成这个光荣任务的。咱们可以去usb_devio_init函数中看一看，它在drivers/usb/devio.c文件中定义：

```
retval = register_chrdev_region(USB_DEVICE_DEV, USB_DEVICE_MAX,
        "usb_device");
if (retval) {
    err("unable to register minors for usb_device");
    goto out;
}
```

register_chrdev_region函数获得了设备usb_device对应的设备编号，设备usb_device对应的驱动当然就是usbfs_driver，参数USB_DEVICE_DEV也在同一个文件中有定义：

```
#define USB_DEVICE_DEV        MKDEV(USB_DEVICE_MAJOR, 0)
```

终于再次见到了USB_DEVICE_MAJOR，也终于明白它是为了usbfs而生，为了让广大人民群众能够在用户空间直接和USB设备通信而生。因此，它并不是我们所要寻找的。

那么答案很明显了，USB_MAJOR就是咱们苦苦追寻的那个她，就是Linux为USB设备预留的主设备号。事实上，前面usb_init函数的880行，usb_major_init函数已经使用USB_MAJOR注册了一个字符设备，名字就叫USB。我们可以在文件/proc/devices里看到它们。

```
localhost:/usr/src/linux/drivers/usb/core # cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
29 fb
116 alsa
128 ptm
136 pts
162 raw
180 usb
189 usb_device
```

/proc/devices文件中显示了所有当前系统里已经分配出去的主设备号，当然上面只是列出了字符设备，块设备被有意的略过了。很明显，咱们前面提到的usb_device和usb都在里面。

不过到这里还没讲完，USB设备有很多种，并不是都会用到这个预留的主设备号。比如我的移动硬盘显示出来的主设备号就是8，你的摄像头在Linux显示的主设备号也绝对不会是这里的USB_MAJOR。坦白地说，咱们经常遇到的大多数USB设备都会与input、video等子系统关联，并不单单只是作为USB设备而存在。如果USB设备没有与其他任何子系统关联，就需要在对应驱动的probe函数中使用usb_register_dev函数来注册并获得主设备号USB_MAJOR，你可以在drivers/usb/misc目录下看到一些例子，drivers/usb/usb-skeleton.c文件也属于这种。如果USB设备关联了其他子系统，则需要在对应驱动程序的probe函数中使用相应的注册函数，USB_MAJOR也就用不着了。比如，USB键盘关联了input子系统，驱动对应drivers/hid/usbhid目录下的usbkbd.c文件，在它的probe函数中可以看到使用了input_register_device来注册一个输入设备。

准确地说，这里的USB设备应该说成USB接口，毕竟一个USB接口才对应着一个USB驱动。当USB接口关联有其他子系统，也就是说不使用USB_MAJOR作为主设备号时，struct usb_interface的字段minor可以简单地忽略。minor只在USB_MAJOR起作用时起作用。

说完了设备号，回到struct usb_interface的151行，condition字段表示接口和驱动的绑定状态，enum usb_interface_condition类型，在include/linux/usb.h中定义。

```
83 enum usb_interface_condition {
84       USB_INTERFACE_UNBOUND = 0,
85       USB_INTERFACE_BINDING,
86       USB_INTERFACE_BOUND,
87       USB_INTERFACE_UNBINDING,
88 };
```

前面介绍Linux设备模型时说了，设备和驱动是相生相依的关系，总线上的每个设备和驱动都在等待着命中的那个她，找到了，执子之手与子偕老；找不到，孤苦伶仃。enum usb_interface_condition形象地描绘了这个过程中接口的各种心情，孤苦、期待、幸福、分开，人生又何尝不是如此？

152行，153行与157行都是关于挂起和唤醒的。协议中规定，所有的USB设备都必须支持挂起状态，就是说为了达到节电的目的，当设备在指定的时间内，大约3 ms吧，如果没有发生总线传输，就要进入挂起状态。当它收到一个non-idle的信号时，就会被唤醒。

152行is_active表示接口是否处于挂起状态。153行needs_remote_wakeup表示是否需要打开远程唤醒功能。远程唤醒功能允许挂起的设备给主机发信号，通知主机它将从挂起状态恢复，注意如果主机处于挂起状态，就会唤醒主机，不然主机仍然在休眠。协议中并没有要求USB设备一定要实现远程唤醒的功能，即使实现了，从主机这边也可以打开或关闭它。

157行中的pm_usage_cnt，pm就是电源管理，usage_cnt就是使用计数，当它为0时，接口允许autosuspend。什么是autosuspend？用过笔记本电脑吧，有时合上笔记本电脑后，它会自动进入休眠，这就叫autosuspend。但不是每次都是这样的，就像这里只有当pm_usage_cnt为0时才会允许接口autosuspend。至于这个计数在哪里统计，暂时还是略过吧。

接下来就剩下155行的struct device dev和156行的struct device *usb_dev，看到struct device，或许就会认为它们是Linux设备模型中的device嵌在这儿的对象。不过这么想是不准确的，这两个成员里面只有dev才是模型中的device嵌在这儿的，usb_dev则不是。当接口使用USB_MAJOR作为主设备号时，usb_dev才会用到。你找遍整个内核，也只在usb_register_dev和usb_deregister_dev两个函数中能够看到它，usb_dev指向的就是usb_register_dev函数中创建的USB class device。

15、设置是接口的设置

最近看了一些韩国的反转剧，什么是反转剧？就是影片儿演到一半时让你猜最后的结局，并且结局十有八九都会出乎你的预料。虽然每片只有短短的20多分钟，但论故事情节，比咱们的帘子幽梦之类的看了几十集还不知道演什么的电视剧精彩得多。

咱们在生活中是不会有这般戏剧性的反转和悬念的，有的只是吃饭的人生，上班的人生，睡觉的人生。思源湖边做了多少次的白日梦，毕业时就被扔到了湖水里，然后我们从平淡走向平庸。

不过这里还是有一点小悬念的，前面介绍struct usb_interface时，表示接口设置的struct usb_host_interface就被有意无意地略过了，现在看一看它的真面目，同样在include/linux/usb.h文件中定义。

```
70 struct usb_host_interface {
71        struct usb_interface_descriptor desc;
72
73        /* array of desc.bNumEndpoint endpoints associated with this
74         * interface setting.  these will be in no particular order.
75         */
76        struct usb_host_endpoint *endpoint;
77
78        char *string;          /* iInterface string, if present */
79        unsigned char *extra;   /* Extra descriptors */
80        int extralen;
81 };
```

71行，desc，接口描述符。什么是描述符？我们的生活就是一个不断遇到人，认识人的过程，有些人注定只是擦肩而过，有些人却深深地留在我们的内心里，比如USB的描述符。实际上，USB的描述符是一个带有预定义格式的数据结构，里面保存了USB设备的各种属性还有相关信息，比如姓名，生产地等，我们可以通过向设备请求获得它们的内容来深刻地了解感知一个USB设备。

主要有四种USB描述符：设备描述符，配置描述符，接口描述符和端点描述符。协议中规定一个USB设备是必须支持这四大描述符的，当然也有其他一些描述符来让设备可以显得个性一些，但这四大描述符是一个都不能少的。

这些描述符放哪儿？当然是在设备中，就等着主机去拿。具体在哪儿？USB设备中都会有一个叫EEPROM的东西，没错，就是放在那儿，它就是用来存储设备本身信息的。如果你的脑海里还残存着一些大学里的美好时光的话，应该还会记得EEPROM就是电可擦写的可编程ROM，它与Flash虽说都是要电擦除的，但它可以按字节擦除，Flash只能一次擦除一个block，所以如果要改动比较少的数据的话，使用它还是比较合适的。但是世界上没有完美的东西，此物成本相对Flash比较高，所以一般来说USB设备中只拿它来存储一些本身特有的信息，要想存储数据，还是使用Flash吧。

具体到接口描述符，自然就是描述接口本身的信息的。一个接口可以有多个设置，使用不同的设置，描述接口的信息会有所不同，所以接口描述符并没有放在struct usb_interface结构中，而是放在表示接口设置的struct usb_host_interface结构中。在include/linux/usb/ch9.h文件中定义。

```
294 /* USB_DT_INTERFACE: Interface descriptor */
295 struct usb_interface_descriptor {
296       __u8  bLength;
297       __u8  bDescriptorType;
298
299       __u8  bInterfaceNumber;
300       __u8  bAlternateSetting;
301       __u8  bNumEndpoints;
302       __u8  bInterfaceClass;
303       __u8  bInterfaceSubClass;
304       __u8  bInterfaceProtocol;
305       __u8  iInterface;
306 } __attribute__ ((packed));
```

又看到了__attribute__，不过在这里改头换面成了__attribute__ ((packed))，意思就是告诉编译器，这个结构的元素都是1字节对齐的，不要再添加填充位了。因为这个结构和spec里的Table 9.12是完全一致的，包括字段的长度。如果不给编译器这个暗示，编译器就会依据你的平台类型在结构的每个元素之间添加一定的填充位，如果你拿这个添加了填充位的结构去向设备请求描述符，你想一想会是什么结果。

296行，bLength，描述符的字节长度。协议中规定，每个描述符必须以一个字节打头来表明描述符的长度。那可以扳着指头数一下，接口描述符的bLength应该是9，两只手就数完了，没错，ch9.h文件中紧挨着接口描述符的定义就定义了这个长度。

```
308 #define USB_DT_INTERFACE_SIZE          9
```

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京|上海|深圳|成都|南京|武汉|西安|广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

297行，bDescriptorType，描述符的类型。各种描述符的类型都在ch9.h文件中有定义，对应spec中的Table 9.5。对于接口描述符来说，值为USB_DT_INTERFACE，也就是0x04。

299行，bInterfaceNumber，接口号。每个配置可以包含多个接口，这个值就是它们的索引值。

300行，bAlternateSetting，接口使用的是哪个可选设置。协议中规定，接口默认使用的设置总为0号设置。

301行，bNumEndpoints，接口拥有的端点数量。这里并不包括端点0，端点0是所有的设备都必须提供的，所以这里就没必要多此一举的包括它了。

302行，bInterfaceClass；303行，bInterfaceSubClass；304行，bInterfaceProtocol。这个世界上有许许多多的USB设备，它们各有各的特点，为了区分它们，USB规范，或者说USB协议把USB设备分成了很多类，然而每个类又分成子类。这很好理解，我们的一个大学也是如此，先是分成很多个学院，然后每个学院又被分为很多个系，然后可能每个系下边又分了各个专业，USB协议也是这样干的，首先每个Device或Interface属于一个Class，然后Class下面又分了SubClass，完了SubClass下面又按各种设备所遵循的不同的通信协议继续细分。USB协议中边为每一种Class，每一种SubClass，每一种Protocol定义一个数值，比如Mass Storage的Class就是0x08，Hub的Class就是0x09。

305行，iInterface，接口对应的字符串描述符的索引值。这里怎么又跳出来一个叫字符串描述符的东西？你没看错我也没说错，除了前面提到的四大描述符，还有字符串描述符。不过四大描述符是每个设备必须支持的，这个字符串描述符却是可有可无的，有了你欢喜，我也欢喜，没有也不是什么问题。使用lsusb命令看一下。

```
localhost:/usr/src/linux/drivers/usb/core # lsusb
Bus 001 Device 013: ID 04b4:1081 Cypress Semiconductor Corp.
Bus 001 Device 001: ID 0000:0000
```

第1行里显示的是我手边儿的Cypress USB开发板，看里面的Cypress Semiconductor Corp.，这么一长串的东西从哪里来？是不是应该从设备中来？设备的那几个标准描述符，整个描述符的大小也不一定放得下这么一长串，所以，一些设备专门准备了一些字符串描述符（string descriptor），就用来记这些长串的东西。字符串描述符主要就是提供一些设备接口相关的描述性信息，比如厂商的名字，产品序列号等。字符串描述符当然可以有多个，这里的索引值就是用来区分它们的。

说过了接口描述符，回到struct usb_host_interface的76行，endpoint，一个数组，表示这个设置所使用到端点。至于端点的结构struct usb_host_endpoint，让它先一边儿凉快凉快吧，咱们先看完struct usb_host_interface再去说它。

78行，string，用来保存从设备中取出来的字符串描述符信息的，既然字符串描述符可有可无，那这里的指针也有可能为空了。

79行，extra；80行，extralen，有关额外的描述符。除了前面提到的四大描述符及字符串描述符外，还有为一组设备也就是一类设备定义的描述符，和厂商为设备特别定义的描述符，extra指的就是它们，extralen表示它们的长度。

16、端点

折腾USB spec的同志应该不会读过庄子，也不会知道于丹这个人物，可别人也知道端点，于是端点成了USB数据传输的终点。看一看它在内核中的定义。

```
59 struct usb_host_endpoint {
60       struct usb_endpoint_descriptor  desc;
61       struct list_head               urb_list;
62       void                          *hcpriv;
63       struct ep_device               *ep_dev;       /* For sysfs info */
65       unsigned char *extra;   /* Extra descriptors */
66       int extralen;
67 };
```

华清远见

60行，desc，端点描述符，四大描述符的第二个隆重登场了。它也在include/linux/usb/ch9.h中定义。

```
312 /* USB_DT_ENDPOINT: Endpoint descriptor */
313 struct usb_endpoint_descriptor {
314         __u8  bLength;
315         __u8  bDescriptorType;
316
317         __u8  bEndpointAddress;
318         __u8  bmAttributes;
319         __le16 wMaxPacketSize;
320         __u8  bInterval;
321
322         /* NOTE:  these two are _only_ in audio endpoints. */
323         /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324         __u8  bRefresh;
325         __u8  bSynchAddress;
326 } __attribute__ ((packed));
328 #define USB_DT_ENDPOINT_SIZE          7
329 #define USB_DT_ENDPOINT_AUDIO_SIZE     9       /* Audio extension */
```

这个结构与spec中的Table 9.13是一一对应的，0号端点仍然保持着它特殊的地位，它没有自己的端点描述符。

314行，bLength，描述符的字节长度，数一下，前面有7个字节，后面又多了两个字节，那是针对音频设备扩展的，紧接着struct usb_host_endpoint定义的就是两个长度值的定义。

315行，bDescriptorType，描述符类型，这里对于端点就是USB_DT_ENDPOINT，0x05。

317行，bEndpointAddress，这个字段描述的信息挺多的，比如这个端点是输入端点还是输出端点，这个端点的地址，以及这个端点的端点号。它的bits 0~3表示的就是端点号，你使用0x0f和它相与就可以得到端点号。不过，开发内核的同志想得都很周到，定义好了一个掩码USB_ENDPOINT_NUMBER_MASK，它的值就是0x0f。当然，这是为了让咱们更容易去读他们的代码，也为了以后的扩展。另外，它的bit 8表示方向，输入还是输出，同样有掩码USB_ENDPOINT_DIR_MASK，值为0x80，将它和bEndpointAddress相与，并结合USB_DIR_IN和USB_DIR_OUT作判断就可以得到端点的方向。

```
48 #define USB_DIR_OUT                     0               /* to device */
49 #define USB_DIR_IN                      0x80            /* to host */
```

318行，bmAttributes，属性，总共8位，其中bit1和bit0共同称为Transfer Type，即传输类型， 00表示控制，01表示等时，10表示批量，11表示中断。前面的端点号还有端点方向都有配对的掩码，这里当然也有，就在struct usb_endpoint_descriptor定义的下面。

```
338 #define USB_ENDPOINT_XFERTYPE_MASK      0x03    /* in bmAttributes */
339 #define USB_ENDPOINT_XFER_CONTROL       0
340 #define USB_ENDPOINT_XFER_ISOC          1
341 #define USB_ENDPOINT_XFER_BULK          2
342 #define USB_ENDPOINT_XFER_INT           3
```

319行，wMaxPacketSize，端点一次可以处理的最大字节数。比如你老板比较看重你，一次给你交代了几个任务，于是你大声疾呼：“神啊，我一次只能做一个。”当然神是听不到的，怎么办？加班加点，一个一个地分开做吧。端点也是，如果你发送的数据量大于端点的这个值，也会分成多次一次一次来传输。友情提醒一下，这个字段还是有点儿门道的，对不同的传输类型也有不同的要求，日后碰到了再说。

320行，bInterval，USB是轮询式的总线，这个值表达了端点一种美好的期待，希望主机轮询自己的时间间隔，但实际上批准不批准就是主机的事了。不同的传输类型bInterval也有不同的意义，暂时就提这么一下，碰到各个实际的传输类型了再去说它。

回到struct usb_host_endpoint，61行，urb_list，端点要处理的urb队列。urb是什么？它可是USB通信的主角，包含了执行USB传输所需要的所有信息，你要想和你的USB通信，就得创建一个urb，并且为它赋好值，交给USB Core，然后USB Core会找到合适的主机控制器，从而进行具体的数据传输。设备中的每个端点都可以处理一个urb队列。当然，urb是内核中对USB传输数据的封装也叫抽象，协议中可不这么叫。基于urb特殊的江湖地位，在后面的章节我会对它大书特书的。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

62行，hcpriv，这是提供给HCD（Host Controller Driver）用的。比如等时端点会在里边儿放一个ehci_iso_stream，什么意思？郑板桥告诉我们要难得糊涂。

63行，ep_dev，这个字段是供sysfs用的。好奇的话可以去/sys下看一看

```
localhost:/usr/src/linux # ls /sys/bus/usb/devices/usb1/ep_00/
bEndpointAddress bmAtributes  direction  subsystem  wMaxpacketSize
bInterval   dev    interval type
bLength    device      power  uevent
```

ep_00端点目录下的这些文件从哪儿来的？就是在usb_create_ep_files函数中使用ep_dev创建的。

65行，extra；66行，extralen，有关一些额外扩展的描述符的，和struct usb_host_interface里差不多，只是这里的是针对端点的，如果你请求从设备中获得描述符信息，它们会跟在标准的端点描述符后面返回给你。

17、设备

struct usb_device结构冗长而又杂乱。struct usb_device还是得说。

```
336 struct usb_device {
337     int             devnum;         /* Address on USB bus */
338     char            devpath [16]; /* Use in messages: /port/port/... */
339     enum usb_device_state   state;/* configured, not attached, etc */
340     enum usb_device_speed   speed;  /* high/full/low (or error) */
341
342     struct usb_tt   *tt;       /* low/full speed dev, highspeed hub */
343     int             ttport;        /* device port on that tt hub */
344
345     unsigned int toggle[2];    /* one bit for each endpoint
346                                    * ([0] = IN, [1] = OUT) */
347
348     struct usb_device *parent;   /* our hub, unless we're the root */
349     struct usb_bus *bus;           /* Bus we're part of */
350     struct usb_host_endpoint ep0;
351
352     struct device dev;             /* Generic device interface */
353
354     struct usb_device_descriptor descriptor;/* Descriptor */
355     struct usb_host_config *config; /* All of the configs */
356
357     struct usb_host_config *actconfig;/* the active configuration */
358     struct usb_host_endpoint *ep_in[16];
359     struct usb_host_endpoint *ep_out[16];
360
361     char **rawdescriptors;       /* Raw descriptors for each config */
362
363     unsigned short bus_mA;       /* Current available from the bus */
364     u8 portnum;                   /* Parent port number (origin 1) */
365     u8 level;                     /* Number of USB hub ancestors */
366
367     unsigned discon_suspended:1;   /* Disconnected while suspended */
368     unsigned have_langid:1;      /* whether string_langid is valid */
369     int string_langid;             /* language ID for strings */
370
371     /* static strings from the device */
372     char *product;                 /* iProduct string, if present */
373     char *manufacturer;         /* iManufacturer string, if present */
374     char *serial;                /* iSerialNumber string, if present */
375
376     struct list_head filelist;
377 #ifdef CONFIG_USB_DEVICE_CLASS
378     struct device *usb_classdev;
379 #endif
380 #ifdef CONFIG_USB_DEVICEFS
381     struct dentry *usbfs_dentry;/*usbfs dentry entry for the device*/
382 #endif
383     /*
384      * Child devices - these can be either new devices
385      * (if this is a hub device), or different instances
386      * of this same device.
```

```
387      *
388      * Each instance needs its own set of data structures.
389      */
391     int maxchild;                   /* Number of ports if hub */
392     struct usb_device *children[USB_MAXCHILDREN];
393
394     int pm_usage_cnt;                /* usage counter for autosuspend */
395     u32 quirks;                     /* quirks of the whole device */
396
397 #ifdef CONFIG_PM
398     struct delayed_work autosuspend; /* for delayed autosuspends */
399     struct mutex pm_mutex;          /* protects PM operations */
400
401     unsigned long last_busy;         /* time of last use */
402     int autosuspend_delay;          /* in jiffies */
403
404     unsigned auto_pm:1;             /* autosuspend/resume in progress */
405     unsigned do_remote_wakeup:1;/* remote wakeup should be enabled */
406     unsigned autosuspend_disabled:1; /* autosuspend and autoresume */
407     unsigned autoresume_disabled:1;  /*  disabled by the user */
408 #endif
409 };
```

337行，devnum：设备的地址。此地址非彼地址，和咱们写程序时说的地址不一样。devnum只是USB设备在一条USB总线上的编号。你的USB设备插到Hub上时，Hub观察到这个变化，于是来了精神，会在一个漫长而又曲折的处理过程中调用一个名叫choose_address的函数，为你的设备选择一个地址。

就像在一个浪漫的季节的某个温馨的下午，你去港汇下边儿的那个必胜客吃饭，同样会领取一个属于自己的编号陪伴自己度过一个漫长的过程。有人说我没有用Hub，我的USB设备直接插到主机的USB接口上了。我哭，即使你没有用Hub，也总要明白主机里还会有个叫Root Hub的东西吧，不管是一般的Hub还是Root Hub，你的USB设备总要通过一个Hub才能在USB的世界里生活。

现在来认识一下USB子系统里面关于地址的游戏规则。在USB世界里，一条总线就是大树一棵，一个设备就是一片叶子。为了记录这棵树上的每一个叶子节点，每条总线设有一个地址映射表，即struct usb_bus结构体里有一个成员struct usb_devmap devmap。

```
268 /* USB device number allocation bitmap */
269 struct usb_devmap {
270     unsigned long devicemap[128 / (8*sizeof(unsigned long))];
271 };
```

什么是usb_bus？前面不是已经有了一个struct bus_type类型的usb_bus_type了吗？没错，在USB子系统的初始化函数usb_init里已经注册了usb_bus_type，不过那是让系统知道有这么一个类型的总线。而一个总线类型和一条总线是两码事儿。从硬件上来讲，一个主机控制器就会连出一条USB总线；而从软件上来讲，不管你有多少个主机控制器，或者说有多少条总线，它们通通属于usb_bus_type这个类型，只是每一条总线对应一个struct usb_bus结构体变量，这个变量在主机控制器的驱动程序中申请。

上面的devmap地址映射表就表示了一条总线上设备连接的情况，假设unsigned long为4bytes，那么unsigned long devicemap[128/(8*sizeof(unsigned long)]]就等价于unsigned long devicemap[128/(8*4)]，进而等价于unsigned long devicemap[4]，而4bytes就是32 bits，因此这个数组最终表示的就是128 bits。而这也对应于一条总线可以连接128个USB设备。之所以这里使用sizeof(unsigned long)，就是为了跨平台应用，不管unsigned long到底是几，总之这个devicemap数组最终可以表示128位，也就是说每条总线上最多可以连上128个设备。

338行，devpath [16]，它显然是用来记录一个字符串的，这个字符串是什么意思？给你看一个直观的东西，如下所示。

```
localhost:~ # ls /sys/bus/usb/devices/
1-0:1.0  2-1      2-1:1.1  4-0:1.0  4-5:1.0  usb2  usb4
2-0:1.0  2-1:1.0  3-0:1.0  4-5      usb1     usb3
```

在sysfs文件系统下，我们可以看到这些乱七八糟的东西，它们都是什么？usb1、usb2、usb3、usb4表示我的计算机上接了4条USB总线，即4个USB主机控制器，事物多了自然就要编号，就跟我们中学或大学里面的学号一样，用于区分多个个体。

而4-0:1.0表示什么？4表示是4号总线，或者说4号Root Hub；0就是这里我们说的devpath，1表示配置为1号，0表示接口号为0。也即是说，4号总线的0号端口的设备，使用的是1号配置，接口号为0。

那么devpath是否就是端口号呢？显然不是，这里我列出来的这个例子是只有Root Hub没有级联Hub的情况，如果在Root Hub上又接了别的Hub，然后一级一级连下去，子又生孙，孙又生子，子又有子，子又有孙。子子孙孙，无穷匮也。那么如何在sysfs里面来表征这整个大家族呢？这就是devpath的作用，顶级的设备其devpath就是其连在Root Hub上的端口号，而次级的设备就是其父hub的devpath后面加上其端口号，即如果4-0:1.0如果是一个Hub，那么它下面的1号端口的设备就可以是4-0.1:1.0；2号端口的设备就可以是4-0.2:1.0；3号端口就可以是4-0.3:1.0。总的来说，就是端口号一级一级往下加。这个思想是很简单的，也是很朴实的。

339行，state：设备的状态。这是一个枚举类型。

```
557 enum usb_device_state {
558      /* NOTATTACHED isn't in the USB spec, and this state acts
559       * the same as ATTACHED ... but it's clearer this way.
560       */
561      USB_STATE_NOTATTACHED = 0,
562
563      /* chapter 9 and authentication (wireless) device states */
564      USB_STATE_ATTACHED,
565      USB_STATE_POWERED,                        /* wired */
566      USB_STATE_UNAUTHENTICATED,                 /* auth */
567      USB_STATE_RECONNECTING,                   /* auth */
568      USB_STATE_DEFAULT,                        /* limited function */
569      USB_STATE_ADDRESS,
570      USB_STATE_CONFIGURED,                      /* most functions */
572      USB_STATE_SUSPENDED
573
574      /* NOTE:  there are actually four different SUSPENDED
575       * states, returning to POWERED, DEFAULT, ADDRESS, or
576       * CONFIGURED respectively when SOF tokens flow again.
577       */
578 };
```

上面定义了9种状态，spec里只定义了6种，Attached，Powered，Default，Address，Configured，Suspended，对应于spec中的Table 9.1。

Attached表示设备已经连接到USB接口上了，是Hub检测到设备时的初始状态。那么这里所谓的USB_STATE_NOTATTACHED就是表示设备并没有Attached。

Powered是加电状态。USB设备的电源可以来自外部电源，协议中叫做self-powered，也可以来自hub，称为bus-powered。尽管self-powered的USB设备可能在连接上USB接口以前已经上电，但它们直到连上USB接口后才能被看作是Powered的，你觉得它已经上电了那是站在你的角度看，可是现在你看的是usbcore，所以要放弃个人的成见，团结在core的周围。

Default默认状态，在Powered之后，设备必须在收到一个复位（reset）信号并成功复位后，才能使用默认地址回应主机发过来的设备和配置描述符的请求。

Address状态表示主机分配了一个唯一的地址给设备，此时设备可以使用默认管道响应主机的请求。真羡慕这些USB设备，住的地方都是包分配的，哪像咱们辛辛苦苦一路小跑着也不一定能达到Address状态。

Configured状态表示设备已经被主机配置过了，也就是协议中说的处理了一个带有非0值的SetConfiguration()请求，此时主机可以使用设备提供的所有功能。

Suspended挂起状态，为了省电，设备在指定的时间内，大约3 ms吧，如果没有发生总线传输，就要进入挂起状态。此时，USB设备要自己维护包括地址、配置在内的信息。

USB设备从生到死都要按照这么几个状态，遵循这么一个过程。

340行，speed：设备的速度，这也是一个枚举变量。

```
550 enum usb_device_speed {
551      USB_SPEED_UNKNOWN = 0,                    /* enumerating */
552      USB_SPEED_LOW, USB_SPEED_FULL,             /* usb 1.1 */
```

```
553     USB_SPEED_HIGH,                        /* usb 2.0 */
554     USB_SPEED_VARIABLE,                     /* wireless (usb 2.5) */
555 };
```

在USB 3.0以前，设备有三种速度，低速、全速、高速。USB1.1在那时只有低速和全速，后来才出现了高速，就是所谓的480 Mb/s。这里还有一个USB_SPEED_VARIABLE，是无线USB的，号称USB 2.5。USB_SPEED_UNKNOWN只是表示现阶段还不知道这个设备究竟什么速度。

342行，tt；343行，ttport。知道tt是干什么的吗？tt全拼为transaction translator。你可以把它想成一块特殊的电路，是Hub里面的电路，确切地说是高速Hub中的电路，我们知道USB设备有三种速度，分别是Low Speed，Full Speed，High Speed。即所谓的低速、全速及高速，就像在抗日战争那会儿，这个世界上只有低速/全速的设备，没有高速的设备，后来解放后，国民生产力的大幅度提升催生了一种高速的设备，包括主机控制器，以前只有两种接口的，OHCI/UHCI，这都是在USB spec 1.0时，后来2.0推出了EHCI，高速设备应运而生。

Hub也有高速的Hub和低速的Hub，但是这里就有一个兼容性问题了，高速的Hub是否能够支持低速/全速的设备呢？一般来说是不支持的，于是有了一个叫做TT的电路，它就负责高速和低速/全速的数据转换。于是，如果一个高速设备中有这么一个TT，那么就可以连接低速/全速设备，要不然，低速/全速设备就没法用，只能连接到OHCI/UHCI那边出来的Hub口里。

345行，toggle[2]，这个数组只有两个元素，分别对应IN和OUT端点，每一个端点占一位。似乎这么说仍是在雾中看花，黑格尔告诉我们，存在就是有价值的，那么这个数组存在的价值是什么？一言难尽，说来话长，那就长话长说好了。

前面说，你要想和你的USB通信，创建一个urb，为它赋好值，交给咱们的USB Core就可以了。这个urb是站在咱们的角度，实际上在USB cable里流淌的根本就不是那么回事儿，咱们提交的是urb，USB cable里流淌的是一个一个的数据包（packet）。

咱们凄苦的人生是从第一声哭开始，所有的packets都从一个SYNC同步字段开始，SYNC是一个8位长的二进制串，只是用来同步用的，它的最后两位标志了SYNC的结束和PID（Packet Identifer）的开始。

PID也是一个8位的二进制串，前四位用来区分不同的packet类型，后面四位只是前四位的反码，校验用的。正如spec的Table 8-1里描述的那样，共有四大类的packet，分别是Token、Data、Handshake和Special，每个大类又包含了几个子类。

主机和设备都是纯理性的东西，完全通过PID来判断送过来的packet是不是自己所需要的，不像咱们，往往缺乏这么一个用来判断的标准，不知道自己究竟需要的是什么。

PID之后紧跟着的是地址字段，每个packet都需要知道自己要往哪里去，它们是一个一个目的明确的精灵，行走在USB cable里。这个地址实际上包括两部分，7位表示了总线上连接的设备或接口的地址，4位表示端点的地址，这就是为什么前面说每条USB总线最多只能有128个设备，即使是高速设备除了0号端点也最多只能有15个in端点和15个out端点。

地址字段再往后是11位的帧号（frame number），值达到7FFH时归零。这个帧号并不是每一个packet都会有，它只在每帧或微帧（Mircoframe）开始的SOF Token包中发送。帧是对于低速和全速模式来说的，一帧就是1 ms，对于高速模式的称呼是微帧，一个微帧为125 ms，每帧或微帧当然不会只能传一个packet。

帧号再往后就是千呼万唤始出来的Data字段了，它可以有0到1024个字节不等。最后还有CRC校验字段来做扫尾工作。

咱们要学习packet，但这里只看一看Data类型的packet。有四种类型的Data包，DATA0，DATA1，DATA2和MDATA。存在就是有价值的，这里分成4种数据包自然有其道理，其中DATA0和DATA1就可以用来实现data toggle同步，看到toggle，好像有点接近不久之前留下的疑问了。

对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为DATA0的，然后为了传输的正确性，就传一次DATA0，再传一次DATA1，一旦哪次打破了这种平衡，主机就可以认为传输出错了。对于等时传输来说，data toggle并不被支持。USB就是在使用这种简单的哲学来判断对与错。

华清远见

我们的struct usb_device中的数组unsigned int toggle[2]就是为了支持这种简单的哲学而生的，它里面的每一位表示的就是每个端点当前发送或接收的数据包是DATA0还是DATA1。

348行，parent，struct usb_device结构体的parent自然也是一个struct usb_device指针。对于Root Hub，前面说过，它是和Host Controller是绑定在一起的。它的parent指针在Host Controller的驱动程序中就已经赋了值，这个值就是NULL。换句话说，对于Root Hub，它不需要再有父指针了，这个父指针就是给从Root Hub连出来的节点用的。USB设备是从Root Hub开始，一个一个往外面连。比如Root Hub有4个口，每个口连一个USB设备，比如其中有一个是Hub，那么这个Hub有可以继续有多个口，于是一级一级地往下连，最终连成了一棵树。

349行，bus：设备所在的那条总线。

350行，ep0：端点0的特殊地位决定了它必将受到特殊的待遇，在struct usb_device对象产生时它就要初始化。

353行，dev：嵌入到struct usb_device结构中的struct device结构。

354行，desc：设备描述符，四大描述符的第三个姗姗而来。它在include/linux/usb/ch9.h中定义。

```
203 /* USB_DT_DEVICE: Device descriptor */
204 struct usb_device_descriptor {
205       __u8  bLength;
206       __u8  bDescriptorType;
207
208       __le16 bcdUSB;
209       __u8  bDeviceClass;
210       __u8  bDeviceSubClass;
211       __u8  bDeviceProtocol;
212       __u8  bMaxPacketSize0;
213       __le16 idVendor;
214       __le16 idProduct;
215       __le16 bcdDevice;
216       __u8  iManufacturer;
217       __u8  iProduct;
218       __u8  iSerialNumber;
219       __u8  bNumConfigurations;
220 } __attribute__ ((packed));
221
222 #define USB_DT_DEVICE_SIZE            18
```

205行，bLength：描述符的长度，可以自己数一数，或者看紧接着的定义USB_DT_DEVICE_SIZE。

206行，bDescriptorType：这里对于设备描述符应该是USB_DT_DEVICE，0x01。

208行，bcdUSB：USB spec的版本号，一个设备如果能够进行高速传输，那么它的设备描述符里的bcdUSB这一项就应该为0200H。

209行，bDeviceClass；210行，bDeviceSubClass；211行，bDeviceProtocol，和接口描述符的意义差不多。

212行，bMaxPacketSize0：端点0一次可以处理的最大字节数，端点0的属性却放到设备描述符里去了，更加彰显了它突出的江湖地位。

前面说端点时说了端点0并没有一个专门的端点描述符，因为不需要，基本上它所有的特性都在spec里规定好了的。然而，别忘了这里说的是“基本上”，有一个特性则是不一样的，这叫做maximum packet size，每个端点都有这么一个特性，即告诉你该端点能够发送或者接收的包的最大值。对于通常的端点来说，这个值被保存在该端点描述符中的wMaxPacketSize这一个field，而对于端点0就不一样了，由于它自己没有一个描述符，而每个设备又都有这么一个端点，所以这个信息被保存在了设备描述符里，所以我们在设备描述符里可以看到这么一项，bMaxPacketSize0。而且spec还规定了，这个值只能是8，16，32或者64这四者之一，如果一个设备工作在高速模式，这个值还只能是64，如果是工作在低速模式，则只能是8，取别的值都不行。

213行，idVendor；214行，idProduct，分别是厂商和产品的ID。

215行，bcdDevice：设备的版本号。

216行，iManufacturer；217行，iProduct；218行，iSerialNumber，分别是厂商，产品和序列号对应的字符串描述符的索引值。

219行，bNumConfigurations：设备当前速度模式下支持的配置数量。有的设备可以在多个速度模式下操作，这里包括的只是当前速度模式下的配置数目，不是总的配置数目。

这就是设备描述符，它和spec中的Table 9-8是一一对应的。咱们回到struct usb_device的355行，config；357行，actconfig，分别表示设备拥有的所有配置和当前激活的，也就是正在使用的配置。USB设备的配置用struct usb_host_config结构来表示，下节再说。

358行，ep_in[16]，359行，ep_out[16]，除了端点0，一个设备即使在高速模式下也最多只能再有15个IN端点和15个OUT端点，端点0太特殊了，对应的管道是message管道，又能进又能出，所以这里的ep_in和ep_out数组都有16个值。

361行，rawdescriptors，这是一个字符指针数组，数组里的每一项都指向一个使用GET_DESCRIPTOR请求去获得配置描述符时所得到的结果。考虑一下，为什么我只说得到的结果，而不直接说得到的配置描述符？不是请求的就是配置描述符吗？这是因为当你使用GET_DESCRIPTOR去请求配置描述符时，设备返回给你的不仅仅只有配置描述符，它把该配置所包括的所有接口的接口描述符，还有接口里端点的端点描述符一股脑的都塞给你了。第一个接口的接口描述符紧跟着这个配置描述符，然后是这个接口下面端点的端点描述符，如果有还有其他接口，它们的接口描述符和端点描述符也跟在后面，这里面，专门为一类设备定义的描述符和厂商定义的描述符跟在它们对应的标准描述符后面。

这里提到了GET_DESCRIPTOR请求，就顺便简单提一下USB的设备请求(device request)。协议中说了，所有的设备通过默认的控制管道来响应主机的请求，既然使用的是控制管道，那当然就是控制传输了，这些请求的底层packet属于Setup类型。协议中同时也定义了一些标准的设备请求，并规定所有的设备必须响应它们，即使它们还处于Default或Address状态。在这些标准的设备请求里，GET_DESCRIPTOR就赫然在列。

363行，bus_mA，这个值是在主机控制器的驱动程序中设置的，通常来讲，计算机的USB端口可以提供500mA的电流。

364行，portnum，不管是Root Hub还是一般的Hub，你的USB设备总归要插在一个Hub的端口上才能用，portnum就是那个端口号。当然，对于Root Hub这个USB设备来说它本身没有portnum这么一个概念，因为它不插在别的Hub的任何一个口上。所以对于Root Hub来说，它的portnum在主机控制器的驱动程序里给设置成了0。

365行，level，层次，也可以说是级别，表征usb设备树的级连关系。Root Hub的level当然就是0，其下面一层就是level 1，再下面一层就是level 2，依此类推。

366行，discon_suspended，Disconnected while suspended。

368行，have_langid，369行，string_langid，USB设备中的字符串描述符使用的是UNICODE编码，可以支持多种语言，string_langid就是用来指定使用哪种语言的，have_langid用来判断string_langid是否有效。

372行，product，373行，manufacturer；374行，serial，分别用来保存产品、厂商和序列号对应的字符串描述符信息。

376行~382行，usbfs相关的。

391行，maxchild，Hub的端口数，注意可不包括上行端口。

392行，children[USB_MAXCHILDREN]，USB_MAXCHILDREN是include/linux/usb.h中定义的一个宏，值为31。

```
324 #define USB_MAXCHILDREN         (31)
```

华清远见

其实Hub可以接一共255个端口，不过实际上遇到的Hub最多的也就是说自己支持10个端口的，所以31基本上够用了。

394行，pm_usage_cnt，struct usb_interface结构中也有。

396行，quirks，简单地说就是大家的常用语“毛病”。

397行，看到#ifdef CONFIG_PM这个标志，我们就知道从这里直到最后的那个#endif都是关于电源管理的。

18、配置

还是接着看USB设备的配置吧，在include/linux/usb.h文件中定义

```
244 struct usb_host_config {
245     struct usb_config_descriptor    desc;
246
247     char *string;           /* iConfiguration string, if present */
248     /* the interfaces associated with this configuration,
249      * stored in no particular order */
250     struct usb_interface *interface[USB_MAXINTERFACES];
251
252     /* Interface information available even when this is not the
253      * active configuration */
254     struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
255
256     unsigned char *extra;   /* Extra descriptors */
257     int extralen;
258 };
```

245行，desc，四大描述符里最后的一个终于出现了，同样是在include/linux/usb/ch9.h中定义。

```
258 struct usb_config_descriptor {
259     __u8  bLength;
260     __u8  bDescriptorType;
261
262     __le16 wTotalLength;
263     __u8  bNumInterfaces;
264     __u8  bConfigurationValue;
265     __u8  iConfiguration;
266     __u8  bmAttributes;
267     __u8  bMaxPower;
268 } __attribute__ ((packed));
269
270 #define USB_DT_CONFIG_SIZE              9
```

259行，bLength，描述符的长度，值为USB_DT_CONFIG_SIZE。

260行，bDescriptorType，描述符的类型，值为USB_DT_CONFIG，0x02。这么说对不对？按照前面接口描述符、端点描述符和设备描述符的习惯来说，应该是没问题。但是，这里的值却并不仅仅可以为USB_DT_CONFIG，还可以为USB_DT_OTHER_SPEED_CONFIG，0x07。这里说的OTHER_SPEED_CONFIG描述符描述的是高速设备操作在低速或全速模式时的配置信息，和配置描述符的结构完全相同，区别只是描述符的类型不同，是只有名字不同的孪生兄弟。

262行，wTotalLength，使用GET_DESCRIPTOR请求从设备中获得配置描述符信息时，返回的数据长度，也就是说对包括配置描述符、接口描述符、端点描述符，class-或vendor-specific描述符在内的所有描述符算了个总账。

263行，bNumInterfaces，这个配置包含的接口数目。

263行，bConfigurationValue，对于拥有多个配置的幸运设备来说，可以用这个值为参数，使用SET_CONFIGURATION请求来改变正在被使用的 USB配置，bConfigurationValue就指明了将要激活哪个配置。设备虽然可以有多个配置，但同一时间却也只能有一个配置被激活。顺便说一下，SET_CONFIGURATION请求也是标准的设备请求之一，专门用来设置设备的配置。

265行，iConfiguration，描述配置信息的字符串描述符的索引值。

266行，bmAttributes，这个字段表征了配置的一些特点，比如bit 6为1表示self-powered，bit 5为1表示这个配置支持远程唤醒。另外，它的bit 7必须为1，ch9.h里有几个相关的定义：

```
272 /* from config descriptor bmAttributes */
273 #define USB_CONFIG_ATT_ONE              (1 << 7)    /* must be set */
274 #define USB_CONFIG_ATT_SELFPOWER        (1 << 6)    /* self powered */
275 #define USB_CONFIG_ATT_WAKEUP           (1 << 5)    /* can wakeup */
276 #define USB_CONFIG_ATT_BATTERY          (1 << 4)    /* battery powered */
```

267行，bMaxPower，设备正常运转时，从总线那里分得的最大电流值，以2mA为单位。设备可以使用这个字段向Hub表明自己需要的电流，但如果设备需求过于多，请求的超出了Hub所能给予的，Hub就会直接拒绝。还记得struct usb_device结构中的bus_mA吗？它就表示Hub所能够给予的。Alan Stern告诉我们：

```
(c->desc.bMaxPower * 2) is what the device requests and udev->bus_mA is what the hub makes available.
```

到此为止，四大标准描述符已经全部登场亮相了，还是回到struct usb_host_config结构的247行，string，这个字符串保存了配置描述符iConfiguration字段对应的字符串描述符信息。

250行，interface[USB_MAXINTERFACES]，配置所包含的接口。注释里说的很明确，这个数组的顺序未必是按照配置里接口号的顺序，所以你要想得到某个接口号对应的struct usb_interface结构对象，就必须使用drivers/usb/usb.c中定义的usb_ifnum_to_if函数。

```
84 struct usb_interface *usb_ifnum_to_if(const struct usb_device *dev,
85                                       unsigned ifnum)
86 {
87         struct usb_host_config *config = dev->actconfig;
88         int i;
89
90         if (!config)
91             return NULL;
92         for (i = 0; i < config->desc.bNumInterfaces; i++)
93             if (config->interface[i]->altsetting[0]
94                          .desc.bInterfaceNumber == ifnum)
95             return config->interface[i];
96
97         return NULL;
98 }
```

这个函数的道理很简单，就是拿你指定的接口号，和当前配置的每一个接口可选设置0里的接口描述符的bInterfaceNumber字段做比较，如果相等，那个接口就是你要寻找的，如果都不相等，不能满足你的要求，虽然它已经尽力了。

如果你看了协议，可能会在9.6.5节里看到，请求配置描述符时，配置里的所有接口描述符是按照顺序一个一个返回的。那为什么这里又明确说明，让咱们不要期待它就会是接口号的顺序？其实以前这里并不是这么说的，这个数组是按照0..desc.bNumInterfaces的顺序，但同时又需要通过usb_ifnum_to_if函数来获得指定接口号的接口对象，Alan Stern质疑了这种有些矛盾的说法，于是David Brownell就把它改成现在这个样子了，为什么改？因为协议归协议，厂商归厂商，有些厂商就是不遵守协议，它非要先返回接口1再返回接口0，所以就不得不增加usb_ifnum_to_if函数。

USB_MAXINTERFACES是drivers/usb/usb.h中定义的一个宏，值为32，不要说不够用，谁见过有很多接口的设备？

```
/* this maximum is arbitrary */
#define USB_MAXINTERFACES    32
```

254行，intf_cache[USB_MAXINTERFACES]，cache是缓存。这是个struct usb_interface_cache对象的结构数组，usb_interface，usb接口，cache，缓存，所以usb_interface_cache就是USB接口的缓存。缓存些什么？查看include/linux/usb/usb.h里的定义。

```
193 struct usb_interface_cache {
194         unsigned num_altsetting;        /* number of alternate settings */
195         struct kref ref;              /* reference counter */
196
```

```
197     /* variable-length array of alternate settings for this interface,
198      * stored in no particular order */
199     struct usb_host_interface altsetting[0];
200 };
```

199行的altsetting[0]是一个可变长数组，按需分配的那种，你对设备说GET_DESCRIPTOR时，内核就根据返回的每个接口可选设置的数目分配给intf_cache数组相应的空间，有多少需要分配多少。

为什么要缓存这些东西？为了在配置被取代之后仍然能够获取它的一些信息，就把日后可能会需要的一些东西放在了intf_cache数组的struct usb_interface_cache对象里。谁会需要？这么说吧，你通过sysfs这个窗口只能看到设备当前配置的一些信息，即使是这个配置下面的接口，也只能看到接口正在使用的可选设置的信息，可是你希望能够看到更多的，怎么办，窗户太小了，可以趴门口看，usbfs就是这个门，里面显示有系统中所有USB设备的可选配置和端点信息，它就是利用intf_cache这个数组里缓存的东西实现的。

256行，extra，257行，extralen，有关额外扩展的描述符的，和struct usb_host_interface里的差不多，只是这里的是针对配置的，如果你使用GET_DESCRIPTOR请求从设备中获得配置描述符信息，它们会紧跟在标准的配置描述符后面返回给你。

19、向左走，向右走

我们回到前面提到的函数usb_device_match，之前做的所有铺垫，只是为了与它再次相见。

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545     /* interface drivers never match devices */
546     if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551
552     } else {
553         struct usb_interface *intf;
554         struct usb_driver *usb_drv;
555         const struct usb_device_id *id;
556
557         /* device drivers never match interfaces */
558         if (is_usb_device_driver(drv))
559             return 0;
560
561         intf = to_usb_interface(dev);
562         usb_drv = to_usb_driver(drv);
563
564         id = usb_match_id(intf, usb_drv->id_table);
565         if (id)
566         return 1;
567
568         id = usb_match_dynamic_id(intf, usb_drv);
569         if (id)
570         return 1;
571     }
572
573     return 0;
574 }
```

在USB的世界里，对于设备和驱动来说只是usb_device_match函数的两端。usb_device_match函数为它们指明向左走还是向右走，为它们指明哪个才是它们命中注定的缘。

543行，第一次遇到这个函数时，我说了这里有两条路，一条给USB设备走，一条给USB接口走。先来查看设备走的这条路，上面只有两个函数。

```
85 static inline int is_usb_device(const struct device *dev)
```

```
86 {
87      return dev->type == &usb_device_type;
88 }
```

drivers/usb/core/usb.h中定义的这个函数就是要告诉你，只有usb_device才能打这儿过。但关键问题不是它让不让你进，而是它怎么知道你是不是usb_device。

关键就在于这个dev->type，设备的类型，看它是不是等于定义的usb_device_type，也就是说USB设备类型，相等的话，那可以通过，不相等，那就此路不是为你开的，你找别的路吧。usb_device_type在drivers/usb/core/usb.c中定义：

```
195 struct device_type usb_device_type = {
196      .name =         "usb_device",
197      .release =      usb_release_dev,
198 };
```

它和咱们前面看到的那个usb_bus_type差不多，一个表示总线的类型，一个表示设备的类型，总线有总线的类型，设备有设备的类型。

假设现在过来一个设备，经过判断，它要走的是设备这条路，可问题是，这个设备的type字段什么时候被初始化成usb_device_type了。这倒是个问题，不过先不说明，继续向前走，带着疑问上路。

546行，又见到一个if，它就在上面的is_usb_device函数后面在drivers/usb/core/usb.h文件中定义：

```
92 static inline int is_usb_device_driver(struct device_driver *drv)
93 {
94      return container_of(drv, struct usbdrv_wrap, driver)->
95                      for_devices;
96 }
```

这个函数用于判断是不是usb device driver，那什么是usb device driver？前面不是一直都是说一个USB接口对应一个USB驱动吗？我可以负责任地告诉你前面说的一点都没有错，一个接口就是要对应一个USB驱动，可是我们不能只钻到接口的那个接口里边儿，我们应该眼光放的更加开阔些，要知道接口在USB的世界里并不是老大，它上边儿还有配置，还有设备，都比它大。

每个接口对应了一个独立的功能，是需要专门的驱动来和它交流，但是接口毕竟整体是作为一个USB设备而存在的，设备还可以有不同的配置，我们还可以为设备指定特定的配置，那谁来做这个事情？struct usb_device _driver，即USB设备驱动，它和USB的接口驱动struct usb_driver都定义在include/linux/usb.h文件中。

```
833 struct usb_driver {
834      const char *name;
835
836      int (*probe) (struct usb_interface *intf,
837                    const struct usb_device_id *id);
838
839      void (*disconnect) (struct usb_interface *intf);
840
841      int (*ioctl) (struct usb_interface *intf, unsigned int code,
842                     void *buf);
843
844      int (*suspend) (struct usb_interface *intf, pm_message_t message);
845      int (*resume) (struct usb_interface *intf);
846
847      void (*pre_reset) (struct usb_interface *intf);
848      void (*post_reset) (struct usb_interface *intf);
849
850      const struct usb_device_id *id_table;
851
852      struct usb_dynids dynids;
853      struct usbdrv_wrap drvwrap;
854      unsigned int no_dynamic_id:1;
855      unsigned int supports_autosuspend:1;
856 };
```

一般来说，我们平时所谓的编写USB驱动指的也就是写USB接口的驱动，需要以一个struct usb_driver结构的对象为中心，以设备的接口提供的功能为基础，开展USB驱动的建设。

834行，name，驱动程序的名字，对应了在/sys/bus/usb/drivers/下面的子目录名称。和我们每个人

一样，它只是彼此区别的一个代号，不同的是我们可以有很多人同名，但这里的名字在所有的USB驱动中必须是唯一的。

836行，probe，用来看一看这个usb驱动是否愿意接受某个接口的函数。每个驱动自诞生起，它的另一半就已经确定了。当然，一个驱动往往可以支持多个接口。

839行，disconnect，当接口失去联系，或使用rmmod卸载驱动将它和接口强行分开时这个函数就会被调用。

841行，ioctl，当你的驱动有通过usbfs和用户空间交流的需要的话，就使用它吧。

844行，suspend，845行，resume，分别在设备被挂起和唤醒时使用。

847行，pre_reset，848行，post_reset，分别在设备将要复位（reset）和已经复位后使用。

850行，id_table，驱动支持的所有设备的列表，驱动就靠这张表来识别是不是支持哪个设备接口的，如果不属于这张表，那就不支持。

852行，dynids，支持动态id的。什么是动态id？本来前面刚说每个驱动诞生时它的另一半在id_table里就已经确定了，可是Greg显然在一年多前加入了动态id的机制。即使驱动已经加载了，也可以添加新的id给它，只要新id代表的设备存在，她就会和他绑定起来。

怎么添加新的id？到驱动所在的地方看一看，也就是/sys/bus/usb/drivers目录下边，那里列出的每个目录就代表了一个USB驱动，随便选一个进去，能够看到一个new_id文件吧，使用echo将厂商和产品id写进去就可以了。查看Greg举的一个例子：

```
echo 0557 2008 > /sys/bus/usb/drivers/foo_driver/new_id
```

就可以将16进制值0557和2008写到foo_driver驱动的设备id表里。

853行，drvwrap，这个字段是，struct usbdrv_wrap结构，也在include/linux/usb.h中定义。

```
779 struct usbdrv_wrap {
780      struct device_driver driver;
781      int for_devices;
782 };
```

近距离观察一下这个结构，它里面内容是比较贫乏的，只有一个struct device_driver结构的对象和一个for_devices的整型字段。回想一下Linux的设备模型，我们就会产生这样的疑问，这里的struct device_driver对象不是应该嵌入到struct usb_driver结构中吗，怎么这里又了一层？

再这么一层当然不是为了美观，这主要还是因为本来驱动在USB的世界里不得已分成了设备驱动和接口驱动两种，为了区分这两种驱动，就中间加了这么一层，添了个for_devices标志来判断是哪种。

不知大家发现没有，之前见识过的结构中，很多不是1就是0的标志使用的是位字段，特别是几个这样的标志放一块儿时，而这里的for_devices虽然也只能有两个值，但却没有使用位字段，为什么？简单地说就是这里没必要，那些使用位字段的是几个在一块儿，可以节省点儿存储空间，而这里只有这么一个，就是使用位字段也节省不了，就不用多此一举了。

其实就这么说为了加一个判断标志就硬生生的塞这么一层，还是会有点模糊的，不过，其他字段不敢说，这个drvwrap以后肯定还会遇到它，这里先简单介绍一下。

854行，no_dynamic_id，可以用来禁止动态id的。

855行，supports_autosuspend，对autosuspend的支持，如果设置为0的话，就不再允许绑定到这个驱动的接口autosuspend。

struct usb_driver结构就暂时了解到这里，咱们再来查看所谓的USB设备驱动与接口驱动到底都有多大的不同。

```
878 struct usb_device_driver {
879      const char *name;
881      int (*probe) (struct usb_device *udev);
```

```
882     void (*disconnect) (struct usb_device *udev);
884     int (*suspend) (struct usb_device *udev, pm_message_t message);
885     int (*resume) (struct usb_device *udev);
886     struct usbdrv_wrap drvwrap;
887     unsigned int supports_autosuspend:1;
888 };
```

这个USB设备驱动比前面的接口驱动要简洁多了，除了少了很多东西外，剩下的将参数中的struct usb_interface换成struct usb_device后就几乎一摸一样了。

友情提醒一下，这里说的是几乎，而不是完全，这是因为probe，它的参数中与接口驱动里的probe相比少了设备的列表，也就是说它不用再去根据列表来判断是不是愿意接受一个USB设备。那么这意味着什么？是它来者不拒，接受所有的USB设备？还是拒绝所有的USB设备？当然只会是前者，不然这个USB设备驱动就完全毫无疑义了，而且我们在内核中找来找去，也就只能找得着它在drivers/usb/core/generic.c文件中定义了usb_generic_driver这个对象：

```
210 struct usb_device_driver usb_generic_driver = {
211     .name = "usb",
212     .probe = generic_probe,
213     .disconnect = generic_disconnect,
214 #ifdef  CONFIG_PM
215     .suspend = generic_suspend,
216     .resume = generic_resume,
217 #endif
218     .supports_autosuspend = 1,
219 };
```

即使这么一个对象也早在usb_init的895行就已经注册给USB子系统了。那么我们该用什么样的言语表达自己的感受？所谓的USB设备驱动完全就是一个博爱的主儿，我们的core用它来与所有的USB设备进行交流，它们都交流些什么？这是后话，我会告诉你的。

不管怎么说，总算是把USB接口的驱动和设备的驱动给讲了一下，还是回到这节开头的usb_device_match函数，目前为止，设备这条路已经比较清晰了。就是如果设备过来了，走到了设备这条路，然后要判断一下驱动是不是设备驱动，是不是针对整个设备的，如果不是的话，对不起，虽然这条路走对了，可是沿这条路，设备找不到对应的驱动，匹配不成功，就直接返回了，那如果驱动也确实是设备驱动呢？代码里是直接返回1，表示匹配成功了，没有再花费哪怕多一点儿的精力。

本来这么说应该是可以了，可是我还是忍不住想告诉你，在之前的内核版本里，是没有很明确的struct usb_device_driver这样一个表示usb设备驱动的结构的，而是直接定义了struct device_driver结构类型的一个对象 usb_generic_driver来处理与整个设备相关的事情，相对应的，usb_device_match这个匹配函数也只有简单的一条路，在接口和对应的驱动之间做匹配。但是在2006年内核中多了struct usb_device_driver结构，usb_device_match这里也多了一条给设备走的路。

是时候也有必要对USB设备在USB世界里的整个人生旅程做一个介绍了。与usb_device_match短暂相遇便要再次分开，不过，最重要的是见得到。

20、设备的生命线

设备的生命线（一）

设备也有它自己的生命线，自你把它插到Hub上始，自你把它从Hub上拔下来终，它的一生是勤勉努力、朴实无华的一生，它的一生是埋头苦干、默默奉献的一生。BH的人生不需要解释，设备的人生值得我们去分析。

当然你将USB设备连接在Hub的某个端口上，Hub检测到有设备连接了进来，它会为设备分配一个struct usb_device结构的对象并初始化，调用设备模型提供的接口将设备添加到USB总线的设备列表里，然后USB总线会遍历驱动列表里的每个驱动，调用自己的match函数看它们和你的设备或接口是否匹配。这不，又走到match函数了。

华清远见

Hub检测到自己的某个端口有设备连接了进来后，它会调用core里的usb_alloc_dev函数为struct usb_device结构的对象申请内存，这个函数在drivers/usb/core/usb.c文件中定义。

```
238 struct usb_device *
239 usb_alloc_dev(struct usb_device *parent, struct usb_bus *bus, unsigned port1)
240 {
241     struct usb_device *dev;
242 
243     dev = kzalloc(sizeof(*dev), GFP_KERNEL);
244     if (!dev)
245         return NULL;
246 
247     if (!usb_get_hcd(bus_to_hcd(bus))) {
248         kfree(dev);
249         return NULL;
250     }
251 
252     device_initialize(&dev->dev);
253     dev->dev.bus = &usb_bus_type;
254     dev->dev.type = &usb_device_type;
255     dev->dev.dma_mask = bus->controller->dma_mask;
256     dev->state = USB_STATE_ATTACHED;
257 
258     INIT_LIST_HEAD(&dev->ep0.urb_list);
259     dev->ep0.desc.bLength = USB_DT_ENDPOINT_SIZE;
260     dev->ep0.desc.bDescriptorType = USB_DT_ENDPOINT;
261     /* ep0 maxpacket comes later, from device descriptor */
262     dev->ep_in[0] = dev->ep_out[0] = &dev->ep0;
263 
264     /* Save readable and stable topology id, distinguishing devices
265      * by location for diagnostics, tools, driver model, etc.  The
266      * string is a path along hub ports, from the root.  Each device's
267      * dev->devpath will be stable until USB is re-cabled, and hubs
268      * are often labeled with these port numbers.  The bus_id isn't
269      * as stable:  bus->busnum changes easily from modprobe order,
270      * cardbus or pci hotplugging, and so on.
271      */
272     if (unlikely(!parent)) {
273         dev->devpath[0] = '';
274 
275         dev->dev.parent = bus->controller;
276         sprintf(&dev->dev.bus_id[0], "usb%d", bus->busnum);
277     } else {
278         /* match any labeling on the hubs; it's one-based */
279         if (parent->devpath[0] == '')
280         snprintf(dev->devpath, sizeof dev->devpath,
281                         "%d", port1);
282         else
283             snprintf(dev->devpath, sizeof dev->devpath,
284                         "%s.%d", parent->devpath, port1);
285 
286         dev->dev.parent = &parent->dev;
287         sprintf(&dev->dev.bus_id[0], "%d-%s",
288                     bus->busnum, dev->devpath);
289 
290         /* hub driver sets up TT records */
291     }
292 
293     dev->portnum = port1;
294     dev->bus = bus;
295     dev->parent = parent;
296     INIT_LIST_HEAD(&dev->filelist);
297 
298 #ifdef  CONFIG_PM
299     mutex_init(&dev->pm_mutex);
300     INIT_DELAYED_WORK(&dev->autosuspend, usb_autosuspend_work);
301     dev->autosuspend_delay = usb_autosuspend_delay * HZ;
302 #endif
303     return dev;
304 }
```

usb_alloc_dev函数就相当于USB设备的构造函数，在参数中边，parent是设备连接的Hub，bus是设备连接的总线，port1就是设备连接在Hub上的端口。

243行，为一个struct usb_device结构的对象申请内存并初始化为0。直到在看到这一行一天前，我还仍在使用kmalloc加me mset这对最佳拍档来申请内存和初始化，但是在看到kzalloc之后，我知道了kzalloc直接取代了kmalloc/me mset，一个函数起到了两个函数的作用。

然后是判断内存是否申请成功，不成功就不用往下走了。那么通过这么几行，咱们应该记住，凡是你想用kmalloc/me mset组合申请内存时，就使用kzalloc代替吧；凡是申请内存的，不要忘了判断是否申请成功了。

247行，这里的两个函数是hcd，是主机控制器驱动里的。要知道USB的世界里一个主机控制器对应着一条USB总线，主机控制器驱动用struct usb_hcd结构表示，一条总线用struct usb_bus结构表示，函数bus_to_hcd是为了获得总线对应的主机控制器驱动，也就是struct usb_hcd结构对象，函数usb_get_hcd只是将得到的这个usb_hcd结构对象的引用计数加1，为什么？因为总线上多了一个设备，当然得为它增加引用计数。如果这俩函数没有很好地完成自己的任务，那整个usb_alloc_dev函数也就没有继续执行下去的必要了，将之前为struct usb_device结构对象申请的内存释放掉就可以了。

252行，device_initialize是设备模型中的函数， 第一个dev是struct usb_device结构体指针，而第二个dev是struct device结构体，这是设备模型中面一个最基本的结构体，使用它必然要先初始化。device_initialize函数的目的就是将struct usb_device结构中嵌入的struct device结构体初始化掉，以后方便调用。

253行，将设备所在的总线类型设置为usb_bus_type。usb_bus_type在前面见过了，USB子系统的初始化函数usb_init里就把它给注册掉了，还记得讲到模型时说过的那个著名的三角关系吗？这里就是把设备和总线这条边给搭上了。

254行，将设备的设备类型初始化为usb_device_type，这是在上节第二次遇到usb_device_match函数，走设备那条路时，使用is_usb_device判断是不是usb设备时留下的疑问，就是在这儿把设备的类型给初始化成usb_device_type了。

255行，这个就是与DMA传输相关的了，设备能不能进行DMA传输，得看主机控制器的脸色，主机控制器不支持的话设备也没法使用。所以这里dma_mask被设置为主机控制器的dma_mask。

256行，将USB设备的状态设置为Attached，表示设备已经连接到USB接口上了，是Hub检测到设备时的初始状态。

258行，端点0实在是太特殊了，struct usb_device里直接就有这么一个成员ep0，这行就将ep0的urb_list给初始化掉了。

259行，260行，分别初始化了端点0的描述符长度和描述符类型。

260行，使struct usb_device结构中的ep_in和ep_out指针数组的第一个成员指向ep0，ep_in[0]和ep_out[0]本来表示的就是端点0。

272行，这里平白无故地多出了一个unlikely，不知道什么意思？先查看它们在include/linux/compiler.h的定义。

```
60 #define likely(x)       __builtin_expect(!!(x), 1)
61 #define unlikely(x)     __builtin_expect(!!(x), 0)
```

除了函数unlikely，还有一个函数likely。定义里那个怪怪的__builtin_expect是GCC里内建的一个函数，具体是做什么用的可以看一看GCC的手册

```
    long __builtin_expect (long exp, long c)
    You may use __builtin_expect to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this ('-fprofile-arcs'), as programmers are notoriously bad at predicting how their progra ms actually perform. However, there are applications in which this data is hard to collect.
    The return value is the value of exp, which should be an integral expression. The value of c must be a compile-time constant. The semantics of the built-in are that it is expected that exp == c. For example:
    if (__builtin_expect (x, 0))
        foo ();
```

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京|上海|深圳|成都|南京|武汉|西安|广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

```
would indicate that we do not expect to call foo, since we expect x to be zero. Since
you are limited to integral expressions for exp, you should use constructions such
as
if (__builtin_expect (ptr != NULL, 1))
    error ();
when testing pointer or floating-point values.
```

大致意思就是由于大部分写代码的在分支预测方面做的比较的糟糕，所以GCC提供了这个内建的函数来帮助处理分支预测，优化程序，它的第一个参数exp为一个整型的表达式，返回值也是这个exp，它的第二个参数c的值必须是一个编译期的常量，那这个内建函数的意思就是exp的预期值为c，编译器可以根据这个信息适当地重排条件语句块的顺序，将符合这个条件的分支放在合适的地方。

具体到unlikely(x)就是告诉编译器条件x发生的可能性不大，那么这个条件块儿里语句的目标码可能就会被放在一个比较远的为止，以保证经常执行的目标码更紧凑。likely则相反。

使用时还是很简单的，就是，if语句你照用，只是如果你觉得if条件为1的可能性非常大时，可以在条件表达式外面包装一个likely()，如果可能性非常小，则用unlikely()包装。那么这里272行的意思就很明显了，就是说写内核的哥们儿觉得你的USB设备直接连接到root hub上的可能性比较小，因为parent指的就是你的设备连接的那个hub。

272行~291行整个的代码就是首先判断你的设备是不是直接连到Root Hub上的，如果是，将dev->devpath[0]赋值为‘0’，以示特殊，然后父设备设为controller，同时把dev->bus_id[]设置为像usb1/usb2/usb3/usb4这样的字符串。如果你的设备不是直接连到Root Hub上的，有两种情况，如果你的设备连接的Hub是直接连到Root Hub上的，则dev->devpath就等于端口号，否则dev->devpath就等于在父Hub的devpath基础上加一个‘.’再加一个端口号，最后把bus_id[]设置成1-/2-/3-/4-这样的字符串后面连接上devpath。

296行，初始化一个队列，usbfs用的。

298~302行，用于电源管理。

设备的生命线（二）

现在设备的struct usb_device结构体已经准备好了，只是还不怎么饱满，Hub接下来就会给它做做整容手术，往里边儿塞点什么，充实一些内容，比如：将设备的状态设置为Powered，也就是加电状态；因为此时还不知道设备支持的速度，于是将设备的speed成员暂时先设置为USB_SPEED_UNKNOWN；设备的级别level当然会被设置为Hub的level加上1了；还有为设备能够从hub那里获得的电流赋值；为了保证通信畅通，Hub还会为设备在总上选择一个独一无二的地址。

表 1.20.1

Devnum	Taken
devpath[16]	Taken
State	USB_STATE_POWERED
Speed	USB_SPEED_UNKNOWN
Parent	设备连接的那个 hub
Bus	设备连接的那条总线
ep0	ep0.urb_list，描述符长度/类型
Dev	dev.bus，dev.type，dev.dma_mask,，dev.parent，dev.bus_id
ep_in[16]	ep_in[0]
ep_out[16]	ep_out[0]
bus_mA	hub->mA_per_port
Portnum	设备连接在 hub 上的那个端口
Level	Hdev->level + 1
Filelist	Taken
pm_mutex	Taken
Autosuspend	Taken
autosuspend_delay	2 * HZ

前面讲过的，设备要想从Powered状态发展到下一个状态Default，必须收到一个复位信号并成功复位。那Hub接下来的动作就很明显了，复位设备，复位成功后，设备就会进入Default状态。

现在就算设备成功复位了，进入了Default状态，同时，Hub也会获得设备真正的速度，那根据这个速度，咱们能知道些什么？起码能够知道端点0一次能够处理的最大数据长度，协议中，对于高速设备，这个值为为64字节，对于低速设备为8字节，而对于全速设备可能为8字节，16字节，32字节，64字节其中的一个。

设备也该进入Address状态了。

设备要想进入Address状态很容易，只要Hub使用core中定义的一个函数usb_control_ msg，发送SET_ADDRESS请求给设备，设备就进入Address状态了。那么设备的address是什么，就是上面的devnum。

那现在咱就来说说usb_control_ msg函数，它在drivers/usb/core/message.c中定义。

```
    120  int  usb_control_ msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8
requesttype,
    121              __u16 value, __u16 index, void *data, __u16 size, int timeout)
    122 {
    123      struct usb_ctrlrequest *dr = kmalloc(sizeof(struct usb_ctrlrequest), GFP_NOIO);
    124      int ret;
    125
    126      if (!dr)
    127      return -ENOMEM;
    128
    129      dr->bRequestType= requesttype;
    130      dr->bRequest = request;
    131      dr->wValue = cpu_to_le16p(&value);
    132      dr->wIndex = cpu_to_le16p(&index);
    133      dr->wLength = cpu_to_le16p(&size);
    134
    135      //dbg("usb_control_ msg");
    136
    137      ret = usb_internal_control_ msg(dev, pipe, dr, data, size, timeout);
    138
    139      kfree(dr);
    140
    141      return ret;
    142 }
```

这个函数主要目的是创建一个控制urb，并把它发送给USB设备，然后等待它完成。urb是什么？前面提到过的，你要想和你的USB通信，就得创建一个urb，并且为它赋值，交给USB Core，它会找到合适的主机控制器，从而进行具体的数据传输。在后面我会具体说明的。

123行，为一个struct usb_ctrlrequest结构体申请了内存，这里又出现了一个新生事物，它在include/linux/usb/ch9.h文件中定义。

```
    140 struct usb_ctrlrequest {
    141      __u8 bRequestType;
    142      __u8 bRequest;
    143      __le16 wValue;
    144      __le16 wIndex;
    145      __le16 wLength;
    146 } __attribute__ ((packed));
```

这个结构完全对应于spec里的Table 9-2，它描述了主机通过控制传输发送给设备的请求（Device Requests）。一直都在羡慕它们，这会儿到体现出咱们的优越了，主机向设备请求些信息必须得按照协议中规定好的格式，不然设备就会不明白主机是什么意思，而咱们就不一样了，不用填这个值那个值。

这个结构描述的request都是在SETUP包中发送的，Setup包是前面说到的Token PID类型中的一种，为了更好理解，这里详细介绍一下控制传输底层的packet情况。控制传输最少要有两个阶段的transaction：SETUP和STATUS，SETUP和STATUS中间的那个DATA阶段是可有可无的。

Transaction这个词在很多地方都有，也算是个跨地区跨学科的热门词汇了，在这里你称它为事务也好，会话也罢，我还是直呼它的原名transaction，可以理解为主机和设备之间形成的一次完整的交流。

USB的transaction可以包括一个Token包、一个Data包和一个Handshake包。

Token、Data和Handshake都属于四种PID类型，前面提到的一个包中的那些部分，如SYNC、PID、地址域、DATA、CRC，并不是所有PID类型的包都会全部包括的。Token包只包括SYNC、PID、地址域、CRC，并没有DATA字段，它的名字起的很形象，就是用来标记所在transaction里接下来动作的，对于Out和Setup Token包，里面的地址域指明了接下来要接收Data包的端点，对于In Token包，地址域指明了接下来哪个端点要发送Data包。

还有，只有主机才有权利发送Token包，协议中就这么规定的。别嫌spec规定太多，又管这个又管那个的，没有规矩不成方圆。

与Token包相比，Data包中没了地址域，多了Data字段，这个Data字段对于低速设备最大为8字节，对于全速设备最大为1023字节，对于高速设备最大为1024字节。里里外外看过去，它就是躲在Token后边儿用来传输数据的。Handshake包的成分就非常简单了，除了SYNC，它就只包含了一个PID，通过PID取不同的值来报告一个transaction的状态，比如数据已经成功接收了等。

控制传输的SETUP transaction一般来说有三个阶段，就是主机向设备发送Setup Token包、然后发送Data0包，如果一切顺利，设备回应ACK Handshake包表示OK，为什么加上“一般”？如果中间的那个Data0包由于某种不可知因素被损坏了，设备就什么都不会回应，这时就成两个阶段了。

SETUP transaction之后，接下来如果控制传输有DATA transaction的话，那就Data0、Data1交叉地发送数据包，前面说过这是为了实现data toggle。最后是STATUS transaction，向主机汇报前面SETUP和DATA阶段的结果，比如表示主机下达的命令已经完成了，或者主机下达的命令没有完成，或者设备正忙着那没工夫去理会主机的那些命令。

这样经过SETUP、DATA、STATUS这三个transaction阶段，一个完整的控制传输完成了。主机接下来可以规划下一次的控制传输。

现在对隐藏在控制传输背后的是是非非摸了个底儿，群众的眼睛是雪亮的，咱们现在应该可以看出之前说requests都在Setup包中发送是有问题的，因为Setup包本身并没有数据字段，严格来说它们应该都是在SETUP transaction阶段里Setup包后的Data0包中发送的。141行，bRequestType，这个字段别看就一个字节，内容很丰富的，大道理往往都包含这种在小地方。它的bit7就表示了控制传输中DATA transaction阶段的方向，当然，如果有DATA阶段的话。bit5~6表示request的类型，是标准的，class-specific的还是vendor-specific的。bit0~4表示了这个请求针对的是设备，接口，还是端点。内核为它们专门量身定做了一批掩码，也在ch9.h文件中。

华清远见

```
42 /*
43  * USB directions
44  *
45  * This bit flag is used in endpoint descriptors' bEndpointAddress field.
46  * It's also one of three fields in control requests bRequestType.
47  */
48 #define USB_DIR_OUT                     0               /* to device */
49 #define USB_DIR_IN                      0x80            /* to host */
50
51 /*
52  * USB types, the second of three bRequestType fields
53  */
54 #define USB_TYPE_MASK                   (0x03 << 5)
55 #define USB_TYPE_STANDARD               (0x00 << 5)
56 #define USB_TYPE_CLASS                  (0x01 << 5)
57 #define USB_TYPE_VENDOR                 (0x02 << 5)
58 #define USB_TYPE_RESERVED               (0x03 << 5)
59
60 /*
61  * USB recipients, the third of three bRequestType fields
62  */
63 #define USB_RECIP_MASK                  0x1f
64 #define USB_RECIP_DEVICE                0x00
65 #define USB_RECIP_INTERFACE             0x01
66 #define USB_RECIP_ENDPOINT              0x02
67 #define USB_RECIP_OTHER                 0x03
68 /* From Wireless USB 1.0 */
69 #define USB_RECIP_PORT                  0x04
```

```
70 #define USB_RECIP_RPIPE                    0x05
```

142行，bRequest，表示具体是哪个request。

143行，wValue，这个字段是request的参数，request不同，wValue就不同。

144行，wIndex，也是request的参数，bRequestType指明request针对的是设备上的某个接口或端点时，wIndex就用来指明是哪个接口或端点。

145行，wLength，控制传输中DATA transaction阶段的长度，方向已经在bRequestType那儿指明了。如果这个值为0，就表示没有DATA transaction阶段，bRequestType的方向位也就无效了。

和struct usb_ctrlrequest的约会暂时就到这里，回到usb_control_ msg函数中。很明显要进行控制传输，得首先创建一个struct usb_ctrlrequest结构体，填上请求的内容。129行到133行就是来使用传递过来的参数初始化这个结构体的。对于刚开始提到的SET_ADDRESS来说，bRequest的值就是USB_REQ_SET_ADDRESS，标准请求之一，ch9.h中定义有。因为SET_ADDRESS请求并不需要DATA阶段，所以wLength为0，而且这个请求是针对设备的，所以wIndex也为0。这么一来，bRequestType的值也只能为0了。因为是设置设备地址的，总得把要设置的地址发给设备，不然设备会不知道主机是什么意思，所以请求的参数wValue就是之前hub已经你的设备指定好的devnum。其实SET_ADDRESS请求各个部分的值spec 9.4.6里都有规定，就和我这里说的一样，不信你去查看。

接下来先看139行，走到这儿就表示成也好败也好，总之这次通信已经完成了，那么struct usb_ctrlrequest结构体也就没用了，没用的东西最好精简掉。

回头看137行，这是引领咱们往深处走了，不过不怕，路有多远，咱们看下去的决心就有多远。

设备的生命线（三）

函数usb_control_ msg调用了usb_internal_control_ msg之后就去一边儿睡大觉了，脏活儿累活儿，全部留给usb_internal_control_ msg去做了。

```
71 static int usb_internal_control_ msg(struct usb_device *usb_dev,
72                               unsigned int pipe,
73                               struct usb_ctrlrequest *cmd,
74                               void *data, int len, int timeout)
75 {
76       struct urb *urb;
77       int retv;
78       int length;
80       urb = usb_alloc_urb(0, GFP_NOIO);
81       if (!urb)
82           return -ENOMEM;
83
84      usb_fill_control_urb(urb, usb_dev, pipe, (unsigned char *)cmd, data,
85                          len, usb_api_blocking_completion, NULL);
86
87       retv = usb_start_wait_urb(urb, timeout, &length);
88       if (retv < 0)
89           return retv;
90       else
91           return length;
92 }
```

这个函数粗看过去，可以概括为一个中心，三个基本点，以一个struct urb结构体为中心，以usb_alloc_urb、usb_fill_control_urb、usb_start_wait_urb三个函数为基本点。

一个中心：struct urb结构体，就是咱们前面多次提到又多次略过，只闻其名不见其形的传说中的urb，全称usb request block，站在咱们的角度看，USB通信靠的就是它这张脸。

第一个基本点：usb_alloc_urb函数，创建一个urb，struct urb结构体只能使用它来创建，它是urb在USB世界里的独家代理。

第二个基本点：usb_fill_control_urb函数，进行初始化控制urb，urb被创建之后，在使用之前必须要正确的初始化。

第三个基本点：usb_start_wait_urb函数，将urb提交给USB Core，以便分配给特定的主机控制器驱动进行处理，然后默默地等待处理结果，或者超时。

```
1126 struct urb
1127 {
1128     /* private: usb core and host controller only fields in the urb */
1129     struct kref kref;              /* reference count of the URB */
1130     spinlock_t lock;               /* lock for the URB */
1131     void *hcpriv;              /* private data for host controller */
1132     atomic_t use_count;          /* concurrent submissions counter */
1133     u8 reject;                    /* submissions will fail */
1134
1135  /* public: documented fields in the urb that can be used by drivers */
1136     struct list_head urb_list;     /* list head for use by the urb's
1137                                     * current owner */
1138     struct usb_device *dev;  /* (in) pointer to associated device */
1139     unsigned int pipe;             /* (in) pipe information */
1140     int status;                  /* (return) non-ISO status */
1141     unsigned int transfer_flags;    /* (in) URB_SHORT_NOT_OK | ...*/
1142     void *transfer_buffer;          /* (in) associated data buffer */
1143     dma_addr_t transfer_dma; /* (in) dma addr for transfer_buffer */
1144     int transfer_buffer_length;     /* (in) data buffer length */
1145     int actual_length;          /* (return) actual transfer length */
1146     unsigned char *setup_packet;/*(in) setup packet (control only)*/
1147     dma_addr_t setup_dma;       /* (in) dma addr for setup_packet */
1148     int start_frame;               /* (modify) start frame (ISO) */
1149     int number_of_packets;          /* (in) number of ISO packets */
1150     int interval;                 /* (modify) transfer interval
1151                                     * (INT/ISO) */
1152     int error_count;               /* (return) number of ISO errors */
1153     void *context;                /* (in) context for completion */
1154     usb_complete_t complete;        /* (in) completion routine */
1155     struct usb_iso_packet_descriptor iso_frame_desc[0];
1156                                    /* (in) ISO ONLY */
1157 };
```

1129行，kref，urb的引用计数。别看它是隐藏在urb内部的一个不起眼的小角色，但小角色做大事情，它决定了一个urb的生死存亡。一个urb有用没用，是继续委以重任还是无情销毁都要看它的脸色。那第一个问题就来了，为什么urb的生死要掌握在这个小小的引用计数手里？

前面说过，主机与设备之间通过管道来传输数据，管道的一端是主机上的一个缓冲区，另一端是设备上的端点。管道之中流动的数据，在主机控制器和设备看来是一个个packets，在咱们看来就是urb。因而，端点之中就有这么一个队列，叫urb队列。不过，这并不代表一个urb只能发配给一个端点，它可能通过不同的管道发配给不同的端点，那么这样一来，我们如何知道这个urb正在被多少个端点使用，如何判断这个urb的生命已经结束？如果没有任何一个端点在使用它，而我们又无法判断这种情况，它就会永远地飘荡在USB的世界里。我们需要寻求某种办法在这种情况下给它们一个好的归宿，这就是引用计数。每多一个使用者，它的引用计数就加1，每减少一个使用者，引用计数就减1，如果连最后一个使用者都释放了这个urb，宣称不再使用它了，那它的生命周期就走到了尽头，会自动销毁。

接下来就是第二个问题，如何来表示这个神奇的引用计数？其实它是一个struct kref结构体，在include/linux/kref.h中定义。

```
23 struct kref {
24       atomic_t refcount;
25 };
```

这个结构与struct urb相比简约到极致了，简直就是迎着咱们的口味来的。不过别看它简单，内核中就是使用它来判断一个对象有没有用。它里边儿只包括了一个原子变量，为什么是原子变量？既然都使用引用计数了，那就说明可能同时有多个地方在使用这个对象，总要考虑一下它们同时修改这个计数的可能性吧，也就是俗称的并发访问，那怎么办？加一个锁？就这么一个整数值专门加个锁未免也大材小用了，所以就使用了原子变量。围绕这个结构，内核中还定义了几个专门操作引用计数的函数，它们在lib/kref.c中定义。

```
21 void kref_init(struct kref *kref)
22 {
23       atomic_set(&kref->refcount,1);
```

```
24       smp_mb();
25 }
26
31 void kref_get(struct kref *kref)
32 {
33       WARN_ON(!atomic_read(&kref->refcount));
34       atomic_inc(&kref->refcount);
35       smp_mb__after_atomic_inc();
36 }
37
52 int kref_put(struct kref *kref, void (*release)(struct kref *kref))
53 {
54       WARN_ON(release == NULL);
55       WARN_ON(release == (void (*)(struct kref *))kfree);
56
57       if (atomic_dec_and_test(&kref->refcount)) {
58       release(kref);
59           return 1;
60       }
61       return 0;
62 }
```

整个kref.c文件就定义了这么三个函数，kref_init初始化，kref_get将引用计数加1，kref_put将引用计数减1并判断是不是为0，为0的话就调用参数中release函数指针指向的函数把对象销毁掉。它们对refcount的操作都是通过原子变量特有的操作函数，原子变量当然要使用专门的操作函数了，编译器还能做些优化，否则直接使用一般的变量就可以了，为什么还要使用原子变量，不是没事找事儿吗？再说如果你直接像对待一般整型值一样对待它，编译器也会看不过去你的行为，直接给你一个error的。

提醒一下，kref_init初始化时，是把refcount的值初始化为1的，不是0。还有一点要说的是kref_put参数中的那个函数指针，你不能传递一个NULL过去，否则这个引用计数就只是计数，而背离了最初的目的。要记住，我们需要在这个计数减为0时将嵌入这个引用计数struct kref结构体的对象给销毁掉，所以这个函数指针也不能为kfree，因为这样的话就只是把这个struct kref结构体给销毁了，而不是整个对象。

第三个问题，如何使用struct kref结构来为我们的对象计数？当然我们需要把这样一个结构嵌入到你希望计数的对象里，不然你根本就无法对对象在它整个生命周期里的使用情况作出判断。但是我们应该是几乎见不到内核中边儿直接使用上面那几个函数来给对象计数的，而是每种对象又定义了自己专用的引用计数函数，比如咱们的urb，在drivers/usb/core/urb.c中定义。

```
31 void usb_init_urb(struct urb *urb)
32 {
33       if (urb) {
34       me mset(urb, 0, sizeof(*urb));
35           kref_init(&urb->kref);
36           spin_lock_init(&urb->lock);
37       }
38 }
39
81 void usb_free_urb(struct urb *urb)
82 {
83       if (urb)
84       kref_put(&urb->kref, urb_destroy);
85 }
86
97 struct urb * usb_get_urb(struct urb *urb)
98 {
99       if (urb)
100          kref_get(&urb->kref);
101      return urb;
102 }
```

usb_init_urb、usb_get_urb、usb_free_urb这三个函数分别调用了前面看到的struct kref结构的三个操作函数来进行引用计数的初始化、加1、减1。什么叫封装？这就叫封装。usb_init_urb和usb_get_urb都没什么好说的，比较感兴趣的是usb_free_urb里给kref_put传递的那个函数urb_destroy，它也在urb.c中定义。

```
9 #define to_urb(d) container_of(d, struct urb, kref)
10
11 static void urb_destroy(struct kref *kref)
```

```
12 {
13     struct urb *urb = to_urb(kref);
14     kfree(urb);
15 }
```

这个urb_destroy首先调用了to_urb，实际上就是一个container_of来获得引用计数关联的那个urb，然后使用kfree将它销毁。

到此，世界如此美丽，引用计数如此简单，不是吗？

回到struct urb，1130行，lock，一把自旋锁。每个urb都有一把自旋锁。

1131行，hcpriv，走到今天，你应该明白这个urb最终还是要提交给主机控制器驱动的，这个字段就是urb里主机控制器驱动的自留地。

1132行，use_count，这里又是一个使用计数，不过此计数非彼计数，它与上面那个用来追踪urb生命周期的kref一点儿血缘关系也没有，连远亲都不是。那它是用来做什么的，凭什么在臃肿的struct urb不断地喊着要瘦身时还仍有一席之地？

先了解下使用urb来完成一次完整的USB通信都要经历哪些阶段。首先，驱动程序发现自己有与USB设备通信的需要，于是创建一个urb，并指定它的目的地是设备上的哪个端点，然后提交给USB Core，USB Core将它修修补补进行一些美化之后再移交给主机控制器的驱动程序HCD，HCD会去解析这个urb，了解它的目的是什么，并与USB设备进行相应的交流，在交流结束，urb的目的达到之后，HCD再把这个urb的所有权移交回驱动程序。

这里的use_count就是在USB Core将urb移交给HCD，办理移交手续时，插上了那么一脚，每当走到这一步，它的值就会加1。什么时候减1？在HCD重新将urb的所有权移交回驱动程序时。这样说吧，只要HCD拥有这个urb的所有权，那么该urb的use_count就不会为0。这么一说，似乎use_count也有一点追踪urb生命周期的味道了，当它的值大于0时，就表示当前有HCD正在处理它，和上面的kref概念上有部分的重叠，不过，显然它们之间是有区别的，没区别的话，这里干吗要用两个计数？

上面的那个kref实现方式是内核中统一的引用计数机制，当计数减为0时，urb对象就被urb_destroy给销毁了。这里的use_count只是用来统计当前这个urb是不是正在被哪个HCD处理，即使它的值为0，也只是说明没有HCD在使用它而已，并不代表就得把它给销毁掉。比方说，HCD利用完了urb，把它还给了驱动，这时驱动还可以对这个urb进行检修，再提交给哪个HCD去使用。

下面的问题就是既然它不会平白无故地多出来，那它究竟是用来干什么的？还要从刚提到的那几个阶段说起。urb驱动也创建了，提交也提交了，HCD正处理着，可驱动反悔了，它不想再继续这次通信了，想将这个urb给终止掉，善解任意的USB Core当然会给驱动提供这样的接口来满足这样的需要。不过这个需要还被写代码的哥们儿细分为两种，一种是驱动只想通过USB Core告诉HCD一声，说这个urb我想终止掉，您就别费心再处理了，然后它不想在那里等着HCD的处理，想忙别的事去，这就是俗称的“异步”，对应的是usb_unlink_urb函数。当然对应的还有另一种同步，驱动会在那里苦苦等候着HCD的处理结果，等待着urb被终止，对应的是usb_kill_urb函数。而HCD将这次通信终止后，同样会将urb的所有权移交回驱动。那么驱动通过什么判断HCD已经终止了这次通信？就是通过这里的use_count，驱动会在usb_kill_urb里面一直等待着这个值变为0。

1133行，reject，拒绝，拒绝什么？又是被谁拒绝？

在目前版本的内核中，只有usb_kill_urb函数有特权对它进行修改，那么，显然reject就与上面说的urb终止有关了。那就看一看drivers/usb/core/urb.c中定义的这个函数。

```
464 void usb_kill_urb(struct urb *urb)
465 {
466     might_sleep();
467     if (!(urb && urb->dev && urb->dev->bus))
468         return;
469     spin_lock_irq(&urb->lock);
470     ++urb->reject;
471     spin_unlock_irq(&urb->lock);
473     usb_hcd_unlink_urb(urb, -ENOENT);
```

```
474     wait_event(usb_kill_urb_queue, atomic_read(&urb->use_count) == 0);
475
476     spin_lock_irq(&urb->lock);
477     --urb->reject;
478     spin_unlock_irq(&urb->lock);
479 }
```

466行，因为usb_kill_urb函数要一直等候着HCD将urb终止掉，它必须是可以休眠的。所以说usb_kill_urb不能用在中断上下文，必须能够休眠将自己占的资源给让出来。

写代码的人于是提供了might_sleep函数，用它来判断一下这个函数是不是处在能够休眠的情况，如果不是，就会打印出一大堆的堆栈信息，比如你在中断上下文调用了这个函数时。不过，它也就是基于调试的目的用一用，方便日后找错，并不能强制哪个函数改变自己的上下文。

467行，这里就是判断一下urb要去的那个设备，还有那个设备现在的总线有没有，如果不存在，就还是返回吧。

469行，去获得每个urb都有的那把锁，然后将reject加1。加1有什么用？其实目前版本的内核中只有两个地方用到了这个值进行判断。第一个地方是在USB Core将urb提交给HCD，正在办移交手续时，如果reject大于0，就不再接着移交了，也就是说这个urb被HCD给拒绝了。这是为了防止这边儿正在终止这个urb，那边儿的某个地方却又妄想将这个urb重新提交给HCD。

473行，这里告诉HCD驱动要终止这个urb了，usb_hcd_unlink_urb函数也只是告诉HCD一声，然后不管HCD怎么处理就返回了。

474行，上面的usb_hcd_unlink_urb是返回了，但并不代表HCD已经将urb给终止了，HCD可能没那么快，所以这里usb_kill_urb要休息一下，等人通知它。这里使用了wait_event宏来实现休眠，usb_kill_urb_queue是在/drivers/usb/core/hcd.h中定义的一个等待队列，专门给usb_kill_urb休息用的。需要注意的是这里的唤醒条件use_count必须等于0，终于看到use_count实战的地方了。

在哪里能唤醒正在睡大觉的usb_kill_urb？这牵扯到了第二个使用reject来做判断的地方。在HCD将urb的所有权返还给驱动时，会对reject进行判断，如果reject大于0，就调用wake_up唤醒在usb_kill_urb_queue上休息的usb_kill_urb。这也好理解，HCD都要将urb的所有权返回给驱动了，那当然就是已经处理完了，放在这里就是已经将这个urb终止了，usb_kill_urb等的就是这一天的到来，当然就要醒过来继续往下走了。

476行，再次获得urb的那把锁，将reject刚才增加的那个1给减掉。urb都已经终止了，也没人再会去拒绝它了，reject还是开始什么样结束时就什么样吧。

与usb_kill_urb相比，usb_unlink_urb函数就简单多了。

```
435 int usb_unlink_urb(struct urb *urb)
436 {
437     if (!urb)
438         return -EINVAL;
439     if (!(urb->dev && urb->dev->bus))
440         return -ENODEV;
441     return usb_hcd_unlink_urb(urb, -ECONNRESET);
442 }
```

usb_unlink_urb函数只是把自己的意愿告诉HCD，然后就非常洒脱返回了。

struct urb结构中的前面这几个函数，只是USB Core和主机控制器驱动需要关心的，实际的驱动里根本用不着也管不着，它们就是usb和HCD的后花园，想种点什么不种点什么都由写这块儿代码的哥们儿决定，它们在里面怎么为所欲为都不关写驱动的人什么事。USB在linux里起起伏伏这么多年，前边儿的这些内容早就变过多少次，说不定你今天还能看到谁，到接下来的哪天就看不到了，不过，变化的是形式，不变的是道理。

而驱动要做的只是创建一个urb，然后初始化，再把它提交给USB Core就可以了，使用不使用引用计数，加不加锁之类的一点都不用去操心。感谢David Brownell，感谢Alan Stern，感谢……没有他们就没有USB在Linux里的今天。

华清远见

设备的生命线（四）

在struct urb中，就是每个写usb驱动的人都需要关心的了，坐这儿看了半天，struct urb才露出来这么一个角儿。

1136行，urb_list，还记得每个端点都会有的那个urb队列吗？那个队列就是由这里的urb_list一个一个的链接起来的。HCD每收到一个urb，就会将它添加到这个urb指定的那个端点的urb队列里去。这个链表的头儿在哪儿？当然是在端点里，就是端点里的那个struct list_head结构体成员。

1138行，dev，它表示urb要去的那个usb设备。

1139行，pipe，urb到达端点之前，需要经过一个通往端点的管道，就是这个pipe。那第一个问题，怎么表示一个管道？人生有两极，管道有两端，一端是主机上的缓冲区，一端是设备上的端点。既然有两端，总要有个方向吧！前面说过，端点有四种类型，那么与端点相生相依的管道也应该不只一种。

这么说来，确定一条管道至少要知道两端的地址、方向和类型，不过这两端里主机是确定的，需要确定的只是另一端设备的地址和端点的地址。那怎么将这些内容揉合起来表示成一个管道？一个包含了各种成员属性的结构再加上一些操作函数？多么完美的封装，但是不需要这么搞，一个整型值再加上一些宏就够了。

先看一看管道，也就是这个整型值的构成，bit7用来表示方向，bit8~14表示设备地址，bit15~18表示端点号，早先说过，设备地址用7位来表示，端点号用4位来表示，剩下来的bit30~31表示管道类型。再看一看围绕管道的一些宏，在include/linux/usb.h中定义。

```
1407 #define PIPE_ISOCHRONOUS              0
1408 #define PIPE_INTERRUPT                1
1409 #define PIPE_CONTROL                  2
1410 #define PIPE_BULK                     3
1411
1412 #define usb_pipein(pipe)        ((pipe) & USB_DIR_IN)
1413 #define usb_pipeout(pipe)       (!usb_pipein(pipe))
1415 #define usb_pipedevice(pipe)    (((pipe) >> 8) & 0x7f)
1416 #define usb_pipeendpoint(pipe)  (((pipe) >> 15) & 0xf)
1418 #define usb_pipetype(pipe)      (((pipe) >> 30) & 3)
1419 #define usb_pipeisoc(pipe)  (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
1420 #define usb_pipeint(pipe)     (usb_pipetype((pipe)) == PIPE_INTERRUPT)
1421 #define usb_pipecontrol(pipe)   (usb_pipetype((pipe)) == PIPE_CONTROL)
1422 #define usb_pipebulk(pipe)      (usb_pipetype((pipe)) == PIPE_BULK)
```

这些宏什么意思，就不用我说了。

现在看第二个问题，如何创建一个管道？主机和设备不是练家子，没练过千里传音什么的绝世神功，要交流必须通过管道，你必须得创建一个管道给urb，它才知道路怎么走。不过在说怎么创建一个管道前，先说个有关管道的故事，咱们也知道一下管道这么热门儿的词汇是怎么来的。

1801年，在意大利中部的小山村，有两个名叫柏波罗和布鲁诺的年轻人，和我现在最大的梦想是看懂内核代码一样，他们的最大梦想是成为村子里最富有的人。有一天，喜鹊在枝头唧唧喳喳地叫，他们的好运也就随着来了。村里决定雇两个人把附近河里的水运到村广场的水缸里去，他们得到了这个机会。“我提一桶水，只收他一分钱，我提10桶水赚多少钱，我一天提他一百桶水！一百桶水！算一下多少钱先，一七得七，二七四十八，三八……妇女节，五一……劳动节，六一……我爹过节，七一……” 布鲁诺激动的盘算着。但柏波罗却想着一天才几分钱的报酬，还要这样来回提水，干脆修一条管道将水从河里引到村里去得了。于是布鲁诺每天辛勤的提着水，很快买了新衣服，买了驴，虽然仍然买不起车也买不起房，但已经被看作是中产阶级了，而柏波罗每天还要抽出一部分提水的时间去挖管道，收入是入不敷出啊，挖管道的同时还要接收很多人对他管道男的嘲笑。两年后，柏波罗的管道完工了，水哗哗的直往村里流，钱哗哗的直往口袋里钻，而此时布鲁诺已经被长时间的提桶生涯给压榨的腰弯背驼。管道男柏波罗成了奇迹的创造者，他没有被赞扬冲昏头脑，他想的是创建更多的管道，他邀请提桶男布鲁诺加入了他的管道事业，从此他们的管道事业芝麻开花节节高，遍布了全球。

这个故事告诉我们，管道很重要啊同志们。显然，写代码的哥们儿也深刻的认识到了这个道理，于是内核的include/linux/usb.h文件中多了很多专门用来创建不同管道的宏。

```
1432 static inline unsigned int __create_pipe(struct usb_device *dev,unsigned int endpoint)
1434 {
1435     return (dev->devnum << 8) | (endpoint << 15);
1436 }
1438 /* Create various pipes... */
1439 #define usb_sndctrlpipe(dev,endpoint)    \
1440         ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint))
1441 #define usb_rcvctrlpipe(dev,endpoint)    \
1442      ((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1443 #define usb_sndisocpipe(dev,endpoint)    \
1444         ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint))
1445 #define usb_rcvisocpipe(dev,endpoint)    \
1446    ((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1447 #define usb_sndbulkpipe(dev,endpoint)    \
1448         ((PIPE_BULK << 30) | __create_pipe(dev,endpoint))
1449 #define usb_rcvbulkpipe(dev,endpoint)    \
1450         ((PIPE_BULK << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
1451 #define usb_sndintpipe(dev,endpoint)    \
1452         ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint))
1453 #define usb_rcvintpipe(dev,endpoint)    \
1454      ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
```

端点是有四种的，对应着管道也就有四种，同时端点是有IN也有OUT的，相应的管道也就有两个方向，于是二四得八，上面就出现了八个创建管道的宏。有了struct usb_device结构体，也就是说知道了设备地址，再加上端点号，你就可以需要什么管道就创建什么管道了。__create_pipe宏只是一个幕后的角色，用来将设备地址和端点号放在管道正确的位置上。

1140行，status，urb的当前状态。urb当然是可以有多种状态的，何况USB通信的顶梁柱urb。urb当前什么状态就是让咱们了解出了什么事情。至于各种具体的状态代表了什么意思，碰到了再说。

1141行，transfer_flags，一些标记，可用的值都在include/linux/usb.h里有定义。

```
942 #define URB_SHORT_NOT_OK       0x0001 /* report short reads as errors */
943 #define URB_ISO_ASAP           0x0002 /* iso-only, urb->start_frame
944                                   * ignored */
945 #define URB_NO_TRANSFER_DMA_MAP 0x0004 /* urb->transfer_dma valid on submit */
946 #define URB_NO_SETUP_DMA_MAP   0x0008/*urb->setup_dma valid on submit*/
947 #define URB_NO_FSBR            0x0020  /* UHCI-specific */
948 #define URB_ZERO_PACKET   0x0040 /* Finish bulk OUT with short packet */
949 #define URB_NO_INTERRUPT       0x0080  /* HINT: no non-error interrupt
950                                   * needed */
```

URB_SHORT_NOT_OK，这个标记只对用来从IN端点读取数据的urb有效，意思就是说如果从一个IN端点那里读取了一个比较短的数据包，就可以认为是错误的。那么这里的short究竟short到什么程度？

之前说到端点时，就知道端点描述符里有一个叫wMaxPacketSize，指明了端点一次能够处理的最大字节数。在另外某个地方也提了，在USB的世界里是有很多种包的，四种PID类型，每种PID下边儿还有一些细分的品种。这四种PID里面，有一个叫Data的，也只有它里边儿有一个数据字段，像其他的Token、Handshake之类的PID类型都是没有这个字段的，所以里里外外看过去，还只有Data PID类型的包最实在，就是用来传输数据的，但是它里面并不是只有一个数据字段，还有SYNC、PID、地址域、CRC等陪伴在数据字段的左右。

那现在又有一个问题出来了，每个端点描述符里的wMaxPacketSize所表示的最大字节数都包括了哪些部分？是整个packet的长度吗？我可以负责任地告诉你，它只包括了Data包中面数据字段，俗称data payload，其他的数据字段都是协议本身需要的信息，和TCP/IP里的报头差不多。

wMaxPacketSize与short有什么关系？关系还不小，short与wMaxPacketSize相比的，如果从IN端点那儿收到了一个比wMaxPacketSize要短的包，同时也设置了URB_SHORT_NOT_OK这个标志，那么就可以认为传输出错了。

本来如果收到一个比较短的包是意味着这次传输到此为止就结束了，你想想，data payload的长度最大必须为wMaxPacketSize，这个规定是不可违背的，但是如果端点想给你的数据不止那么多，怎么办？这就需要分成多个wMaxPacketSize大小的data payload来传输，事情有时不会那么凑巧刚好能平分成多个整份。这时，最后一个data payload的长度就会比wMaxPacketSize小，这种情况本来意味着端点已经传完了

它想传的数据，释放完了自己的需求，这次传输就该结束了，不过如果你设置了URB_SHORT_NOT_OK标志，HCD这边就会认为发生了错误。

URB_ISO_ASAP，这个标志只是为了方便等时传输使用。等时传输和中断传输在spec里都被认为是periodic transfers，也就是周期传输，咱们都知道在USB的世界里都是主机占主导地位，设备是没多少发言权的，但是对于等时传输和中断传输，端点可以对主机表达自己一种美好的期望，希望主机能够隔多长时间访问自己一次，这个期望的时间就是这里说的周期。

当然，期望与现实是有一段距离的，如果期望的都能成为现实，咱们还研究USB干什么，端点的这个期望能不能得到满足，要看主机控制器答应不答应。

对于等时传输，一般来说也就一帧（微帧）一次，主机也很忙，再多也抽不出空儿来。那么如果你有一个用于等时传输的urb，你提交给HCD时，就得告诉HCD它应该从哪一帧开始的，就要对下面要讲到的start_frame赋值，也就是说告诉HCD等时传输开始的那一帧（微帧）的帧号，如果你留心，应该还会记得前面说过在每帧或微帧（Mircoframe）的开始都会有一个SOF Token包，这个包中就含有一个帧号字段，记录了那一帧的编号。

这样的话，一是要去设置这个start_frame，二是到你设置的那一帧时，如果主机控制器没空儿开始等时传输，怎么办，要知道USB的世界里它可是老大。于是，就出现了URB_ISO_ASAP，它的意思就是告诉HCD什么时候不忙就什么时候开始，就不用指定什么开始的帧号了，是不是感觉特轻松？所以说，你如果想进行等时传输，又不想标新立异的话，就还是设置start_frame吧。

URB_NO_TRANSFER_DMA_MAP，还有URB_NO_SETUP_DMA_MAP，这两个标志都是有关DMA的。什么是DMA？就是外设，比如USB摄像头，和内存之间直接进行数据交换，把CPU给放到一边儿了。本来，在咱们的电脑里，CPU自认为是老大，什么事都要去插一脚，都要经过它去协调处理。可是这样的话就影响了数据传输的速度，有DMA和没有DMA区别就是这么大。

USB的世界里也是要与时俱进，所以DMA也是少不了的。一般来说，都是驱动里提供了kmalloc等分配的缓冲区，HCD进行一定的DMA映射处理，DMA映射是干什么的？外设和内存之间进行数据交换，总要互相认识吧，外设是通过各种总线连到主机里边的，使用的是总线地址，而内存使用的是虚拟地址，它们之间本来就是两条互不相交的平行线，要让它们中间产生连接点，必须得将一个地址转化为另一个地址，这样才能找得到对方，才能互通有无，而DMA映射就是干这个的。

这只是粗略说法，实际上即使千言万语也道不完的。它可是高技术含量的活儿，所以在某些平台上非常的费时费力。为了分担HCD的压力，于是就有了这里的两个标志，告诉HCD不要再自己做DMA映射了，驱动提供的urb里已经提供有DMA缓冲区地址。具体提供了哪些DMA缓冲区？就涉及到下面的transfer_buffer，transfer_dma，还有setup_packet，setup_dma这两对函数了。

URB_NO_FSBR，这是给UHCI用的函数。

URB_ZERO_PACKET，这个标志表示批量的OUT传输必须使用一个short packet来结束。批量传输的数据大于批量端点的wMaxPacketSize时，需要分成多个Data包来传输，最后一个data payload的长度可能等于wMaxPacketSize，也可能小于wMaxPacketSize，当等于wMaxPacketSize时，如果同时设置了URB_ZERO_PACKET标志，就需要再发送一个长度为0的数据包来结束这次传输，如果小于wMaxPacketSize就没必要多此一举了。如果你要问，当批量传输的数据小于wMaxPacketSize时呢？也没必要再发送0长的数据包，因为此时发送的这个数据包本身就是一个short packet。

URB_NO_INTERRUPT，这个标志用来告诉HCD，在URB完成后，不要请求一个硬件中断，当然这就意味着你的结束处理函数可能不会在urb完成后立即被调用，而是在之后的某个时间被调用， USB Core会保证为每个urb调用一次结束处理函数。

还是回到struct urb，1142~1144行，transfer_buffer，transfer_dma，transfer_buffer_length，前面说过管道的一端是主机上的缓冲区，另一端是设备上的端点，这三个家伙就是描述主机上的缓冲区的。

transfer_buffer是使用kmalloc分配的缓冲区，transfer_dma是使用usb_buffer_alloc分配的dma缓

冲区。HCD不会同时使用它们两个，如果你的urb自带了transfer_dma，就要同时设置URB_NO_TRANSFER_DMA_MAP来告诉HCD一声，不用它再费心做DMA映射了。transfer_buffer 是必须要设置的，因为不是所有的主机控制器都能够使用DMA的，万一遇到这样的情况，也好有一个备用。transfer_buffer_length指的就是transfer_buffer或transfer_dma的长度。

1145行，actual_length，urb结束之后，会用这个字段告诉你实际上传输了多少数据。

1146~1147行，setup_packet，setup_dma，同样是两个缓冲区，一个是kmalloc分配的，一个是用usb_buffer_alloc分配的，不过，这两个缓冲区是控制传输专用的，记得struct usb_ctrlrequest吗？它们保存的就是一个struct usb_ctrlrequest结构体，如果你的urb设置了setup_dma，同样要设置URB_NO_SETUP_DMA_MAP标志来告诉HCD。如果进行的是控制传输，setup_packet是必须要设置的，也是为了防止出现主机控制器不能使用DMA的情况。

1148行，start_frame，如果你没有指定URB_ISO_ASAP标志，就必须自己设置start_frame，指定等时传输在哪个帧或哪个微帧开始。如果指定了URB_ISO_ASAP，urb结束时会使用这个值返回实际的开始帧号。

1150行，interval，等时传输和中断传输专用。Interval是间隔时间的意思。什么的间隔时间？就是上面说的端点希望主机轮询自己的时间间隔。这个值和端点描述符里的bInterval是一样的，你不能随便地指定一个，然后就去做春秋大梦，以为到时间了梦里的名车美女都会跑出来，协议中对你能指定的值是有范围限制的。对于中断传输，全速时，这个范围为1 ms ~255 ms，低速时为10 ms ~255 ms，高速时为1 ms ~16 ms，这个1 ms ~16 ms只是bInterval可以取的值，实际的间隔时间为2的（bInterval-1）次方乘以125 ms，也就是2的（bInterval-1）次方个微帧。

对于等时传输，没有低速等时传输根本就不是低速端点负担得起的，对于全速和高速，这个范围也是为1~16，间隔时间由2的（bInterval-1）次方算出来，单位为帧或微帧。

这样看来，每一帧或微帧里，你最多只能期望有一次等时传输和中断传输，不能再多了。

不过即使完全按照上面的范围来取，你的期望也并不是就肯定可以实现的，因为对于高速来说，最多有80%的总线时间用于这两种传输，对于低速和全速要多一点，达到90%。这个时间怎么分配，都由主机控制器掌握着，所以你的期望能不能实现还要看主机控制器的脸色，没办法，它就有这种权力。

1153行，context，驱动设置了给下面的结束处理函数用的。比如可以将自己驱动里描述自己设备的结构体放在里面，在结束处理函数中就可以取出来。

1154行，complete，一个指向结束处理函数的指针，传输成功完成，或者中间发生错误时就会调用它，驱动可以在这里边儿检查urb的状态，并做一些处理。比如可以释放这个urb，或者重新提交给HCD。就说摄像头吧，你向HCD提交了一个等时的urb，从摄像头那里读取视频数据，传输完成时调用了你指定的这个结束处理函数，并在里面取出了urb里面获得的数据进行解码等处理，然后怎么办？总不会这一个urb读取的数据就够了吧，所以需要获得更多的数据，那你也总不会再去创建、初始化一个等时的urb吧，很明显刚刚的那个urb可以继续用，只要将它再次提交给HCD就可以了。这个函数指针的定义在include/linux/usb.h。

```
961 typedef void (*usb_complete_t)(struct urb *);
```

还有三个函数，都是等时传输专用的，等时传输与其他传输不一样，可以指定传输多少个packet，每个packet使用struct usb_iso_packet_descriptor结构来描述。1155行的iso_frame_desc就表示了一个变长的struct usb_iso_packet_descriptor结构体数组，而1149行的number_of_packets指定了这个结构体数组的大小，也就是要传输多少个packet。

要说明的是这里说的packet不是说在一次等时传输里传输了多个Data packet，而是说在一个urb里指定了多次的等时传输，每个struct usb_iso_packet_descriptor结构体都代表了一次等时传输。

这里对等时传输底层的packet情况说明一下。不像控制传输最少要有SETUP和STATUS两个阶段的transaction，等时传输只有Isochronous transaction，即等时transaction一个阶段，一次等时传输就是一次等时transaction的过程。

而等时transaction也只有两个阶段，就是主机向设备发送OUT Token包，然后发送一个Data包，或者是主机向设备发送IN Token包，然后设备向主机发送一个Data包，这个Data包中data payload的长度只能小于或者等于等时端点的wMaxPacketSize值。

这里没有了Handshake包，是因为不需要，等时传输是不保证数据完全正确无误到达的，没有什么错误重传机制，也就不需要使用Handshake包来汇报。对它来说实时要比正确性重要得多，你的摄像头一秒钟少给你一帧和多给你一帧没有本质的区别，如果等时传输延迟几秒，就明显的感觉不同了。

所以对于等时传输来说，在完成了number_of_packets次传输之后，会去调用你的结束处理函数，在里面对数据做处理，而1152行的error_count记录了这么多次传输中发生错误的次数。

现在看一看struct usb_iso_packet_descriptor结构的定义，在include/linux/usb.h中定义。

```
952 struct usb_iso_packet_descriptor {
953     unsigned int offset;
954     unsigned int length;          /* expected length */
955     unsigned int actual_length;
956     int status;
957 };
```

offset表示transfer_buffer里的偏移位置，你不是指定了要进行number_of_packets次等时传输吗，那么也要准备够这么多次传输用的缓冲区，当然不是说让你准备多个缓冲区。没必要，都放transfer_buffer或者transfer_dma里面好了，只要记着每次传输对应的数据偏移就可以。length是预期的这次等时传输Data包中数据的长度，注意这里说的是预期，因为实际传输时因为各种原因可能不会有那么多数据，urb结束时，每个struct usb_iso_packet_descriptor结构体的actual_length就表示了各次等时传输实际传输的数据长度，而status分别记录了它们的状态。

最后，不管你理解不理解，struct urb都是暂且说到这里了，最后听一听David Brownell大侠的呼声：“I'd rather see 'struct urb' start to shrink, not grow!”

设备的生命线（五）

下面接着看内核代码的三个基本点。

第一个基本点，usb_alloc_urb函数，创建urb的专用函数，为一个urb申请内存并做初始化，在drvi ers/usb/core/urb.c中定义。

```
56 struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags)
57 {
58      struct urb *urb;
60      urb = kmalloc(sizeof(struct urb) +
61              iso_packets * sizeof(struct usb_iso_packet_descriptor),
62              mem_flags);
63      if (!urb) {
64          err("alloc_urb: kmalloc failed");
65          return NULL;
66      }
67      usb_init_urb(urb);
68      return urb;
69 }
```

这函数只做了两件事情，拿kmalloc来为urb申请内存，然后调用usb_init_urb进行初始化。usb_init_urb函数在前面说struct urb中的引用计数时已经讲过了，它主要的作用就是初始化urb的引用计数，还用me mset顺便把这里给urb申请的内存清零。

没什么说的了吗？usb_alloc_urb说：“别看我简单，我也是很有内涵的。”先看第一个问题，它的第一个参数iso_packets，表示struct urb结构最后那个变长数组iso_frame_desc的元素数目，也就是应该与number_of_packets的值相同，所以对于控制/中断/批量传输，这个参数都应该为0。这也算是给咱们示范了一下变长数组的用法

第二个问题是参数mem_flags的类型gfp_t，早几个版本的内核，类型还是int，当然这里变成gfp_t是因为kmalloc参数中的标志参数的类型从int变成gfp_t了，你要用kmalloc来申请内存，就得遵守它的规则。

不过这里要说的不是kmalloc，而是gfp_t，它在江湖上也没出现多久，名号还没打出来，很多人不了解，咱们来调查一下它的背景。它在include/linux/types.h中定义。

```
193 typedef unsigned __bitwise__ gfp_t;
```

很显然，要了解gfp_t，关键是要了解__bitwise__，它也在types.h中定义。

```
170 #ifdef __CHECKER__
171 #define __bitwise__ __attribute__((bitwise))
172 #else
173 #define __bitwise__
174 #endif
```

__bitwise__的含义又取决于是否定义了__CHECKER__，如果没有定义__CHECKER__，那__bitwise__就什么也不是。在哪中定义了__CHECKER__？内核代码里就没有哪个地方定义了__CHECKER__，它是有关Sparse工具的，内核编译时的参数决定了是不是使用Sparse工具来做类型检查。那Sparse又是什么？它是一种静态分析工具(static analysis tool)，用于在Linux内核源代码中发现各种类型的漏洞，一直都是比较神秘的角色，最初由Linus Torvalds写的，后来Linus没有继续维护，直到去年的冬天，它才又有了新的维护者Josh Triplett。有关Sparse再多的东东，咱们还是各自去研究吧，这里不多说了。

可能还会有第三个问题，usb_alloc_urb也没做多少事，它做的那些咱们自己很容易就能做了，为什么还说驱动里一定要使用它来创建urb呢？按照C++的说法，它就是urb的构造函数，构造函数是创建对象的唯一方式，如果你说C++里面使用位复制去复制一个简单对象给新对象就没使用构造函数，那是你不知道，C++的ARM里将这时的构造函数称为trivial copy constructor。再说，现在它做的这些事儿，以后还是做这些吗？它将创建urb的工作给包装了，咱们只管调用就是了，孙子兵法里有“以不变应万变”。

对应的，当然还会有一个析构函数，销毁urb的，也在urb.c中定义。

```
81 void usb_free_urb(struct urb *urb)
82 {
83         if (urb)
84                 kref_put(&urb->kref, urb_destroy);
85 }
```

usb_free_urb更潇洒，只调用kref_put将urb的引用计数减1，减了之后如果变为0，也就是没人再用它了，就调用urb_destroy将它销毁掉。

接着看第二个基本点，usb_fill_control_urb函数，初始化刚才创建的控制urb，你要想使用urb进行USB传输，不是光为它申请点内存就够的，你得为它初始化，填充实点实实在在的内容。它是在include/linux/usb.h中定义的内联函数。

```
1175 static inline void usb_fill_control_urb (struct urb *urb,
1176                                    struct usb_device *dev,
1177                                    unsigned int pipe,
1178                                    unsigned char *setup_packet,
1179                                    void *transfer_buffer,
1180                                    int buffer_length,
1181                                    usb_complete_t complete_fn,
1182                                    void *context)
1183 {
1184     spin_lock_init(&urb->lock);
1185     urb->dev = dev;
1186     urb->pipe = pipe;
1187     urb->setup_packet = setup_packet;
1188     urb->transfer_buffer = transfer_buffer;
1189     urb->transfer_buffer_length = buffer_length;
1190     urb->complete = complete_fn;
1191     urb->context = context;
1192 }
```

这个函数基本上都是赋值语句，把你在参数中指定的值充实给刚刚创建的urb，urb的元素有很多，这里只是填充了一部分，剩下那些不是控制传输管不着的，就是自有安排可以不用去管的。

你想想，一个struct urb结构要应付四种传输类型，每种传输类型总会有一点自己特别的要求，总会有些元素专属于某种传输类型，而其他传输类型不用管的。如果按C++的做法，这称不上是一个好的设计思想，应该有一个基类urb，里面放点儿四种传输类型公用的函数，比如pipe，transfer_buffer等，再搞

几个子类，如control_urb，bulk_urb等，专门应付具体的传输类型，如果不用什么虚函数，实际的时间空间消耗也不会增加什么。但是实在没必要这么搞，这年头儿内核的结构已经够多了，你创建什么类型的urb，就填充相关的一些字段好了，况且写USB Core的哥们儿已经给咱们提供了不同传输类型的初始化函数，就像上面的usb_fill_control_urb函数，对于批量传输有usb_fill_bulk_urb函数，对于中断传输有usb_fill_int_urb函数，一般来说这也就够了，下面就查看usb_fill_control_urb函数的这俩孪生兄弟。

```
1207 static inline void usb_fill_bulk_urb (struct urb *urb,
1208                                       struct usb_device *dev,
1209                                       unsigned int pipe,
1210                                       void *transfer_buffer,
1211                                       int buffer_length,
1212                                       usb_complete_t complete_fn,
1213                                       void *context)
1214 {
1215     spin_lock_init(&urb->lock);
1216     urb->dev = dev;
1217     urb->pipe = pipe;
1218     urb->transfer_buffer = transfer_buffer;
1219     urb->transfer_buffer_length = buffer_length;
1220     urb->complete = complete_fn;
1221     urb->context = context;
1222 }
1223
1242 static inline void usb_fill_int_urb (struct urb *urb,
1243                                      struct usb_device *dev,
1244                                      unsigned int pipe,
1245                                      void *transfer_buffer,
1246                                      int buffer_length,
1247                                      usb_complete_t complete_fn,
1248                                      void *context,
1249                                      int interval)
1250 {
1251     spin_lock_init(&urb->lock);
1252     urb->dev = dev;
1253     urb->pipe = pipe;
1254     urb->transfer_buffer = transfer_buffer;
1255     urb->transfer_buffer_length = buffer_length;
1256     urb->complete = complete_fn;
1257     urb->context = context;
1258     if (dev->speed == USB_SPEED_HIGH)
1259         urb->interval = 1 << (interval - 1);
1260     else
1261         urb->interval = interval;
1262     urb->start_frame = -1;
1263 }
```

负责批量传输的usb_fill_bulk_urb函数和负责控制传输的usb_fill_control_urb函数相比，只是少初始化了一个setup_packet，因为批量传输里没有Setup包的概念，中断传输里也没有，所以usb_fill_int_urb里也没有初始化setup_packet。不过usb_fill_int_urb函数比那两个函数还是多了点儿内容的，因为它有一个interval，比控制传输和批量传输多了一个表达自己期望的权利，1258行还做了一次判断，如果是高速就怎么办，否则又该怎么办，主要是高速和低速/全速的间隔时间单位不一样，低速/全速的单位为帧，高速的单位为微帧，还要经过2的（bInterval-1）次方这么算一下。至于1262行start_frame，它是给等时传输用的，这里自然就设置为-1，关于为什么既然start_frame是等时传输用的这里还要设置那么一下。

我们很快发现usb_fill_control_urb的孪生兄弟里，少了等时传输对应的初始化函数。对于等时传输，urb里是可以指定进行多次传输的，你必须一个一个地对那个变长数组iso_frame_desc里的内容进行初始化。难道你能想出一个办法搞出一个适用于各种情况等时传输的初始化函数？我是不能。如果想不出来，使用urb进行等时传输时，还是老老实实地对里面相关的字段一个一个填内容吧。如果想找个例子参考一下别人是怎么初始化的，可以去找个摄像头驱动或者其他USB音视频设备的驱动看一看，内核中也有一些。

现在，你应该还记得是因为要设置设备的地址，让设备进入Address状态，调用了usb_control_ msg，才遇到usb_fill_control_urb的，参数中的setup_packet就是之前创建和赋好值的struct usb_ctrlrequest结构体，设备的地址已经在struct usb_ctrlrequest结构体wValue字段里了。这次控制传输并没有DATA

transaction阶段，也并不需要urb提供什么transfer_buffer缓冲区，所以transfer_buffer应该传递一个NULL，当然transfer_buffer_length也就为0了。有意思的是这时候传递进来的结束处理函数usb_api_blocking_completion，可以看一下当这次控制传输已经完成，设备地址已经设置好后，接着做了些什么，它的定义在drivers/usb/core/message.c里：

```
21 static void usb_api_blocking_completion(struct urb *urb)
22 {
23       complete((struct completion *)urb->context);
24 }
```

这个函数更简洁，就一句，没有前面说的释放urb，也没有重新提交它。设置一个地址就可以了，没必要再将它提交给HCD，你就是再提交多少次，设置多少次，也只能有一个地址，USB的世界里不提倡囤积居奇。那在这里仅仅一句里面都做了些什么？接着往下看。

然后就是第三个基本点，usb_start_wait_urb函数，将前面历经千辛万苦创建和初始化的urb提交给USB core，让它移交给特定的主机控制器驱动进行处理，然后等待HCD回馈的结果，如果等待的时间超过了预期的限度，它不会再等。它在message.c中定义：

```
33 static int usb_start_wait_urb(struct urb *urb, int timeout, int *actual_length)
34 {
35       struct completion done;
36       unsigned long expire;
37       int status;
39       init_completion(&done);
40       urb->context = &done;
41       urb->actual_length = 0;
42       status = usb_submit_urb(urb, GFP_NOIO);
43       if (unlikely(status))
44           goto out;
45
46      expire = timeout ? msecs_to_jiffies(timeout) : MAX_SCHEDULE_TIMEOUT;
47       if (!wait_for_completion_timeout(&done, expire)) {
48
49           dev_dbg(&urb->dev->dev,
50                     "%s timed out on ep%d%s len=%d/%d\n",
51                     current->comm,
52                     usb_pipeendpoint(urb->pipe),
53                     usb_pipein(urb->pipe) ? "in" : "out",
54                     urb->actual_length,
55                     urb->transfer_buffer_length);
57           usb_kill_urb(urb);
58           status = urb->status == -ENOENT ? -ETIMEDOUT : urb->status;
59       } else
60           status = urb->status;
61 out:
62       if (actual_length)
63           *actual_length = urb->actual_length;
64
65       usb_free_urb(urb);
66       return status;
67 }
```

35行，定义了一个struct completion结构体。completion是内核中一个比较简单的同步机制，一个线程可以通过它来通知另外一个线程某件事情已经做完了。

completion机制也同样是这么回事儿，你的代码执行到某个地方，需要再其他的事情，就新开一个线程，让它去忙活，然后接着忙自己的，想知道那边儿忙活的结果了，就停在某个地方等着，那边儿忙活完了会通知一下已经有结果了，于是你的代码又可以继续往下走。

completion机制围绕struct completion结构去实现，有两种使用方式，一种是通过DECLARE_COMPLETION宏在编译时就创建好struct completion的结构体，另外一种就是上面的使用形式，运行时才创建的。先在35行定义一个struct completion结构体，然后在39行使用init_completion去初始化。光是创建struct completion的结构体没用，关键的是如何通知任务已经完成了，和怎么去等候完成的好消息。片子下载完了可能会用声音、对话框等多种方式来通知你，同样这里用来通知已经完成了的函数也不止一个。

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

complete只通知一个等候的线程，complete_all可以通知所有等候的线程。

你不可能会毫无限度地等下去，所以针对不同的情况，等候的方式就有好几种，都在kernel/sched.c中定义：

```
    void wait_for_completion(struct completion *c);
    unsigned long wait_for_completion_timeout(struct completion *x, unsigned long timeout);
    int wait_for_completion_interruptible(struct completion *x);
    unsigned long wait_for_completion_interruptible_timeout(struct completion *x, unsigned long
timeout);
```

上面47行使用的就是wait_for_completion_timeout，设定一个时间限度，然后在那里候着，直到得到通知，或者超过时间。既然有等的一方，也总得有通知的一方吧，不然岂不是每次都超时？记得上面刚出现过的结束处理函数usb_api_blocking_completion吗？它里面唯一的一句complete((struct completion *)urb->context)就是用来通知这里的47行。有疑问的话看40行，将刚刚初始过的struct completion结构体done的地址赋值给了urb->context，47行等的就是这个done。再看42行，usb_submit_urb函数将这个控制urb提交给USB Core，它是异步的，也就是说提交了之后不会等到传输完成了才返回。

现在就比较清晰了，usb_start_wait_urb函数将urb提交给USB Core去处理后，就停在47行等候USB Core和HCD的处理结果，而这个urb代表的控制传输完成之后会调用结束处理函数usb_api_blocking_completion，从而调用complete来通知usb_start_wait_urb说不用再等了，传输已经完成了，当然还有一种可能是usb_start_wait_urb已经等的超过了时间限度仍然没有接到通知。不管是哪种情况，usb_start_wait_urb都可以不用再等，继续往下走了。

42行，提交urb，并返回一个值表示是否提交成功了，显然，成功的可能性要远远大于失败的可能性，所以接下来的判断加上了unlikely，如果真的失败，那也就没必要在47行等通知了，直接到后面去吧。

46行，计算超时值。超时值在参数中不是已经给了吗，还计算什么？没错，你是在参数中是指定了自己能够忍受的最大时间限度，不过那是以 ms为单位的，不过在Linux里时间概念必须得加上一个jiffy，因为函数wait_for_completion_timeout里的超时参数是必须以jiffy为单位的。

jiffy，意思是瞬间，短暂的时间跨度，短暂到什么程度？Linux里它表示两次时钟中断的间隔，时钟中断是由定时器周期性产生的，关于这个周期，内核中有一个很形象的变量来描述，就是HZ，它是一个体系结构相关的常数。内核中还提供了专门的计数器来记录从系统引导开始所度过的jiffy值，每次时钟中断发生时，这个计数器就增加1。

既然你指定的时间和wait_for_completion_timeout需要的时间单位不一致，就需要转换一下，msecs_to_jiffies函数可以完成这个工作，它将 ms值转化为相应的jiffy值。在46行里MAX_SCHEDULE_TIMEOUT比较陌生，在include/linux/sched.h里它被定义为LONG_MAX，最大的长整型值， 在include/linux/kernel.h里看一看：

```
    23 #define INT_MAX         ((int)(~0U>>1))
    24 #define INT_MIN         (-INT_MAX - 1)
    25 #define UINT_MAX        (~0U)
    26 #define LONG_MAX        ((long)(~0UL>>1))
    27 #define LONG_MIN        (-LONG_MAX - 1)
    28 #define ULONG_MAX       (~0UL)
    29 #define LLONG_MAX       ((long long)(~0ULL>>1))
    30 #define LLONG_MIN       (-LLONG_MAX - 1)
    31 #define ULLONG_MAX      (~0ULL)
```

各种整型数的最大值最小值都在这里了，‘~’是按位取反，‘UL’是无符号长整型，‘ULL’是64位的无符号长整型，‘<<’左移运算就是直接把所有位往左边儿移若干位，‘>>’右移运算比较容易搞混，主要是怎么去补缺。在C里主要就是无符号整数和有符号整数的之间的冲突，在往右移之后，空出来的那些空缺，对于无符号整数得补0，对于有符号的，得补上符号位。

还是讲一下LONG_MAX，上边定义为((long)(~0UL>>1))，0UL按位取反之后全为1的无符号长整型，向右移1位，左边空出来的位置补0，这个数对于无符号的long来说不算什么，但是再经过long这么强制转化一下变为有符号的长整型，它就是老大了。

现在你可以很明白地知道儿在46行指定的超时时间被转化为相应的jiffy值，或者直接被设定为最大的long值。

47行，等待通知，我们需要知道的是怎么去判断等待的结果，也就是wait_for_completion_timeout的返回值代表什么意思？一般来说，一个函数返回了0代表一切顺利，可是wait_for_completion_timeout返回0恰恰表示坏消息，表示直到超过了忍耐的极限仍没有接到任何的回馈；而返回了一个大于0的值则表示接到通知了，不管是完成了还是出错了总归是告诉这边儿不用再等了，这个值具体的含义就是距你设定的时限提前了多少时间。为什么会这样？你去看一看wait_for_completion_timeout的定义就知道了，它是通过schedule_timeout来实现超时的，schedule_timeout的返回值就是这个意思。

现在就很明显了，如果超时了，就会打印一些调试信息提醒一下，然后调用usb_kill_urb函数终止这个urb，再将返回值设定一下。如果收到了通知，直接设定返回值，就接着往下走。

62行，actual_length是用来记录实际传输的数据长度的，是usb_control_ msg需要的。不要说这个值urb里本来就有保存，接下来的65行就用usb_free_urb把这个urb给销毁了，到那时还到哪里去找这个值。actual_length是从上头儿传递过来的一个指针，这里要先判断一下actual_length是不是空的。

设备的生命线（六）

现在来讲一下usb_submit_urb函数，这是一个有几百行的函数。

```
220 int usb_submit_urb(struct urb *urb, gfp_t mem_flags)
221 {
222     int                     pipe, temp, max;
223     struct usb_device       *dev;
224     int                     is_out;
225
226     if (!urb || urb->hcpriv || !urb->complete)
227         return -EINVAL;
228     if (!(dev = urb->dev) ||
229           (dev->state < USB_STATE_DEFAULT) ||
230           (!dev->bus) || (dev->devnum <= 0))
231         return -ENODEV;
232     if (dev->bus->controller->power.power_state.event != PM_EVENT_ON
233                     || dev->state == USB_STATE_SUSPENDED)
234         return -EHOSTUNREACH;
235
236     urb->status = -EINPROGRESS;
237     urb->actual_length = 0;
238
239     /* Lots of sanity checks, so HCDs can rely on clean data
240      * and don't need to duplicate tests
241      */
242     pipe = urb->pipe;
243     temp = usb_pipetype(pipe);
244     is_out = usb_pipeout(pipe);
245
246     if (!usb_pipecontrol(pipe) && dev->state < USB_STATE_CONFIGURED)
247         return -ENODEV;
248
249     /* FIXME there should be a sharable lock protecting us against
250      * config/altsetting changes and disconnects, kicking in here.
251      * (here == before maxpacket, and eventually endpoint type,
252      * checks get made.)
253      */
254
255     max = usb_maxpacket(dev, pipe, is_out);
256     if (max <= 0) {
257         dev_dbg(&dev->dev,
258             "bogus endpoint ep%d%s in %s (bad maxpacket %d)\n",
259                 usb_pipeendpoint(pipe), is_out ? "out" : "in",
260                 __FUNCTION__, max);
261         return -EMSGSIZE;
262     }
263
264     /* periodic transfers limit size per frame/uframe,
265      * but drivers only control those sizes for ISO.
266      * while we're checking, initialize return status.
```

```
267          */
268         if (temp == PIPE_ISOCHRONOUS) {
269         int     n, len;
270 
271             /* "high bandwidth" mode, 1-3 packets/uframe? */
272             if (dev->speed == USB_SPEED_HIGH) {
273                 int     mult = 1 + ((max >> 11) & 0x03);
274                 max &= 0x07ff;
275                 max *= mult;
276             }
277 
278             if (urb->number_of_packets <= 0)
279                 return -EINVAL;
280             for (n = 0; n < urb->number_of_packets; n++) {
281                 len = urb->iso_frame_desc[n].length;
282                 if (len < 0 || len > max)
283                     return -EMSGSIZE;
284                 urb->iso_frame_desc[n].status = -EXDEV;
285                 urb->iso_frame_desc[n].actual_length = 0;
286             }
287         }
288 
289         /* the I/O buffer must be mapped/unmapped, except when length=0 */
290         if (urb->transfer_buffer_length < 0)
291             return -EMSGSIZE;
292 
293 #ifdef DEBUG
294         /* stuff that drivers shouldn't do, but which shouldn't
295          * cause proble ms in HCDs if they get it wrong.
296          */
297         {
298         unsigned int    orig_flags = urb->transfer_flags;
299         unsigned int    allowed;
300 
301         /* enforce simple/standard policy */
302         allowed = (URB_NO_TRANSFER_DMA_MAP | URB_NO_SETUP_DMA_MAP |
303                         URB_NO_INTERRUPT);
304         switch (temp) {
305         case PIPE_BULK:
306             if (is_out)
307             allowed |= URB_ZERO_PACKET;
308             /* FALLTHROUGH */
309         case PIPE_CONTROL:
310             allowed |= URB_NO_FSBR; /* only affects UHCI */
311             /* FALLTHROUGH */
312         default:                    /* all non-iso endpoints */
313             if (!is_out)
314                 allowed |= URB_SHORT_NOT_OK;
315             break;
316         case PIPE_ISOCHRONOUS:
317             allowed |= URB_ISO_ASAP;
318             break;
319         }
320         urb->transfer_flags &= allowed;
321 
322         /* fail if submitter gave bogus flags */
323         if (urb->transfer_flags != orig_flags) {
324             err("BOGUS urb flags, %x --> %x",
325                     orig_flags, urb->transfer_flags);
326             return -EINVAL;
327         }
328         }
329 #endif
330         /*
331          * Force periodic transfer intervals to be legal values that are
332          * a power of two (so HCDs don't need to).
333          *
334          * FIXME want bus->{intr,iso}_sched_horizon values here.  Each HC
335          * supports different values... this uses EHCI/UHCI defaults (and
336          * EHCI can use smaller non-default values).
337          */
338         switch (temp) {
339         case PIPE_ISOCHRONOUS:
340         case PIPE_INTERRUPT:
341             /* too small? */
```

```
342         if (urb->interval <= 0)
343             return -EINVAL;
344         /* too big? */
345         switch (dev->speed) {
346         case USB_SPEED_HIGH:    /* units are microframes */
347         // NOTE usb handles 2^15
348             if (urb->interval > (1024 * 8))
349                 urb->interval = 1024 * 8;
350             temp = 1024 * 8;
351             break;
352         case USB_SPEED_FULL:    /* units are frames/ msec */
353     case USB_SPEED_LOW:
354             if (temp == PIPE_INTERRUPT) {
355                 if (urb->interval > 255)
356                     return -EINVAL;
357                     // NOTE ohci only handles up to 32
358                 temp = 128;
359             } else {
360                 if (urb->interval > 1024)
361                 urb->interval = 1024;
362                     // NOTE usb and ohci handle up to 2^15
363                 temp = 1024;
364         }
365             break;
366         default:
367             return -EINVAL;
368         }
369         /* power of two? */
370         while (temp > urb->interval)
371         temp >>= 1;
372         urb->interval = temp;
373     }
375     return usb_hcd_submit_urb(urb, mem_flags);
376 }
```

这个函数虽然很长，目标却很简单，就是对urb做些前期处理后扔给HCD。

226行，一些有关存在性的判断，这个函数在开始就要履行一下常规的检验，urb为空，都没有初始化是不可以提交给core的，core很生气，后果很严重，hcpriv本来说好了留给HCD用的，自己不能偷偷先用了，HCD很生气，后果也会很严重，complete，每个urb结束了都必须的调用一次complete代表的函数。

228行，上226行是对urb本身的检验，这里是对urb的目的地USB设备的检验。要想让设备回应，它起码得达到Default状态。

设备编号devnum的值肯定是不能为负的了，那为什么为0也不行呢？Token包的地址域里有7位是表示设备地址的，也就是说总共可以有128个地址来分配给设备，但是其中0号地址是被保留作为默认地址用的，任何一个设备处于Default状态还没有进入Address状态时都需要通过这个默认地址来响应主机的请求，所以0号地址不能分配给任何一个设备，Hub为设备选择一个地址时，只有选择到一个大于0的地址，设备的生命线才会继续，因此说这里的devnum的值是不可能也不应该为0的。

因为要设置设备的地址，让设备进入Address状态，所以针对SET_ADDRESS请求再查看这个devnum。主机向设备发送SET_ADDRESS请求时，如果设备处于Default状态，就是它现在的状态，指定一个非0值时，设备将进入Adress状态；指定0值时，设备仍然会处于Default状态。所以说这里的devnum也是不能为0的。如果设备已经处于Adress状态，指定一个非0值时，设备仍然会处于Address状态，只是将使用新分配的地址，一个设备只能占用一个地址，是分配的，如果指定了一个0值，则设备将离开Address状态退回到Default状态。

232行，power，power_state，event，还有PM_EVENT_ON都是电源管理核心里的内容，这里的目的是判断设备所在的那条总线的主机控制器有没有挂起，然后再判断设备本身是不是处于Suspended状态。

236行，常规检查都做完了，core和HCD已经认同了这个urb，就将它的状态设置为-EINPROGRESS，表示从现在开始urb的控制权就在core和HCD手里了，在驱动那里看不到这个状态。

237行，这时还没开始传输，实际传输的数据长度当然为0了，这里进行初始化，也是为了防止以后哪里出错返回了，在驱动里可以检查。

242行，这几行获得管道的类型还有方向。

246行，在设备进入Configured状态之前，主机只能使用控制传输，通过默认管道，也就是管道0来和设备进行交流。

255行，获得端点的wMaxPacketSize，看一看include/linux/usb.h中定义的这个函数：

```
1458 static inline __u16
1459 usb_maxpacket(struct usb_device *udev, int pipe, int is_out)
1460 {
1461     struct usb_host_endpoint       *ep;
1462     unsigned                        epnum = usb_pipeendpoint(pipe);
1464     if (is_out) {
1465         WARN_ON(usb_pipein(pipe));
1466         ep = udev->ep_out[epnum];
1467     } else {
1468         WARN_ON(usb_pipeout(pipe));
1469         ep = udev->ep_in[epnum];
1470     }
1471     if (!ep)
1472         return 0;
1474     /* NOTE:  only 0x07ff bits are for packet size... */
1475     return le16_to_cpu(ep->desc.wMaxPacketSize);
1476 }
```

这个函数是很简单的。要根据现有的信息获得一个端点的wMaxPacketSize当然是必须得获得该端点的描述符，我们知道每个struct usb_device结构体里都有两个数组ep_out和ep_in，它们保存了各个端点对应的struct usb_host_endpoint结构体，只要知道该端点对应了这两个数组里的哪个元素就可以获得它的描述符了，这就需要知道该端点的端点号和方向，而端点的方向就管道的方向，端点号也保存在pipe里。

你是不是会担心ep_out或ep_in数组都还空着，或者说没有保存对应的端点信息？不用担心它还是空的，即使是现在设备还刚从Powered走到Default，连Address都没有，但是在使用usb_alloc_dev构造这个设备的struct usb_device时，就把它里面端点0的struct usb_host_endpoint结构体ep0指定给ep_out[0]和ep_in[0]了，而且后来还对ep0的wMaxPacketSize指定了值。不过如果真的没有从它们里面找到想要的端点的信息，那肯定就是哪里出错了，指定了错误的端点号，或其他什么原因，也就不用再继续走了。

268行，如果是等时传输就要进行一些特别的处理。272到276这几行涉及高速、高带宽端点（high speed，high bandwidth endpoint）。前面提到interval时，说过每一帧或微帧最多只能有一次等时传输，完成一次等时transaction，那时这么说主要是因为还没遇到高速高带宽的等时端点。高速高带宽等时端点每个微帧可以进行2到3次等时transaction，它和一般等时端点的主要区别也在这儿，没必要专门为它设置个描述符类型，端点描述符wMaxPacketSize字段的bit 11~12就是用来指定可以额外有几次等时transaction的，00表示没有额外的transaction，01表示额外有1次，10表示额外有2次，11被保留。wMaxPacketSize字段的前10位就是实际的端点每次能够处理的最大字节数。所以这几行意思就是如果是高速等时端点，获得它允许的额外等时transaction次数，和每次能够处理的最大字节数，再将它们相乘就得出了该等时端点每个微帧的所能传输的最大字节数。

278行，number_of_packets不大于0就表示这个等时urb没有指定任何一次等时传输，可以直接返回了。

280~286行，对等时urb里指定的各次等时则传输分别进行处理。如果它们预期传输的数据长度比上面算出来的max还要大，返回。然后将它们实际传输的数据长度先置为0，状态都先初始化为-EXDEV，表示这次等时传输仅仅部分完成了，因为走到这里时传输都还没开始。

290行，transfer_buffer_length长度不能小于0，等于0倒是可以的，毕竟不是什么时候都是有数据要传的。

293行，见到#ifdef DEBUG我们都应该很高兴，这意味着一直到下面对应的#endif之间的代码都是调试时用的，对整体函数作用无关痛痒。

338行，temp是上面计算出来的管道类型，那下面的各个case肯定是针对四种传输类型的了。不过可以发现，这里只“case”了等时传输和中断传输两种周期性的传输类型，因为是关于interval的处理，所以就没有控制传输和批量传输了。

342行，这里保证等时和中断urb的interval必须是大于0的，不然主机那边看不懂这是什么意思。

345行，这里的switch根据目的设备的速度去case，速度有三种，case也有三个。前面已经说过，不同的速度，urb->interval可以取不同的范围，不过你可能会发现那时说的最大值要比这里的限制要大一些，这是因为协议归协议，实现归实现。比如，对于UHCI来说，中断传输的interval不能比128更大，而协议规定的最大值为255。那么现在的问题是，temp又是做什么用的？要注意urb->interval的单位是帧或者微帧，temp只是为了调整它的值为2的次幂，这点从370行就可以看出来。

375行，将urb交给HCD，然后就进入HCD。

本来usb_submit_urb函数到此应该结束了，但是它对于写驱动的来说太重要了，驱动里做的所有铺垫就是为了使用usb_submit_urb提交一个合适的urb给设备，满怀期待地等待着设备回馈你需要的信息，然后才有接下来的处理，不然usb驱动只是一纸空谈毫无用处。

首先还是要再次强调一下，在调用usb_submit_urb提交你的urb之前，一定必须不得不要对它正确初始化，对于控制/中断/批量传输，core都提供了usb_fill_control_urb的几个孪生兄弟供你初始化使用，对于等时传输要自己手工一步一步小心翼翼地对urb的相关元素逐个赋值。urb决定了整个usb驱动能否顺利运转。

第二，对于驱动来说，usb_submit_urb是异步的，也就是说不用等传输完全完成就返回了，只要usb_submit_urb的返回值表示为0，就表示已经提交成功了，urb已经被core和HCD认可了，接下来core和HCD怎么处理就是它们的事了。

只要你提交成功了，不管是中间出了差错还是顺利完成，你指定的结束处理函数总是会调用，只有到这个时候，你才能够重新拿到urb的控制权，检查它是不是出错了，需要不需要释放或者是重新提交。那么，第三就是，什么时候需要在结束处理函数中重新提交这个urb？其实，我更想问的是对于中断/等时传输，是怎么实现让主机按一定周期去访问端点的？端点的描述符里已经指定了这个间隔时间，urb里也有interval描述了这个间隔周期。可是 urb一次只完成一次传输，即使等时传输也只完成有限次的传输，然后就在结束处理函数中返回了，urb的控制权就完全属于驱动了，接下来的周期访问是怎么做到的？难道脱离urb主机自己就去智能化的自动的与端点通信了？即使是这样了，那通信的数据又在哪里，又怎么去得到这些数据？

事实上，第一次提交一个中断或等时的urb时，HCD会根据interval判断一下自己是否能够满足你的需要，如果不能够安排足够的带宽来完成这种周期性的传输，它是不可能会批准请求的，如果它估量一下觉得可以满足，就会保留足够的带宽。但是这并不是就表明万事大吉了，HCD是保留带宽了，可是驱动得保证在对应端点要处理的urb队列里总是有urb，不能是空的，否则这个保留的带宽就会被“cancel”掉。

那么对于中断/等时传输，如何保证对应端点的urb队列里总是会有urb？这就回到最开始的问题了。驱动需要在结束处理函数中重新初始化和提交刚刚完成的urb，提醒一下，这个时候你是不能够修改interval的值，否则等待你的只能是错误。中断传输的例子可以去看一看触摸屏驱动，等时传输的例子可以去看一看摄像头驱动，看一看它们在结束处理函数中都做了些什么，你就知道了。

第四个要说的是，对于控制/批量/中断传输，实际上很多时候你可以不用创建urb，不用对它初始化，不用调用usb_submit_urb来提交，USB Core将这个过程分别封装在了usb_control_ msg、usb_bulk_ msg和usb_interrupt_ msg这三个函数中，不同的是它们的实现是同步的，会去等待传输完全结束。咱们就是从usb_control_ msg走过来的，所以这里只查看另外两个，它们都定义在drivers/usb/core/message.c里。

```
169 int usb_interrupt_ msg(struct usb_device *usb_dev, unsigned int pipe,
170                     void *data, int len, int *actual_length, int timeout)
171 {
172   return usb_bulk_ msg(usb_dev, pipe, data, len, actual_length, timeout);
173 }
```

usb_interrupt_ msg全部都借助usb_bulk_ msg去完成了。

```
207 int usb_bulk_ msg(struct usb_device *usb_dev, unsigned int pipe,
208                     void *data, int len, int *actual_length, int timeout)
209 {
```

```
210     struct urb *urb;
211     struct usb_host_endpoint *ep;
212
213     ep = (usb_pipein(pipe) ? usb_dev->ep_in : usb_dev->ep_out)
214                     [usb_pipeendpoint(pipe)];
215     if (!ep || len < 0)
216     return -EINVAL;
217
218     urb = usb_alloc_urb(0, GFP_KERNEL);
219     if (!urb)
220         return -ENOMEM;
221
222     if ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
223                     USB_ENDPOINT_XFER_INT) {
224         pipe = (pipe & ~(3 << 30)) | (PIPE_INTERRUPT << 30);
225         usb_fill_int_urb(urb, usb_dev, pipe, data, len,
226                         usb_api_blocking_completion, NULL,
227                         ep->desc.bInterval);
228     } else
229         usb_fill_bulk_urb(urb, usb_dev, pipe, data, len,
230                         usb_api_blocking_completion, NULL);
231
232     return usb_start_wait_urb(urb, timeout, actual_length);
233 }
```

首先根据指定的pipe获得端点的方向和端点号，然后从设备struct usb_device结构体的ep_in或ep_out数组里得到端点对应的struct usb_host_endpoint结构体，接着调用usb_alloc_urb创建urb。

因为这个函数可能是从usb_interrupt_ msg调用过来的，所以接下来要根据端点描述符的bmAttributes字段获取传输的类型，判断究竟是中断传输还是批量传输，是中断传输的话还要修改pipe的类型，防止万一被其他函数直接调用usb_bulk_ msg来完成中断传输。预防一下总归是没错的。

不管是中断传输还是批量传输，都要调用usb_fill_xxx_urb函数来初始化，最后，和usb_control_ msg一样，调用usb_start_wait_urb函数。

设备的生命线（七）

在HCD这个片区域里，王中之王就是drivers/usb/core/hcd.h中定义的struct usb_hcd。

```
58 struct usb_hcd {
59
60      /*
61       * housekeeping
62       */
63      struct usb_bus          self;           /* hcd is-a bus */
64      struct kref             kref;           /* reference counter */
65
66      const char              *product_desc;  /* product/vendor string */
67      char                    irq_descr[24];  /* driver + bus # */
68
69      struct timer_list       rh_timer;       /* drives root-hub polling */
70      struct urb              *status_urb;    /* the current status urb */
71 #ifdef CONFIG_PM
72      struct work_struct      wakeup_work;    /* for remote wakeup */
73 #endif
74
75      /*
76       * hardware info/state
77       */
78      const struct hc_driver  *driver;        /* hw-specific hooks */
79
80      /* Flags that need to be manipulated atomically */
81      unsigned long           flags;
82 #define HCD_FLAG_HW_ACCESSIBLE  0x00000001
83 #define HCD_FLAG_SAW_IRQ        0x00000002
84
85      unsigned                rh_registered:1;/* is root hub registered? */
86
87      /* The next flag is a stopgap, to be removed when all the HCDs
88       * support the new root-hub polling mechanism. */
89      unsigned                uses_new_polling:1;
90      unsigned                poll_rh:1;      /* poll for rh status? */
```

```
 91      unsigned              poll_pending:1;  /* status has changed? */
 92      unsigned              wireless:1;     /* Wireless USB HCD */
 93
 94      int                   irq;           /* irq allocated */
 95      void __iomem           *regs;         /* device memory/io */
 96      u64                   rsrc_start;     /* memory/io resource start */
 97      u64                   rsrc_len;      /* memory/io resource length */
 98      unsigned              power_budget;   /* in mA, 0 = no limit */
 99
100 #define HCD_BUFFER_POOLS       4
101      struct dma_pool        *pool [HCD_BUFFER_POOLS];
102
103      int                   state;
104 #define  __ACTIVE               0x01
105 #define  __SUSPEND              0x04
106 #define  __TRANSIENT            0x80
107
108 #define  HC_STATE_HALT          0
109 #define  HC_STATE_RUNNING       (__ACTIVE)
110 #define  HC_STATE_QUIESCING     (__SUSPEND|__TRANSIENT|__ACTIVE)
111 #define  HC_STATE_RESUMING      (__SUSPEND|__TRANSIENT)
112 #define  HC_STATE_SUSPENDED     (__SUSPEND)
113
114 #define HC_IS_RUNNING(state) ((state) & __ACTIVE)
115 #define HC_IS_SUSPENDED(state) ((state) & __SUSPEND)
116
117      /* more shared queuing code would be good; it should support
118       * smarter scheduling, handle transaction translators, etc;
119       * input size of periodic table to an interrupt scheduler.
120       * (ohci 32, uhci 1024, ehci 256/512/1024).
121       */
122
123      /* The HC driver's private data is stored at the end of
124       * this structure.
125       */
126      unsigned long hcd_priv[0]
127                   __attribute__ ((aligned (sizeof(unsigned long))));
128 };
```

63行，又一个结构体，struct usb_bus里还有self。

为什么这里会用这么一个戏剧性的词汇self？我在前面提到过，一个主机控制器就会连出一条USB总线，主机控制器驱动用struct usb_hcd结构表示，一条总线用struct usb_bus结构表示。struct usb_bus在include/linux/usb.h中定义。

```
276 struct usb_bus {
277      struct device *controller;     /* host/master side hardware */
278      int busnum;                   /* Bus number (in order of reg) */
279      char *bus_name;               /* stable id (PCI slot_name etc) */
280      u8 uses_dma;               /* Does the host controller use DMA? */
281      u8 otg_port;                  /* 0, or number of OTG/HNP port */
282      unsigned is_b_host:1;      /* true during some HNP roleswitches */
283      unsigned b_hnp_enable:1;       /* OTG: did A-Host enable HNP? */
284
285      int devnum_next;              /* Next open device number in
286                                     * round-robin allocation */
288      struct usb_devmap devmap;      /* device address allocation map */
289      struct usb_device *root_hub;    /* Root hub */
290      struct list_head bus_list;      /* list of busses */
292      int bandwidth_allocated;       /* on this bus: how much of the time
293                                     * reserved for periodic (intr/iso)
294                                     * requests is used, on average?
295                                     * Units: microseconds/frame.
296                                    * Limits: Full/low speed reserve 90%,
297                                     * while high speed reserves 80%.
298                                     */
299      int bandwidth_int_reqs;        /* number of Interrupt requests */
300      int bandwidth_isoc_reqs;       /* number of Isoc. requests */
302 #ifdef CONFIG_USB_DEVICEFS
303      struct dentry *usbfs_dentry; /* usbfs dentry entry for the bus */
304 #endif
305      struct class_device *class_dev; /* class device for this bus */
306
307 #if defined(CONFIG_USB_MON)
```

```
    308     struct mon_bus *mon_bus;        /* non-null when associated */
    309     int monitored;                 /* non-zero when monitored */
    310 #endif
    311 };
```

277行，controller，struct usb_hcd包含了一个usb_bus，这里就回应了一个controller。那现在通过struct usb_hcd里的self和struct usb_bus里的controller这两个词儿，它们到底是什么关系？其实对于写代码的人来说一个主机控制器和一条总线差不多是一码事，不用分得那么清楚，可以简单的说它们都是用来描述主机控制器的，那为什么又分成了两个结构，难道Greg他们现在又不信奉简约主义了？

USB主机控制器是一个设备，而且更多时它还是一个PCI设备，那它就应该纳入这个设备模型范畴之内，struct usb_hcd结构中就得嵌入类似struct device或struct pci_dev这样的一个结构体，但是你仔细看一看，能不能在它里面发现这么一个成员？不能。但是再看一看struct usb_bus，第一个就是一个struct device结构体。好，第一条线索就先到这儿。

再以UHCI为例说明一下，都在host目录下的uhci-族文件中，首先它是一个pci设备，要使用pci_register_driver注册一个struct pci_driver结构体uhci_pci_driver，uhci_pci_driver里又有一个probe，在这个probe里，它调用usb_create_hcd来创建一个usb_hcd，初始化里面的self，还将这个self里的controller设定为描述主机控制器的pci_dev里的struct device结构体，从而将usb_hcd、usb_bus和pci_dev，甚至设备模型都连接起来了。

再接着看一下uhci-文件中定义的函数。只用看它们的参数，你会发现参数中不是struct usb_hcd就是struct uhci_hcd，而且那些函数的前面几行常常会有hcd_to_uhci或者uhci_to_hcd这样的函数在struct usb_hcd和struct uhci_hcd之间转换。struct uhci_hcd是什么？它是uhci自己私有的一个结构体，每个具体的主机控制器都有这么一个类似的结构体。顺便看一下hcd_to_uhci或者uhci_to_hcd的定义你就会明白，每个主机控制器的这个私有结构体都藏在struct usb_hcd结构最后的hcd_priv变长数组里。

对于具体的主机控制器驱动来说，它们的眼里只有struct usb_hcd，struct usb_hcd结构之于主机控制器驱动，就如同struct usb_device或struct usb_interface之于USB驱动。没有usb_create_hcd去创建usb_hcd，就不会有usb_bus的存在。

而对于Linux设备模型来说，struct usb_bus无疑要更亲切一些。总之，你可以把struct usb_bus当作只是嵌入到struct usb_hcd里面的一个结构体，它将struct usb_hcd要完成的一部分工作进行了封装，因为要描述一个主机控制器太复杂太难，于是就开了struct usb_bus去专门面对设备模型、sysfs等。这也就是在前面说struct usb_hcd才是王中之王的原因。

你知道Greg他们是怎么描述这种奇妙的关系吗？他们把这个叫作HCD bus-glue layer，并致力于flatten out it。 这个关系早先是比较混沌的，现在要清晰一些，以后只会更清晰，struct usb_hcd越来越走上台前，struct usb_bus越来越走向幕后。

278行，busnum，总线编号，你的电脑里总可以有多个主机控制器吧，自然也就可以有多条usb总线了，既然可以有多条，就要编个号方便确认了。有关总线编号，可以看一看定义在drivesr/usb/core/hcd.c里的这几行。

```
    88 /* used when allocating bus numbers */
    89 #define USB_MAXBUS              64
    90 struct usb_busmap {
    91      unsigned long busmap [USB_MAXBUS / (8*sizeof (unsigned long))];
    92 };
    93 static struct usb_busmap busmap;
```

讲struct usb_device的devnum时候，说到过一个devicemap，这里又有个busmap，当时分析说devicemap一共有128位，同理可知，这里的busmap一共有64位，也就是说最多可以有64条USB总线。

279行，bus_name，bus，总线；name，名字，bus_name即总线的名字。什么样的名字？要知道大多数情况下主机控制器都是一个PCI设备，那么bus_name应该就是用来在PCI总线上标识usb主机控制器的名字，PCI总线使用标准的PCI ID来标识PCI设备，所以bus_name里保存的应该就是主机控制器对应的PCI ID。UHCI等调用usb_create_hcd创建usb_hcd时确实是将它们对应PCI ID赋给了bus_name。

现在简单介绍一下PCI ID。PCI spec允许单个系统可以最多有256条PCI总线，对咱们当然是太多了，但是对于一些特殊的系统，它可能还觉得这满足不了要求，于是所有的PCI总线又被划分为domain，每个PCI domain又可以最多拥有256条总线，而且每条总线上又可以支持32个设备，这些设备中边儿还都可以是多功能板，它们还都可以最多支持8种功能。那系统怎么来区分每种功能的呢？总要知道它在哪个domain，哪条总线，哪个设备板上吧。这么说还是太笼统了，你可以用lspci命令看一下。

```
    00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge (rev 01)
    00:01.0 PCI bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev 01)
    00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 08)
    00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
    00:07.2 USB Controller: Intel Corporation 82371AB/EB/MB PIIX4 USB
    00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
    00:0f.0 VGA compatible controller: VMware Inc [VMware SVGA II] PCI Display Adapter
    00:10.0 SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320
SCSI (rev 01)
    00:11.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE] (rev 10)
    00:12.0 Multimedia audio controller: Ensoniq ES1371 [AudioPCI-97] (rev 02)
```

每行前面的数字就是所谓的“PCI ID”，每个PCI ID由domain号（16位），总线编号（8位），设备号（5位），功能号（3位）组成，不过这里lspci没有标明domain号，但对于一台普通电脑而言，一般也就只有一个domain，0x0000。

280行，uses_dma，表明这个主机控制器支持不支持DMA。主机控制器的一项重要工作就是在内存和USB总线之间传输数据，这个过程可以使用DMA或者不使用DMA。不使用DMA的方式即所谓的PIO方式。DMA代表着Direct Memory Access，即直接内存访问，不需要CPU去干预。具体去看一看PCI DMA吧，因为一般来说主机控制器都是PCI设备，uses_dma都在它们自己的probe函数中设置了。

281行~283行，有关otg的。

285行，devnum_next，288行，devmap，早就说过devmap这张表了，devnum_next中记录的就是这张表里下一个为0的位，里面为1的位都是已经被这条总线上的usb设备占据了的。

289行，root_hub， Root Hub在所有的Hub里面是那么特殊，还记得USB的那颗树吗？它就是那颗树的根，和USB主机控制器绑定在一起，其他的hub和设备都必须从它这儿延伸出去。正是因为这种特殊的关系，写代码的人就直接将它放在了struct usb_bus结构中。USB主机控制器：USB总线：Root Hub为1：1：1。

290行，bus_list，在drivers/usb/core/hcd.c中定义有一个全局队列usb_bus_list。

```
    84 /* host controllers we manage */
    85 LIST_HEAD (usb_bus_list);
    86 EXPORT_SYMBOL_GPL (usb_bus_list);
```

它就是所有USB总线的组织。每次一条总线新添加进来，都要向这个组织靠拢，都要使用bus_list字段链接在这个队列上。

292行，bandwidth_allocated，表明总线为中断传输和等时传输预留了多少带宽，协议中规定了，对于高速来说，最多可以有80%，对于低速和全速要多一点，可以达到90%。它的单位是 ms，表示一帧或微帧内有多少 ms可以留给中断/等时传输用。

299行，bandwidth_int_reqs；300行，bandwidth_isoc_reqs，分别表示当前中断传输和等时传输的数量。

302行~304行，是usbfs的，每条总线都对应于/proc/bus/usb下的一个目录。

305行，class_dev，这里又牵涉设备模型中的一个概念——设备的class，即设备的类。像前面提到的设备模型中的总线、设备、驱动三个核心概念，纯粹是从写驱动的角度看的，而这里的类则是面向于Linux的广大用户，它不管你是用什么接口，怎么去连接，它只管对用户来说提供了什么功能。一个SCSI硬盘和一个ATA硬盘对驱动来说是不相关的两个东西，但是对于用户来说，它们都是硬盘，都是用来备份文件。

设备模型与sysfs是分不开的，class在sysfs里的体现就在/sys/class下面，可以去看一看：

```
    atm      dma graphics hwmon        i2c-adapter input
```

华清远见

```
    mem      miscnet pci_bus      scsi_device scsi_disk
    scsi_host    sound    spi_host spi_master  spi_transport    tty
    usb_device   usb_endpoint usb_host     vc       vtconsole
```

看到里面的usb_host了吧，它就是所有USB主机控制器的类，这些目录都是怎么来的那？咱们还要追溯一下USB子系统的初始化函数usb_init，它里面有这么一段，

```
    877      retval = usb_host_init();
    878      if (retval)
    879          goto host_init_failed;
```

当时只是简单说这是用来初始化主机控制器的，在hcd.c里：

```
    671 static struct class *usb_host_class;
    672
    673 int usb_host_init(void)
    674 {
    675      int retval = 0;
    676
    677      usb_host_class = class_create(THIS_MODULE, "usb_host");
    678      if (IS_ERR(usb_host_class))
    679          retval = PTR_ERR(usb_host_class);
    680      return retval;
    681 }
```

usb_host_init所作的一切就是调用class_create创建了一个usb_host这样的类，你只要加载了usbcore模块就能在/sys/class下面看到有usb_host目录出现。既然usb_host目录表示usb主机控制器的类，那么它下面应该就对应各个具体的主机控制器了，你用ls命令就能看到usb_host1、usb_host2等这样的目录，它们每个都对应一个在你系统里实际存在的主机控制器，实际上在hcd.c里的usb_register_bus函数有以下这几行：

```
    735  bus->class_dev = class_device_create(usb_host_class, NULL, MKDEV(0,0),
    736                  bus->controller, "usb_host%d", busnum);
```

这两行就是使用class_device_create在/sys/class/usb_host下面为每条总线创建了一个目录，目录名里的数字代表的就是每条总线的编号，usb_register_bus函数是每个主机控制器驱动在probe里调用的，向USB Core注册一条总线，也可以说是注册一个主机控制器。

在struct usb_bus的最后，307行~310行，CONFIG_USB_MON是干什么用的？这要查看drivers/usb/mon目录下的Kconfig。

```
    5 config USB_MON
    6        bool "USB Monitor"
    7        depends on USB!=n
    8        default y
    9        help
    10         If you say Y here, a component which captures the USB traffic
    11         between peripheral-specific drivers and HC drivers will be built.
    12         For more information, see <file:Documentation/usb/usbmon.txt>.
    13
    14         This is somewhat experimental at this time, but it should be safe.
    15
    16         If unsure, say Y.
```

文件中就这么多内容，从里面咱们可以知道，如果定义了CONFIG_USB_MON，一个所谓的USB Monitor，也就是USB监视器就会编进内核。这个Monitor是用来监视USB总线上的底层通信流的，相关的文件都在drivers/usb/mon下面。

设备的生命线（八）

这个世界上不需要努力就能得到的东西只有一样，那就是年龄。所以要不怕苦不怕累，看完struct usb_bus，回到struct usb_hcd，继续努力的往下看。

64行，又见kref，USB主机控制器的引用计数。struct usb_hcd也有自己专用的引用计数函数，看hcd.c文件。

```
    1526 static void hcd_release (struct kref *kref)
    1527 {
    1528     struct usb_hcd *hcd = container_of (kref, struct usb_hcd, kref);
```

```
1529
1530     kfree(hcd);
1531 }
1532
1533 struct usb_hcd *usb_get_hcd (struct usb_hcd *hcd)
1534 {
1535     if (hcd)
1536         kref_get (&hcd->kref);
1537     return hcd;
1538 }
1539 EXPORT_SYMBOL (usb_get_hcd);
1540
1541 void usb_put_hcd (struct usb_hcd *hcd)
1542 {
1543     if (hcd)
1544         kref_put (&hcd->kref, hcd_release);
1545 }
1546 EXPORT_SYMBOL (usb_put_hcd);
```

66行，product_desc，主机控制器的产品描述字符串，对于UHCI，它为“UHCI Host Controller”，对于EHCI，它为“EHCI Host Controller”。

67行，irq_descr[24]，这里边儿保存的是“ehci-hcd:usb1”之类的字符串，也就是驱动的名称再加上总线编号。

71行~73行，电源管理。

78行，driver，每个主机控制器驱动都有一个struct hc_driver结构体。查看它在hcd.h里的定义。

```
149 struct hc_driver {
150     const char      *description;   /* "ehci-hcd" etc */
151     const char      *product_desc;  /* product/vendor string */
152     size_t          hcd_priv_size;  /* size of private data */
153
154     /* irq handler */
155     irqreturn_t     (*irq) (struct usb_hcd *hcd);
156
157     int     flags;
158 #define HCD_MEMORY      0x0001       /* HC regs use memory (else I/O) */
159 #define HCD_USB11       0x0010          /* USB 1.1 */
160 #define HCD_USB2        0x0020          /* USB 2.0 */
161
162     /* called to init HCD and root hub */
163     int     (*reset) (struct usb_hcd *hcd);
164     int     (*start) (struct usb_hcd *hcd);
165
166     /* NOTE:  these suspend/resume calls relate to the HC as
167      * a whole, not just the root hub; they're for PCI bus glue.
168      */
169     /* called after suspending the hub, before entering D3 etc */
170     int     (*suspend) (struct usb_hcd *hcd, pm_message_t message);
171
172     /* called after entering D0 (etc), before resuming the hub */
173     int     (*resume) (struct usb_hcd *hcd);
174
175     /* cleanly make HCD stop writing memory and doing I/O */
176     void    (*stop) (struct usb_hcd *hcd);
177
178     /* shutdown HCD */
179     void    (*shutdown) (struct usb_hcd *hcd);
180
181     /* return current frame number */
182     int     (*get_frame_number) (struct usb_hcd *hcd);
184     /* manage i/o requests, device state */
185     int     (*urb_enqueue) (struct usb_hcd *hcd,
186                             struct usb_host_endpoint *ep,
187                             struct urb *urb,
188                             gfp_t mem_flags);
189     int     (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb);
191     /* hw synch, freeing endpoint resources that urb_dequeue can't */
192     void    (*endpoint_disable)(struct usb_hcd *hcd,
193                 struct usb_host_endpoint *ep);
195     /* root hub support */
196     int             (*hub_status_data) (struct usb_hcd *hcd, char *buf);
```

```
197     int             (*hub_control) (struct usb_hcd *hcd,
198                             u16 typeReq, u16 wValue, u16 wIndex,
199                             char *buf, u16 wLength);
200     int             (*bus_suspend)(struct usb_hcd *);
201     int             (*bus_resume)(struct usb_hcd *);
202     int         (*start_port_reset)(struct usb_hcd *, unsigned port_num);
203     void            (*hub_irq_enable)(struct usb_hcd *);
204                 /* Needed only if port-change IRQs are level-triggered */
205 };
```

与usb_driver和pci_driver一样，所有的“xxx_driver”都有一堆函数指针，具体的主机控制器驱动就靠它们，这里只说一下函数指针之外的东西，也就是开头儿的那三个。

description直白点说就是驱动的大名，比如对于UHCI，它是“uhci_hcd”，对于EHCI，它就是“ehci_hcd”。product_desc和struct usb_hcd里的是一样。hcd_priv_size还是有点儿意思的，每个主机控制器驱动都会有一个私有结构体，藏在struct usb_hcd最后的那个变长数组里，这个变也是相对的，在创建usb_hcd时也得知道它能变多长，不然谁知道要申请多少内存啊，这个长度就hcd_priv_size。

81行，flags，属于HCD的一些标志，可用值就在82行和83行。它们什么意思？书到用时方恨少，flags到用时才可知。

69行，7行0，85行~91行，这几行都是专为root hub服务的。一个主机控制器对应一个Root Hub，即使在嵌入式系统里，硬件上主机控制器没有集成Root Hub，软件上也需要虚拟一个出来，也就是所谓的Virtual Root Hub。

它位置是特殊的，但需要提供的功能和其他Hub是没有什么差别的，仅仅是在和主机控制器的软硬件接口上有一些特别的规定。root hub再怎么特别也始终是一个hub，是一个USB设备，也不能脱离USB这个大家庭，也要向组织注册，也要有自己的设备生命线。

看92行，wireless，是无线USB。

94行~97行这几行都是与主机控制器的娘家PCI有关的，说到PCI就不得不说到如图8所示的那张著名的表。

	0x0–0x1	0x2–0x3	0x4–0x5	0x6–0x7	0x8	0x9–0xb	0xc	0xd	0xe	0xf
0x00	厂商ID	设备ID	命令寄存器	状态寄存器	版本修订ID	类代号	高速缓存线	延期定时器	头类型	BIST
0x10	基地址0		基地址1		基地址2		基地址3			
0x20	基地址4		基地址5		CardBus CIS指针		子系统厂商ID		子系统设备ID	
0x30	扩展ROM基地址		保留				中断线	中断引脚	Min_Gnt	Max_Lat

图 8　PCI 配置寄存器

这张表里中断号啊等很多有用的东西都在里面准备好了。94行的irq就躲在上面表儿里的倒数第四个byte，HCD可以直接拿来用，根本就不用再去申请。接下来的regs，rsrc_start，rsrc_len就与中间的Base Address0~5脱不开关系了，牵涉到所谓的I/O内存和I/O端口，下面简单说一下。

大家都知道CPU，是众人瞩目的焦点，但是它也不可能一个人战斗，电脑运转是个集体项目，CPU也需要跟各种外设配合交流，它需要访问外设里的寄存器或者内存。

主要空间的差别。一些CPU芯片有两个空间，即I/O空间和内存空间，提供有专门访问外设I/O端口的指令；而另外一些CPU只有一个空间，即内存空间。外设的I/O端口可以映射在I/O空间也可以映射到内存空间，CPU通过访问这两个空间来访问外设，I/O空间有I/O空间访问的接口，内存空间有内存空间访问的接口。当然一些外设不但有寄存器，还有内存，也就是I/O内存，比如EHCI/OHCI，它们需要映射到内存空间。但是不管映射到哪个空间，访问I/O端口还是I/O内存，CPU必须知道它们映射后的地址，不然没有办法配合交流。

在图8中，中间的那些Base Address就是保存PCI设备中I/O内存或I/O端口的首尾位置还有长度。驱动使用时要首先把它们给读出来，如果要映射到I/O空间，则要根据读到的值向系统申请I/O端口资源，如果要映射到内存空间，除了要申请内存资源，还要使用ioremap等进行映射。

96行的rsrc_start和97行的rsrc_len保存的就是从表里读出来的主机控制器的I/O端口或内存的首地址和长度，95行的regs保存的是调用ioremap_nocache映射后的内存地址。

98行，power_budget，能够提供的电流。

101行，*pool [HCD_BUFFER_POOLS]，几个dma池。因为HCD_BUFFER_POOLS在100行定义为4，所以这里就表示每个主机控制器可以有4个dma池。

我们知道主机控制器是可以进行DMA传输的，再回到前面讲的struct urb，它有两个成员transfer_dma和setup_dma，前面只是说你可以使用usb_buffer_alloc分配好DMA缓冲区给它们，然后再告诉HCD urb已经有了，HCD就可以不用再进行复杂的DMA映射了。但并没有提到你这个获取的DMA缓冲区是从哪里来的。这里所说的DMA池子里来的了。

像创建或销毁线程、数据库连接时比较的耗资源，可以先建一个池子预先创建好一批线程放里边儿，用时就从里面取出来，不用时就再放里面，用的越多就越划的来。

当然，DMA池还有其他的作用。一般来说DMA映射获得的都是以页为单位的内存，urb需要不了这么大，如果需要比较小的DMA缓冲区，就离不开DMA池了。还是查看主机控制器的这几个池子是怎么创建的，在buffer.c文件中。

```
52 int hcd_buffer_create(struct usb_hcd *hcd)
53 {
54         char            name[16];
55         int             i, size;
56 
57         if (!hcd->self.controller->dma_mask)
58             return 0;
59 
60         for (i = 0; i < HCD_BUFFER_POOLS; i++) {
61             if (!(size = pool_max [i]))
62                 continue;
63             snprintf(name, sizeof name, "buffer-%d", size);
64             hcd->pool[i] = dma_pool_create(name, hcd->self.controller,size, size, 0);
66             if (!hcd->pool [i]) {
67                 hcd_buffer_destroy(hcd);
68                 return -ENOMEM;
69             }
70         }
71         return 0;
72 }
```

这里首先要判断一下这个主机控制器支持不支持DAM，如果不支持的话再创建DMA池就是纯粹无稽之谈了。如果主机控制器支持DMA，就逐个使用dma_pool_alloc来创建DMA池，如果创建失败了，就调用同一个文件中的hcd_buffer_destroy来将已经创建成功的池子给销毁掉。

```
82 void hcd_buffer_destroy(struct usb_hcd *hcd)
83 {
84         int             i;
85 
86         for (i = 0; i < HCD_BUFFER_POOLS; i++) {
87             struct dma_pool         *pool = hcd->pool[i];
88             if (pool) {
89                 dma_pool_destroy(pool);
90                 hcd->pool[i] = NULL;
91             }
92         }
93 }
```

这里调用的dma_pool_destroy和dma_pool_alloc。带翅膀的丘比特说了，每个人的人生都要找到四个人，第一个是自己，第二个是你最爱的人，第三个是最爱你的人，第四个是共度一生的人。每个DMA池子都要找到四个函数，一个用来创建，一个用来销毁，一个用来取内存，一个用来放内存。上边儿只遇到了创建和销毁的函数，还少两个取内存和放内存的函数，再去同一个文件中找找。

```
100 void *hcd_buffer_alloc(
101     struct usb_bus  *bus,
102     size_t                  size,
103     gfp_t                   mem_flags,
104     dma_addr_t               *dma
105 )
106 {
107     struct usb_hcd          *hcd = bus_to_hcd(bus);
108     int                     i;
110     /* some USB hosts just use PIO */
111     if (!bus->controller->dma_mask) {
112         *dma = ~(dma_addr_t) 0;
113         return kmalloc(size, mem_flags);
114     }
115
116     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
117         if (size <= pool_max [i])
118         return dma_pool_alloc(hcd->pool [i], mem_flags, dma);
119     }
120     return dma_alloc_coherent(hcd->self.controller, size, dma, 0);
121 }
122
123 void hcd_buffer_free(
124     struct usb_bus  *bus,
125     size_t                  size,
126     void                    *addr,
127     dma_addr_t               dma
128 )
129 {
130     struct usb_hcd          *hcd = bus_to_hcd(bus);
131     int                     i;
133     if (!addr)
134         return;
136     if (!bus->controller->dma_mask) {
137         kfree(addr);
138         return;
139     }
141     for (i = 0; i < HCD_BUFFER_POOLS; i++) {
142         if (size <= pool_max [i]) {
143             dma_pool_free(hcd->pool [i], addr, dma);
144             return;
145         }
146     }
147     dma_free_coherent(hcd->self.controller, size, addr, dma);
148 }
```

又可以找到两个函数，即dma_pool_alloc和dma_pool_free，现在可以凑齐四个函数了。不过不用管它们四个到底什么长相，只要它们能够干活儿就成了，这里要注意的是以下几个问题。

第一个问题是即使你的主机控制器不支持DMA，这几个函数也是可以用的，只不过创建的不是DMA池子，取的也不是DMA缓冲区。此时，DMA池子不存在，hcd_buffer_alloc获取的只是适用kmalloc申请的普通内存。当然，相应地，你必须在它没有利用价值时使用hcd_buffer_free将它释放掉。

第二个问题是size的问题，它们里面都有一个pool_max，这是个同一个文件中定义的数组。

```
26 static const size_t     pool_max [HCD_BUFFER_POOLS] = {
27      /* platfor ms without dma-friendly caches might need to
28       * prevent cacheline sharing...
29       */
30      32,
31      128,
32      512,
33      PAGE_SIZE / 2
34      /* bigger --> allocate pages */
35 };
```

这个数组中定义的就是四个池子中每个池子里保存的DMA缓冲区的size。注意这里虽说只定义了四种size，但是并不说明你使用hcd_buffer_alloc获取DMA缓冲区时不能指定更大的size，如果这几个池子都满足不了要求，那就会使用dma_alloc_coherent建立一个新的DMA映射。还有，每个人的情况都不一样，不可能都会完全恰好和上面定义的四种size一致，那也不用怕，使用这个size获取DMA缓冲区时，池子会选择一个略大一些的回馈过去。

还是让咱们抛开size，回到struct usb_hcd中来。103行，state，主机控制器的状态，紧挨着它的下面那些行就是相关的可用值和宏定义。

设备的生命线（九）

说完了struct usb_hcd和struct usb_bus，接下来就该查看usb_submit_urb()最后的那个遗留问题usb_hcd_submit_urb()了。

现在内核中有个很不好的现象，设计结构比复杂，函数比长。

```
921 int usb_hcd_submit_urb (struct urb *urb, gfp_t mem_flags)
922 {
923      int                    status;
924      struct usb_hcd          *hcd = bus_to_hcd(urb->dev->bus);
925      struct usb_host_endpoint *ep;
926      unsigned long           flags;
927
928      if (!hcd)
929          return -ENODEV;
930
931      usbmon_urb_submit(&hcd->self, urb);
932
933      /*
934       * Atomically queue the urb,  first to our records, then to the HCD.
935       * Access to urb->status is controlled by urb->lock ... changes on
936       * i/o completion (normal or fault) or unlinking.
937       */
938
939      // FIXME:  verify that quiescing hc works right (RH cleans up)
940
941      spin_lock_irqsave (&hcd_data_lock, flags);
942      ep = (usb_pipein(urb->pipe) ? urb->dev->ep_in : urb->dev->ep_out)
943                  [usb_pipeendpoint(urb->pipe)];
944      if (unlikely (!ep))
945          status = -ENOENT;
946      else if (unlikely (urb->reject))
947          status = -EPERM;
948      else switch (hcd->state) {
949      case HC_STATE_RUNNING:
950      case HC_STATE_RESUMING:
951 doit:
952      list_add_tail (&urb->urb_list, &ep->urb_list);
953          status = 0;
954          break;
955      case HC_STATE_SUSPENDED:
956      /* HC upstream links (register access, wakeup signaling) can work
957       * even when the downstream links (and DMA etc) are quiesced; let
958       * usbcore talk to the root hub.
959       */
960        if (hcd->self.controller->power.power_state.event==PM_EVENT_ON
961                              && urb->dev->parent == NULL)
962                      goto doit;
963          /* FALL THROUGH */
964      default:
965          status = -ESHUTDOWN;
966          break;
967      }
968      spin_unlock_irqrestore (&hcd_data_lock, flags);
969      if (status) {
970      INIT_LIST_HEAD (&urb->urb_list);
971          usbmon_urb_submit_error(&hcd->self, urb, status);
972          return status;
973      }
974
975      /* increment urb's reference count as part of giving it to the HCD
976       * (which now controls it).  HCD guarantees that it either returns
977       * an error or calls giveback(), but not both.
978       */
979      urb = usb_get_urb (urb);
980      atomic_inc (&urb->use_count);
981
982      if (urb->dev == hcd->self.root_hub) {
983          /* NOTE:  requirement on hub callers (usbfs and the hub
```

```
984          * driver, for now) that URBs' urb->transfer_buffer be
985          * valid and usb_buffer_{sync,unmap}() not be needed, since
986          * they could clobber root hub response data.
987          */
988         status = rh_urb_enqueue (hcd, urb);
989         goto done;
990     }
992     /* lower level hcd code should use *_dma exclusively,
993      * unless it uses pio or talks to another transport.
994      */
995     if (hcd->self.uses_dma) {
996         if (usb_pipecontrol (urb->pipe)
997                 && !(urb->transfer_flags & URB_NO_SETUP_DMA_MAP))
998             urb->setup_dma = dma_map_single (
999                             hcd->self.controller,
1000                                urb->setup_packet,
1001                                sizeof (struct usb_ctrlrequest),
1002                            DMA_TO_DEVICE);
1003        if (urb->transfer_buffer_length != 0
1004                && !(urb->transfer_flags & URB_NO_TRANSFER_DMA_MAP))
1005        urb->transfer_dma = dma_map_single (
1006                                hcd->self.controller,
1007                            urb->transfer_buffer,
1008                                urb->transfer_buffer_length,
1009                                usb_pipein (urb->pipe)
1010                                    ? DMA_FROM_DEVICE
1011                                    : DMA_TO_DEVICE);
1012    }
1014    status = hcd->driver->urb_enqueue (hcd, ep, urb, mem_flags);
1015 done:
1016    if (unlikely (status)) {
1017        urb_unlink (urb);
1018        atomic_dec (&urb->use_count);
1019        if (urb->reject)
1020            wake_up (&usb_kill_urb_queue);
1021        usbmon_urb_submit_error(&hcd->self, urb, status);
1022        usb_put_urb (urb);
1023    }
1024    return status;
1025 }
```

usb_hcd_submit_urb是hcd.c里的，目标也很明确，就是将提交过来的urb指派给合适的主机控制器驱动程序。core目录下面以hcd打头的几个文件严格来说不能算是HCD，只能算HCDI，即主机控制器驱动的接口层，这用来衔接具体的主机控制器驱动和USB Core的。

924行，bus_to_hcd在哪里提到过一下，是用来获得struct usb_bus结构体对应的struct usb_hcd结构体，urb要去的那个设备所在的总线是在设备生命线的开始就初始化好了的，忘了可以再复习一下。bus_to_hcd还有一个兄弟hcd_to_bus，都在hcd.h中定义。

```
131 static inline struct usb_bus *hcd_to_bus (struct usb_hcd *hcd)
132 {
133     return &hcd->self;
134 }
136 static inline struct usb_hcd *bus_to_hcd (struct usb_bus *bus)
137 {
138     return container_of(bus, struct usb_hcd, self);
139 }
```

继续看928行，也是大多数函数开头儿必备的常规检验，如果usb_hcd都还是空的，那就返回吧。

931行，usbmon_urb_submit就是与前面Greg孕育出来的USB Monitor有关的，如果你编译内核时没有配置上CONFIG_USB_MON，它就是一个空函数。

941行，去获得一把锁，这把锁在hcd.c的开头儿就已经初始化好了，所以说是把全局锁。

```
102 /* used when updating hcd data */
103 static DEFINE_SPINLOCK(hcd_data_lock);
```

前面多次遇到过自旋锁，现在就简单介绍一下。它和信号量，还有前面提到的completion一样都是Linux里用来进行代码同步。为什么要进行同步？你要知道在Linux这个庞大负责的世界里，可能同时有多个线程，那么只要它们互相之间有一定的共享，就必须要保证一个线程操作这个共享时让其他线程知道。

自旋锁身为同步机制的一种，自然也有它独特的本事，它可以用在中断上下文或者原子上下文使用。上下文就是代码运行的环境，Linux的这个环境使用二分法可以分成两种，能休眠的环境和不能休眠的环境。像信号量和completion就只能用在可以休眠的环境，而自旋锁就用在不能休眠的环境里。

而usb_submit_urb还有usb_hcd_submit_urb必须得在两种环境里都能够使用，所以使用的是自旋锁，想一想urb的结束处理函数，它就是不能休眠的，但它里面必须得能够重新提交urb。

再说一说hcd_data_lock这把锁都是用来保护什么的，为什么要使用它？主机控制器的struct usb_hcd结构体在它的驱动里早就初始化好了，但同一时刻是可能有多个urb向同一个主机控制器申请进行传输，可能有多个地方都希望访问它里面的内容的。比如948行的state元素，显然就要同步了，hcd_data_lock这把锁就是专门用来保护主机控制器的这个结构体的。

942行，遇到多次也说过多次了，知道了pipe，就可以从struct usb_device里的两个数组ep_in和ep_out里拿出对应端点的struct usb_host_endpoint结构体。

944行~973行，这些行都是做检验的。

944行，显然都走到这一步了，目的端点为空的可能性太小了，所以加上了unlikely。

946行，前面还说过，urb里的这个reject，只有usb_kill_urb有特权修改它，如果走到这里发现它的值大于0了，那就说明哪里调用了usb_kill_urb要终止这次传输，所以还是返回吧，不过这种可能性比较小，没人无聊到那种地步，总是刚提交就终止，吊主机控制器胃口，所以仍然加上unlikely。

948行，如果上面那两个检验都通过了，现在就“case”一下主机控制器的状态，如果为HC_STATE_RUNNING或HC_STATE_RESUMING就说明主机控制器这边儿没问题，尽管将这个urb往端点的urb队列里塞好了，952行就是完成这个的，这行是小case了。如果主机控制器的状态为HC_STATE_SUSPENDED，但它的上行链路能够工作，而且这个urb是送往Root Hub的，则将其塞到Root Hub的urb队列里。

然后判断上面几次检验的结果，如果一切正常，则继续往下走，否则就返回吧。

979行，检验都通过了，可以放心地增加urb的引用计数了。

980行，将urb的use_count也增加1，表示urb已经被HCD接受了，正在被处理着。你如果对这两个引用计数什么差别还有疑问，再前面讲解struct urb时。

982行，判断这个urb是不是流向Root Hub的，如果是，它就走向了Root Hub的生命线。不过，毕竟你更关注的是USB设备，应该很少有机会直接和Root Hub交流些什么。

995行，如果这个主机控制器支持DMA，可你却没有告诉它URB_NO_SETUP_DMA_MAP或URB_NO_TRANSFER_DMA_MAP这两个标志，它就会认为你在urb里没有提供DMA的缓冲区，就会调用dma_map_single将setup_packet或transfer_buffer映射为DMA缓冲区。

1014行，终于可以将urb交给具体的主机控制器驱动程序了。

到现在，设备已经可以进入Address状态。该继续看设备的那条生命线了。

设备的生命线（十）

跟着设备的生命线走到现在，我算是明白了，什么东西的发展都是越往后越高级越复杂，再查看下面这张小表，比较下和上次那张表出现时有什么变化。

表 1.20.2

State	USB_STATE_ADDRESS
Speed	taken
ep0	ep0.urb_list，描述符长度/类型，wMaxPacketSize

接下来设备的目标当然就是Configured了。

要进入Configured状态，就得去配置设备，当然不能是盲目地去配置，要知道设备是可能有多个配置的，所以要有选择、有目的、有步骤、有计划地去配置，但先去获得设备的设备描述符，message.c中的usb_get_device_descriptor()就是core里专门干这个的。

```
    860 int usb_get_device_descriptor(struct usb_device *dev, unsigned int size)
    861 {
    862     struct usb_device_descriptor *desc;
    863     int ret;
    864
    865     if (size > sizeof(*desc))
    866         return -EINVAL;
    867     desc = kmalloc(sizeof(*desc), GFP_NOIO);
    868     if (!desc)
    869         return -ENOMEM;
    870
    871     ret = usb_get_descriptor(dev, USB_DT_DEVICE, 0, desc, size);
    872     if (ret >= 0)
    873         memcpy(&dev->descriptor, desc, size);
    874     kfree(desc);
    875     return ret;
    876 }
```

这个函数比较的精练，先是准备了一个struct usb_device_descriptor结构体，然后就用它去调用message.c里的usb_get_descriptor()获得设备描述符，获得之后再把得到的描述符复制到设备struct usb_device结构体的descriptor成员里。因此，这个函数成功与否的关键就在usb_get_descriptor()。其实对于写驱动的来说，眼里是只有usb_get_descriptor()没有usb_get_device_descriptor()的，不管你想获得哪种描述符都是要通过usb_get_descriptor()，而usb_get_device_descriptor()是专属内核用的接口。

```
    618 int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned char index,
void *buf, int size)
    619 {
    620     int i;
    621     int result;
    623     me mset(buf,0,size);     // Make sure we parse really received data
    625     for (i = 0; i < 3; ++i) {
    626         /* retry on length 0 or stall; some devices are flakey */
    627         result = usb_control_ msg(dev, usb_rcvctrlpipe(dev, 0),
    628                         USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
    629                         (type << 8) + index, 0, buf, size,
    630                         USB_CTRL_GET_TIMEOUT);
    631         if (result == 0 || result == -EPIPE)
    632             continue;
    633         if (result > 1 && ((u8 *)buf)[1] != type) {
    634             result = -EPROTO;
    635             continue;
    636         }
    637         break;
    638     }
    639     return result;
    640 }
```

参数type就是用来区分不同的描述符的，协议中说了，GET_DESCRIPTOR请求主要就是适用于三种描述符，设备描述符、配置描述符和字符串描述符。参数index是要获得的描述符的序号，如果希望得到的这种描述符设备中可以有多个，你需要指定获得其中的哪个，比如配置描述符就可以有多个，不过对于设备描述符来说，只有一个，所以这里的index应该为0。参数buf和size就是描述你用来放置获得的描述符的缓冲区的。

这个函数的内容挺单调的，主要就是调用了一个usb_control_ msg()。这里要说的第一个问题是它的一堆参数，这就需要认真了解一下如图9所示的这张表。

bmRequestType	bRequest	wValue	wLength	Data
10000000B	GET DESCRIPTOR	描述符的类型和序号(index)	描述符长度	描述符

图 1.20.1　GET_DESCRIPTOR 请求

GET_DESCRIPTOR请求的数据传输方向是device-to-host，而且还是协议中规定所有设备都要支持的标准请求，也不是针对端点或者接口，而是针对设备的。所以bRequestType只能为0x80，就是上面表里的10000000B，也等于628行的USB_DIR_IN。wValue的高位字节表示描述符的类型，低位字节表示描述符的序号，所以就有629行的(type << 8) + index。wIndex对于字符串描述符应该设置为使用语言的ID，对于其他的描述符应该设置为0，所以也有了629行中间的那个0。至于wLength，就是描述符的长度，对于设备描述符，一般来说你都会指定为USB_DT_DEVICE_SIZE。

USB_CTRL_GET_TIMEOUT是定义在include/linux/usb.h里的一个宏，值为5000，表示有5s的超时时间。

```
1330 #define USB_CTRL_GET_TIMEOUT    5000
1331 #define USB_CTRL_SET_TIMEOUT    5000
```

第二个问题就是为什么会有3次循环。这个又要归咎于一些不守规矩的厂商了，搞出的设备古里古怪的，比如一些USB读卡器，一次请求还不一定能成功，但是设备描述符拿不到接下来就没法子走了，所以这里多试几次。至于631行到636行之间的代码都是判断是不是成功得到请求的描述符的，这个版本的内核在这里的判断还比较混乱，就不多说了，你只要知道((u8 *)buf)[1] != type是用来判断获得描述符是不是请求的类型就可以了。

现在设备描述符已经有了，但是只有设备描述符是远远不够的，你从设备描述符里只能知道它一共支持几个配置，但是要配置一个设备时要知道具体每个配置有什么用，所以接下来就要获得各个配置的配置描述符，并且拿结果去充实struct usb_device的config、rawdescriptors等相关元素。

core内部并不直接调用上面的usb_get_descriptor()去完成这个任务，而是调用config.c里的usb_get_configuration()，为什么？core总是需要做更多的事情，不然就不叫core了。

```
476 int usb_get_configuration(struct usb_device *dev)
477 {
478     struct device *ddev = &dev->dev;
479     int ncfg = dev->descriptor.bNumConfigurations;
480     int result = -ENOMEM;
481     unsigned int cfgno, length;
482     unsigned char *buffer;
483     unsigned char *bigbuffer;
484     struct usb_config_descriptor *desc;
485
486     if (ncfg > USB_MAXCONFIG) {
487         dev_warn(ddev, "too many configurations: %d, "
488                 "using maximum allowed: %d\n", ncfg, USB_MAXCONFIG);
489         dev->descriptor.bNumConfigurations = ncfg = USB_MAXCONFIG;
490     }
491
492     if (ncfg < 1) {
493         dev_err(ddev, "no configurations\n");
494         return -EINVAL;
495     }
496
497     length = ncfg * sizeof(struct usb_host_config);
498     dev->config = kzalloc(length, GFP_KERNEL);
499     if (!dev->config)
500         goto err2;
501
502     length = ncfg * sizeof(char *);
503     dev->rawdescriptors = kzalloc(length, GFP_KERNEL);
504     if (!dev->rawdescriptors)
505         goto err2;
506
507     buffer = kmalloc(USB_DT_CONFIG_SIZE, GFP_KERNEL);
508     if (!buffer)
509         goto err2;
510     desc = (struct usb_config_descriptor *)buffer;
511
512     for (cfgno = 0; cfgno < ncfg; cfgno++) {
513         /* We grab just the first descriptor so we know how long
514          * the whole configuration is */
515         result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
516                 buffer, USB_DT_CONFIG_SIZE);
517         if (result < 0) {
518             dev_err(ddev, "unable to read config index %d "
```

```
519                          "descriptor/%s\n", cfgno, "start");
520                 dev_err(ddev, "chopping to %d config(s)\n", cfgno);
521                 dev->descriptor.bNumConfigurations = cfgno;
522                 break;
523             } else if (result < 4) {
524                 dev_err(ddev, "config index %d descriptor too short "
525                          "(expected %i, got %i)\n", cfgno,
526                          USB_DT_CONFIG_SIZE, result);
527                 result = -EINVAL;
528                 goto err;
529             }
530             length = max((int) le16_to_cpu(desc->wTotalLength),
531                     USB_DT_CONFIG_SIZE);
532 
533         /* Now that we know the length, get the whole thing */
534             bigbuffer = kmalloc(length, GFP_KERNEL);
535             if (!bigbuffer) {
536                 result = -ENOMEM;
537                 goto err;
538             }
539             result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno,
540                     bigbuffer, length);
541             if (result < 0) {
542                 dev_err(ddev, "unable to read config index %d "
543                         "descriptor/%s\n", cfgno, "all");
544                 kfree(bigbuffer);
545                 goto err;
546             }
547             if (result < length) {
548                 dev_warn(ddev, "config index %d descriptor too short "
549                     "(expected %i, got %i)\n", cfgno, length, result);
550                 length = result;
551             }
552 
553             dev->rawdescriptors[cfgno] = bigbuffer;
554 
555             result = usb_parse_configuration(&dev->dev, cfgno,
556                     &dev->config[cfgno], bigbuffer, length);
557             if (result < 0) {
558                 ++cfgno;
559                 goto err;
560             }
561         }
562         result = 0;
563 
564 err:
565         kfree(buffer);
566         dev->descriptor.bNumConfigurations = cfgno;
567 err2:
568         if (result == -ENOMEM)
569             dev_err(ddev, "out of memory\n");
570         return result;
571 }
```

要想得到配置描述符，最终都不可避免要向设备发送GET_DESCRIPTOR请求，这就需要以USB_DT_CONFIG为参数调用usb_get_descriptor函数，也就需要知道该为获得的描述符准备多大的一个缓冲区。本来这个长度应该很明确地为USB_DT_CONFIG_SIZE，它表示的就是配置描述符的大小，但是实际上不是这么回事儿，USB_DT_CONFIG_SIZE只表示配置描述符本身的大小，并不表示GET_DESCRIPTOR请求返回结果的大小。因为向设备发送GET_DESCRIPTOR请求时，设备并不只是返回一个配置描述符，而是将这个配置下面的所有接口描述符、端点描述还有class-或vendor-specific描述符都返回了。

那么这个总长度如何得到呢？在神秘的配置描述符里有一个神秘的字段wTotalLength，它里面记录的就是这个总长度。那么问题就简单了，可以首先发送USB_DT_CONFIG_SIZE个字节的请求过去，获得这个配置描述符的内容，从而获得那个总长度，然后以这个长度再请求一次，这样就可以获得一个配置下面所有的描述符内容了。上面的usb_get_configuration()采用的就是这个处理方法。

479行，获得设备理配置描述符的数目。

486行，USB_MAXCONFIG是config.c中定义的。

```
14 #define USB_MAXCONFIG                    8       /* Arbitrary limit */
```

限制了一个设备最多只能支持8种配置拥有8个配置描述符，如果超出了这个限制，489行就强制它为这个最大值。不过如果设备中没有任何一个配置描述符，什么配置都没有，那是不可能的，492行这关就过不去。

498行，struct usb_device里的config表示设备拥有的所有配置，你设备有多少个配置就为它准备多大的空间。

503行，rawdescriptors还认识吧，这是一个字符指针数组里的每一项都指向一个使用GET_DESCRIPTOR请求去获取配置描述符时所得到的结果。

507行，准备一个大小为USB_DT_CONFIG_SIZE的缓冲区，第一次发送GET_DESCRIPTOR请求要用的。

512行，剩下的主要就是这个for循环了，获取每一个配置的那些描述符。

515行，如上面所说的，首先发送USB_DT_CONFIG_SIZE个字节请求，获得配置描述符的内容。然后对返回的结果进行检验，知道为什么523行会判断结果是不是小于4吗？答案尽在配置描述符中，里面的第3字节和4字节就是wTotalLength，只要得到前4个字节，就已经完成任务能够获得总长度了。

534行，既然总长度已经有了，那么这里就为接下来的GET_DESCRIPTOR请求准备一个大点的缓冲区。

539行，现在可以获得这个配置相关的所有描述符了。然后是对返回结果的检验，将得到的那一堆数据的地址赋给rawdescriptors数组里的指针。

555行，从这个颇有韵味的数字555开始，你将会遇到另一个超级变态的函数，它将对前面GET_DESCRIPTOR请求获得的那堆数据做处理。

设备的生命线（十一）

现在已经使用GET_DESCRIPTOR请求取到了包含一个配置里所有相关描述符内容的一堆数据，这些数据是raw，即原始的，所有数据不管是配置描述符、接口描述符还是端点描述符都挤在一起，所以得想办法将它们给分开，于是用到了usb_parse_configuration()。

```
264 static int usb_parse_configuration(struct device *ddev, int cfgidx,
265     struct usb_host_config *config, unsigned char *buffer, int size)
266 {
267      unsigned char *buffer0 = buffer;
268      int cfgno;
269      int nintf, nintf_orig;
270      int i, j, n;
271      struct usb_interface_cache *intfc;
272      unsigned char *buffer2;
273      int size2;
274      struct usb_descriptor_header *header;
275      int len, retval;
276      u8 inu ms[USB_MAXINTERFACES], nalts[USB_MAXINTERFACES];
277
278      memcpy(&config->desc, buffer, USB_DT_CONFIG_SIZE);
279      if (config->desc.bDescriptorType != USB_DT_CONFIG ||
280          config->desc.bLength < USB_DT_CONFIG_SIZE) {
281          dev_err(ddev, "invalid descriptor for config index %d: "
282              "type = 0x%X, length = %d\n", cfgidx,
283                  config->desc.bDescriptorType, config->desc.bLength);
284          return -EINVAL;
285      }
286      cfgno = config->desc.bConfigurationValue;
287
288      buffer += config->desc.bLength;
289      size -= config->desc.bLength;
290
291      nintf = nintf_orig = config->desc.bNumInterfaces;
292      if (nintf > USB_MAXINTERFACES) {
293          dev_warn(ddev, "config %d has too many interfaces: %d, "
294                  "using maximum allowed: %d\n",
295                  cfgno, nintf, USB_MAXINTERFACES);
296          nintf = USB_MAXINTERFACES;
297      }
```

```
298
299     /* Go through the descriptors, checking their length and counting the
300      * number of altsettings for each interface */
301     n = 0;
302     for ((buffer2 = buffer, size2 = size);
303           size2 > 0;
304           (buffer2 += header->bLength, size2 -= header->bLength)) {
305
306          if (size2 < sizeof(struct usb_descriptor_header)) {
307               dev_warn(ddev, "config %d descriptor has %d excess "
308                         "byte%s, ignoring\n",
309                            cfgno, size2, plural(size2));
310               break;
311          }
312
313          header = (struct usb_descriptor_header *) buffer2;
314          if ((header->bLength > size2) || (header->bLength < 2)) {
315               dev_warn(ddev, "config %d has an invalid descriptor "
316                         "of length %d, skipping remainder of the config\n",
317                         cfgno, header->bLength);
318               break;
319          }
320
321          if (header->bDescriptorType == USB_DT_INTERFACE) {
322               struct usb_interface_descriptor *d;
323               int inum;
324
325               d = (struct usb_interface_descriptor *) header;
326               if (d->bLength < USB_DT_INTERFACE_SIZE) {
327                    dev_warn(ddev, "config %d has an invalid "
328                                 "interface descriptor of length %d, "
329                                 "skipping\n", cfgno, d->bLength);
330                    continue;
331               }
332
333               inum = d->bInterfaceNumber;
334               if (inum >= nintf_orig)
335                    dev_warn(ddev, "config %d has an invalid "
336                             "interface number: %d but max is %d\n",
337                                 cfgno, inum, nintf_orig - 1);
338
339               /* Have we already encountered this interface?
340                * Count its altsettings */
341               for (i = 0; i < n; ++i) {
342               if (inu ms[i] == inum)
343                         break;
344               }
345               if (i < n) {
346                    if (nalts[i] < 255)
347                         ++nalts[i];
348               } else if (n < USB_MAXINTERFACES) {
349                    inu ms[n] = inum;
350                    nalts[n] = 1;
351                    ++n;
352               }
353
354          } else if (header->bDescriptorType == USB_DT_DEVICE ||
355                         header->bDescriptorType == USB_DT_CONFIG)
356               dev_warn(ddev, "config %d contains an unexpected "
357                         "descriptor of type 0x%X, skipping\n",
358                         cfgno, header->bDescriptorType);
359
360     }      /* for ((buffer2 = buffer, size2 = size); ...) */
361     size = buffer2 - buffer;
362     config->desc.wTotalLength = cpu_to_le16(buffer2 - buffer0);
363
364     if (n != nintf)
365          dev_warn(ddev, "config %d has %d interface%s, different from "
366                    "the descriptor's value: %d\n",
367                    cfgno, n, plural(n), nintf_orig);
368     else if (n == 0)
369          dev_warn(ddev, "config %d has no interfaces?\n", cfgno);
370     config->desc.bNumInterfaces = nintf = n;
371
372     /* Check for missing interface numbers */
```

```
373     for (i = 0; i < nintf; ++i) {
374         for (j = 0; j < nintf; ++j) {
375             if (inu ms[j] == i)
376                 break;
377         }
378         if (j >= nintf)
379             dev_warn(ddev, "config %d has no interface number "
380                     "%d\n", cfgno, i);
381     }
382 
383     /* Allocate the usb_interface_caches and altsetting arrays */
384     for (i = 0; i < nintf; ++i) {
385         j = nalts[i];
386         if (j > USB_MAXALTSETTING) {
387             dev_warn(ddev, "too many alternate settings for "
388                     "config %d interface %d: %d, "
389                     "using maximum allowed: %d\n",
390                       cfgno, inu ms[i], j, USB_MAXALTSETTING);
391             nalts[i] = j = USB_MAXALTSETTING;
392         }
393 
394         len = sizeof(*intfc) + sizeof(struct usb_host_interface) * j;
395         config->intf_cache[i] = intfc = kzalloc(len, GFP_KERNEL);
396         if (!intfc)
397             return -ENOMEM;
398         kref_init(&intfc->ref);
399     }
400 
401     /* Skip over any Class Specific or Vendor Specific descriptors;
402      * find the first interface descriptor */
403     config->extra = buffer;
404     i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
405           USB_DT_INTERFACE, &n);
406     config->extralen = i;
407     if (n > 0)
408         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
409                 n, plural(n), "configuration");
410     buffer += i;
411     size -= i;
412 
413     /* Parse all the interface/altsetting descriptors */
414     while (size > 0) {
415         retval = usb_parse_interface(ddev, cfgno, config,
416                 buffer, size, inu ms, nalts);
417         if (retval < 0)
418             return retval;
419 
420         buffer += retval;
421         size -= retval;
422     }
423 
424     /* Check for missing altsettings */
425     for (i = 0; i < nintf; ++i) {
426         intfc = config->intf_cache[i];
427         for (j = 0; j < intfc->num_altsetting; ++j) {
428             for (n = 0; n < intfc->num_altsetting; ++n) {
429                 if (intfc->altsetting[n].desc.
430                         bAlternateSetting == j)
431                     break;
432             }
433             if (n >= intfc->num_altsetting)
434                 dev_warn(ddev, "config %d interface %d has no "
435                         "altsetting %d\n", cfgno, inu ms[i], j);
436         }
437     }
438 
439     return 0;
440 }
```

其实前面也说到过的，使用GET_DESCRIPTOR请求时，得到的数据并不是杂乱无序的，而是有规可循的。一般来说，配置描述符后面跟的是第一个接口的接口描述符，接着是这个接口里第一个端点的端点描述符，如果有class-和vendor-specific描述符的话，会紧跟在对应的标准描述符后面，不管接口有多少端点都是按照这个规律顺序排列。

当然有些厂商会特立独行一些，非要先返回第二个接口然后再返回第一个接口，但配置描述符后面总归先是接口描述符再是端点描述符。

267行，buffer里保存的就是GET_DESCRIPTOR请求获得的数据，要解析这些数据，不可避免地要对buffer指针进行操作，这里先将它备份一下。

278行，config是参数中传递过来的，是设备struct usb_device结构体里的struct usb_host_config结构体数组config中的一员。不出意外的话，buffer的前USB_DT_CONFIG_SIZE个字节对应的就是配置描述符，那么这里的意思就很明显了。

然后检验一下，查看这USB_DT_CONFIG_SIZE字节的内容究竟是不是正如我们所期待的那样是个配置描述符，如果不是，那buffer里的数据问题可就大了，没什么利用价值了，还是返回吧，不必要再接着解析了。

288行，buffer的前USB_DT_CONFIG_SIZE个字节已经理清了，接下来该解析剩下的数据了，buffer需要紧跟形势的发展，位置和长度都要做相应的修正。

291行，获得这个配置所拥有的接口数目，不能简单赋值就行了，得知道系统里对这个数目是有USB_MAXINTERFACES这样的限制。如果数目比这个限制还大，就改为USB_MAXINTERFACES。

302行~360行，这函数就是统计记录一下这个配置里每个接口所拥有的设置数目。所以这里会提醒你一下，千万别被迷惑了，这个循环里使用的是buffer2和size2，buffer和size的两个替身。

306行，这里遇到一个新的结构struct usb_descriptor_header，在include/linux/usb/ch9.h中定义。

```
195 struct usb_descriptor_header {
196         __u8  bLength;
197         __u8  bDescriptorType;
198 } __attribute__ ((packed));
```

这个结构就包括了两个成员，它们的前两个字节都是一样的，一个表示描述符的长度，一个表示描述符的类型。

那么为什么要专门搞这么一个结构？试想一下，有一块数据缓冲区，让你判断一下里面保存的是哪个描述符，或者是其他什么东西，你怎么做？当然可以直接将它的前两个字节内容读出来，判断bDescriptorType，再判断bLength。不过这样的代码就好像你自己画的一副抽象画，太艺术化了，过个若干年自己都不知道什么意思，更别说别人了。313行把buffer2指针转化为struct usb_descriptor_header的结构体指针，然后就可以使用‘->’来取出bLength和bDescriptorType。

那么306行就表示如果GET_DESCRIPTOR请求返回的数据里除了包括一个配置描述符外，连两个字节都没有，能看不能用。

321行，如果这是个接口描述符就说明这个配置的某个接口拥有一个设置是没有什么所谓的设置描述符的，一个接口描述符就代表了存在一个设置，接口描述里的bInterfaceNumber会指出这个设置隶属于哪个接口。那么这里除了是接口描述符还有可能是class-和vendor-specific描述符。

325行，既然觉得这是个接口描述符，就把这个指针转化为struct usb_interface_descriptor结构体指针，你可别被C语言里的这种指针游戏给转晕了，一个地址如果代码不给它赋予什么意义，它除了表示一个地址外就什么都不是。同样一个地址上面转化为struct usb_descriptor_header结构体指针和这里转化为struct usb_interface_descriptor结构体指针，它就不再仅仅是一个地址，而是代表了不同的含义。

326行，bDescriptorType等于USB_DT_INTERFACE并不说明它就一定是接口描述符了，它的bLength还必须要等于USB_DT_INTERFACE_SIZE。bLength和bDescriptorType一起才能决定一个描述符。

341行~352行，这几行首先要明白n、inu ms和nalts这表示什么，n记录的是接口的数目，数组inu ms里的每一项都表示一个接口号，数组nalts里的每一项记录的是每个接口拥有的设置数目，inu ms和nalts两个数组里的元素是一一对应的，inu ms[0]就对应nalts[0]，inu ms[1]就对应nalts[1]。其次还要记住，发送GET_DESCRIPTOR请求时，设备并不一定会按照接口1，接口2这样的顺序循规蹈矩的返回数据。

361行，buffer的最后边儿可能会有些垃圾数据，为了去除这些垃圾数据，这里需要将size和配置描述符里的wTotalLength修正一下。

364行，经过上面的循环之后，如果统计得到的接口数目和配置描述符里的bNumInterfaces不符，或者干脆就没有发现配置里有什么接口，就警告一下。

373行，一个for循环，目的是看一看是不是遗漏了哪个接口号，比如说配置6个接口，每个接口号都应该对应数组inu ms里的一项，如果在inu ms里面没有发现这个接口号，比如2吧，那2这个接口号就神秘失踪了，你找不到接口2。这个当然也属于违章驾驶，需要警告一下。

384行，又是一个for循环， USB_MAXALTSETTING的定义在config.c里。

```
11 #define USB_MAXALTSETTING           128     /* Hard limit */
```

一个接口最多可以有128个设置，足够了。394行根据每个接口拥有的设置数目为对应的intf_cache数组项申请内存。

403行，配置描述符后面紧跟的不一定就是接口描述符，还可能是class-和vendor-specific描述符。不管有没有，先把buffer的地址赋给extra，如果没有扩展的描述符，则404行返回的i就等于0，extralen也就为0。

404行，调用find_next_descriptor()在buffer里寻找配置描述符后面跟着的第一个接口描述符。它也在config.c中定义，进去看一看。

```
22 static int find_next_descriptor(unsigned char *buffer, int size,
23     int dt1, int dt2, int *num_skipped)
24 {
25       struct usb_descriptor_header *h;
26       int n = 0;
27       unsigned char *buffer0 = buffer;
28
29       /* Find the next descriptor of type dt1 or dt2 */
30       while (size > 0) {
31           h = (struct usb_descriptor_header *) buffer;
32           if (h->bDescriptorType == dt1 || h->bDescriptorType == dt2)
33               break;
34           buffer += h->bLength;
35           size -= h->bLength;
36           ++n;
37       }
38
39       /* Store the number of descriptors skipped and return the
40        * number of bytes skipped */
41       if (num_skipped)
42           *num_skipped = n;
43       return buffer - buffer0;
44 }
```

这个函数需要传递两个描述符类型的参数，32行已经清清楚楚地表明它不是专一地去寻找一种描述符，而是去寻找两种描述符，比如你指定dt1为USB_DT_INTERFACE，dt2为USB_DT_ENDPOINT时，只要能够找到接口描述符或端点描述符中的一个，这个函数就返回。usb_parse_configuration函数的404行只需要寻找下一个接口描述符，所以dt1和dt2都设置为USB_DT_INTERFACE。

这个函数结束后，num_skipped里记录的是搜索过程中忽略的dt1和dt2之外其他描述符的数目，返回值表示搜索结束时buffer的位置比搜索开始时前进的字节数。其他没什么好讲的，还是回到usb_parse_configuration函数。

410行，根据find_next_descriptor的结果修正buffer和size。你可能对C语言里的按引用传递和按值传递已经烂熟于心，看到find_next_descriptor()那里传递的是buffer，一个指针，条件反射地觉得它里面对buffer的修改必定影响了外面的buffer，所以认为buffer已经指向了寻找到的接口描述符。但是find_next_descriptor里修改的只是参数中buffer的值，并没有修改它指向的内容，对于地址本身来说仍然只能算是按值传递，怎么修改都影响不到函数外边，所以这里的410行仍然要对buffer的位置进行修正。

414行，事不过三，三个for循环之后轮到了一个while循环。如果size大于0，就说明配置描述符后面找到了一个接口描述符，根据这个接口描述符的长度，已经可以解析出一个完整的接口描述符了，但是仍然没到乐观时，这个接口描述符后面还会跟着一群端点描述符，再然后还会有其他的接口描述符。

所以我们又迎来了另一个函数usb_parse_interface，先不管这个它长什么样子，毕竟usb_parse_configuration()就快到头了，暂时只需要知道它返回时，buffer的位置已经在下一个接口描述符那里了，同理对buffer地址本身来说是按值传递的，所以420行要对这个位置和长度进行下调整以适应新形势。那么这个while循环的意思就很明显了，对buffer一段一段的解析，直到再也找不到接口描述符了。

425行，最后这个for循环没什么实质性的内容，就是找一下每个接口是不是有哪个设置编号给漏过去了，只要有耐心，你就能看得懂。咱们接下来还是看config.c里的那个usb_parse_interface()。

```
158 static int usb_parse_interface(struct device *ddev, int cfgno,
159     struct usb_host_config *config, unsigned char *buffer, int size,
160     u8 inu ms[], u8 nalts[])
161 {
162      unsigned char *buffer0 = buffer;
163      struct usb_interface_descriptor *d;
164      int inum, asnum;
165      struct usb_interface_cache *intfc;
166      struct usb_host_interface *alt;
167      int i, n;
168      int len, retval;
169      int num_ep, num_ep_orig;
170
171      d = (struct usb_interface_descriptor *) buffer;
172      buffer += d->bLength;
173      size -= d->bLength;
174
175      if (d->bLength < USB_DT_INTERFACE_SIZE)
176          goto skip_to_next_interface_descriptor;
177
178      /* Which interface entry is this? */
179      intfc = NULL;
180      inum = d->bInterfaceNumber;
181      for (i = 0; i < config->desc.bNumInterfaces; ++i) {
182          if (inu ms[i] == inum) {
183              intfc = config->intf_cache[i];
184              break;
185          }
186      }
187      if (!intfc || intfc->num_altsetting >= nalts[i])
188          goto skip_to_next_interface_descriptor;
189
190      /* Check for duplicate altsetting entries */
191      asnum = d->bAlternateSetting;
192      for ((i = 0, alt = &intfc->altsetting[0]);
193          i < intfc->num_altsetting;
194            (++i, ++alt)) {
195          if (alt->desc.bAlternateSetting == asnum) {
196              dev_warn(ddev, "Duplicate descriptor for config %d "
197                         "interface %d altsetting %d, skipping\n",
198                         cfgno, inum, asnum);
199              goto skip_to_next_interface_descriptor;
200          }
201      }
202
203      ++intfc->num_altsetting;
204      memcpy(&alt->desc, d, USB_DT_INTERFACE_SIZE);
205
206      /* Skip over any Class Specific or Vendor Specific descriptors;
207       * find the first endpoint or interface descriptor */
208      alt->extra = buffer;
209      i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
210      USB_DT_INTERFACE, &n);
211      alt->extralen = i;
212      if (n > 0)
213          dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
214                  n, plural(n), "interface");
215      buffer += i;
216      size -= i;
```

```
217
218     /* Allocate space for the right(?) number of endpoints */
219     num_ep = num_ep_orig = alt->desc.bNumEndpoints;
220     alt->desc.bNumEndpoints = 0;           // Use as a counter
221     if (num_ep > USB_MAXENDPOINTS) {
222         dev_warn(ddev, "too many endpoints for config %d interface %d "
223                 "altsetting %d: %d, using maximum allowed: %d\n",
224                 cfgno, inum, asnum, num_ep, USB_MAXENDPOINTS);
225         num_ep = USB_MAXENDPOINTS;
226     }
227
228     if (num_ep > 0) {       /* Can't allocate 0 bytes */
229         len = sizeof(struct usb_host_endpoint) * num_ep;
230         alt->endpoint = kzalloc(len, GFP_KERNEL);
231         if (!alt->endpoint)
232             return -ENOMEM;
233     }
234
235     /* Parse all the endpoint descriptors */
236     n = 0;
237     while (size > 0) {
238         if (((struct usb_descriptor_header *) buffer)->bDescriptorType
239              == USB_DT_INTERFACE)
240             break;
241         retval = usb_parse_endpoint(ddev, cfgno, inum, asnum, alt,num_ep, buffer, size);
243         if (retval < 0)
244             return retval;
245         ++n;
247         buffer += retval;
248         size -= retval;
249     }
251     if (n != num_ep_orig)
252         dev_warn(ddev, "config %d interface %d altsetting %d has %d "
253                 "endpoint descriptor%s, different from the interface "
254                 "descriptor's value: %d\n",
255                 cfgno, inum, asnum, n, plural(n), num_ep_orig);
256     return buffer - buffer0;
258 skip_to_next_interface_descriptor:
259     i = find_next_descriptor(buffer, size, USB_DT_INTERFACE,
260         USB_DT_INTERFACE, NULL);
261     return buffer - buffer0 + i;
262 }
```

171行，传递过来的buffer里开头儿那部分只能是一个接口描述符，所以这里将地址转化为struct usb_interface_descriptor结构体指针，然后调整buffer的位置和size。

175行，“只能是”并不说明“它就是”，只有bLength等于USB_DT_INTERFACE_SIZE才说明开头儿的USB_DT_INTERFACE_SIZE字节确实是一个接口描述符。否则就没必要再对这些数据进行什么处理了，直接跳到最后吧。先看一看这个函数的最后都发生了什么，从新的位置开始再次调用find_next_descriptor()在buffer里寻找下一个接口描述符。

179行，因为数组inu ms并不一定是按照接口的顺序来保存接口号的，inu ms[1]对应的可能是接口1也可能是接口0。所以这里要用for循环来寻找这个接口对应着inu ms里的哪一项，从而根据在数组里的位置获得接口对应的struct usb_interface_cache结构体。usb_parse_configuration()已经告诉了我们，同一个接口在inu ms和intf_cache这两个数组里的位置是一样的。

191行，获得这个接口描述符对应的设置编号，然后根据这个编号从接口的cache里搜索看这个设置是不是已经遇到过了。如果设置已经遇到过，就没必要再对这个接口描述符进行处理，直接跳到最后。否则就意味着发现了一个新的设置，要将它添加到cache里，并cache里的设置数目num_altsetting加1。要记住，设置是用struct usb_host_interface结构来表示的，一个接口描述符就对应一个设置。

208行，这段代码很熟悉。现在buffer开头儿的接口描述符已经理清了，现在要解析它后面的那些数据了。先把位置赋给这个刚解析出来的接口描述符的extra，然后再从这个位置开始去寻找下一个距离最近的一个接口描述符或端点描述符。如果这个接口描述符后面还跟有class-或vendor-specific描述符，则find_next_descriptor的返回值会大于0，buffer的位置和size也要进行相应调整，来指向新找到的接口描述符或端点描述符。

这里find_next_descriptor的dt1参数和dt2参数就不再一样了，因为如果一个接口只用到端点0，它的接口描述符后边儿是不会跟有端点描述符的。

219行，获得这个设置使用的端点数目，然后将相应接口描述符里的bNumEndpoints置0，为什么？你要往下看。USB_MAXENDPOINTS在config.c中定义：

```
12 #define USB_MAXENDPOINTS                30      /* Hard limit */
```

为什么这个最大上限为30？前面提到过，请参考前面章节的讲解。然后根据端点数为接口描述符里的endpoint数组申请内存。

237行，走到这里，buffer开头儿的那个接口描述符已经理清了，而且也找到了下一个接口描述符或端点描述符的位置，该从这个新的位置开始解析了，于是又遇到了一个似曾相识的while循环。

238行，先判断一下前面找到的是接口描述符还是端点描述符，如果是接口描述符就中断这个while循环，返回与下一个接口描述符的距离。否则说明在buffer当前的位置上是一个端点描述符，因此就要迎来另一个函数usb_parse_endpoint对面紧接着的数据进行解析。usb_parse_endpoint()返回时，buffer的位置已经在下一个端点描述符里了，247行调整buffer的位置长度，这个while循环的也很明显了，对buffer一段一段的解析，直到遇到下一个接口描述符或者已经走到buffer结尾。现在看一看config.c中定义的usb_parse_endpoint函数。

```
46 static int usb_parse_endpoint(struct device *ddev, int cfgno, int inum,
47    int asnum, struct usb_host_interface *ifp, int num_ep,
48    unsigned char *buffer, int size)
49 {
50      unsigned char *buffer0 = buffer;
51      struct usb_endpoint_descriptor *d;
52      struct usb_host_endpoint *endpoint;
53      int n, i, j;
54
55      d = (struct usb_endpoint_descriptor *) buffer;
56      buffer += d->bLength;
57      size -= d->bLength;
58
59      if (d->bLength >= USB_DT_ENDPOINT_AUDIO_SIZE)
60          n = USB_DT_ENDPOINT_AUDIO_SIZE;
61      else if (d->bLength >= USB_DT_ENDPOINT_SIZE)
62          n = USB_DT_ENDPOINT_SIZE;
63      else {
64          dev_warn(ddev, "config %d interface %d altsetting %d has an "
65                 "invalid endpoint descriptor of length %d, skipping\n",
66                 cfgno, inum, asnum, d->bLength);
67          goto skip_to_next_endpoint_or_interface_descriptor;
68      }
69
70      i = d->bEndpointAddress & ~USB_ENDPOINT_DIR_MASK;
71      if (i >= 16 || i == 0) {
72          dev_warn(ddev, "config %d interface %d altsetting %d has an "
73                 "invalid endpoint with address 0x%X, skipping\n",
74                 cfgno, inum, asnum, d->bEndpointAddress);
75          goto skip_to_next_endpoint_or_interface_descriptor;
76      }
77
78      /* Only store as many endpoints as we have room for */
79      if (ifp->desc.bNumEndpoints >= num_ep)
80          goto skip_to_next_endpoint_or_interface_descriptor;
81
82      endpoint = &ifp->endpoint[ifp->desc.bNumEndpoints];
83      ++ifp->desc.bNumEndpoints;
84
85      memcpy(&endpoint->desc, d, n);
86      INIT_LIST_HEAD(&endpoint->urb_list);
87
88      /* If the bInterval value is outside the legal range,
89       * set it to a default value: 32  ms */
90      i = 0;          /* i = min, j = max, n = default */
91      j = 255;
92      if (usb_endpoint_xfer_int(d)) {
93          i = 1;
94          switch (to_usb_device(ddev)->speed) {
```

```
95          case USB_SPEED_HIGH:
96              n = 9;          /* 32  ms = 2^(9-1) uframes */
97              j = 16;
98              break;
99          default:                /* USB_SPEED_FULL or _LOW */
100             /* For low-speed, 10  ms is the official minimum.
101              * But some "overclocked" devices might want faster
102              * polling so we'll allow it. */
103             n = 32;
104             break;
105         }
106     } else if (usb_endpoint_xfer_isoc(d)) {
107         i = 1;
108         j = 16;
109         switch (to_usb_device(ddev)->speed) {
110         case USB_SPEED_HIGH:
111             n = 9;          /* 32  ms = 2^(9-1) uframes */
112             break;
113         default:                /* USB_SPEED_FULL */
114             n = 6;          /* 32  ms = 2^(6-1) frames */
115             break;
116         }
117     }
118     if (d->bInterval < i || d->bInterval > j) {
119         dev_warn(ddev, "config %d interface %d altsetting %d "
120                 "endpoint 0x%X has an invalid bInterval %d, "
121                 "changing to %d\n",
122                 cfgno, inum, asnum,
123                 d->bEndpointAddress, d->bInterval, n);
124         endpoint->desc.bInterval = n;
125     }
126
127     /* Skip over any Class Specific or Vendor Specific descriptors;
128      * find the next endpoint or interface descriptor */
129     endpoint->extra = buffer;
130     i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
131           USB_DT_INTERFACE, &n);
132     endpoint->extralen = i;
133     if (n > 0)
134         dev_dbg(ddev, "skipped %d descriptor%s after %s\n",
135                 n, plural(n), "endpoint");
136     return buffer - buffer0 + i;
137
138 skip_to_next_endpoint_or_interface_descriptor:
139     i = find_next_descriptor(buffer, size, USB_DT_ENDPOINT,
140           USB_DT_INTERFACE, NULL);
141     return buffer - buffer0 + i;
142 }
```

55行，buffer开头儿只能是一个端点描述符，所以这里将地址转化为struct usb_endpoint_descriptor结构体指针，然后调整buffer的位置和size。

59行，这里要明白的是端点描述符与配置描述符、接口描述符不一样，它是可能有两种大小的。

70行，得到端点号。这里的端点号不能为0，因为端点0是没有描述符的，也不能大于16。

79行，要知道这个bNumEndpoints在usb_parse_interface()的220行是被赋为0了的。

82行，要知道这个endpoint数组在usb_parse_interface()的230行也是已经申请好内存了的。从这里你应该明白bNumEndpoints是被当成了一个计数器，发现一个端点描述符，它就加1，并把找到的端点描述符copy到设置的endpoint数组里。

86行，初始化端点的urb队列urb_list。

88行~125行，这堆代码的目的是处理端点的bInterval，你要想不被它们给忽悠了，得明白几个问题。第一个就是，*i*，*j*，*n*分别表示什么。90行~117行这么多行就为了给它们选择一个合适的值，*i*和*j*限定了bInterval的一个范围，bInterval如果在这里边儿，像124行做的那样将*n*赋给它，那么*n*表示的就是bInterval的一个默认值。*i*和*j*的默认值分别为0和255，也就是说合法的范围默认是0~255。对于批量端点和控制端点，bInterval对你我来说并没有太大的用处，不过协议中还是规定了，这个范围只能为0~255。对于中断端点和等时端点，bInterval表演的舞台就很大了，对这个范围也要做一些调整。

第二个问题就是如何判断端点是中断的还是等时的。这涉及两个函数usb_endpoint_xfer_int和usb_endpoint_xfer_isoc，它们都在include/linux/usb.h中定义。

```
596 static inline int usb_endpoint_xfer_int(const struct usb_endpoint_descriptor *epd)
597 {
598     return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
599             USB_ENDPOINT_XFER_INT);
600 }
601
609 static inline int usb_endpoint_xfer_isoc(const struct usb_endpoint_descriptor *epd)
610 {
611     return ((epd->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) ==
612             USB_ENDPOINT_XFER_ISOC);
613 }
```

这两个函数的意思简单明了，另外两个函数就是usb_endpoint_xfer_bulk和usb_endpoint_xfer_control，用来判断批量端点和控制端点的。

第三个问题是to_usb_device。usb_parse_endpoint()的参数是struct device结构体，要获得设备的速度就需要使用to_usb_device将它转化为struct usb_device结构体，这是一个include/linux/usb.h中定义的宏：

```
410 #define to_usb_device(d) container_of(d, struct usb_device, dev)
```

接着看usb_parse_endpoint的129行，现在你对这几行的意思明白了。这里接着在buffer里寻找下一个端点描述符或者接口描述符。

经过usb_parse_configuration、usb_parse_interface和usb_parse_endpoint这三个函数一步一营的层层推进，通过GET_DESCRIPTOR请求所获得那堆数据现在已经解析地清清白白。现在，设备的各个配置信息已经了然于胸，那接下来设备的那条生命线该怎么去走？它已经可以进入Configured状态了吗？事情没这么简单，光是获得设备各个配置信息没用，要进入Configured状态，你还得有选择、有目的、有步骤、有计划地去配置设备，那怎么去有选择、有目的、有步骤、有计划？这好像就不是core能够答复的问题了，毕竟它并不知道你希望设备采用哪种配置，只有设备的驱动才知道，所以接下来设备要做的是去在设备模型中寻找属于自己的驱动。

华清远见

不要忘记设备的struct usb_device结构体在出生时就带有usb_bus_type和usb_device_type这样的胎记，Linux设备模型根据总线类型usb_bus_type将设备添加到USB总线的那条有名的设备链表里，然后去轮询USB总线的另外一条有名的驱动链表，针对每个找到的驱动去调用USB总线的match函数，也就是usb_device_match()，去为设备寻找另一个匹配的驱动。match函数会根据设备的自身条件和类型usb_device_type安排设备走设备那条路，从而匹配到那个对所有usb_device_type类型的设备驱动，USB世界里唯一的那个USB设备驱动（不是USB接口驱动）struct device_driver结构体对象usb_generic_driver。

21、驱动的生命线

驱动的生命线（一）

usb_generic_driver是一个USB设备都会拜倒在她的石榴裙下，争先恐后地找配对儿。

现在开始就沿着usb_generic_driver的成名之路走一走，设备的生命线你可以想当然地认为是从你的USB设备连接到Hub的某个端口时开始，驱动的生命线就必须得回溯到USB子系统的初始化函数usb_init了。

```
895  retval = usb_register_device_driver(&usb_generic_driver, THIS_MODULE);
896  if (!retval)
897      goto out;
```

在USB子系统初始化时就调用driver.c里的usb_register_device_driver函数将usb_generic_driver注册给系统了，下面来看一看。

```
671 int usb_register_device_driver(struct usb_device_driver *new_udriver,
672             struct module *owner)
```

```
673 {
674     int retval = 0;
675
676     if (usb_disabled())
677         return -ENODEV;
678
679     new_udriver->drvwrap.for_devices = 1;
680     new_udriver->drvwrap.driver.name = (char *) new_udriver->name;
681     new_udriver->drvwrap.driver.bus = &usb_bus_type;
682     new_udriver->drvwrap.driver.probe = usb_probe_device;
683     new_udriver->drvwrap.driver.remove = usb_unbind_device;
684     new_udriver->drvwrap.driver.owner = owner;
685
686     retval = driver_register(&new_udriver->drvwrap.driver);
687
688     if (!retval) {
689         pr_info("%s: registered new device driver %s\n",
690                 usbcore_name, new_udriver->name);
691         usbfs_update_special();
692     } else {
693         printk(KERN_ERR "%s: error %d registering device "
694                 "       driver %s\n",
695                 usbcore_name, retval, new_udriver->name);
696     }
697
698     return retval;
699 }
```

676行，判断一下USB子系统是不是在你启动内核时就被禁止了，如果是的话，它的生命也就太短暂了。

679行，for_devices就是在这儿被初始化为1的，有了它，match里的is_usb_device_driver才有章可循，有凭可依。

下面就是充实了一下usb_generic_driver里嵌入的struct device_driver结构体，usb_generic_driver就是通过它和设备模型搭上关系的。name就是usb_generic_driver的名字，即usb，所属的总线类型同样被设置为usb_bus_type，然后是指定probe函数和remove函数。

686行，调用设备模型的函数driver_register将usb_generic_driver添加到USB总线的那条驱动链表里。

usb_generic_driver和USB设备匹配成功后，就会调用682行指定的probe函数usb_probe_device()，现在看一看driver.c中定义的这个函数。

```
152 static int usb_probe_device(struct device *dev)
153 {
154     struct usb_device_driver *udriver =
                to_usb_device_driver(dev->driver);
155     struct usb_device *udev;
156     int error = -ENODEV;
157
158     dev_dbg(dev, "%s\n", __FUNCTION__);
159
160     if (!is_usb_device(dev))        /* Sanity check */
161         return error;
162
163     udev = to_usb_device(dev);
164
165     /* TODO: Add real matching code */
166
167     /* The device should always appear to be in use
168      * unless the driver suports autosuspend.
169      */
170     udev->pm_usage_cnt = !(udriver->supports_autosuspend);
171
172     error = udriver->probe(udev);
173     return error;
174 }
```

154行，to_usb_device_driver是include/linux/usb.h中定义的一个宏，和前面遇到的那个to_usb_device有异曲同工之妙，

```
889 #define to_usb_device_driver(d) container_of(d, struct usb_device_driver, \
```

```
890                     drvwrap.driver)
```

160行，match函数543行的函数又调到这儿来“把门儿”了，如果你这个设备的类型不是usb_device_type，那怎么从前面走到这儿的它管不着，但是到它这里就不可能再蒙混过关了。你的设备虽然和usb_generic_driver是match成功了，但是还要经过判断。

163行，前面刚提到to_usb_device就到这来了。

170行，pm_usage_cnt和supports_autosuspend这两个前面都提到过一下。每个struct usb_interface或struct usb_device里都有一个pm_usage_cnt，每个struct usb_driver或struct usb_device_driver里都有一个supports_autosuspend。只有pm_usage_cnt为0时才会允许接口autosuspend，如果supports_autosuspen为0就不再允许绑定到这个驱动的接口autosuspend。

接口？设备？需要时，接口是接口设备是设备；不需要时，接口设备一个样。这里就是不需要时，所以将上面的话里的接口换成设备套一下就是：pm_usage_cnt为0时才会允许设备autosuspend，supports_autosuspend为0，就不再允许绑定到这个驱动的设备autosuspend。

所有的USB设备都是绑定到usb_generic_driver上面的，usb_generic_driver的supports_autosuspend字段又是为1的，所以这行就是将设备struct usb_device结构体的pm_usage_cnt置为了0，也就是说允许设备autosuspend。但是不是说将pm_usage_cnt轻轻松松置为0，设备就能够autosuspend了，驱动必须得实现一对suspend/resume函数供PM子系统驱使，usb_generic_driver里的这对函数就是generic_suspend/generic_resume。

172行，这里要调用usb_generic_driver自己私有的probe函数generic_probe()对你的设备进行进一步的审查，它在generic.c中定义。

```
153 static int generic_probe(struct usb_device *udev)
154 {
155     int err, c;
156
157     /* put device-specific files into sysfs */
158     usb_create_sysfs_dev_files(udev);
159
160     /* Choose and set the configuration.  This registers the interfaces
161      * with the driver core and lets interface drivers bind to them.
162      */
163     c = choose_configuration(udev);
164     if (c >= 0) {
165         err = usb_set_configuration(udev, c);
166         if (err) {
167             dev_err(&udev->dev, "can't set config #%d, error %d\n",
168                                     c, err);
169             /* This need not be fatal.  The user can try to
170              * set other configurations. */
171         }
172     }
173
174     /* USB device state == configured ... usable */
175     usb_notify_add_device(udev);
176
177     return 0;
178 }
```

这函数说简单也简单，说复杂也复杂，简单的是外表，复杂的是内心。用一句话去概括它的中心思想，就是从设备众多配置中选择一个合适的，然后去配置设备，从而让设备进入期待已久的Configured状态。先看一看是怎么选择一个配置的，调用的是generic.c里的choose_configuration函数。

```
42 static int choose_configuration(struct usb_device *udev)
43 {
44      int i;
45      int num_configs;
46      int insufficient_power = 0;
47      struct usb_host_config *c, *best;
48
49      best = NULL;
50      c = udev->config;
51      num_configs = udev->descriptor.bNumConfigurations;
```

```
52       for (i = 0; i < num_configs; (i++, c++)) {
53           struct usb_interface_descriptor *desc = NULL;
54
55           /* It's possible that a config has no interfaces! */
56           if (c->desc.bNumInterfaces > 0)
57               desc = &c->intf_cache[0]->altsetting->desc;
58
59           /*
60            * HP's USB bus-powered keyboard has only one configuration
61            * and it clai ms to be self-powered; other devices may have
62            * similar errors in their descriptors.  If the next test
63            * were allowed to execute, such configurations would always
64            * be rejected and the devices would not work as expected.
65            * In the meantime, we run the risk of selecting a config
66            * that requires external power at a time when that power
67            * isn't available.  It see ms to be the lesser of two evils.
68            *
69            * Bugzilla #6448 reports a device that appears to crash
70            * when it receives a GET_DEVICE_STATUS request!  We don't
71            * have any other way to tell whether a device is self-powered,
72            * but since we don't use that information anywhere but here,
73            * the call has been removed.
74            *
75            * Maybe the GET_DEVICE_STATUS call and the test below can
76            * be reinstated when device firmwares become more reliable.
77            * Don't hold your breath.
78            */
79 #if 0
80           /* Rule out self-powered configs for a bus-powered device */
81           if (bus_powered && (c->desc.bmAttributes &
82                                   USB_CONFIG_ATT_SELFPOWER))
83               continue;
84 #endif
85
86           /*
87            * The next test may not be as effective as it should be.
88            * Some hubs have errors in their descriptor, claiming
89            * to be self-powered when they are really bus-powered.
90            * We will overestimate the amount of current such hubs
91            * make available for each port.
92            *
93            * This is a fairly benign sort of failure.  It won't
94            * cause us to reject configurations that we should have
95            * accepted.
96            */
97
98           /* Rule out configs that draw too much bus current */
99           if (c->desc.bMaxPower * 2 > udev->bus_mA) {
100              insufficient_power++;
101              continue;
102          }
103
104      /* When the first config's first interface is one of Microsoft's
105           * pet nonstandard Ethernet-over-USB protocols, ignore it unless
106           * this kernel has enabled the necessary host side driver.
107           */
108          if (i == 0 && desc && (is_rndis(desc)||is_activesync(desc))) {
109 #if !defined(CONFIG_USB_NET_RNDIS_HOST) && !defined(CONFIG_USB_NET_RNDIS_HOST_MODULE)
110              continue;
111 #else
112              best = c;
113 #endif
114      }
115
116          /* From the remaining configs, choose the first one whose
117           * first interface is for a non-vendor-specific class.
118           * Reason: Linux is more likely to have a class driver
119           * than a vendor-specific driver. */
120          else if (udev->descriptor.bDeviceClass !=
121                                  USB_CLASS_VENDOR_SPEC &&
122                          (!desc || desc->bInterfaceClass !=
123                                  USB_CLASS_VENDOR_SPEC)) {
124              best = c;
125              break;
126          }
```

```
127
128             /* If all the remaining configs are vendor-specific,
129              * choose the first one. */
130             else if (!best)
131                 best = c;
132         }
133
134         if (insufficient_power > 0)
135             dev_info(&udev->dev, "rejected %d configuration%s "
136                         "due to insufficient available bus power\n",
137                         insufficient_power, plural(insufficient_power));
138
139         if (best) {
140             i = best->desc.bConfigurationValue;
141             dev_info(&udev->dev,
142                         "configuration #%d chosen from %d choice%s\n",
143                         i, num_configs, plural(num_configs));
144         } else {
145             i = -1;
146             dev_warn(&udev->dev,
147                         "no configuration chosen from %d choice%s\n",
148                         num_configs, plural(num_configs));
149         }
150         return i;
151 }
```

设备各个配置的详细信息在设备自身的漫漫人生旅途中就已经获取存放在相关的几个成员里了，怎么从中挑选一个让人满意的？显然谁都会说去一个一个地浏览每个配置，查看有没有称心如意的，于是就有了52行的for循环。

刚看到这个for循环就有点傻眼了，居然注释要远远多于代码，这么一个for循环，我们把它分成三大段，59行~84行这一段，你什么都可以不看，就是不能不看那个#if 0，一见到它，就意味着你可以略过这么一大段代码了。

第二段是98行~102行，这一段牵扯到人世间最让人无可奈何的一对矛盾，索取与给予。一个配置索取的电流比Hub所能给予的还要大，显然它不会是一个让人满意的配置。

第三段是108行~131行，关于这段只说一个原则，Linux更喜欢那些标准的东西，比如USB_CLASS_VIDEO、USB_CLASS_AUDIO等这样子的设备和接口就更讨人喜欢一些，所以就会优先选择非USB_CLASS_VENDOR_SPEC的接口。

for循环之后，剩下的那些部分都是调试用的，输出一些调试信息，不需要去关心，不过里面出现了个有趣的函数plural，它是一个在generic.c开头儿定义的内联函数。

```
23 static inline const char *plural(int n)
24 {
25         return (n == 1 ? "" : "s");
26 }
```

参数*n*为1返回一个空字符串，否则返回一个‘s’，瞄一下使用了这个函数的打印语句，就明白它是用来打印一个英语名词的单复数的，复数的话就加上一个“s”。

不管你疑惑也好，满意也好，choose_configuration就是这样按照自己的标准挑选了一个比较合自己心意的配置，接下来当然就是要用这个配置去配置设备以便让它迈进Configured状态了。

驱动的生命线（二）

core配置设备使用的是message.c里的usb_set_configuration函数。

```
1430 int usb_set_configuration(struct usb_device *dev, int configuration)
1431 {
1432         int i, ret;
1433         struct usb_host_config *cp = NULL;
1434         struct usb_interface **new_interfaces = NULL;
1435         int n, nintf;
1436
1437         if (configuration == -1)
1438             configuration = 0;
```

```
1439     else {
1440         for (i = 0; i < dev->descriptor.bNumConfigurations; i++) {
1441             if (dev->config[i].desc.bConfigurationValue ==
1442                                     configuration) {
1443                 cp = &dev->config[i];
1444                 break;
1445             }
1446         }
1447     }
1448     if ((!cp && configuration != 0))
1449         return -EINVAL;
1450 
1451     /* The USB spec says configuration 0 means unconfigured.
1452      * But if a device includes a configuration numbered 0,
1453      * we will accept it as a correctly configured state.
1454      * Use -1 if you really want to unconfigure the device.
1455      */
1456     if (cp && configuration == 0)
1457         dev_warn(&dev->dev, "config 0 descriptor??\n");
1458 
1459     /* Allocate memory for new interfaces before doing anything else,
1460      * so that if we run out then nothing will have changed. */
1461     n = nintf = 0;
1462     if (cp) {
1463         nintf = cp->desc.bNumInterfaces;
1464         new_interfaces = kmalloc(nintf * sizeof(*new_interfaces),
1465                                 GFP_KERNEL);
1466         if (!new_interfaces) {
1467             dev_err(&dev->dev, "Out of memory");
1468             return -ENOMEM;
1469         }
1470 
1471         for (; n < nintf; ++n) {
1472             new_interfaces[n] = kzalloc(
1473                                     sizeof(struct usb_interface),
1474                                     GFP_KERNEL);
1475             if (!new_interfaces[n]) {
1476                 dev_err(&dev->dev, "Out of memory");
1477                 ret = -ENOMEM;
1478 free_interfaces:
1479                 while (--n >= 0)
1480                     kfree(new_interfaces[n]);
1481                 kfree(new_interfaces);
1482                 return ret;
1483             }
1484         }
1485 
1486         i = dev->bus_mA - cp->desc.bMaxPower * 2;
1487         if (i < 0)
1488             dev_warn(&dev->dev, "new config #%d exceeds power "
1489                                     "limit by %dmA\n",
1490                                     configuration, -i);
1491     }
1492 
1493     /* Wake up the device so we can send it the Set-Config request */
1494     ret = usb_autoresume_device(dev);
1495     if (ret)
1496         goto free_interfaces;
1497 
1498     /* if it's already configured, clear out old state first.
1499      * getting rid of old interfaces means unbinding their drivers.
1500      */
1501     if (dev->state != USB_STATE_ADDRESS)
1502         usb_disable_device (dev, 1);    // Skip ep0
1503 
1504     if ((ret = usb_control_ msg(dev, usb_sndctrlpipe(dev, 0),
1505                     USB_REQ_SET_CONFIGURATION, 0, configuration, 0,
1506                     NULL, 0, USB_CTRL_SET_TIMEOUT)) < 0) {
1507 
1508         /* All the old state is gone, so what else can we do?
1509          * The device is probably useless now anyway.
1510          */
1511         cp = NULL;
1512     }
1513 
```

```
1514      dev->actconfig = cp;
1515      if (!cp) {
1516          usb_set_device_state(dev, USB_STATE_ADDRESS);
1517          usb_autosuspend_device(dev);
1518          goto free_interfaces;
1519      }
1520      usb_set_device_state(dev, USB_STATE_CONFIGURED);
1521
1522      /* Initialize the new interface structures and the
1523       * hc/hcd/usbcore interface/endpoint state.
1524       */
1525      for (i = 0; i < nintf; ++i) {
1526          struct usb_interface_cache *intfc;
1527          struct usb_interface *intf;
1528          struct usb_host_interface *alt;
1529
1530          cp->interface[i] = intf = new_interfaces[i];
1531          intfc = cp->intf_cache[i];
1532          intf->altsetting = intfc->altsetting;
1533          intf->num_altsetting = intfc->num_altsetting;
1534          kref_get(&intfc->ref);
1535
1536          alt = usb_altnum_to_altsetting(intf, 0);
1537
1538          /* No altsetting 0?  We'll assume the first altsetting.
1539           * We could use a GetInterface call, but if a device is
1540           * so non-compliant that it doesn't have altsetting 0
1541           * then I wouldn't trust its reply anyway.
1542           */
1543          if (!alt)
1544              alt = &intf->altsetting[0];
1545
1546          intf->cur_altsetting = alt;
1547          usb_enable_interface(dev, intf);
1548          intf->dev.parent = &dev->dev;
1549          intf->dev.driver = NULL;
1550          intf->dev.bus = &usb_bus_type;
1551          intf->dev.type = &usb_if_device_type;
1552          intf->dev.dma_mask = dev->dev.dma_mask;
1553          device_initialize (&intf->dev);
1554          mark_quiesced(intf);
1555          sprintf (&intf->dev.bus_id[0], "%d-%s:%d.%d",
1556                      dev->bus->busnum, dev->devpath,
1557                      configuration, alt->desc.bInterfaceNumber);
1558      }
1559      kfree(new_interfaces);
1560
1561      if (cp->string == NULL)
1562          cp->string = usb_cache_string(dev, cp->desc.iConfiguration);
1563
1564      /* Now that all the interfaces are set up, register them
1565       * to trigger binding of drivers to interfaces.  probe()
1566       * routines may install different altsettings and may
1567       * claim() any interfaces not yet bound.  Many class drivers
1568       * need that: CDC, audio, video, etc.
1569       */
1570      for (i = 0; i < nintf; ++i) {
1571          struct usb_interface *intf = cp->interface[i];
1572
1573          dev_dbg (&dev->dev,
1574                      "adding %s (config #%d, interface %d)\n",
1575                      intf->dev.bus_id, configuration,
1576                      intf->cur_altsetting->desc.bInterfaceNumber);
1577          ret = device_add (&intf->dev);
1578      if (ret != 0) {
1579              dev_err(&dev->dev, "device_add(%s) --> %d\n",
1580                          intf->dev.bus_id, ret);
1581              continue;
1582          }
1583          usb_create_sysfs_intf_files (intf);
1584      }
1585
1586      usb_autosuspend_device(dev);
1587      return 0;
1588  }
```

这个函数可以分成三个部分，从1432行到1491行的这几十行是准备阶段，做常规检查，申请申请内存。1498行到1520行这部分可是重头戏，就是在这里设备从Address发展到了Configured。1522行到1584行这阶段也挺重要的，主要就是充实设备的每个接口并提交给设备模型，为它们寻找命中注定的接口驱动，过了这个阶段，usb_generic_driver也就彻底从设备那儿得到满足了，generic_probe的历史使命也就完成了。

先看第一阶段，1437行，configuration是前边儿choose_configuration()那里返回回来的，找到合意的配置的话，就返回那个配置的bConfigurationValue值，没有找到称心的配置的话，就返回-1，所以这里的configuration值就可能有两种情况，或者为-1，或者为配置的bConfigurationValue值。

当configuration为-1时这里为什么又要把它改为0呢？要知道configuration这个值是要在后面的高潮阶段里发送SET_CONFIGURATION请求时用的，关于SET_CONFIGURATION请求，spec里说，这个值必须为0或者与配置描述符的bConfigurationValue一致，如果为0，则设备收到SET_CONFIGURATION请求后，仍然会待在Address状态。这里当configuration为-1也就是没有发现满意的配置时，设备不能进入Configured，所以要把configuration的值改为0，以便满足SET_CONFIGURATION请求的要求。

那接下来的问题就出来了，在没有找到合适配置时直接给configuration这个参数设置为0，也就是让choose_configuration()返回个0不就得了，干吗还这么麻烦先返回个-1再把它改成0？有些设备就是有拿0当配置bConfigurationValue值的毛病，你又不能不让它用。想让设备回到Address状态时，usb_set_configuration()就别传递0了，传递个-1，里边儿去处理一下。如果configuration值为0或大于0的值，就从设备struct usb_device结构体的config数组里将相应配置的描述信息，也就是struct usb_host_config结构体给取出来。

1448行，如果没有拿到配置的内容，configuration值就必须为0了，让设备待在Address那儿别动。这也很好理解，配置的内容都找不到了，还配置什么。当然，如果拿到了配置的内容，但同时configuration为0，这就是对应了上面说的那种有毛病的设备的情况，就提出警告出现不正常现象了。

1461行，过了这个if函数，第一阶段就告结束了。如果配置是实实在在存在的，就为它使用的接口都准备一个struct usb_interface结构体。new_interfaces是开头儿就定义好的一个struct usb_interface结构体指针数组，数组的每一项都指向了一个struct usb_interface结构体，所以这里申请内存也要分两步走，先申请指针数组的，再申请每一项的。

驱动的生命线（三）

现在查看第二阶段的重头戏，查看设备是怎么从Address进入Configured的。1501行，如果已经在Configured状态了，退回到Address状态。仔细研究一下message.c里的usb_disable_device函数。

```
1034 void usb_disable_device(struct usb_device *dev, int skip_ep0)
1035 {
1036     int i;
1037 
1038     dev_dbg(&dev->dev, "%s nuking %s URBs\n", __FUNCTION__,
1039                 skip_ep0 ? "non-ep0" : "all");
1040     for (i = skip_ep0; i < 16; ++i) {
1041         usb_disable_endpoint(dev, i);
1042         usb_disable_endpoint(dev, i + USB_DIR_IN);
1043     }
1044     dev->toggle[0] = dev->toggle[1] = 0;
1045 
1046     /* getting rid of interfaces will disconnect
1047      * any drivers bound to them (a key side effect)
1048      */
1049     if (dev->actconfig) {
1050         for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1051             struct usb_interface    *interface;
1052 
1053             /* remove this interface if it has been registered */
1054             interface = dev->actconfig->interface[i];
1055             if (!device_is_registered(&interface->dev))
1056                 continue;
1057         dev_dbg (&dev->dev, "unregistering interface %s\n",
```

```
1058                            interface->dev.bus_id);
1059                usb_remove_sysfs_intf_files(interface);
1060                device_del (&interface->dev);
1061            }
1063            /* Now that the interfaces are unbound, nobody should
1064             * try to access them.
1065             */
1066            for (i = 0; i < dev->actconfig->desc.bNumInterfaces; i++) {
1067                put_device (&dev->actconfig->interface[i]->dev);
1068                dev->actconfig->interface[i] = NULL;
1069            }
1070            dev->actconfig = NULL;
1071            if (dev->state == USB_STATE_CONFIGURED)
1072                usb_set_device_state(dev, USB_STATE_ADDRESS);
1073        }
1074 }
```

经过研究我们可以发现，usb_disable_device函数的工作主要有两部分，一是将设备中所有端点给disable掉，另一个是将设备当前配置使用的每个接口都从系统里给unregister掉，也就是将接口和它对应的驱动给分开。

先说一下第二部分的工作，1409行，actconfig表示设备当前激活的配置，只有它不为空时才有接下来清理的必要。

1050行~1061行这个for循环就是将这个配置的每个接口从设备模型的体系中删除掉，将它们和对应的接口驱动分开，没有驱动了，这些接口也就丧失了能力，当然也就什么作用都发挥不了了，这也是名字里那个disable的真正含意所在。

1066行~1070行，将actconfig的interface数组置为空，然后再将actconfig置为空，这里你可能会有的一个问题是，为什么只是置为空，既然要清理actconfig，为什么不直接将它占用的内存给释放掉？你应该注意到actconfig只是一个指针，一个地址，你应该首先弄清楚这个地址里保存的是什么东西再决定是不是将它给释放掉，那这个指针指向哪儿？它指向设备struct usb_device结构的config数组里的其中一项，当前被激活的是哪一个配置，它就指向config数组里的哪一项。你这里只是不想让设备当前激活任何一个配置而已，没必要将actconfig指向的那个配置给释放掉。

那这么说的话另一个问题就出来了，既然actconfig指向了config里的一项，那为什么要把interface数组给置为空，这不是修改了配置的内容，从而也修改了config数组的内容吗？你先别着急，在设备生命线那里取配置描述符，解析返回的那堆数据时，只是把每个配置里的cache数组，也就是intf_cache数组初始化了，并没有为interface数组充实任何的内容，这里做清理工作的目的就是要恢复原状，当然要将它置为空了。那么配置的interface数组又在哪里被充实了呢？usb_set_configuration函数中第二个高潮阶段之后不是还有个第三个阶段呢，就在那里，激活了哪个配置，就为哪个配置的interface数组，填了点儿东西。

1071行，如果这个设备此时确实是在Configured状态，就让它回到Address。

现在回头来说说第一部分的清理工作。这个部分主要就是为每个端点调用了usb_disable_endpoint函数，将挂在它们上面的urb给取消掉。为什么要这么做？你想想，能调用到usb_disable_device这个函数，一般来说设备的状态要发生变化了，设备的状态都改变了，那设备的端点状态要不要改变？还有挂在它们上面的那些urb需不需要给取消掉？这些都是很显然的事情，就拿现在让设备从Configured回到Address来说吧，在Address时，你只能通过默认管道也就是端点0对应的管道与设备进行通信的，但是在Configured时，设备的所有端点都是能够使用的，它们上面可能已经挂了一些urb正在处理或者将要处理，那么这时让设备要从Configured变到Address，是不是应该先将这些urb给取消掉？

参数skip_ep0是什么意思？这里for循环的i是从skip_ep0开始算起，也就是说skip_ep0为1的话，就不需要对端点0调用usb_disable_endpoint函数了，按常理来说，设备状态改变了，是需要把每个端点上面的urb取消掉的，这里面当然也要包括端点0，但是写代码的人在这里搞出一个skip_ep0自然有他们的玄机，usb_set_configuration()调用这个函数时参数skip_ep0的值是什么？是1，因为这时候是从Configured回到Address，这个过程中，其他端点是从能够使用变成了不能使用，但端点0却是一直都很强势，虽说

是设备发生了状态的变化，但在这两个状态里它都是要正常使用的，所以就没必要disable它了。

什么时候需要disable端点0？在目前版本的内核中我只发现了两种情况，一种是设备要断开时，另一种是设备从Default进化到Address时。虽说不管是Default还是Address，端点0都是需要能够正常使用的，但因为地址发生改变了，毫无疑问，你需要将挂在它上面的urb清除掉。当时讲设备生命线时，在设置完设备地址，设备进入Address后，第二种情况的这个步骤给忽略了，主要是当时也不影响理解，现在既然遇到了，就讲一讲吧。

在设备生命线的过程中，设置完设备地址，让设备进入Address状态后，立马就调用了hub.c里一个名叫ep0_reinit的函数。

```
2066 static void ep0_reinit(struct usb_device *udev)
2067 {
2068     usb_disable_endpoint(udev, 0 + USB_DIR_IN);
2069     usb_disable_endpoint(udev, 0 + USB_DIR_OUT);
2070     udev->ep_in[0] = udev->ep_out[0] = &udev->ep0;
2071 }
```

这个函数中只对端点0调用了usb_disable_endpoint()，但是端点0接下来还是要使用的，不然你就取不到设备那些描述符了，所以接着重新将ep0赋给ep_in[0]和ep_out[0]。多说无益，还是到usb_disable_endpoint()里面去查看吧。

```
987 void usb_disable_endpoint(struct usb_device *dev, unsigned int epaddr)
988 {
989     unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
990     struct usb_host_endpoint *ep;
991 
992     if (!dev)
993         return;
994 
995     if (usb_endpoint_out(epaddr)) {
996         ep = dev->ep_out[epnum];
997         dev->ep_out[epnum] = NULL;
998     } else {
999         ep = dev->ep_in[epnum];
1000        dev->ep_in[epnum] = NULL;
1001    }
1002    if (ep && dev->bus)
1003        usb_hcd_endpoint_disable(dev, ep);
1004 }
```

这个函数先获得端点号和端点的方向，然后从ep_in或ep_out两个数组里取出端点的struct usb_host_endpoint结构体，并将数组里的对应项置为空，要注意的是这里同样不是释放掉数组里对应项的内存而是置为空。这两个数组里的ep_in[0]和ep_out[0]是早就被赋值了，至于剩下的那些项是在什么时候被赋值的，又是指向了什么东西，就是usb_set_configuration函数第三个阶段的事了。

最后1003行调用了一个usb_hcd_endpoint_disable函数，主要的工作还得它来做，不过这已经深入HCD的腹地了，就不多说了，还是飘回usb_disable_device()吧。在为每个端点都调用了usb_disable_endpoint()之后，还有一个小步骤要做，就是将设备struct usb_device结构体的toggle数组置为0。至于toggle数组有什么用，为什么要被初始化为0，还是回首到设备那节去看吧。我要直接讲usb_set_configuration()了。

1504行，又一次与usb_control_ msg()相遇了，每当我们需要向设备发送请求时它就会适时出现。

usb_control_ msg这次出现的目的当然是为了SET_CONFIGURATION请求，这里只说一下它的那堆参数，看一下如图10所示的这张表

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET CONFIGURATION	配置值 Configuration Value	0	0	NULL

图 10 SET_CONFIGURATION 请求

SET_CONFIGURATION请求不需要DATA transaction，而且还是协议中规定所有设备都要支持的标准请求，也不是针对端点或者接口，而是针对设备的，所以bRequestType只能为0x80，就是上面表里的00000000B，也就是1505行的第一个0。wValue表示配置的bConfigurationValue，就是1505行的configuration。

1514行，将激活的配置的地址赋给actconfig。如果cp为空，重新设置设备的状态为Address，并将之前准备的那些struct usb_interface结构体和new_interfaces释放掉，然后返回。扫一下前面的代码，cp有三种可能为空，一种是参数configuration为-1，一种是参数configuration为0，而且从设备的config数组里拿出来的就为空，还有一种是SET_CONFIGURATION请求除了问题。不管怎么说，走到1515行，cp还是空的，你就要准备返回了。

1520行，事情在这里发展达到了高潮的顶端，设置设备的状态为Configured。

驱动的生命线（四）

设备自从有了Address，拿到了各种描述符，就在那儿看usb_generic_driver忙活了，不过还算没白忙，设备总算是幸福的进入“Configured”了。从设备这儿咱们应该学到点幸福生活的秘诀，就是找到你所喜欢的事，然后找到愿意为你来做这件事的人。

Address有点像你合几代人之力辛辛苦苦才弄到的一套新房子，如果不装修，它对你来说的意义可以对人说你的地址是哪儿了，可实际上你在里边儿什么也干不了，你还得想办法去装修它，Configured就像是已经装修好的房子，下面咱就看一看usb_generic_driver又对设备做了些什么。

1525行，nintf就是配置里接口的数目，那么这个for循环显然就是在对配置里的每个接口做处理。

1530行，这里要明白的是，cp里的两个数组interface和intf_cache，一个是没有初始化的，一个是已经动过手术很饱满的。正像前面提到的，这里需要为interface动手术，拿intf_cache的内容去充实它。

1536行，获得这个接口的0号设置。因为某些厂商有特殊的癖好，导致了struct usb_host_config结构中的数组interface并不一定是按照接口号的顺序存储的，必须使用usb_ifnum_to_if()来获得指定接口号对应的struct usb_interface结构体。现在咱们需要再多知道一点，接口里的altsetting数组也不一定是按照设置编号来顺序存储的，必须使用usb_altnum_to_altsetting()来获得接口里的指定设置。它在usb.c中定义。

```
118 struct usb_host_interface *usb_altnum_to_altsetting(const struct usb_interface *intf,
119                                                     unsigned int altnum)
120 {
121     int i;
122
123     for (i = 0; i < intf->num_altsetting; i++) {
124         if (intf->altsetting[i].desc.bAlternateSetting == altnum)
125             return &intf->altsetting[i];
126     }
127     return NULL;
128 }
```

这个函数依靠一个for循环来解决问题，就是轮询接口里的每个设置，比较它们的编号与你指定的编号是不是一样。原理简单，操作也简单，不简单的是前面调用它时为什么指定编号0，也就是获得0号设置。这还要归咎于spec里，接口的默认设置总是0号设置，所以这里的目的就是获得接口的默认设置，如果没有拿到设置0，接下来就在1544行拿altsetting数组里的第一项来充数。

1546行，指定刚刚拿到的设置为当前要使用的设置。

1547行，前面遇到过device和endpoint的disable函数，这里遇到个interface的enable函数，同样在message.c中定义。

```
1111 static void usb_enable_interface(struct usb_device *dev,
1112                                 struct usb_interface *intf)
1113 {
1114     struct usb_host_interface *alt = intf->cur_altsetting;
1115     int i;
1116
1117     for (i = 0; i < alt->desc.bNumEndpoints; ++i)
```

```
    1118        usb_enable_endpoint(dev, &alt->endpoint[i]);
    1119 }
```

这个函数同样也是靠一个for循环来解决问题，轮询前面获得的那个接口设置使用到的每个端点，调用message.c里的usb_enable_endpoint()将它们enable。

```
    1085 static void
    1086 usb_enable_endpoint(struct usb_device *dev, struct usb_host_endpoint *ep)
    1087 {
    1088     unsigned int epaddr = ep->desc.bEndpointAddress;
    1089     unsigned int epnum = epaddr & USB_ENDPOINT_NUMBER_MASK;
    1090     int is_control;
    1092     is_control = ((ep->desc.bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
    1093                  == USB_ENDPOINT_XFER_CONTROL);
    1094     if (usb_endpoint_out(epaddr) || is_control) {
    1095         usb_settoggle(dev, epnum, 1, 0);
    1096         dev->ep_out[epnum] = ep;
    1097     }
    1098     if (!usb_endpoint_out(epaddr) || is_control) {
    1099         usb_settoggle(dev, epnum, 0, 0);
    1100         dev->ep_in[epnum] = ep;
    1101     }
    1102 }
```

这个函数的前面儿几行没什么好说的，分别获得端点地址，端点号，还有是不是控制端点。后面的两个if函数，它们分别根据端点的方向来初始化设备的ep_in和ep_out数组，还有就是调用了include/linux/usb.h里的一个叫usb_settoggle的宏。

```
    1425 #define usb_gettoggle(dev, ep, out) (((dev)->toggle[out] >> (ep)) & 1)
    1426 #define usb_dotoggle(dev, ep, out)  ((dev)->toggle[out] ^= (1 << (ep)))
    1427 #define usb_settoggle(dev, ep, out, bit) \
    1428         ((dev)->toggle[out] = ((dev)->toggle[out] & ~(1 << (ep))) | \
    1429         ((bit) << (ep)))
```

这三个宏都是用来处理端点的toggle位的，也就是struct usb_device里的数组toggle[2]。toggle[0]对应的是IN端点，toggle[1]对应的是OUT端点，上面宏参数中的out用来指定端点是IN还是OUT，ep指的不是端点的结构体，而仅仅是端点号。

usb_gettoggle用来得到端点对应的toggle值，usb_dotoggle用来将端点的toggle位取反，也就是原来为1就置为0，原来为0就置为1，usb_settoggle看起来就要复杂点儿，意思是如果bit为0，则将ep 所对应的toggle 位reset 成0，如果bit为1，则reset为1。((dev)->toggle[out] & ~(1<<(ep)))就是把1左移ep位，比如ep为3，那么就是得到了1000，然后取反，得到0111，（当然高位还有更多个1），然后(dev)->toggle[out]和0111相与，这就是使得toggle[out]的第3位清零而其他位都不变。这里调用usb_settoggle时，bit传递的是0，用来将端点的toggle位清零，原因早先也提到过，就是对于批量传输、控制传输和中断传输来说，数据包最开始都是被初始化为DATA0的，然后才一次传DATA0，一次传DATA1。

现在看一看为什么两个if语句里都出现有is_control。控制传输使用的是message管道，message管道必须对应两个相同号码的端点，一个用来“in”，一个用来“out”。这里使用两个if，而不是if-else组合，目的就是加进去一个is_control，表示只要是控制端点，就将它的端点号对应的IN和OUT两个方向上的toggle位还有ep_int和ep_out都给初始化了。当然，所谓的控制端点一般也就是指端点0。

endpoint的enable函数要比disable函数简单得多，disable时还要深入到HCD的腹地去撤销挂在它上面的各个urb，而enable时就是简单设置一下toggle位还有那两个数组就好了，要知道它的urb队列urb_list是早在从设备那里获取配置描述符并去解析那一大堆数据时就初始化好了的。enable之后，接口里的各个端点便都处于了可用状态，你就可以在驱动里向指定的端点提交urb了。当然，到目前这个时候接口还仍然是接口，驱动（接口驱动）还仍然是驱动。

然后看一看跟在usb_enable_interface()后面的那几行，接口所属的总线类型仍然为usb_bus_type，设备类型变为usb_if_device_type，dma_mask被设置为你的设备的dma_mask，而你设备的dma_mask很早以前就被设置为了host controller的dma_mask。

1553行，device_initialize在初始化设备struct usb_device结构体时遇到过一次，这里初始化接口时又遇到了。

1554行，将接口的is_active标志初始化为0，表示它还没有与任何驱动绑定，就是还没有找到另一半。

1559行，for循环结束了，new_interfaces的历史使命也就结束了。这里的kfree释放的只是new_interfaces指针数组的内存，而不包括它里面各个指针所指向的内存，至于那些数据，都已经在前面被赋给配置里的interface数组了。

1561行，获得配置的字符串描述符。

1570行，这个for循环结束，usb_set_configuration()的三个阶段也就算结束了，设备和usb_generic_driver上上下下忙活了这么久也都很累了，接下来就该接口和接口驱动去忙活了。

这个for循环将前面那个for循环准备好的每个接口送给设备模型，Linux设备模型会根据总线类型usb_bus_type将接口添加到USB总线的那条有名的设备链表里，然后去轮询USB总线的另外一条有名的驱动链表，针对每个找到的驱动去调用USB总线的match函数，也就是usb_device_match()，去为接口寻找另一个匹配的半圆。你说这个时候设备和接口两条路它应该走哪条？它的类型已经设置成usb_if_device_type了，设备那条路把门儿的根本就不会让它进，所以它必须得去走接口那条路。

22、字符串描述符

字符串描述符的地位仅次于设备/配置/接口/端点四大描述符，那么四大设备必须得支持它。而字符串描述符对设备来说则是可选的。

这并不是就说字符串描述符不重要，对咱们来说，字符串要比数字亲切得多，提供字符串描述符的设备也要比没有提供的设备亲切得多，不会有人会专门去记前面使用lsusb列出的04b4表示Cypress Semiconductor Corp.。

一提到字符串，不可避免就得提到字符串使用的语言。字符串亲切是亲切，但不像数字那样全球通用。当然这并不是说设备中就要存储这么多不同语言的字符串描述符，这未免要求过高了一些，代价也昂贵了一些，要知道这些描述符不会凭空生出来，是要存储在设备的EEPROM里的，此物是需要记忆的。所以说只提供几种语言的字符串描述符就可以了，甚至说只提供一种语言，比如英语就可以了。

其实咱们现在说的语言就是太多了。不过不管哪种语言，在PC里或者设备中存放都只能用二进制数字，这就需要在语言与数字之间进行编码，这个所谓的编码和这个世界上其他事物一样，都是有多种的。

Spec里就说了，字符串描述符使用的就是UNICODE编码，USB设备中的字符串可以通过它来支持多种语言，不过你需要在向设备请求字符串描述符时指定一个你期望看到的一种语言，俗称语言ID，即Language ID。这个语言ID使用两个字节表示，所有可以使用的语言ID在http://www.usb.org/developers/docs/USB_LANGIDs.pdf文档里都有列出来，从这个文档里你也可以明白为啥要使用两个字节，而不是一个字节表示。

这么说吧，比如中文是0X04，但是中文还有好几种，所以就需要另外一个字节表示是哪种中文，简体就是0X02。注意合起来表示简体中文并不是0X0402或者0X0204，因为这两个字节并不是分得那么清，bit0~9一共10位去表示Primary语言ID，其余6位去表示Sub语言ID，毕竟一种语言的Sub语言不可能会特别的多，没必要分去整整一半8bits，所以简体中文的语言ID就是0X0804。

不多罗嗦，还是结合代码，从usb_cache_string说起，看一看如何去获得一个字符串描述符，它在message.c中定义。

```
825 char *usb_cache_string(struct usb_device *udev, int index)
826 {
827     char *buf;
828     char *smallbuf = NULL;
829     int len;
830
831     if (index > 0 && (buf = kmalloc(256, GFP_KERNEL)) != NULL) {
832         if ((len = usb_string(udev, index, buf, 256)) > 0) {
833             if ((smallbuf = kmalloc(++len, GFP_KERNEL)) == NULL)
834                 return buf;
835             memcpy(smallbuf, buf, len);
```

```
836             }
837             kfree(buf);
838         }
839         return smallbuf;
840 }
```

每个成年人都有一个身份证号码，每个字符串描述符都有一个序号。身份证号可能会重复，字符串描述符这个序号是不能重复的。

也好理解，什么东西一多了，最好最节约、最省事的区分方式就是编号，字符串描述符当然可以有很多个，参数的index就是表示了你希望获得其中的第几个。但是不可疏忽大意的是，不能指定index为0，0编号是有特殊用途的，指定为0就什么也得不到。

有关这个函数，还需要明白两点，第一是它采用的方针策略，就是苦活儿累活儿找usb_string()去做。这个usb_string()怎么工作的之后再看，现在只要注意下它的参数，比usb_cache_string()的参数多了两个，就是buf和size，也就是需要传递一个存放返回的字符串描述符的缓冲区。但是你调用usb_cache_string()时并没有指定一个明确的size，usb_cache_string()也就不知道你想要的字符串描述符有多大，于是它就采用了这么一个技巧，先申请一个足够大的缓冲区，这里是256字节，拿这个缓冲区去调用usb_string()，通过usb_string()的返回值会得到字符串描述符的真实大小，然后再拿这个真实的大小去申请一个缓冲区，并将大缓冲区里放的字符串描述符数据复制过来，这时那个大缓冲区当然就没什么利用价值了，于是再把它给释放掉。

第二就是申请小缓冲区时，使用的并不是usb_string()的返回值，而是多了1个字节，也就是说要从大缓冲区里多复制一个字节到小缓冲区里，为什么？这牵涉到C语言里字符串方面字符串结束符。

字符串都需要结束符，但并不是每个人都能记得给字符串加上结束符。下面举一个例子。

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 int main(int argc, char *argv[])
 5 {
 6   #define MAX (100)
 7
 8   char buf[512], tmp[32];
 9   int i, count = 0;
10
11   for ( i = 0; i< MAX; ++i){
12           sprintf(tmp, "0x%.4X,", i);
13
14       strcat(buf, tmp);
15
16       if (count++ == 10){
17               printf("%s\n", buf);
18               buf[0] = '\0';
19               count = 0;
20               }
21   }
22
23   return 0;
24 }
```

这程序简单直白，打印100个数，每行10个，你觉得它有什么毛病没有？当然我不是说算法上的问题，这本来就演示用的，你只需要查看它能不能得到预期的结果。

当然这程序是有问题的，在第10行少了下面这一句：

```
    buf[0] = '\0'; ////////////////// here !
```

就是说忘记将buf初始化了，传递给strcat的是一个没有初始化的buf，这个时候buf的内容都非0，strcat不知道它的结尾在哪里，它能猜到buf的开始，却猜不到buf的结束。me mset就比较的消耗CPU了，而这里buf[0] = '\0'就足够用了。为了更好地说明问题，这里贴一下内核中对strcat的定义。

```
171 char *strcat(char *dest, const char *src)
172 {
173     char *tmp = dest;
174
```

```
175     while (*dest)
176         dest++;
177     while ((*dest++ = *src++) != '\0')
178         ;
179     return tmp;
180 }
```

strcat会从dest的起始位置开始去寻找一个字符串的结束符，只有找到，175行的while循环才会结束。但是如果dest没有初始化过，义无反顾的strcat并不会有好结果。本来strcat的目的是将tmp追加到buf里字符串的后面，但是因为buf没有初始化，没有一个结束符，strcat就会一直找下去，就算它在哪儿停了下来，如果这个停下的位置超出了buf的范围，就会把src的数据写到不应该的地方，就可能会破坏其他很重要的数据，你的系统可能就“死”掉了。

解决这种问题的方法很简单，就是记着指针、数组使用前首先统统初始化掉，到最后真觉得哪里不必要影响性能了再去优化它。

问一个大家都会笑的问题，这个字符串结束符具体是什么？C和C++里一般是指'\0'，像上面的那句buf[0] = '\0'。这里再引用spec里的一句话：The UNICODE string descriptor is not NULL-terminated. 什么是NULL-terminated字符串？其实就是以'\0'结尾的字符串，咱们查看内核中对NULL的定义。

```
6 #undef NULL
7 #if defined(__cplusplus)
8 #define NULL 0
9 #else
10 #define NULL ((void *)0)
11 #endif
```

0是一个整数，但它又不仅仅是一个整数。由于各种标准转换，0可以被用于作为任意的整型、浮点类型、指针。0的类型将由上下文来确定。典型情况下0被表示为一个适当大小的全零二进制位的模式。所以，无论NULL是定义为常数0还是((void *)0)这个零指针，NULL都是指的是0值，而不是非0值。而字符的'\0'和整数的0也可以通过转型来相互转换。再多说一点，'\0'是C语言定义的用'\'加8进制ASCII码来表示字符的一种方法，'\0'就是表示一个ASCII码值为0的字符。

所以更准确地说，NULL-terminated字符串就是以0值结束的字符串，那么spec的那句话说字符串描述符不是一个NULL-terminated字符串，意思也就是字符串描述符没有一个结束符，你从设备那里得到字符串之后得给它追加一个结束符。本来usb_string()里已经为buf追加好了，但是它返回的长度里还是没有包括进这个结束符的1个字节，所以usb_cache_string()为smallbuf申请内存时就得多准备那么一个字节，以便将buf里的那个结束符也给复制过来。现在就查看usb_string()的细节，定义在message.c里。

```
757 int usb_string(struct usb_device *dev, int index, char *buf, size_t size)
758 {
759     unsigned char *tbuf;
760     int err;
761     unsigned int u, idx;
762
763     if (dev->state == USB_STATE_SUSPENDED)
764         return -EHOSTUNREACH;
765     if (size <= 0 || !buf || !index)
766         return -EINVAL;
767     buf[0] = 0;
768     tbuf = kmalloc(256, GFP_KERNEL);
769     if (!tbuf)
770         return -ENOMEM;
771
772     /* get langid for strings if it's not yet known */
773     if (!dev->have_langid) {
774         err = usb_string_sub(dev, 0, 0, tbuf);
775         if (err < 0) {
776             dev_err (&dev->dev,
777                     "string descriptor 0 read error: %d\n",
778                         err);
779             goto errout;
780         } else if (err < 4) {
781             dev_err (&dev->dev, "string descriptor 0 too short\n");
782                     err = -EINVAL;
783             goto errout;
```

```
    784          } else {
    785              dev->have_langid = 1;
    786              dev->string_langid = tbuf[2] | (tbuf[3]<< 8);
    787              /* always use the first langid listed */
    788              dev_dbg (&dev->dev, "default language 0x%04x\n",
    789                              dev->string_langid);
    790          }
    791      }
    793      err = usb_string_sub(dev, dev->string_langid, index, tbuf);
    794      if (err < 0)
    795          goto errout;
    797      size--;  /* leave room for trailing NULL char in output buffer */
    798      for (idx = 0, u = 2; u < err; u += 2) {
    799          if (idx >= size)
    800              break;
    801          if (tbuf[u+1])                  /* high byte */
    802              buf[idx++] = '?';  /* non ISO-8859-1 character */
    803          else
    804              buf[idx++] = tbuf[u];
    805      }
    806      buf[idx] = 0;
    807      err = idx;
    809      if (tbuf[1] != USB_DT_STRING)
    810          dev_dbg(&dev->dev, "wrong descriptor type %02x for string %d (\"%s\")\n", tbuf[1],
index, buf);
    811
    812  errout:
    813      kfree(tbuf);
    814      return err;
    815  }
```

763行，这几行做些例行检查，设备不能是挂起的，index也不能是0，只要传递了指针就需要检查。

767行，初始化buf，usb_cache_string()并没有对这个buf初始化，所以这里必须要加上这么一步。当然usb_string()并不仅仅只有在usb_cache_string()里调用，可能会在很多地方调用到它，不过不管在哪里，这里谨慎起见，还是需要这一步。

768行，申请一个256字节大小的缓冲区。前面一直强调要初始化，怎么到这里没有去初始化tbuf？这是因为没必要。为什么没必要？你查看看usb_string()最后面的那些代码就明白了。

773行，struct usb_device里有have_langid和string_langid这么两个字段是和字符串描述符有关的，string_langid用来指定使用哪种语言，have_langid用来指定string_langid是否有效。如果have_langid为空，就说明没有指定使用哪种语言，那么获得的字符串描述符使用的是哪种语言就完全看设备了。你可能会疑惑，为什么当have_langid为空时，要在774行和793行调用两次usb_string_sub()？就像usb_string()是替usb_cache_string()一样，usb_string_sub()是替usb_string()做苦工的，也就是说usb_string()是靠usb_string_sub()去获得字符串描述符的，那问题就变成为什么have_langid为空时，要获取两遍的字符串描述符？

你可以比较一下两次调用usb_string_sub()的参数有什么区别。第一次调用参数时，语言ID和index都为0，第二次调用参数时就明确指定了语言ID和index。这里的玄机就在index为0时，也就是0编号的字符串描述符是什么，前面只说了它有特殊的用途，现在必须得解释一下。

spec在9.6.7节说了，编号0的字符串描述符包括了设备所支持的所有语言ID，对应的就是如图11所示的表。

偏移(Offset)	字段(Field)	字节(Size)	描述(Description)
0	bLength	1	描述符的字节数
1	bDescriptorType	1	描述符的类型
2	wLANGID[0]	2	语言ID 0
...	...	...	...
N	wLANGID[x]	2	语言ID x

图 11　0 号字符串描述符

第一次调用usb_string_sub()就是为了获得这张表，获得这张表做什么用？接着往下看。

775行，usb_string_sub()返回一个负数，就表示没有获得这张表，没有取到0号字符串描述符。如果返回值比4要小，就表示获得的表里没有包含任何一个语言ID。要知道一个语言ID占用2个字节，还有前两个字节表示表的长度及类型，所以得到的数据至少要为4，才能够得到一个有效的语言ID。如果返回值比4要大，就使用获得的数据的第3字节和第4字节设置string_langid，同时设置have_langid为1。

现在很明显了，773行~791行这些代码就是在你没有指定使用哪种语言时，去获取设备中默认使用的语言，也就是0号字符串描述符里的第一个语言ID所指定的语言。如果没有找到这个默认的语言ID，即usb_string_sub()返回值小于4的情况，就没有办法再去获得其他字符串描述符了。因为没有指定语言，设备不知道你是要英语还是中文或是其他的。

793行，使用指定的语言ID，或者前面获得的默认语言ID去获得想要的字符串描述符。现在看一看定义在message.c里的usb_string_sub函数。

```
696 static int usb_string_sub(struct usb_device *dev, unsigned int langid,
697                 unsigned int index, unsigned char *buf)
698 {
699     int rc;
700
701     /* Try to read the string descriptor by asking for the maximum
702      * possible number of bytes */
703     if (dev->quirks & USB_QUIRK_STRING_FETCH_255)
704         rc = -EIO;
705     else
706         rc = usb_get_string(dev, langid, index, buf, 255);
707
708     /* If that failed try to read the descriptor length, then
709      * ask for just that many bytes */
710     if (rc < 2) {
711         rc = usb_get_string(dev, langid, index, buf, 2);
712         if (rc == 2)
713             rc = usb_get_string(dev, langid, index, buf, buf[0]);
714     }
715
716     if (rc >= 2) {
717         if (!buf[0] && !buf[1])
718             usb_try_string_workarounds(buf, &rc);
719
720         /* There might be extra junk at the end of the descriptor */
721         if (buf[0] < rc)
722             rc = buf[0];
723
724         rc = rc - (rc & 1); /* force a multiple of two */
725     }
726
727     if (rc < 2)
728         rc = (rc < 0 ? rc : -EINVAL);
730     return rc;
731 }
```

这个函数首先检查一下你的设备是不是属于遵纪守法的合格设备，然后就调用usb_get_string()去获得字符串描述符。USB_QUIRK_STRING_FETCH_255就是在include/linux/usb/quirks.h中定义的那些形形色色的毛病之一，表示设备在获取字符串描述符时会crash。

usb_string_sub()的核心就是message.c中定义usb_get_string函数。

```
664 static int usb_get_string(struct usb_device *dev, unsigned short langid,
665                     unsigned char index, void *buf, int size)
666 {
667     int i;
668     int result;
669
670     for (i = 0; i < 3; ++i) {
671         /* retry on length 0 or stall; some devices are flakey */
672         result = usb_control_ msg(dev, usb_rcvctrlpipe(dev, 0),
673                 USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
674                 (USB_DT_STRING << 8) + index, langid, buf, size,
675                 USB_CTRL_GET_TIMEOUT);
676         if (!(result == 0 || result == -EPIPE))
```

```
677                break;
678        }
679        return result;
680 }
```

我已经不记得这是第多少次遇到usb_control_ msg()了。老习惯，还是简单说一下它的一堆参数。wValue的高位字节表示描述符的类型，低位字节表示描述符的序号，所以有674行的(USB_DT_STRING << 8) + index，wIndex。对于字符串描述符应该设置为使用语言的ID，所以有674行的langid。至于wLength，就是描述符的长度，对于字符串描述符很难有一个统一的确定的长度，所以一半来说上头儿传递过来的通常是一个比较大的255字节。

和获得设备描述符时一样，因为一些厂商搞出的设备古灵精怪，可能需要多试几次才能成功。要容许设备犯错误。

还是回过头去看usb_string_sub函数，如果usb_get_string()成功的得到了期待的字符串描述符，则返回获得的字节数，如果这个数目小于2，就再读两个字节试一试，要想明白这两个字节是什么内容，需要查看如图12所示的表。

偏移(Offset)	字段(Field)	字节(Size)	描述(Description)
0	bLength	1	描述符的字节数
1	bDescriptorType	1	描述符的类型
2	bString	N	UNICODE编码的字符串

图 1.22.1　其他字符串描述符

图10描述的是0号字符串描述符的格式，这个图11描述的是其他字符串描述符的格式。很明显可以看到，它的前两个字节分别表示了长度和类型，如果读两个字节成功的话，就可以准确地获得这个字符串描述符的长度，然后再拿这个准确的长度去请求一次。

该尝试的都尝试了，现在查看716行，分析一下前面调用usb_get_string()的结果，如果几次尝试之后，它的返回值还是小于2，那就返回一个错误码。rc大于等于2，说明终于获得了一个有效的字符串描述符。

717行，buf的前两个字节有一个为空时，也就是字符串描述符的前两个字节有一个为空时，调用了message.c中定义的usb_try_string_workarounds函数。

```
682 static void usb_try_string_workarounds(unsigned char *buf, int *length)
683 {
684        int newlength, oldlength = *length;
685
686        for (newlength = 2; newlength + 1 < oldlength; newlength += 2)
687            if (!isprint(buf[newlength]) || buf[newlength + 1])
688                break;
689
690        if (newlength > 2) {
691            buf[0] = newlength;
692            *length = newlength;
693        }
694 }
```

这个函数的目的是从usb_get_string()返回的数据里计算出前面有效的长度。它的核心就是686行的for循环，不过要搞清楚这个循环，还真不是一件容易的事情，得有相当的理论功底。

前面刚说了字符串描述符使用的是UNICODE编码，其实UNICODE指的是包含了字符集、编码、字型等很多规范的一整套系统，字符集仅仅描述符系统中存在那些字符，并进行分类，并不涉及如何用数字编码表示的问题。

UNICODE使用的编码形式主要有两种UTF，即UTF-8和UTF-16。使用usb_get_string()获得的字符串使用的是UTF-16编码规则，而且是little-endpoint的，每个字符都需要使用两个字节来表示。在这个for循环里newlength每次加2，就是表示每次处理一个字符的，但是要弄明白怎么处理的，还需要知道这两个字节分别是什么，这就不得不提及ASCII、ISO-8859-1等几个名词。

ASCII是用来表示英文的一种编码规范，表示的最大字符数为256，每个字符占1个字节。但是英文字符没那么多，一般来说128个也就够了（最高位为0），这就已经完全包括了控制字符、数字、大小写字母，还有其他一些符号。对于法语、西班牙语和德语之类的西欧语言都使用叫做ISO-8859-1，它扩展了ASCII码的最高位，来表示像n上带有一个波浪线（241），和u上带有两个点（252）这样的字符。而Unicode的低字节，也就是在0到255上同ISO-8859-1完全一样，它接着使用剩余的数字，256到65535，扩展到表示其他语言的字符。所以可以说ISO-8859-1就是Unicode的子集，如果Unicode的高字节为0，则它表示的字符就和ISO-8859-1完全一样了。

再看一看这个for循环，newlength从2开始，是因为前两个字节应该是表示长度和类型的，这里只逐个儿对上面Table 9-16里的bString中的每个字符做处理。还要知道usb_get_string()得到的结果是little-endpoint的，所以buf[newlength]和buf[newlength + 1]分别表示一个字符的低字节和高字节，那么isprint(buf[newlength]就是用来判断一下这个Unicode字符的低字节是不是可以print的。如果不是，就没必要再往下循环了，后边儿的字符也不再看了，然后就到了690行的if，将newlength赋给buf[0]，即bLength。length指向的是usb_get_string()返回的原始数据的长度，692行使用for循环计算出的有效长度将它修改了。isprint在include/linux/ctype.h中定义，你可以去查看，这里就不多说了。

这个for循环终止的条件有两个，另外一个就是buf[newlength + 1]，也就是这个Unicode字符的高字节不为0，这时它不存在对应的ISO-8859-1形式，为什么加上这个判断？你接着看。

usb_string_sub()的721行，buf[0]表示的就是bLength的值，如果它小于usb_get_string()获得的数据长度，说明这些数据里存在一些垃圾，要把它们给揪出来排除掉。要知道这个rc是要做为真实有效的描述符长度返回的，所以这个时候需要将buf[0]赋给rc。

724行，每个Unicode字符需要使用两个字节来表示，所以rc必须为偶数，即2的整数倍。如果为奇数，就得将最后那一个字节给去掉，也就是将rc减1。咱们可以学习一下这里将一个数字转换为偶数时采用的技巧，(rc & 1)在rc为偶数时等于0，为奇数时等于1，再使用rc减去它，得到的就是一个偶数。

从716行~725行这几行，咱们应该看得出，在成功获得一个字符串描述符时，usb_string_sub()返回的是一个NULL-terminated字符串的长度，并没有涉及结束符。牢记这一点，回到usb_string函数的797行，先将size，也就是buf的大小减1，目的就是为结束符保留1个字节的位置。

798行，tbuf里保存的是GET_DESCRIPTOR请求返回的原始数据，err是usb_string_sub()的返回值，一切顺利的话，它表示的就是一个有效的描述符的大小。这里idx的初始值为0，而u的初始值为2，目的是从tbuf的第三个字节开始复制给buf，毕竟这里的目的是获得字符串描述符里的bString，而不是整个描述符的内容。u每次都是增加2，这是因为采用的UTF-16是用两个字节表示一个字符的，所以循环一次要处理两个字节。

801行，这个if-else组合你可能比较糊涂，要搞清楚，还要看一下前面刚讲过的一些理论。tbuf里每个Unicode字符两个字节，又是little-endpoint的，所以801行就是判断这个Unicode字符的高位字节是不是为0，如果不为0，则ISO-8859-1里没有这个字符，就设置buf里的对应字符为‘?’。如果它的高位字节为0，就说明这个Unicode字符ISO-8859-1里也有，就直接将它的低位字节赋给buf。这么一个for循环下来，就将tbuf里的Unicode字符串转化成了buf里的ISO-8859-1字符串。

806行，为buf追加一个结束符。

23、接口的驱动

我们已经知道，usb_generic_driver在自己的生命线里，以一己之力将设备的各个接口送给了Linux的设备模型，让USB总线的match函数，也就是usb_device_match()，在自己的驱动链表里为它们寻找一个合适的接口驱动程序。现在让我问一句：“这些接口驱动都从哪里来？”

这就要说到几个著名的命令insmod，modprobe，rmmod。当insmod或modprobe驱动时，经过一个曲折的过程，会调用到你驱动里的那个xxx_init函数，进而去调用usb_register()将你的驱动提交给设备模型，添加到USB总线的驱动链表里。rmmod驱动时候，同样经过一个曲折的过程之后，调用到驱动里的那个xxx_cleanup函数，进而调用usb_deregister()将你的驱动从USB总线的驱动链表里删除掉。

现在就查看include/linux/usb.h中定义的usb_register函数。

```
916 static inline int usb_register(struct usb_driver *driver)
917 {
918     return usb_register_driver(driver, THIS_MODULE, KBUILD_MODNAME);
919 }
```

看到这个函数，让人不得不感叹一下，现在什么都要讲究包装，在内核中注册一个驱动也要包装个两层，不过这个包装不是为了出名，而是方便大伙儿的。

本来在两年以前没有usb_register_driver这个函数，只有个usb_register()。而且那个时候struct usb_driver结构中还有一个有名的owner字段，每个在那个岁月里写过USB驱动的人都会认得它，并且都会毫无犹豫的将它设置为THIS_MODULE。但是经过岁月的洗礼，在早先贴出来的struct usb_driver结构内容里，你会发现owner已经无影无踪了。要搞清楚这个历史变迁的来龙去脉，你得知道这个owner和THIS_MODULE都代表了什么。

从那个时代走过来的人，都应该知道owner是一个struct module *类型的结构体指针，现在告诉你的是每个struct module结构体在内核中都代表了一个内核模块，每个struct module结构体代表了什么模块，对它进行初始化的模块才知道。

当然，初始化这个结构不是写驱动的人该做的事，是在刚才那个从insmod或modprobe到你驱动的xxx_init函数的曲折过程中做的事。insmod命令执行后，会在kernel/module.c里的一个系统调用sys_init_module，它会调用load_module函数，将用户空间传入的整个内核模块文件创建成一个内核模块，并返回一个struct module结构体，从此，内核中便以这个结构体代表这个内核模块。

再查看THIS_MODULE宏是什么意思，它在include/linux/module.h里的定义：

```
85 #define THIS_MODULE (&__this_module)
```

而是一个struct module变量，代表当前模块，与著名的current有几分相似，可以通过THIS_MODULE宏来引用模块的struct module结构。比如使用THIS_MODULE->state可以获得当前模块的状态。现在你应该明白为什么在以前，你需要毫不犹豫毫不迟疑地将struct usb_driver结构中的owner设置为THIS_MODULE了吧，这个owner指针指向的就是模块自己。

那现在owner怎么就没了呢？这个说来可就话长了，咱就长话短说吧。不知道那个时候你有没有忘记过初始化owner，反正是很多人都会忘记，大家都把注意力集中到probe、disconnect等需要动脑子变量上面了，这个不需要动脑子，只需要花个几秒钟指定一下的owner反倒常常被忽视。

于是在2006年的春节前夕，Greg去掉了owner，于是千千万万个写usb驱动的人再也不用去时刻谨记初始化owner了。咱们是不用设置owner了，可core里不能不设置，struct usb_driver结构中不是没有owner了吗，可它里面嵌的struct device_driver结构中还有，设置了它就可以了。于是Greg同时又增加了usb_register_driver()这么一层，usb_register()可以通过将参数指定为THIS_MODULE去调用它，所有的事情都挪到里面去做。反正usb_register()也是内联的，并不会增加调用的开销。现在是时机看一看usb_register_driver函数了。

```
734 int usb_register_driver(struct usb_driver *new_driver, struct module *owner,
735                         const char *mod_name)
736 {
737     int retval = 0;
738
739     if (usb_disabled())
740         return -ENODEV;
741
742     new_driver->drvwrap.for_devices = 0;
743     new_driver->drvwrap.driver.name = (char *) new_driver->name;
744     new_driver->drvwrap.driver.bus = &usb_bus_type;
```

```
745     new_driver->drvwrap.driver.probe = usb_probe_interface;
746     new_driver->drvwrap.driver.remove = usb_unbind_interface;
747     new_driver->drvwrap.driver.owner = owner;
748     new_driver->drvwrap.driver.mod_name = mod_name;
749     spin_lock_init(&new_driver->dynids.lock);
750     INIT_LIST_HEAD(&new_driver->dynids.list);
751
752     retval = driver_register(&new_driver->drvwrap.driver);
753
754     if (!retval) {
755         pr_info("%s: registered new interface driver %s\n",
756                 usbcore_name, new_driver->name);
757         usbfs_update_special();
758         usb_create_newid_file(new_driver);
759     } else {
760         printk(KERN_ERR "%s: error %d registering interface "
761                 "       driver %s\n",
762                 usbcore_name, retval, new_driver->name);
763     }
764
765     return retval;
766 }
```

这函数和前面见过的usb_register_device_driver长得很像，你如果是从那里一路看过来的话，不用我说什么，你都会明明白白它的意思。for_devices在742行设置成了0，有了这行，match里的is_usb_device_driver把门儿的才不会把它当成设备驱动放过去。

然后就是在752行将你的驱动提交给设备模型，从而添加到USB总线的驱动链表里，从此之后，接口和接口驱动就可以通过USB总线的match函数传达数据了。

24、还是那个match

从前面遇到USB总线的match函数usb_device_match()开始到现在，遇到了设备，遇到了设备驱动，遇到了接口，也遇到了接口驱动，期间还多次遇到usb_device_match()。

其实每个人都有一条共同之路，与正义和良知初恋，失身于上学，嫁给了钱，被世俗包养。每个设备也都有一条共同之路，与Hub初恋，失身于usb_generic_driver，嫁给了接口驱动，被usb总线保养。

人类从没有真正自由过，少年时我们坐在课室里动弹不得，稍后又步入办公室，无论外头阳光多好，还得超时加班，终于铅华洗尽，遍历人间沧桑，又要为子女忙碌，有几个人可以真正做自己想做的事？设备也没有真正自由过，刚开始时在Default状态动弹不得，稍后步入Address，无论外头风光多好，都得与usb_generic_driver长厢厮守，没得选择，终于达到了Configured，又必须为自己的接口殚精竭虑，以便usb_device_match()能够为它们找一个好人家。

不管怎么说，在这里我们会再次与usb_device_match()相遇，查看它怎么在接口和驱动之间搭起那座桥。

```
540 static int usb_device_match(struct device *dev, struct device_driver *drv)
541 {
542     /* devices and interfaces are handled separately */
543     if (is_usb_device(dev)) {
544
545     /* interface drivers never match devices */
546     if (!is_usb_device_driver(drv))
547             return 0;
548
549         /* TODO: Add real matching code */
550         return 1;
551
552     } else {
553         struct usb_interface *intf;
554         struct usb_driver *usb_drv;
555         const struct usb_device_id *id;
556
557         /* device drivers never match interfaces */
558         if (is_usb_device_driver(drv))
```

```
559                 return 0;
560
561             intf = to_usb_interface(dev);
562             usb_drv = to_usb_driver(drv);
563
564             id = usb_match_id(intf, usb_drv->id_table);
565             if (id)
566             return 1;
567
568             id = usb_match_dynamic_id(intf, usb_drv);
569             if (id)
570             return 1;
571         }
572
573         return 0;
574 }
```

设备那条路已经走过了，现在走一走552行接口这条路。558行，接口驱动的for_devices在usb_register _driver()里被初始化为0，所以这个把门儿的会痛痛快快地放行，继续往下走。

561行，遇到一对似曾相识的宏to_usb_interface和to_usb_driver，之所以说似曾相识，是因为前面已经遇到过一对儿to_usb_device和to_usb_device_driver。这两对宏一对儿用于接口和接口驱动，一对儿用于设备和设备驱动，意思都很直白，还是看一看include/linux/usb.h里的定义。

```
159 #define to_usb_interface(d) container_of(d, struct usb_interface, dev)
857 #define to_usb_driver(d) container_of(d, struct usb_driver, drvwrap.driver)
```

再往下走，就是两个函数usb_match_id和usb_match_dynamic_id，它们都是用来完成实际的匹配工作的，只不过前一个是从驱动的id_table里找，看接口是不是被驱动所支持，后一个是在驱动的动态id链表dynids里找。驱动的id表分id_table和dynids两种。显然564行~570行这几行的意思就是将id_table放在一个比较高的优先级的位置，从它里面找不到接口了才再从动态id链表里找。

当时讲到struct usb_driver结构时并没有详细讲它里面表示动态id的结构体struct usb_dynids，所以现在补充一下，这个结构的定义在include/linux/usb.h里。

```
760 struct usb_dynids {
761     spinlock_t lock;
762     struct list_head list;
763 };
```

它只有两个字段，一把锁，一个链表，都是在usb_register _driver()里面初始化的，这个list是驱动动态id链表的头儿，它里面的每个节点是用另外一个结构struct usb_dynid来表示。

```
765 struct usb_dynid {
766     struct list_head node;
767     struct usb_device_id id;
768 };
```

这里面就出现了一个struct usb_device_id结构体，也就是设备的id，每次添加一个动态id，就会向驱动的动态id链表里添加一个struct usb_dynid结构体。你现在应该可以想像到usb_match_id和usb_match_dynamic_id这两个函数除了查找的地方不一样，其他应该是没什么差别的。所以接下来咱们只深入探讨一下usb_match_id函数，至于usb_match_dynamic_id()，它们都在driver.c中定义。

```
518 const struct usb_device_id *usb_match_id(struct usb_interface *interface,
519                                          const struct usb_device_id *id)
520 {
521     /* proc_connectinfo in devio.c may call us with id == NULL. */
522     if (id == NULL)
523         return NULL;
524
525     /* It is important to check that id->driver_info is nonzero,
526         since an entry that is all zeroes except for a nonzero
527     id->driver_info is the way to create an entry that
528         indicates that the driver want to examine every
529         device and interface. */
530     for (; id->idVendor || id->bDeviceClass || id->bInterfaceClass ||
531         id->driver_info; id++) {
532         if (usb_match_one_id(interface, id))
533             return id;
534     }
```

```
535
536     return NULL;
537 }
```

522行，参数id指向的是驱动的设备花名册，即id_table，如果它为空，那肯定就是不可能会匹配成功了，这样的驱动就如usb驱动里的修女，起码表面上是对所有的接口不感兴趣的。

530行，你可能会问为什么这里不详细介绍一下struct usb_device_id结构，主要是它里面的元素的含意都相当明白。

那么这个for循环就是轮询设备花名册里的每个设备，如果符合了条件id->idVendor || id->bDeviceClass || id->bInterfaceClass || id->driver_info，就调用函数usb_match_one_id做深层次的匹配。本来，在动态id出现之前这个地方是没有usb_match_one_id这么一个函数的，所有的匹配都在这个for循环里直接做了，但是动态id出现之后，同时出现了前面提到的usb_match_dynamic_id函数，要在动态id链表里做同样的匹配，这就要避免代码重复，于是就将那些重复的代码提出来，组成了usb_match_one_id函数。

for循环的条件里可能出现的一种情况是，id的其他字段都为空，只有driver_info字段有实实在在的内容，这种情况下匹配是肯定成功的，不信的话等会儿你可以看usb_match_one_id()，这种驱动对usb接口来说是比较随便的，不管什么接口都能和它对得上。为什么会出现这种情况？咱们已经知道，接口匹配成功后，接着就会调用驱动自己的probe函数，驱动在它里面还会对接口做进一步的检查，如果真出现了这里所说的情况，意思也就是驱动将所有的检查接口，和接口培养感情的步骤都揽在自己的probe函数中了，它会在那个时候将driver_info的内容取出来，然后想怎么处理就怎么处理，本来id里边的driver_info就是给驱动保存数据用的。

还是看一看usb_match_one_id()究竟是怎么匹配的吧，定义也在driver.c里。

```
405 int usb_match_one_id(struct usb_interface *interface,
406                      const struct usb_device_id *id)
407 {
408     struct usb_host_interface *intf;
409     struct usb_device *dev;
410
411     /* proc_connectinfo in devio.c may call us with id == NULL. */
412     if (id == NULL)
413         return 0;
414
415     intf = interface->cur_altsetting;
416     dev = interface_to_usbdev(interface);
417
418     if (!usb_match_device(dev, id))
419         return 0;
420
421     /* The interface class, subclass, and protocol should never be
422      * checked for a match if the device class is Vendor Specific,
423      * unless the match record specifies the Vendor ID. */
424     if (dev->descriptor.bDeviceClass == USB_CLASS_VENDOR_SPEC &&
425             !(id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
426             (id->match_flags & (USB_DEVICE_ID_MATCH_INT_CLASS |
427                 USB_DEVICE_ID_MATCH_INT_SUBCLASS |
428                     USB_DEVICE_ID_MATCH_INT_PROTOCOL)))
429     return 0;
430
431     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
432         (id->bInterfaceClass != intf->desc.bInterfaceClass))
433         return 0;
434
435     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&
436         (id->bInterfaceSubClass != intf->desc.bInterfaceSubClass))
437         return 0;
438
439     if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&
440         (id->bInterfaceProtocol != intf->desc.bInterfaceProtocol))
441         return 0;
442
443     return 1;
444 }
```

412行，这个id指向的就是驱动id_table里的某一项了。

415行，获得接口采用的设置，设置里可是有接口描述符的，要匹配接口和驱动，接口描述符里的信息是必不可少的。

416行，从接口的struct usb_interface结构体获得USB设备的struct usb_device结构体，interface_to_usbdev的定义在include/linux/usb.h里。

```
160 #define interface_to_usbdev(intf) \
161         container_of(intf->dev.parent, struct usb_device, dev)
```

USB设备和它里面的接口是怎么关联起来的呢？就是上面的那个parent，接口的parent早在usb_generic_driver的generic_probe函数向设备模型提交设备中的每个接口时就被初始化好了，而且指定为接口所在的USB设备。那么，interface_to_usbdev的意思就很明显了。

418行，这里又冒出来一个usb_match_device()，接口和驱动之间的感情还真不是那么好培养的，一层一层的。不过既然存在就是有理由的，它也不会毫无根据的出现，这里虽说是在接口和接口驱动之间匹配，但是接口的parent也是必须要符合条件的，这也合情合理。所以说接口要想得到驱动，自己的parent符合驱动的条件也是很重要的，usb_match_device()就是专门来匹配接口parent的。同样在driver.c中定义。

```
369 int usb_match_device(struct usb_device *dev, const struct usb_device_id *id)
370 {
371     if ((id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
372         id->idVendor != le16_to_cpu(dev->descriptor.idVendor))
373         return 0;
375     if ((id->match_flags & USB_DEVICE_ID_MATCH_PRODUCT) &&
376         id->idProduct != le16_to_cpu(dev->descriptor.idProduct))
377         return 0;
379     /* No need to test id->bcdDevice_lo != 0, since 0 is never
380         greater than any unsigned number. */
381     if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_LO) &&
382         (id->bcdDevice_lo > le16_to_cpu(dev->descriptor.bcdDevice)))
383         return 0;
385     if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_HI) &&
386         (id->bcdDevice_hi < le16_to_cpu(dev->descriptor.bcdDevice)))
387         return 0;
389     if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_CLASS) &&
390         (id->bDeviceClass != dev->descriptor.bDeviceClass))
391         return 0;
393     if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_SUBCLASS) &&
394         (id->bDeviceSubClass!= dev->descriptor.bDeviceSubClass))
395         return 0;
397     if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_PROTOCOL) &&
398         (id->bDeviceProtocol != dev->descriptor.bDeviceProtocol))
399         return 0;
401     return 1;
402 }
```

这个函数采用了排比的修辞手法，美观的同时也增加了可读性。这一个个的if条件里都有一部分是将id的match_flags和一个宏相与，所以弄明白match_flags的意思就很关键。这里再说一下这个match_flags。

驱动的花名册里每个设备都对应了一个struct usb_device_id结构体，这个结构体里有很多字段，都是驱动设定好的条条框框，接口必须完全满足里面的条件才能够被驱动所接受，所以说匹配的过程就是检查接口是否满足这些条件的过程。

当然你可以每次都按照id的内容一个一个地比较，但是经常来说，一个驱动往往只是想设定其中的某几项，并不要求struct usb_device_id结构中的所有那些条件都要满足。

match_flags就是为了方便各种各样的需求而生的，驱动可以将自己的条件组合起来，match_flags的每一位对应一个条件，驱动在意哪个条件了，就将那一位置1，否则就置0。当然，内核中对每个驱动可能会在意的条件都定义成了宏，供驱动去组合，它们都在include/linux/mod_devicetable.h中定义。

```
123 #define USB_DEVICE_ID_MATCH_VENDOR          0x0001
124 #define USB_DEVICE_ID_MATCH_PRODUCT         0x0002
```

```
125 #define USB_DEVICE_ID_MATCH_DEV_LO          0x0004
126 #define USB_DEVICE_ID_MATCH_DEV_HI          0x0008
127 #define USB_DEVICE_ID_MATCH_DEV_CLASS        0x0010
128 #define USB_DEVICE_ID_MATCH_DEV_SUBCLASS      0x0020
129 #define USB_DEVICE_ID_MATCH_DEV_PROTOCOL      0x0040
130 #define USB_DEVICE_ID_MATCH_INT_CLASS        0x0080
131 #define USB_DEVICE_ID_MATCH_INT_SUBCLASS      0x0100
132 #define USB_DEVICE_ID_MATCH_INT_PROTOCOL      0x0200
```

很容易能看出来这些数字分别表示了一个u16整数，也就是match_flags中的某一位。驱动比较在意哪个方面，就可以将match_flags里对应的位置1，在和接口匹配时自动就会去比较驱动设置的条件是否满足。那整个usb_match_device()函数就没什么说的了，就是从match_flags那里得到驱动都在意那些条件，然后将设备保存在描述符里的自身信息与id里的相应条件进行比较，有一项比较不成功就说明匹配失败，如果一项符合了就接着看下一项，接口parent都满足条件了，就返回1，表示匹配成功了。

还是回到usb_match_one_id()继续往下看，假设parent满足了驱动的所有条件，得以走到了424行。这行还要对接口的parent做点最后的确认，这行的意思就是，如果接口的parent、USB设备是属于厂商定义的class，也就是不属于storage等标准的class，就不再检查接口的class，subclass和protocol了，除非match_flags里指定了条件USB_DEVICE_ID_MATCH_VENDOR。431行之后的三个if函数也不用多说，前面是检查接口parent的，这里就是检查接口本身是不是满足驱动的条件的。

当上面各个函数进行的所有检查都完全匹配时，USB总线的match函数usb_device_match就会返回1表示匹配成功，之后接着就会去调用驱动的probe函数做更深入的处理，什么样的处理？这是每个驱动才知道的事情，反正到此为止，core的任务是已经圆满完成了，咱们的故事也就该结束了。

25、结束语

这个 core 的故事，从 match 开始，到 match 结束，在 match 的两端是设备和设备的驱动，是接口和接口的驱动，这个故事里遇到的人，遇到的事，早就安排在那里了，由不得我们去选择。

华清远见

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067