本书基于 2.6.22 内核，对 USB 子系统的大部分源代码逐行进行分析，系统地阐释了 Linux 内核中 USB 子系统是如何运转的，子系统内部的各个模块之间是如何互相协作、配合的。

2012 年 3 月第 1 次印刷

本次改版修改了第 1 版中出 现的错误，增加了一个附录，主要内容是关于 Linux 内核的学习方法，是作者的经验总结，值得一读。

本书使用幽默诙谐的笔调对Linux内核中的USB子系统源代码进行了分析，形象且详尽地介绍了USB 在 Linux 中的实现。本书从 U 盘、Hub、USB Core 到主机控制器覆盖了 USB 实现的方方面面，被一些网 友誉为 USB 开发的“圣经”。

对于 Linux 初学者，可以通过本书掌握学习内核、浏览内核代码的方法；对于 Linux 驱动开发者，可 以通过本书对设备模型有形象深刻的理解；对于 USB 开发者，可以通过本书全面理解 USB 在一个操作系 统中的实现；对于 Linux 内核开发者，也可以通过本书学习到很多 Linux 高手开发和维护一个完整子系统 时的编程思想。

或许，大家早已经默认技术本是一个沉重或者枯燥的话题，我们无法用一种娱乐的心态去 看待它，甚至说很多人早已丧失了从中获取乐趣的能力。但是，一切本不该如此的，对于不管 什么原因踏入这个行业的我们，愿意或不愿意，技术都已经是我们生命不可分割的一部分。

# 前 言

当成一个有 **生命的实体，把它放在对等的地位上，我们才能够更好地认识和理解到它的精髓**

# USB Core

为什么要开发 USB？ 在 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接口，所以扩展能力不足，而且速度也确实很有限。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。 USB 正是为了解决速度、扩展能力、易用性等问题应景而生的

USB 初的设计目标就是替代串行、并行等各种低速总线，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。

从 USB 1.0、USB 1.1、USB 2.0 到 USB 3.0， 漫漫辛酸路，一把辛酸泪。 USB 2.0 的高速模式（High-Speed）高已经达到了 480 MB/s，也就是说，以这个速度， 你将自己从网上下载的短片备份到自己的移动硬盘上的时间长约为一秒钟。而 USB 3.0 的 Super-Speed 模式比这个速度提高了几乎 10 倍，达到了 4.8GB/s。

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

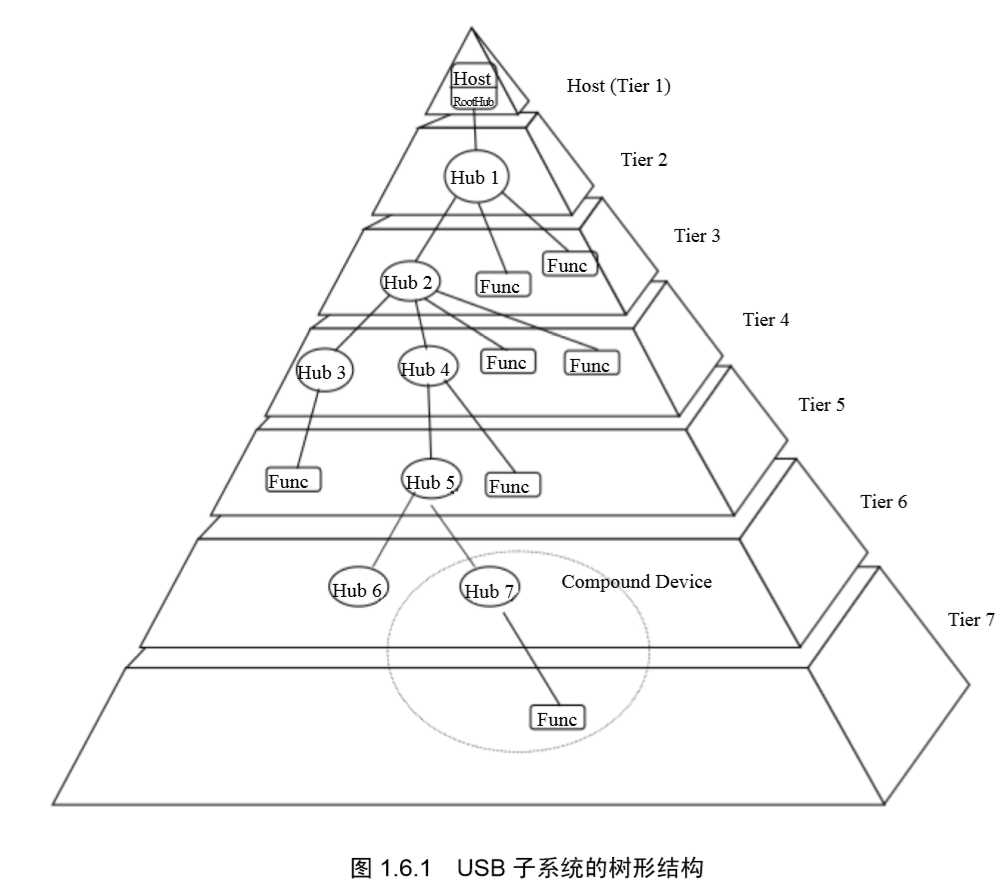
USB 的各个版本都是兼容的，。每个 USB 2.0 控制器带有 3 个芯片，根据设备的识别方式， 将信号发送到正确的控制芯片。我们可以将 USB 1.1 设备连接到 USB 2.0 的控制器上使用，不 过它只能达到 USB 1.1 的速度。同时也可以将 USB 2.0 的设备连接到 USB 1.1 的控制器上，不 过不能指望它能以 USB 2.0 的速度运行。

USB 支持热插拔，而其他的比如 SCSI 设备等只有在关掉主机的前提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，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，不然移动硬盘

IPod 等常备外设也用不了了。相对应，有的 USB 设备也可以使用普通的电源供电。 USB 能够支持从每秒几十千字节到几十兆字节的传输速率，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。

## 6．我是一棵树

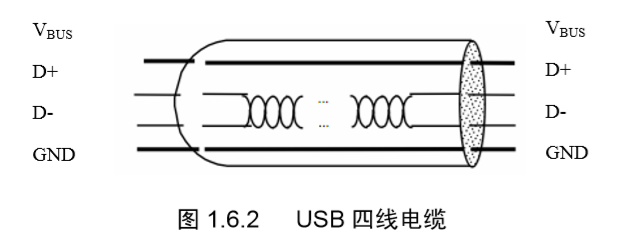
USB 子系统的拓扑也是一棵树，它并不以总线的方式来部署



这棵大树主要包括 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（也就是图 1.6.1 中的 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。控制器替它去完成剩 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。 那么 Hub 是什么？在大学里，有的宿舍里网口有限，所以会有网口不够用的情况出现，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 的计算机不会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，同样还可以插入 USB 键盘，然后再插一个 USB 鼠标，因为你会 发现你的计算机里并不只是一个 USB 接口。这些接口实际上就是所谓的 Hub 口。 而现实中经常是一个 USB 控制器和一个 Hub 绑定在一起，专业一点称为“集成”，而这个 Hub也被称做Root Hub。换而言之，和USB控制器绑定在一起的Hub就是系统中根本的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）和两根信号线。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 1.6.2 所示

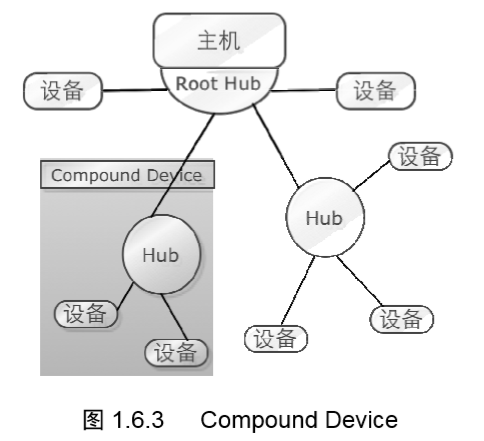
现在，如图 1.6.3 所示的 USB 大树里只有 Compound Device 还没有说。那么，Compound Device又是什么样的设备？其实， 在USB的世界里，不仅仅有Compound Device，还有Composite Device，简单的中文名字已经无法形象地表达它们的区别。正如图 1.6.3 所示，Compound Device 是将 Hub 和连在 Hub 上的设备封装在一起所组成的设备。而 Composite Device 则是包含彼此独 立的多个接口的设备。从主机的角度看，一个 Compound Device 和单独的一个 Hub，然后连接 了多个 USB 设备是一样的，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 里不管有多少接口，它都只有一个地址



USB 大树要想茁壮成长，离不开 USB 协议。USB 总线是一种轮询式总线。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点或者接收点，要发送数据，只需把数据发送到正确的端点就可以了。 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字。 严格来说，管道的另一端应该是 USB 主机，USB 协议也是这么说的，协议说管道代表着在主 机和设备上的端点之间移动数据的能力。

端点不但是有方向的，而且这个方向还是确定的，要么是 in，要么是 out，没有既是 in 又 是 out 的，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，不是想要多少就有多少，除了端点 0，低速设备多只能拥有两个 端点，高速设备也多只能拥有 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 就特殊呢？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。协议中规定，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，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。

USB 端点有四种类型，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）和等时传输 （Isochronous Data Transfers）。

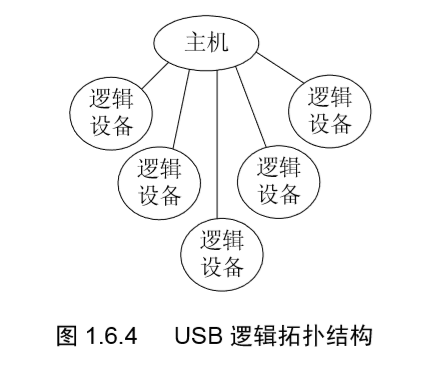
y 控制传输用来控制对 USB 设备不同部分的访问，通常用于配置设备，获取设备信息，发送 命令到设备，或者获取设备的状态报告。总之，就是用来传送控制信息的，每个 USB 设备 都会有一个名为“端点 0”的控制端点，内核中的 USB Core 使用它在设备插入时进行设备 的配置。

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 的触摸屏也是使用这种方式，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，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。

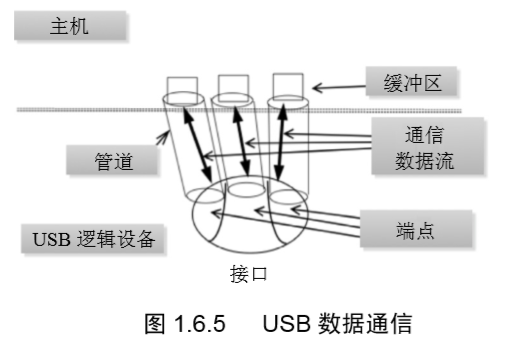
等时传输同样用来传输大量的数据，但并不保证数据是否到达，以稳定的速率发送和接收实 时的信息，对传送延迟非常敏感，显然是用于音频和视频一类的设备。这类设备期望能够有 一个比较稳定的数据流，比如在用 QQ 视频聊天时，肯定希望每分钟传输的图像/声音速率 是比较稳定的，不能说这一分钟前对方看到你在向她或向你深情表白，可是下一分钟却看见 画面停滞在那里，只能看到你在那里一动不动，这不是浪费感情吗？

如图 1.6.1 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 和设备都被看做是一个个的逻辑设备（Logical Device） ，如图 1.6.4 所示， 好像它们本来就直接连接在 Root Hub 上一样。



如图 1.6.5 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 1.6.5 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，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。



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 1.7.1 所示，就是该设备对应的 sysfs 目录树。

ag411:/sys/devices/platform # ls | grep usb

fe380000.usb

fe3a0000.usb

fe3c0000.usb

fe3e0000.usb

usb@fe800000

usb@fe900000



图 1.7.1 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，设备的目录下包括表示设备接口的 目录。目录里的各个文件表示设备或接口的描述，大都对应了设备描述符、接口描述符等相应 值，可以通过这些值获得您感兴趣的信息。什么是设备描述符和接口描述符？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，弄清 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 USB 2。 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/proc/bus/usb 目录中，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。

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。那么首先应该关注什么？那就是 Kconfig、Makefile、README

drivers/usb/README 文件描述了前面使用 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函数，并美其名曰 为“USB Core”。

USB Core：

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/目录下放了很多文件，USB Core 与其他各种设备驱动程序代码都 堆砌在这里，后来，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，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 目录下面负责各种 主机控制器的人去维护。

那么 USB gadget 呢？gadget 说白了就是配件的意思，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 制外围设备作为一个 USB 设备和主机通信。比如，我们的嵌入式主板上支持 SD 卡，如果我们 希望将主板通过 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 目录下大概可以分为两个模块：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，比如常见的支持 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。理论上讲，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驱动程 序来使用，对 U 盘的普及起了极大的推动作用，制定其他的 Class 也是同样的目的。

# 不一样的 Core@ lnx.ker.drivers.usb

## 设置

### usb\_device

#### devnum

**int** devnum;// 设备的地址, 设备的地址,。你的 usb 设备插到 hub 上时，hub 观 察到这个变化, 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。我的 usb 设备直接插到主机的 usb 接口上了, 没有用 hub，也总要明白主机里还会有个叫 root hub

**char** devpath[16];  
 u32 route;  
 **enum** usb\_device\_state state;  
 **enum** usb\_device\_speed speed;  
 **unsigned int** rx\_lanes;  
 **unsigned int** tx\_lanes;  
  
 **struct** usb\_tt \*tt;  
 **int** ttport;  
  
 **unsigned int** toggle[2];  
  
 **struct** usb\_device \*parent;  
 **struct** usb\_bus \*bus;  
 **struct** usb\_host\_endpoint ep0;  
  
 **struct** device dev;  
  
 **struct** usb\_device\_descriptor descriptor;  
 **struct** usb\_host\_bos \*bos;  
 **struct** usb\_host\_config \*config;  
  
 **struct** usb\_host\_config \*actconfig;  
 **struct** usb\_host\_endpoint \*ep\_in[16];  
 **struct** usb\_host\_endpoint \*ep\_out[16];  
  
 **char** \*\*rawdescriptors;  
  
 **unsigned short** bus\_mA;  
 u8 portnum;  
 u8 level;  
  
 **unsigned** can\_submit:1;  
 **unsigned** persist\_enabled:1;  
 **unsigned** have\_langid:1;  
 **unsigned** authorized:1;  
 **unsigned** authenticated:1;  
 **unsigned** wusb:1;  
 **unsigned** lpm\_capable:1;  
 **unsigned** usb2\_hw\_lpm\_capable:1;  
 **unsigned** usb2\_hw\_lpm\_besl\_capable:1;  
 **unsigned** usb2\_hw\_lpm\_enabled:1;  
 **unsigned** usb2\_hw\_lpm\_allowed:1;  
 **unsigned** usb3\_lpm\_u1\_enabled:1;  
 **unsigned** usb3\_lpm\_u2\_enabled:1;  
 **int** string\_langid;  
  
 */\* static strings from the device \*/* **char** \*product;  
 **char** \*manufacturer;  
 **char** \*serial;  
  
 **struct** list\_head filelist;  
  
 **int** maxchild;  
  
 u32 quirks;  
 atomic\_t urbnum;  
  
 **unsigned long** active\_duration;  
  
#ifdef CONFIG\_PM  
 **unsigned long** connect\_time;  
  
 **unsigned** do\_remote\_wakeup:1;  
 **unsigned** reset\_resume:1;  
 **unsigned** port\_is\_suspended:1;  
#endif  
 **struct** wusb\_dev \*wusb\_dev;  
 **int** slot\_id;  
 **enum** usb\_device\_removable removable;  
 **struct** usb2\_lpm\_parameters l1\_params;  
 **struct** usb3\_lpm\_parameters u1\_params;  
 **struct** usb3\_lpm\_parameters u2\_params;  
 **unsigned** lpm\_disable\_count;  
  
 u16 hub\_delay;

#### usb\_devmap

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，一个设备就是叶子一片。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，

*/\* USB device number allocation bitmap \*/***struct** usb\_devmap {  
 **unsigned long** devicemap[128 / (8\***sizeof**(**unsigned long**))];  
};

## 设备的生命线

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 usb.c 文件里定义

凡是你想用 kmalloc/memset 组合申请内存的时候，就使用 kzalloc 代替吧

## 31．驱动的生命线（一） ······················105

## 35．字符串描述符··································119

## 36．接口的驱动······································127

## 37．还是那个 match·······························129

## 38．结束语··············································134

# OTG

# 附录 A – sysfs

# 附录 B Linux 内核高效学习法

既然有高效，相对的就有低效。学习本身就是一件很玄乎的事情，有些人整天潇潇洒洒没 见怎么用心就能够获得很好的成绩，而有些人则相反，即使头悬梁锥刺骨也还是成绩平平收获 无几。这里面很大一部分的原因就是学习的方法

，因为基于每个人不同的情况，并没有那样一个标准的 方法存在，所以讲起来就很容易成为一场大忽悠。找到自己最为合适的方法才是最好的。很像一句句的口号，我们也可以将它们看做内核学习的大字 报。

## B.1 高效学习 Linux 内核

首先是第一句话：把内核当朋友。

我们只有把内核当朋友，把它放在对等的地位上，而不仅 仅是一堆死气沉沉的代码，我们才能够更好地认识和理解到它的精髓。

然后是第二句话：先学会使用它。 意思就是我们在学习内核前首先要会用 Linux，依照一个由上至下循序渐进的过程，在能 够熟练地使用 Linux 操作系统之后再去研究内核中的实现。这也是 linus 本人的观点。

第三句是依照四个层次进行内核学习。 笛卡儿在 17 世纪的某一天，闲极无聊写了这么一本书，书名就叫《方法论》，在这本目前 来说绝大部分人都不知道的书里将方法上升到了理论的高度。笛卡儿在他的这本书里将研究问 题的方法归纳为简单的一句话，就是“复杂问题要简单化”。就是说要将复杂的问题分解为很多 个简单的小问题，一个个地分开解决。这句话当然可以借鉴运用到内核的学习上，不过需要做 些改动，不是分解为多个简单的小问题，而是将内核学习这么一件很复杂的事情划分为由低到 高多个不同的层次，每一层次都有自己需要达到的目标和要求。这也是我自己认为比较好的认 识学习内核的方法。

第四句是走出心理误区。 对于学习这种复杂的事情来说，无论是我们在学校的课堂学习，还是这里说的内核学习， 它的效果好与坏，主要取决于两个方面：一个是学习的方法，另一个就是学习时的心理。注 意，在这儿我无视了智商的差异，智商这玩意儿太玄了，可以将它归于迷信的范畴。而我们在 学习时经常会产生一系列的问题或者说误区，只有走出这些误区，在学习中养成一个坚强的心 理，我们才能够真正做到高效。

第五句是使用 vim+cscope+ctags 浏览内核源码。 其实这句话更主要的意思是说我们需要一个好的工具去浏览内核的代码。在 Windows 下 面，我们或许可以很容易地找到很多比较好的 IDE，用来浏览代码，比如 source insight，它可 以很方便地在代码之间进行关联阅读。但是对于 Linux 新人来说，有没有一个功能类似的浏览 代码的工具就成为一个很常见的问题。

第六句是使用 kernel 地图定位目标代码。 应该说学习内核就是学习内核的源代码，但是内核代码千千万，又到处像个迷宫一样，不 迷路都很难，又怎么去直面它？这时我们就需要这样的一幅内核地图来帮助我们去定位要分析 的目标代码，并缩小目标代码的范围与代码量。

接下来是第七句话：分析内核源码，态度决定一切。 我们很多人或许有这样的困惑，也分析浏览了很多内核的源码，可总是觉得分析完浏览完 脑子里还是空空的，并没有感觉到多大的收获。这个时候我们或许可以去看看是不是自己在分 析代码时的态度出现了问题。我们在分析内核源码时，只有遵循严谨的态度，去理解每一段代 码的实现，多问多想多记，而不是抱着走马观花、得过且过的态度，终必然会有很大的收获。

后一句是：以内核源码为中心，坚持学习资源建设。 在我们内核学习的过程中，内核源码本身就是好的参考资料，其他任何经典或非经典的 书多只是起到个辅助作用，不能也不应该取代内核代码在我们学习过程中的主导地位。但是 这些辅助的作用也是不可忽视的，我们需要以内核源码为中心，坚持各种学习资源的长期建设 不动摇。 除了这里的八句话，其他的可能会对大家有帮助的感悟或者方法还有很多。

## B.2 Kernel 地图：Kconfig 与 Makefile

基本上， Linux 内核中每一个目录下边都会有一个 Kconfig 文件和一个 Makefile 文件。 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

去香港，通过海关的时候，总会有免费的地图和各种指南拿，有了它们在手里我们才 不至于无头苍蝇般迷茫地行走在陌生的街道上。即使在内地出去旅游的时候一般来说也总是会 首先找份地图，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， 只不 过有的特色是服务，有的特色是索取

比如我们打算研究 U 盘驱动的实现，因为 U 盘是一种 storage 设备，所以我们应该先进入 到 drivers/usb/storage/目录。但是该目录下的文件很多，那么究竟哪些文件才是我们需要关注 的？

## B.3 分析内核源码如何入手·················423

### B.3.1 分析 README····················423

### B.3.2 分析 Kconfig 和 Makefile····425

### B.3.3 态度决定一切：从初始化函数 开始······································427

## B.4 内核学习的心理问题·····················432

## B.5 高效学习 Linux 驱动开发··············433

## B.6 设备模型（上）·····························434

## B.7 设备模型（下）·····························438

### B.7.1 内核中 USB 子系统 的结构··································438

### B.7.2 USB 子系统与设备模型······440

## B.8 驱动开发三件宝·····························440