USB 协议有两种：USB1.1 和 USB2.0。USB2.0 和 USB1.1 完全兼容。USB1.1 支持的数据传输率为 12Mbps 和 1.5Mbps（用于慢速外设），USB2.0 支持的数据传速率可达 480Mbps。想要全面了解 USB 协议以及更多的相关文档，请访问站点

http://www.usb.org。
在普通用户看来，USB 系统就是外设通过一根 USB 电缆和 PC 机连接起来。通常把外设称为 USB 设备，把其所连接的 PC 机称为 USB 主机。将指向 USB 主机的数据传输方向称为上行通信，把指向 USB 设备的数据传输方向称为下行通信。

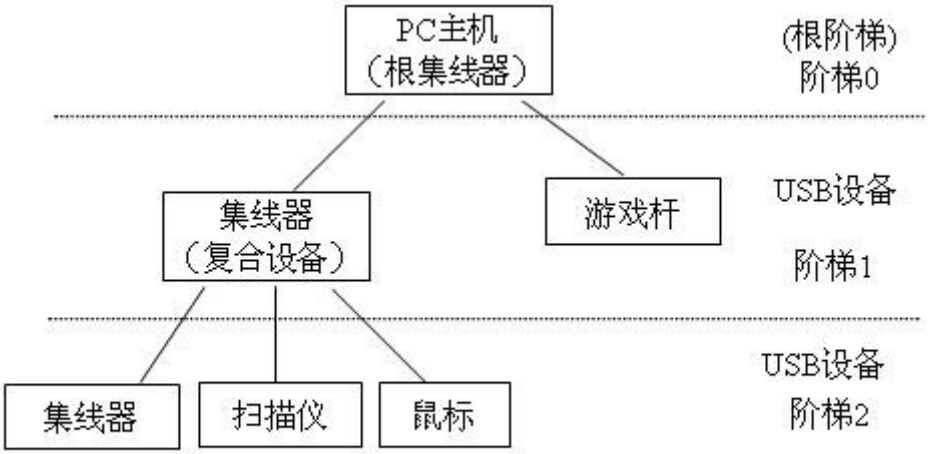

图16-1 USB主机和USB设备的连接

USB 网络采用阶梯式星形拓扑结构，如图 16-1。一个 USB 网络中只能有一个主机。主机内设置了一个根集线器，提供了主机上的初始附属点。这些集 线器成为网络中的节点，这些节点与设备或其他集线器连接，在 USB 通信中它们基本上是透明的，即集线器的存在并不会影响设备和主机之间的通信。集线器用来 扩展 USB 网络。比如，一个特定的计算机主机有 5 个 USB 端口，通过在主机的端口上连接集线器（每个集线器有多个端口），系统的物理连接能力就提高了。 USB 网络中，包括根集线器在内，最多允许连接 127 个外设和集线器。许多 USB 设备（比如键盘）都内置有集线器，既可实现其本身的功能又可进行扩展。

主机定时对集线器的状态进行查询。当一个新设备接入集线器时，主机会检测到集线器状态改变，主机发出一个命令使该端口有效并对其进行设置。位于这个端口上 的设备进行响应，主机收到关于设备的信息，主机的操作系统确定对这个设备使用那种驱动程序，接着设备被分配一个唯一标识的地址，范围从 0～127，其中 0 为所有的设备在没有分配惟一地址时使用的默认地址。主机向它发出内部设置请求。当一个设备从总线上移走时，主机就从其可用资源列表中将这个设备删除。

16.2.1 USB 主机

USB 的所有数据通信（不论是上行通信还是下行通信）都由 USB 主机启动，所以 USB 主机在整个数据传输过程中占据着主导地位。在 USB 系统中只允许有一个主机。从开发人员的角度看，USB 主机可分为三个不同的功能模块：客户软件、USB 系统软件和 USB 总线接口。

(1) 客户软件
客户软件负责和 USB 设备的功能单元进行通信，以实现其特定功能。一般由开发人员自行开发。客户软件不能直接访问 USB 设备，其与 USB 设备功能单元的通信必须经过 USB 系统软件和 USB 总线接口模块才能实现。客户软件一般包括 USB 设备驱动程序和界面应用程序两部分。

USB 设备驱动程序负责和 USB 系统软件进行通信。通常，它向 USB 总线驱动程序发出 I/O 请求包（IRP）以启动一次 USB 数据传输。此外，根据数据传输的方向，它还应提供一个或空或满的数据缓

冲区以存储这些数据。
界面应用程序负责和 USB 设备驱动程序进行通信，以控制 USB 设备。它是最上层的软件，只能看到向 USB 设备发送的原始数据和从 USB 设备接收的最终数据。

(2) USB 系统软件
USB 系统软件负责和 USB 逻辑设备进行配置通信，并管理客户软件启动的数据传输。USB 逻辑设备是程序员与 USB 设备打交道的部分。USB 系统软件一般包括 USB 总线驱动程序和 USB 主控制器驱动程序这两部分。这些软件通常由操作系统提供，开发人员不必掌握。

(3) USB 总线接口
USB 总线接口包括主控制器和根集线器两部分。根集线器为 USB 系统提供连接起点，用于给 USB 系统提供一个或多个连接点（端口）。主控制器负责完成主机 和 USB 设备之间数据的实际传输，包括对传输的数据进行串行编解码、差错控制等。该部分与 USB 系统软件的接口依赖于主控制器的硬件实现，开发人员不必掌 握。

16.2.2 USB 设备

在终端用户看来，USB 设备为主机提供了多种多样的附加功能，如文件传输、声音播放等，但对 USB 主机来说，它与所有 USB 设备的接口都是一致的。一个 USB 设备由三个功能模块组成：USB 总线接口、USB 逻辑设备和功能单元。这里的 USB 总线接口指的是 USB 设备中的串行接口引擎（SIE）；USB 逻 辑设备被 USB 系统软件看作是一个端点的集合；功能单元被客户软件看作是一个接口的集合。SIE、端点和接口都是 USB 设备的组成单元。

为了更好地描述 USB 设备的特征，USB 提出了设备架构的概念。从这个角度来看，可以认为 USB 设备是由一些配置、接口和端点组成的，即一个 USB 设备可 以含有一个或多个配置，在每个配置中可含有一个或多个接口，在每个接口中可含有若干个端点。其中，配置和接口是对 USB 设备功能的抽象，实际的数据传输由 端点来完成。在使用 USB 设备前，必须指明其采用的配置和接口。这个步骤一般是在设备接入主机时设备进行自检时完成的，我们在后面会进一步介绍。
USB 设备使用各种描述符来说明其设备架构，包括设备描述符、配置描述符、接口描述符、端点描述符以及字符串描述符，它们通常被保存在 USB 设备的固件程序中。

① 设备
设备代表一个 USB 设备，它由一个或多个配置组成。设备描述符用于说明设备的总体信息，并指明其所含的配置的个数。一个 USB 设备只能有一个设备描述符。
② 配置
一个 USB 设备可以包含一个或多个配置，如 USB 设备的低功耗模式和高功耗模式可分别对应一个配置。在使用 USB 设备前，必须为其选择一个合适的配置。配 置描述符用于说明 USB 设备中各个配置的特性，如配置所含接口的个数等。USB 设备的每一个配置都必须有一个配置描述符。
③ 接口
一个配置可以包含一个或多个接口，如对一个光驱来说，当用于文件传输时使用其大容量存储接口；而当用于播放 CD 时，使用其音频接口。接口是端点的集合，可 以包含一个或多个可替换设置，用户能够在 USB 处于配置状态时，改变当前接口所含的个数和特性。接口描述符用于说明 USB 设备中各个接口的特性，如接口所 属的设备类及其子类等。USB 设备的每个接口都必须有一个接口描述符。
④ 端点

端点是 USB 设备中的实际物理单元，USB 数据传输就是在主机和 USB 设备各个端点之间进行的。端点一般由 USB 接口芯片提供，例如 Freescale 的 MC68HC908JB8。USB 设备中的每一个端点都有唯一的端点号，每个端点所支持的数据传输方向一般而言也是确定的：或是输入（IN）或是输出（OUT），也有些芯片提供的端点的数据方向是可以配置的，例如 MC68HC908JB8 包含有两个用于数据收发的端点：端点 1 和端点 2。其中端点 1 只能 用于数据发送，即支持输入（IN），端点 2 既能用于数据发送也可用于数据接收，即支持输入（IN）和输出（OUT）操作。需要注意的是，在这里数据的传输 方向是站在主机的立场上来看的。比如端点 1 只能发送数据，在主机看来是端点 1 向主机输

入数据，即 IN 操作；当端点 2 配置为接收数据时，主机向端点 2 输出数 据，即 OUT 操作。这一点是初学者比较容易产生混淆的地方。

利用设备地址、端点号和传输方向就可以指定一个端点，并和它进行通信。端点的传输特性还决定了其与主机通信时所采用的传输类型，如控制端点只能使用控制传输。根据端点的不同用途，可将端点分为两类：0 号端点和非 0 号端点。

0 号端点比较特殊，它有数据输入 IN 和数据输出 OUT 两个物理单元，且只能支持控制传输。所有的 USB 设备都必须含有一个 0 号端点，用作缺省控制 管道。USB 系统软件就是使用该管道和 USB 逻辑设备进行配置通信的。0 号端点在 USB 设备上电以后就可以使用，而非 0 号端点必须要在配置以后才可以使 用。

根据具体应用的需要，USB 设备还可以含有多个除 0 号端点以外的其他端点。对于低速设备，其附加的端点数最多为 2 个；对于全速/高速设备，其附加的端点数最多为 15 个。

⑤ 字符串
在 USB 设备中通常还含有字符串描述符，以说明一些专用信息，如制造商的名称、设备的序列号等。它的内容以 UNICODE 的形式给出，且可以被客户软件所读取。对 USB 设备来说，字符串描述符是可选的。
⑥ 管道
在 USB 系统结构中，可以认为数据传输是在主机软件（USB 系统软件或客户软件）和 USB 设备的各个端点之间直接进行的，它们之间的连接称为管道。管道是 在 USB 设备的配置过程中建立的。管道是对主机和 USB 设备间通信流的抽象，它表示主机的数据缓冲区和 USB 设备的端点之间存在着逻辑数据传输，而实际的 数据传输是由 USB 总线接口层来完成的。
管道和 USB 设备中的端点一一对应。一个 USB 设备含有多少个端点，其和主机进行通信时就可以使用多少条管道，且端点的类型决定了管道中数据的传输类型， 如中断端点对应中断管道，且该管道只能进行中断传输。传输类型在后面会介绍。不论存在着多少条管道，在各个管道中进行的数据传输都是相互独立的。

16.3.1 USB 接口

USB 使用一根屏蔽的 4 线电缆与网络上的设备进行互联。数据传输通过一个差分双绞线进行，这两根线分别标为 D+和 D-，另外两根线是 Vcc 和 Ground，其中 Vcc 向 USB 设备供电。使用 USB 电源的设备称为总线供电设备，而使用自己外部电源的设备叫做自供电设备。为了避免混淆，USB 电缆中的线都用不同的颜色标记，如表 16-1 所示。

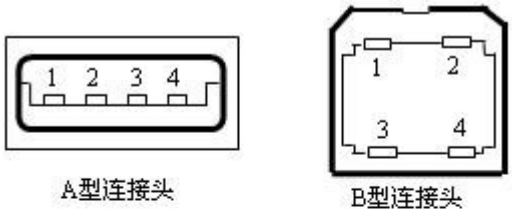

图16-2 USB连接头

表16-1 USB缆线的信号与颜色

引脚编号	信号名称	缆线颜色
1	Vcc	红
2	Data-(D-)	白
3	Data+(D+)	绿
4	Ground	黑

引脚编号 信号名称 缆线颜色

1 Vcc 红
2 Data-(D-) 白
3 Data+(D+) 绿
4 Ground 黑

从一个设备连回到主机，称为上行连接；从主机到设备的连接，称为下行连接。为了防止回环情况的发生，上行和下行端口使用不同的连接器，所以 USB 在电缆和 设备的连接中分别采用了两种类型的连接头，即图 16-2 所示的 A 型连接头和 B 型连接头。每个连接头内的电线号与图 16-2 的引脚编号是一致的，请读者对照 阅读。A 型连接头，用于上行连接，即在主机或集线器上有一个 A 型插座，而在连接到主机或集线器的电缆的一端是 A 型插头。在 USB 设备上有 B 型插座，而 B 型 插头在从主机或集线器接出的下行电缆的一端。采用这种连接方式，可以确保 USB 设备、主机/集线器和 USB 电缆始终以正确的方式连接，而不可能出现电缆接 入方式出错，或直接将两个 USB 设备连接到一起的情况。

16.3.2 USB 信号

数据在 USB 总线上实际传输时，使用的是 NRZI（反向不归零）编码的差分信号，这种信号有利于保证数据的完整性和消除噪声干扰。

1．差分信号技术
我们知道，传统的传输方式大多使用“正信号”或者“负信号”二进制表达机制，这些信号利用单线传输。用不同的信号电平范围来分别表示 1 和 0，它们之间有一 个临界值，如果在数据传输过程中受到中低强度的干扰，高低电平不会突破临界值，那么信号传输可以正常进行。但如果遇到强干扰，高低电平突破临界值，由此造 成数据传输出错。一般说来，总线频率越高，线路间的电磁干扰就越厉害，数据传输失败的发生机率也就越高。因此这种信号表达技术无法应用于高速总线传输，而 差分信号技术能有效克服这种缺点。

差分信号技术最大的特点是：必须使用两条线路才能表达一个比特位，用两条线路传输信号的压差作为判断 1 还是 0 的依据。这种做法的优点是具有极强的抗干扰 性。倘若遭受外界强烈干扰，两条线路对应的电平同样会出现大幅度提升或降低的情况，但二者的电平改变方向和幅度几乎相同，电压差值就可始终保持相对稳定， 因此数据的准确性并不会因干扰噪声而有所降低。当然，由于 1 个比特位需要两条线路，在总线宽度相等的条件下，差分技术需要的信号线条数就是“正/负信号” 技术的两倍。

2．**USB** 编码格式
USB 的数据包使用反向不归零编码（NRZI）。图 16-3 描述了在 USB 电缆段上传输信息的步骤。反向不归零编码由传送信息的 USB 代理程序完成；然 后，被编码的数据通过差分驱动器送到 USB 电缆上；接着，接收器将输入的差分信号进行放大，将其送给解码器。使用该编码和差动信号传输方式可以更好地保证 数据的完整性并减少噪声干扰。

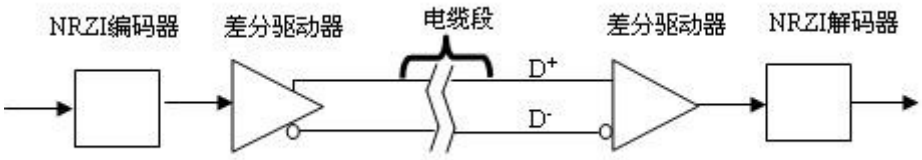

图16-3 在USB电缆上使用双向不归零编码和差动信号的传输

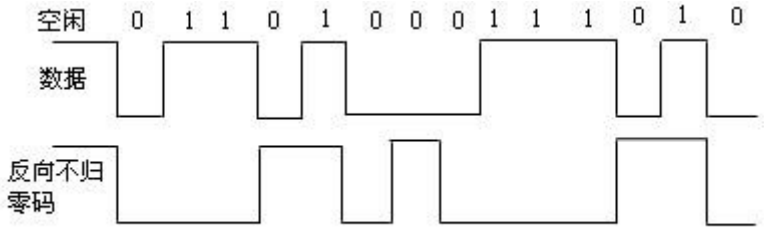

图16-4 反向不归零编码

使用反向不归零编码方式可以保证数据传输的完整性，而且不要求传输过程中有独立的时钟信号。反向不归零编码不是一个新的编码方式。它在许多方面都有应 用。图 16-4 给出了一个数据流和编码之后的结果。在反向不归零编码时，遇到“0”转换，遇到“1”保持。反向不归零码必须保持与输入数据的同步性，以确 保数据采样正确。反向不归零码数据流必须在一个数据窗口被采样，无论前一个位时间是否发生过转换。解码器在每个位时间采样数据以检查是否有转换。

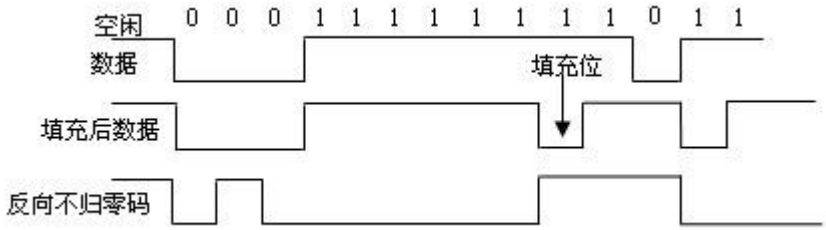

图16-5 在USB电缆上使用反向不归零编码和差动信号的传输

但这样的编码方式会遇到一个很严重的问题，就是若重复相同的“1”信号一直进入时，就会造成数据长时间无法转换，逐渐的积累，而导致接收器最终丢失同步信 号的状况，使得读取的时序会发生严重的错误。因此，在 NRZI 编码之前，还需执行所谓的位填充的工作。位填充要求数据流中如果有连续的六个“1”就要强行 转换。这样接收器在反向不归零码数据流中最多每七个位就检测到一次跳转。这样就保证了接收器与输入数据流保持同步。反向不归零码的发送器要把“0”（填充 位）插到数据流中。接收器必须被设计成能够在连续的六个“1”之后识别一个自动跳转，并且立即扔掉这六个“1”之后的“0”位。

图 16-5 的第一行是送到接收器的原始数据。注意数据流包括连续的八个“1”。第二行表示对原始数据进行了位填充，在原始的第六个和第七个“1”之间填入 了一个“0”。第七个“1”延时一个位时间让填充位插入。接收器知道连续六个“1”之后将是一个填充位，所以该位就要被忽略。注意，如果原始数据的第七个位是“0”，填充位也同样插入，在填充过的数据流中就会有两个连续的“0”。

16.3.3 检测设备连接和速度

在 USB 设备连接时，USB 系统能自动检测到这个连接，并识别出其采用的数据传输速率。USB 采用在 D+或 D-线上增加上拉电阻的方法来识别低速和全速设 备。USB 支持三种类型的传输速率：1.5Mb/s 的低速传输、12Mb/s 的全速传输和 480Mb/s 的高速传输。如图 16-6 和图 16-7 所示。

图16-6 低速USB设备电缆和电阻的连接

图16-7 全速USB设备电缆和电阻的连接

当主控制器或集线器的下行端口上没有 USB 设备连接时，其 D+和 D-线上的下拉电阻使得这两条数据

线的电压都是近地的（0V）；当低速/全速设备连接以 后，电流流过由集线器的下拉电阻和设备在D+/D-的上拉电阻构成的分压器。由于下拉电阻的阻值是15KΩ，上拉电阻的阻值是1.5KΩ，所以在D+ /D-线上会出现大小为（Vcc*15/（15+1.5））的直流高电平电压。当USB主机探测到D+/D-线的电压已经接近高电平，而其它的线保持接地 时，它就知道全速/低速设备已经连接了。
高速设备在连接起始时需要以全速速率与主机进行通信，以完成其配置操作，这时需要在D+线上连接1.5KΩ的上拉电阻。当高速设备正常工作时，如果采用高 速传输的话，D+线不可上拉；但如果仍采用全速传输，则在D+线上必须使用上拉电阻。所以，为识别出高速设备，需要在上拉电阻和D+线之间连接一个由软件 控制的开关，它通常被集成在USB设备接口芯片的内部。

USB事务处理是主机和设备之间数据传输的基本单位，由一系列具有特定格式的信息包组成。因此，要了解完整的USB通信协议，必须从USB的信息传输单元包及其数据域谈起。通过由下而上，从简单至复杂的通信协议单位组成各种复杂的通信协议，进而构建出完整的通信协议。

16.4.1 包

包（Packet）是USB系统中信息传输的基本单元，所有数据都是经过打包后在总线上传输的。首先了解一下包的组成。
USB包由五部分组成，即同步（SYNC）字段、包标识符（PID）字段、数据字段、循环冗余校验（CRC）字段和包结尾（EOP）字段，包的基本格式如下：

同步字段（SYNC） PID字段 数据字段 CRC字段 包结尾字段（EOP）
在USB的数据传输中，所有的传输包都起始于SYNC，接着是PID，后面是包中所包含的数据信息，接下来是用来检测包中数据错误的循环冗余校验信息，最后以包结尾作为结束标志。下面我们将一一介绍每个字段。

1．同步（**SYNC**）字段
SYNC字段由8位组成，作为每个数据信息包的前导。顾名思义，它是用来产生同步作用的，目的是使USB设备与总线的包传输率同步，它的数值固定为00000001。
2．包标识符（**PID**）字段
PID字段是紧随在SYNC字段后面，用来表示数据信息包的类型。在USB协议中，根据PID的不同，USB包有着不同的类型，分别表示具有特定的意义。如下所示：
PID0 PID1 PID2 PID3

包标识符长度为一个字节（8个数据位），由4个位的包类型字段和4个位的校验字段构成。PID是USB包类型的唯一标志，USB主机和USB设备在接收到 包后，必须首先对包标识符解码得到包的类型，并判断其意义从而做出下一个反应。包标识符中的校验字段是通过对类型字段的每个位求反码产生的，它是用来对包 类型字段进行错误检测用的，旨在保证对包的标识符译码的可靠性，如果4个检验位不是它们各自的类型位的反码，则说明标识符中的信息有错误。
表16-2中列出了信息包的类型，包括令牌、数据、握手或特殊四种信息包类型。为简化对USB的认识，有关高速传输的部分没有在表中列出。

表16-2 各种信息包的类型与规范

信息包类型	PID名称	PID编码	意义
令牌	OUT	0001B	从主机到设备的数据传输
	IN	1001B	从设备到主机的数据传输
	SOF	0101B	帧的起始标记与帧码
	SETUP	1101B	从主机到设备，表示要进行控制传输
数据	DATA0	0011B	偶数数据信息包
	DATA1	1011B	奇数数据信息包
握手	ACK	0010B	接收器收到无错误的数据信息包
	NAK	1010B	接收器无法接收数据或发射器无法送出数据
	STALL	1110B	端点产生停滞的状况
特殊	PRE	1100B	使能下游端口的USB总线的数据传输切换到低速的设备

3．数据字段

在 USB 包中，数据字段是用来携带主机与设备之间要传递的信息，其内容和长度根据包标识符、传输类型的不同而各不相同。并非所有的 USB 包都必须有数据字 段，例如握手包、专用包和 SOF 令牌包就没有数据字段。在 USB 包中，数据字段可以包含设备地址、端点号、帧序列号以及数据等内容。在总线传输中，总是首 先传输字节的最低位，最后传输字节的最高位。

(1) 设备地址（**ADDR**）数据域

ADDR 数据域由 7 位组成，可用来寻址多达 127 个外围设备。

(2) 端点（**ENDP**）数据域

ENDP 数据域由 4 位组成。通过这 4 个位最多可寻址出 32 个端点。这个 ENDP 数据域仅用在 IN、OUT 与 SETUP 令牌信息包中。对于慢速设备可支持端 点 0 以及端点 1 作为中断传输模式，而全速设备则可以拥有 16 个输入端点（IN）与 16 个输出端点（OUT）共 32 个端点。

(3) 帧序列号

当 USB 令牌包的 PID 为 SOF 时，其数据字段必须为 11 位的帧序列号。帧序列号由主机产生，且每个数据帧自动加一，最大数值为 0x7FF。当帧序列号达到最大数时将自动从 0 开始循环。

(4) 数据

它仅存于 DATA 信息包内，根据不同的传输类型，拥有不同的字节大小，从 0 到 1023 字节（实时传输）。

4．循环冗余校验（**CRC**）字段

根据不同的信息包类型，CRC 数据域由不同数目的位所组成。其中重要的数据信息包采用 CRC16 的数据域（16 个位），而其余的信息包类型则采用 CRC5 的数据域（5 个位）。其中的循环冗余码校验 CRC，是一种错误检测技术。由于数据在传输时，有时候会发生错误，因此 CRC 可根据数据算出一个校验值，然后 依此判断数据的正确性。

5．包结尾（**EOP**）字段

包的发送方在包的结尾发出包结尾信号。它表现为差分线路的两根数据线保持 2 比特低位时间和 1 比特空闲位时间。USB 主机根据 EOP 判断数据包的结束。

16.4.2 信息包格式

根据信息包所实现的功能，其可以分为 3 种类型：令牌包、数据包和握手包。其中，令牌包定义了数据传输的类型，数据包中含有需要传输的数据，握手包指明了数据接收是否成功。

1．令牌（**token**）包

在 USB 系统中，只有主机才能发出令牌包。令牌包定义了数据传输的类型，它是事务处理的第一阶段。令牌包格式如下：

8 位 8 位 7 位 4 位 5 位

SYNC PID
ADDR ENDP CRC5
令牌包中较为重要的是 SETUP、IN 和 OUT 这三个令牌包。它们用来在根集线器和设备端点之间建立数据传输。一个 IN 包用来建立一个从设备到根集线器的 数据传送，一个 OUT 包用来建立从根集线器到设备的数据传输。IN 包和 OUT 包可以对任何设备上的任何端点编址。一个 SETUP 包是一个 OUT 包的特殊情 形，它是“高优先级的”，也就是说设备必须接受它，即使设备正在进行数据传输操作的过程中也要对其进行响应。SETUP 包总是指向端点 0 的。

2．数据（**data**）包
一个数据信息包包含了 4 个数据域：SYNC、PID、DATA 与 CRC16。在这里要注意的是 DATA 数据域内所放置的位值，需根据 USB 设备的传输速度（慢速、高速与全速）以及传输类型（中断传输、批量传输、等时传输）而定，且须以 8 字节为基本单位。也就是，若传输的数据不足 8 字节，或传输到最后所剩余 的也不足 8 字节，仍须传输 8 字节的数据域。格式如下：
8 位 8 位 0~1023 字节 16 位
SYNC PID
DATA CRC16

3．握手（**Handshake**）包
握手信息包是最简单的信息包类型。在这个握手信息包中仅包含一个 PID 数据域而已，它的格式如下所列：
8 位 8 位
SYNC PID
其中仅包含 SYNC 与 PID 两个数据域。

16.4.3 事务

在 USB 上数据信息的一次接收或发送的处理过程称为事务处理（Transaction）。事务处理的类型包括输入(IN)事务处理、输出(OUT)事务处 理、设置(SETUP)事务处理和帧开始、帧结尾等类型。在输出(OUT)事务处理和设置(SETUP)事务处理中，紧接着 SETUP 和 OUT 包后的是 DATA 包，DATA0 和 DATA1 包是交替地发送的，在 DATA 包后面，设备将回应一个握手信号，如果设备可以接收数据，就回应 ACK 包，如果设备忙， 就回应 NAK 包，如果设备出错，则回应 STALL 包；在 IN 事务中，IN 包后面是设备发来的 DATA 包或 NAK 包或 STALL 包，若设备忙或出错，就发 NAK 包或 STALL 包给主机，若设备数据准备好发送，则发 DATA 包，DATA0 和 DATA1 包也是交替地发送的，紧接着 DATA 包后面是主机发给设备 的握手包，ACK 表示主机可以接收数据，NAK 包代表主机忙，STALL 包代表主机出错。下面我们再分别介绍这些事务。

1．输入（**IN**）事务处理
输入事务处理表示 USB 主机从总线上的某个 USB 设备接收一个数据包的过程，接下来分析输入事务处理的各种典型情况：
(1) 正常的输入事务处理
1．主机->设备(令牌信息包) SYNC IN ADDR ENDP CRC5
2．设备->主机(数据信息包) SYNC DATA0 DATA CRC16
3．主机->设备(握手信息包) SYNC ACK
(2) 设备忙时的输入事务处理
1．主机->设备(令牌信息包) SYNC IN ADDR ENDP CRC5
2．设备->主机(握手信息包) SYNC NAK
(3) 设备出错时的输入事务处理
1．主机->设备(令牌信息包) SYNC IN ADDR ENDP CRC5
2．设备->主机(握手信息包)
SYNC STALL

2．输出（**OUT**）事务处理

(1) 正常的输出事务处理
1．主机->设备(令牌信息包) SYNC OUT ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC ACK
(2) 设备忙时的输出事务处理
1．主机->设备(令牌信息包) SYNC OUT ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC NAK
(3) 设备出错时的输入事务处理
1．主机->设备(令牌信息包) SYNC OUT ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC STALL

3．设置（**SETUP**）事务处理
(1) 正常的设置事务处理
1．主机->设备(令牌信息包) SYNC SETUP ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC ACK
(2) 设备忙时的设置事务处理
1．主机->设备(令牌信息包) SYNC SETUP ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC NAK
(3) 设备出错时的设置事务处理
1．主机->设备(令牌信息包) SYNC SETUP ADDR ENDP CRC5
2．主机->设备(数据信息包) SYNC DATA0 DATA CRC16
3．设备->主机(握手信息包) SYNC STALL

16.4.4 USB 传输类型

在 USB 的传输中，制定了 4 种传输类型：控制传输、中断传输、批量传输以及实时传输。这里只详细介绍控制传输，其他传输类型只作简要说明。

1．控制传输
控制传输是 USB 传输中最重要的传输，唯有正确地执行完控制传输，才能进一步正确地执行其他传输模式。
由于每个 USB 设备可能速度、传输的包的大小等信息有可能不同，因此每个 USB 设备内部都记录着该设备的一些信息（也就是接下来将要介绍的设备描述符）， 当在主机上检测到 USB 设备时，系统软件必须读取设备描述符，以确定该设备的类型和操作特性，以及对该设备进行相应的配置，这些工作都是通过控制传输来完 成。每个 USB 设备都必须实现一个缺省的控制端点，该端点总是 0 号端点。

控制传输类型分为 2 ~ 3 个阶段：设置阶段、数据阶段（无数据控制没有此阶段）以及状态阶段。根据数据阶段的数据传输的方向，控制传输又可分为 3 种类型：控制读取（读取 USB 描述符）、控制写入（配置 USB 设备）以及无数据控制。以下介绍各阶段的工作。

阶段一：设置阶段
USB 设备在正常使用之前，必须先配置，本阶段由主机将信息传送给 USB 设备，定义对 USB 设备的请求信息（如：读设备描述符）。主机一般会从 USB 设备获取配置信息后再确定此设备有哪些功能。作为配置的一部分，主机会设置设备的配置值。我们统称这一阶段为设置阶段。

设置阶段由设置事务完成，也就是该阶段包含了 SETUP 令牌信息包、紧随其后的 DATA0 数据信息包（该信息包里的数据即为设备请求，本章将后续介绍）以及 ACK 握手信息包。它的作用是执行一个

设置的数据交换，并定义此控制传输的内容。
阶段二：数据传输阶段

数据传输阶段是用来传输主机与设备之间的数据。

图16-8 控制读取过程示意图

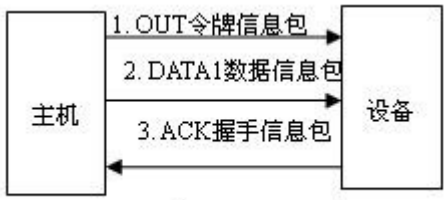

图16-9 控制写入过程示意图

控制读取是将数据从设备读到主机上，读取的数据 USB 设备描述符。该过程如图 16-8 所示。对每一个数据信息包而言，首先，主机会发送一个 IN 令牌信息 包，表示要读数据进来。然后，设备将数据通过 DATA1 数据信息包回传给主机。最后，主机将以下列的方式加以响应：当数据已经正确接收时，主机送出 ACK 令牌信息包；当主机正在忙碌时，发出 NAK 握手信息包；当发生了错误时，主机发出 STALL 握手信息包。

控制写入则是将数据从主机传到设备上，所传的数据即为对 USB 设备的配置信息，该过程如图 16-9 所示。对每一个数据信息包而言，主机将会送出一个 OUT 令牌信息包，表示数据要送出去。紧接着，主机将数据通过 DATA0 数据信息包传递至设备。最后，设备将以下列方式加以响应：当数据已经正确接收时，设备送 出 ACK 令牌信息包；当设备正在忙碌时，设备发出 NAK 握手信息包；当发生了错误时，设备发出 STALL 握手信息包。

阶段三：状态阶段
状态阶段用来表示整个传输的过程已经完全结束了。请注意，状态阶段传输的方向必须与数据阶段的方向相反。即原来是 IN 令牌信息包，这个阶段应为 OUT 令牌 信息包；反之，原来是 OUT 令牌信息包，这个阶段应为 IN 令牌信息包。对于控制读取而言，主机会送出 OUT 令牌信息包，其后再跟着 0 长度的 DATA1 信息 包。而此时，设备也会做出相对应的动作，送 ACK 握手信息包、NAK 握手信息包或 STALL 握手信息包。相对地对于控制写入传输，主机会送出 IN 令牌信息 包，然后设备送出表示完成状态阶段的 0 长度的 DATA1 信息包，主机再做出相对应的动作：送 ACK 握手信息包、NAK 握手信息包或 STALL 握手信息包。

2．实时传输
实时传输适用于必须以固定速率抵达或在指定时刻抵达，可以容忍偶尔错误的数据上。实时传输一般用于麦克风、喇叭等设备。
实时传输只需令牌与数据两个信息包阶段，没有握手包，故数据传错时不会重传。

3．批量传输
用于传输大量数据，要求传输不能出错，但对时间没有要求，适用于打印机、存储设备等。

4．中断传输
中断传输方式总是用于对设备的查询，以确定是否有数据需要传输。因此中断传输的方向总是从 USB 设备到主机。

16.4.5 设备列举

1．描述符
USB 描述符就好像是 USB 外围设备的“身份证”一样，详细地记录着外围设备相关的一切信息。为了描述不同的数据，就需以不同类型的 USB 描述符来加以描 述，它共有以下几种类型：设备描述符、配置描述符、接口描述符和端点描述符，这几个描述符是必须具有的，下面将结合实例详细介绍；其他的描述符，例如，字 符串描述符、数种不同的群组描述符以及报告描述符则可以根据不同的设备进行选择。

(1) 设备描述符

设备描述符具有 18 字节的长度，并且是主机向设备请求的第一个描述符。以下列出设备描述符的范例、数值以及各个字段的意义：

```
DeviceDesc: ;表示设备描述符
.byte $12；bLength 域，表示该描述符的字节长度为 18 个字节
.byte $01；bDescriptorType 域描述符类型，1 代表设备
.byte $10,$01；bcdUSB 域表示符合 USB 1.10 规范，$210 代表 2.10 规范
.byte $00；bDeviceClass 域，群组码，0 表示每个接口都有自身的群信息，
;不同的接口操作相互独立
.byte $00；bDeviceSubClass 域，设备次群组（如果群组码为 0，设备次群组必须为 0）
.byte $00；bDeviceProtocol 域，设备协议（0 表示无群组特定协议）
.byte $08；bMaxPacketSize0 域，端点 0 的最大信息包大小（只能为 8、16、32 或 64）
.byte $70,$0C；idVendor 域，制造商 ID（2 字节）
.byte $00,$00；idProduct 域，产品 ID（2 字节）
.byte $00,$01；bcdDevice 域，以 BCD 表示设备发行序号（2 字节）
.byte $01；iManufacturer 域，制造商的字符串描述符索引（1 字节）,本实例中指
; String1Desc 处的字符串描述符
.byte $02；iProduct 域，产品的字符串描述符索引（1 字节），本实例中指
; String2Desc 处的字符串描述符
.byte $00；iSerialNumber 域，设备序号的字符串描述符索引（1 字节），
;本实例中指 String0Desc 处的字符串描述符
.byte $01；bNumConfigurations 域，配置数目为 1
```

(2) 配置描述符

配置描述符具有 9 字节的长度，针对设备给予配置的信息。以下列出配置描述符的范例、数值以及各个字段的意义：

```
ConfigDesc: ;表示配置描述符
.byte $09；bLength 域，表示该描述符的字节长度为 9 个字节（1 字节）
.byte $02；bDescriptorType 域，描述符类型，2 代表配置（1 字节）
.byte $20,$00；wTotalLength 域，描述符的总长度为 32 个字节，（包括配置描述符
; 9 字节，接口描述符 9 字节，两个端点描述符各 7 字节）
.byte $01；bNumInterfaces 域，该配置支持的接口数目（1 字节）
.byte $01；bConfigurationValue 域，配置值,作为 Set Configuration 请求的配置值
.byte $00；iConfiguration 域，配置的字符串描述符的索引（1 字节）
.byte $C0；bmAttributes 域，配置的属性（自供电，不具有远程唤醒的特征）
.byte $00；MaxPower 域，表示当 USB 设备操作时，它从总线上
;获得的最大电源（以 2mA 为单位）
```

(3) 接口描述符

接口描述符具有 9 字节的长度，用来描述每一个设备的接口特性。以下列出接口描述符的范例、数值以及各个字段的意义：

```
InterfaceDesc: ;表示接口描述符
.byte $09；bLength 域，表示该描述符的字节长度为 9 个字节
.byte $04；bDescriptorType 域，描述符类型，4 代表接口
.byte $00；bInterfaceNumber 域，接口数目以 0 为基值（表示一个接口）
.byte $00；bAlternateSetting 域，交互设置值为 0（因为只有一个接口）
.byte $02；bNumEndpoints 域，端点数目设置为 2
.byte $FF；bInterfaceClass 域，接口群组，$FF 表示是供应商说明书
.byte $01；bInterfaceSubClass 域，接口次群组
.byte $FF；bInterfaceProtocol 域，接口协议，$FF 表示该接口使用的
;是供应商说明的协议
```

.byte $00；iInterface 域，接口的字符串描述符的索引，本实例没有

(4) 端点描述符

端点描述符具有 7 字节长度，用来描述端点的属性以及各个端点的位置。该实例中有两个端点，我们首先介绍端点 1 描述符：

```
Endpoint1Desc: ;表示端点 1 描述符
.byte $07；bLength 域，表示该描述符的字节长度为 7 个字节
.byte $05；bDescriptorType 域，描述符类型，5 代表端点
.byte $81；bEndpointAddress 域，端点地址（[0x81 = IN,0x02=OUT]），
;本实例端点编号为 1 且为 IN 端点
.byte $03；bmAttributes 域，传输类型的属性设置为中断传输
;（0 = 控制，1 = 实时，2 = 批量，3 = 中断）
.byte $08,$00；wMaxPacketSize 域，最大信息包的大小设置为 8 个字节
.byte $0A；bInterval 域，轮询间隔，以 ms 为单位，在此设置为 10ms
```

接下来介绍端点 2 描述符：

```
Endpoint2Desc: ;表示端点 2 描述符
.byte $07；bLength 域，表示该描述符的字节长度为 7 个字节
.byte $05；bDescriptorType 域，描述符类型，5 代表端点
.byte $02；bEndpointAddress 域，端点地址（[0x81 = IN,0x02=OUT]），
;本实例端点编号为 2 且为 OUT 端点
.byte $03；bmAttributes 域，传输类型的属性设置为中断传输
;（0 = 控制，1 = 实时，2 = 批量，3 = 中断）
.byte $08,$00；wMaxPacketSize 域，最大信息包的大小设置为 8 个字节
.byte $0A；bInterval 域，轮询间隔，以 ms 为单位，在此设置为 10ms
```

2．USB 设备请求

在 USB 接口的通信协议中，由于主机是取得绝对的主控权，因此，主机与设备之间就必须遵循某种已沟通的特定命令格式，以达到通信的目的。而这个命令格式就 是 USB 规范书中所制定的“设备请求”。这个设备请求的设置、清除与取得都须通过控制传输的数据交换来完成。表 16-3 中列出了标准设备请求的数据格式内 容。

表16-3 数据请求的数据格式内容

位移量	字段值	大小(字节)	描述
0	bmRequestType	1	D7-数据传输方向：0= 主机至设备，1= 设备至主机 D[6:5]-类型：0= 标准，1= 群组，2= 供应商 3= 保留 D[4:0]-接收端：0= 设备，1= 接口，2= 端点，3= 其他，
1	bRequest	1	特定请求
2	wValue	2	传递一个参数给设备
4	wIndex	2	指定一个接口或端点
6	wLength	2	如果有数据阶段，该域表示所要传输的字节个数

表 16-3 各字段含义解释如下。

bmRequestType 域决定了特定请求的特征，该域的 D7 表示在控制传输的数据阶段，数据传输的方向。如果 wLength 的值为 0，表示没有数据阶 段，该位可以忽略；D[6:5]表示了该请求所属的类型，USB 标准中定义了所有的 USB 设备必须支持的一系列的标准请求，此外，群组和供应商也可以定义一些其他的请求；D[4:0]表示接收端，请求可以针对设备、接口或设备的一个端点，当针对一个接口或端点时，wIndex 域决定了是哪个接口或端点。

bRequest 域表示特定请求，如果 bmRequestType 域的类型字段为 0，可以根据 USB 指定的一系列的标准请求，结合该域的值，知道这是什么请求。将在下面介绍这些请求。

wValue 域的值根据请求的不同而不同，用来传递一个参数给设备。
wIndex 域的值根据请求的不同而不同，用来指定一个接口或端点。
wLength 域表示控制传输的数据阶段中传输数据的字节大小，数据传输的方向由 bmRequestType 域中的 D7 位决定，如果该域的值为 0，表示没有数据阶段。

下面结合实例分别介绍几个主要的设备请求。

(1) 清除特性(Clear Feature)

该请求是用来取消一个特性，其格式如下：
bmRequestType bRequest wValue wIndex wLength Data
00000000B(设备)
00000001B(接口)
00000010B(端点) CLEAR_FEATURE（01H）特性选择 0
接口
端点 0 无
该请求中的 wValue 表示特性选择器，它对应的值为：0 = 端点，1 = 设备。当某个特性不允许取消，或该特性根本不存在，或者是指向一个根本不存在的接口 或端点时，该请求将会导致设备请求失败。如果端点被固件设为停止状态，主机软件(总线驱动程序)也可以发送一个值为 0 的 CLEAR_FEATURE 命令清 除该端点的停止状态，本实例中就是这样使用该请求的。

(2) 取得描述符(Get Descriptor)

该请求可以取得 USB 设备中存在的特定的描述符，其格式如下：
bmRequestType bRequest wValue wIndex wLength Data
10000000B GET_DESCRIPTOR
（06H）描述符类型与描述符指针 0 或语言 ID 描述符的长度 各个描述符
该请求中的 wValue 的高字节表示要取得描述符类型，低字节表示描述符的索引值，描述的类型有：1 表示设备描述符，2 表示配置描述符，3 表示字符串描述 符，4 表示接口描述符，5 表示端点描述符。wIndex 的值为 0 或语言 ID；当要取得描述符是字符串描述符时，该域的值为语言 ID；当为其他的描述符时， 该域为 0。wLength 表示要返回的数据长度，如果 SETUP 阶段的地址使用的是预设地址 0（ENDP 字段为 0），这时的 wLength 值会大于实际的 描述的值。这是为什么呢？原因是用户以预设的地址 0 来取得设备描述符时，不管设多少字节，用户最多只取其前 8 字节，即在控制传输过程只有一次数据阶段。但 是，如果用户以新的地址（ENDP 字段不为 0）来取得设备描述符时，这时 wLength 的值就要注意了。

(3) 设置地址(Set Address)

该请求给 USB 设备设置地址，从而可以对该 USB 设备进行进一步的访问。其格式如下：
bmRequestType bRequest wValue wIndex wLength Data
00000000B SET_ADDRESS（05H）设备地址 0 0 无
该请求与其他的请求有一个重要的不同点，该请求下，USB 设备一直不改变它的地址，直到该请求的状态阶段被成功地完成，而其他请求的操作都是在状态阶段之 前完成，可以阅读本实例加深对该点的理解。若特定的设备地址大于 127，或者 wIndex 或 wLength 为非 0 值，那么该请求不执行。

(4) 设置配置(Set Configuration)

该请求对设备进行设置。其格式如下：
bmRequestType bRequest wValue wIndex wLength Data
00000000B SET_CONFIGURATION（09H）设置值 0 0 无
该请求中的 wValue 域的低字节表示设置的值，该值必须为 0 或者与配置描述符中的配置值相匹配。如果设置值等于 0，表示设备在地址状态。如果 wIndex 或 wLength 为非 0 值，那么该请求不执行。
另外还有其他的请求，这里不再详细介绍，读者可以参考相关的资料。
在学习了描述符和 USB 设备请求的基础上，接下来就能进行设备列举了。

3. 设备列举

设备列举可以简单地概括为这样的一个过程：主机通过 USB 设备请求来取得设备描述符并对该设备进

行配置。该过程可以简化为如下 5 个步骤：
第一步，使用预设的地址 0 取得设备描述符。
第二步，设置设备的新地址。
第三步，使用新地址取得设备描述符。
第四步，取得配置描述符。
第五步，设置配置描述符。
设备列举使用的是控制传输。上述的 5 个步骤必须符合控制传输的基本架构，第一步、第三步和第四步使用的是控制读取，第二步和第五步使用的是无数据控制。

USB 子系统

内核使用 2.6.29.4

拓扑结构上，一个 USB 子系统并不是以总线的方式来分布；它是一棵由几个点对点连接构成的树。这些连接是连接设备和集线器的 4 线电缆(地，电源，和 2 个差分信号线)，如同以太网的双绞线。USB 主控制器负责询问每个 USB 设备是否有数据需要发送。
由 于这个拓扑结构，一个 USB 设备在没有主控制器要求的情况下不能发送数据. 也就是说：USB 是单主方式的实现，主机轮询各外设。但是设备也可以要求一个固定的数据传输带宽，以保证可靠的音视频 I/O。USB 只作为数据传输通道， 对他所收发的数据格式没有特殊的内容和结构上的要求，也就是类似于透传。

Linux 内核支持两种主要类型的 USB 驱动程序：Host 系统上的驱动程序（USB device driver）和 device 上的驱动程序（USB gadget driver）（设备端驱动）。
USB 驱动程序存在于不同的内核子系统和 USB 硬件控制器之中。USB 核心为 USB 驱动程序提供了一个用于访问和控制 USB 硬件的接口，它隐藏了 USB 控制器的 硬件细节。从这里我们要知道：《LDD3》所谓的 USB 驱动是针对 USB 核心提供的接口而写的，并不是真正去操纵 USB 硬件控制器中的寄存器。这样你必须 保证你的板子上 CPU 的 USB 硬件控制器的驱动是可用的。否则您就得先搞定 CPU 的 USB 硬件控制器的驱动才行。

以下是 Linux 内核中 USB 驱动的软件构架：

如左下图所示，从主机侧的观念去看，在 Linux 驱动中，USB 驱动处于最底层的是 USB 主机控制器硬件，在其之上运行的是 USB 主机控制器驱动， 主机控制器之上为 USB 核心层，再上层为 USB 设备驱动层（插入主机上的 U 盘、鼠标、USB 转串口等设备驱动）。因此，在主机侧的层次结构中，要实现的 USB 驱动包括两类：USB 主机控制器驱动和 USB 设备驱动，前者控制插入其中的 USB 设备，后者控制 USB 设备如何与主机通信。Linux 内核 USB 核 心负责 USB 驱动管理和协议处理的主要工作。主机控制器驱动和设备驱动之间的 USB 核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备 驱动提供编程接口，向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息；完成设备热插拔控制、总线数据传输控制等。

如右下图所示，Linux 内核中 USB 设备侧驱动程序分为 3 个层次：UDC 驱动程序、Gadget API 和 Gadget 驱动程序。UDC 驱动程序直接访问硬件，控制 USB 设备和主机间的底层通信，向上层提供与硬件相关操作的回调函数。当前 Gadget API 是 UDC 驱动程序回调函数的简单包装。Gadget 驱动程序具体控制 USB 设备功能的实现，使设备表现出“网络连接”、“打印机”或“USB Mass Storage”等特性，它使用 Gadget API 控制 UDC 实现上述功能。Gadget API 把下层的 UDC 驱动程序和上层的 Gadget 驱动程序隔离开，使得在 Linux 系统中编写 USB 设备侧驱动程序时能够把功能的实现和底层通信分离。

《LDD3》中的 USB 驱动的介绍分以下几块：
－－－－－－－－－－－－基础知识部分－－－－－－－－－－－－－－－－
（1）USB 设备基础
端点
接口
配置
（2）USB 和 sysfs
（3）USB urb
struct urb
创建和销毁 urb
中断 urb
批量 urb
控制 urb
等时 urb
提交 urb
结束 urb：结束回调处理例程
取消 urb
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

－－－－－－－－－驱动编写部分（一）－－－－－－－－－－－－－－－－
（4）编写 USB 驱动程序
驱动支持什么设备
注册 USB 驱动程序
探测和断开的细节
提交和控制 urb
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

－－－－－－－－－驱动编写部分（二）－－－－－－－－－－－－－－－－
（5）不使用 urb 的 USB 传输
usb_bulk_msg
usb_control_msg
（6）其他 USB 数据函数
－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－－

USB 设备其实很复杂，但是 Linux 内核提供了一个称为 USB core 的子系统来处理了大部分的复杂工作，所以这里所描述的是驱动程序和 USB core 之间的接口。

在 USB 设备组织结构中，从上到下分为设备（device）、配置（config）、接口（interface）和端点（endpoint）四个层次。
对于这四个层次的简单描述如下：
设备通常具有一个或多个的配置

配置经常具有一个或多个的接口
接口通常具有一个或多个的设置
接口没有或具有一个以上的端点

设备

很明显，地代表了一个插入的 USB 设备，在内核使用数据结构 struct usb_device 来描述整个 USB 设备。（include/linux/usb.h）

```
   struct usb_device {
    int devnum; //设备号，是在 USB 总线的地址
    char devpath [16]; //用于消息的设备 ID 字符串
    enum usb_device_state state; //设备状态：已配置、未连接等等
    enum usb_device_speed speed; //设备速度：高速、全速、低速或错误

    struct usb_tt *tt; //处理传输者信息；用于低速、全速设备和高速 HUB
    int ttport; //位于 tt HUB 的设备口

    unsigned int toggle[2]; //每个端点的占一位，表明端点的方向([0] =
IN, [1] = OUT)
    struct usb_device *parent; //上一级 HUB 指针
    struct usb_bus *bus; //总线指针
    struct usb_host_endpoint ep0; //端点 0 数据

    struct device dev; //一般的设备接口数据结构

    struct usb_device_descriptor descriptor; //USB 设备描述符
    struct usb_host_config *config; //设备的所有配置

    struct usb_host_config *actconfig; //被激活的设备配置
    struct usb_host_endpoint *ep_in[16]; //输入端点数组
    struct usb_host_endpoint *ep_out[16]; //输出端点数组

    char **rawdescriptors; //每个配置的 raw 描述符

    unsigned short bus_mA; //可使用的总线电流


    u8 portnum;//父端口号
    u8 level; //USB HUB 的层数

    unsigned can_submit:1; //URB 可被提交标志
    unsigned discon_suspended:1; //暂停时断开标志
    unsigned persist_enabled:1; //USB_PERSIST 使能标志
    unsigned have_langid:1; //string_langid 存在标志
    unsigned authorized:1;
    unsigned authenticated:1;
    unsigned wusb:1; //无线 USB 标志

    int string_langid; //字符串语言 ID
```

```
    /* static strings from the device */ //设备的静态字符串
    char *product; //产品名
    char *manufacturer; //厂商名
    char *serial; //产品串号

    struct list_head filelist; //此设备打开的 usbfs 文件

   #ifdef CONFIG_USB_DEVICE_CLASS
    struct device *usb_classdev; //用户空间访问的为 usbfs 设备创建的 USB
类设备
   #endif
   #ifdef CONFIG_USB_DEVICEFS
    struct dentry *usbfs_dentry; //设备的 usbfs 入口
   #endif

    int maxchild; //（若为 HUB）接口数
    struct usb_device *children[USB_MAXCHILDREN];//连接在这个 HUB
上的子设备
    int pm_usage_cnt; //自动挂起的使用计数
    u32 quirks;
    atomic_t urbnum; //这个设备所提交的 URB 计数

    unsigned long active_duration; //激活后使用计时

   #ifdef CONFIG_PM //电源管理相关
    struct delayed_work autosuspend; //自动挂起的延时
    struct work_struct autoresume; //（中断的）自动唤醒需求
    struct mutex pm_mutex; //PM 的互斥锁


    unsigned long last_busy; //最后使用的时间
    int autosuspend_delay;
    unsigned long connect_time; //第一次连接的时间

    unsigned auto_pm:1; //自动挂起/唤醒
    unsigned do_remote_wakeup:1; //远程唤醒
    unsigned reset_resume:1; //使用复位替代唤醒
    unsigned autosuspend_disabled:1; //挂起关闭
    unsigned autoresume_disabled:1; //唤醒关闭
    unsigned skip_sys_resume:1; //跳过下个系统唤醒
   #endif
    struct wusb_dev *wusb_dev; //（如果为无线 USB）连接到 WUSB 特定的数据
结构

   };
```

配置

一个 USB 设备可以有多个配置，并可在它们之间转换以改变设备的状态。比如一个设备可以通过下载固件（firmware）的方式改变设备的使用状态（我感觉 类似 FPGA 或 CPLD），那么 USB 设备就

要切换配置，来完成这个工作。一个时刻只能有一个配置可以被激活。Linux 使用结构 struct usb_host_config 来描述 USB 配置。我们编写的 USB 设备驱动通常不需要读写这些结构的任何值。可在内核源码的文件 include/linux/usb.h 中找到对它们的 描述。

```
struct usb_host_config {
struct usb_config_descriptor desc; //配置描述符

char *string; /* 配置的字符串指针（如果存在） */
struct usb_interface_assoc_descriptor
*intf_assoc[USB_MAXIADS]; //配置的接口联合描述符链表
struct usb_interface *interface[USB_MAXINTERFACES]; //接口描述符链
表
struct usb_interface_cache *intf_cache[USB_MAXINTERFACES];
unsigned char *extra; /* 额外的描述符 */
int extralen;
};
```

接口

USB 端点被绑为接口，USB 接口只处理一种 USB 逻辑连接。一个 USB 接口代表一个基本功能，每个 USB 驱动控制一个接口。所以一个物理上的硬件设备可能需要 一个以上的驱动程序。这可以在“晕到死 差屁”系统中看出，有时插入一个 USB 设备后，系统会识别出多个设备，并安装相应多个的驱动。USB 接口可以有其他的设置,它是对接口参数的不同选择. 接口的初始化的状态是第一个设置,编号为 0。 其他的设置可以以不同方式控制独立的端点。

USB 接口在内核中使用 struct usb_interface 来描述。USB 核心将其传递给 USB 驱动，并由 USB 驱动负责后续的控制。

```
struct usb_interface {
struct usb_host_interface *altsetting; /* 包含所有可用于该接口的可选设
置的接口结构数组。每个 struct usb_host_interface 包含一套端点配置（即
struct usb_host_endpoint 结构所定义的端点配置。这些接口结构没有特别的顺序。*/

struct usb_host_interface *cur_altsetting; /* 指向 altsetting 内部
的指针，表示当前激活的接口配置*/
unsigned num_altsetting; /* 可选设置的数量*/

/* If there is an interface association descriptor then it
will list the associated interfaces */
struct usb_interface_assoc_descriptor *intf_assoc;

int minor; /* 如果绑定到这个接口的 USB 驱动使用 USB 主设备号，这个变量包含由
USB 核心分配给接口的次设备号. 这只在一个成功的调用 usb_register_dev 后才有效。
*/
/*以下的数据在我们写的驱动中基本不用考虑，系统会自动设置*/
enum usb_interface_condition condition; /* state of binding */
unsigned is_active:1; /* the interface is not suspended */
```

```
unsigned sysfs_files_created:1; /* the sysfs attributes exist
*/
unsigned ep_devs_created:1; /* endpoint "devices" exist */
unsigned unregistering:1; /* unregistration is in progress */
unsigned needs_remote_wakeup:1; /* driver requires remote
wakeup */
unsigned needs_altsetting0:1; /* switch to altsetting 0 is
pending */
unsigned needs_binding:1; /* needs delayed unbind/rebind */
unsigned reset_running:1;

struct device dev; /* 接口特定的设备信息 */
struct device *usb_dev;
int pm_usage_cnt; /* usage counter for autosuspend */
struct work_struct reset_ws; /* for resets in atomic context
*/
};

struct usb_host_interface {
struct usb_interface_descriptor desc; //接口描述符

struct usb_host_endpoint *endpoint; /* 这个接口的所有端点结构体的联合数
组*/
char *string; /* 接口描述字符串 */
unsigned char *extra; /* 额外的描述符 */
int extralen;
};
```

端点

USB 通讯的最基本形式是通过一个称为端点的东西。一个 USB 端点只能向一个方向传输数据（从主机到设备(称为输出端点)或者从设备到主机(称为输入端点)）。端点可被看作一个单向的管道。

一个 USB 端点有 4 种不同类型，分别具有不同的数据传送方式：

控制 CONTROL

控 制端点被用来控制对 USB 设备的不同部分访问. 通常用作配置设备、获取设备信息、发送命令到设备或获取设备状态报告。这些端点通常较小。每个 USB 设备都有一个控制端点称为"端点 0"，被 USB 核心用来在插入时配置设备。USB 协议保证总有足够的带宽留给控制端点传送数据到设备.

中断 INTERRUPT

每当 USB 主机向设备请求数据时，中断端点以固定的速率传送小量的数据。此为 USB 键盘和鼠标的主要的数据传送方法。它还用以传送数据到 USB 设备来控制设备。通常不用来传送大量数据。USB 协议保证总有足够的带宽留给中断端点传送数据到设备.

批量 **BULK**

批 量端点用以传送大量数据。这些端点常比中断端点大得多. 它们普遍用于不能有任何数据丢失的数据。USB 协议不保证传输在特定时间范围内完成。如果总线上没有足够的空间来发送整个 BULK 包，它被分为多个包进行传输。这些端点普遍用于打印机、USB Mass Storage 和 USB 网络设备上。

等时 **ISOCHRONOUS**

等时端点也批量传送大量数据，但是这个数据不被保证能送达。这些端点用在可以处理数据丢失的设备中，并且更多依赖于保持持续的数据流。如音频和视频设备等等。

控制和批量端点用于异步数据传送，而中断和同步端点是周期性的。这意味着这些端点被设置来在固定的时间连续传送数据，USB 核心为它们保留了相应的带宽。

端点在内核中使用结构 struct usb_host_endpoint 来描述，它所包含的真实端点信息在另一个结构中：struct usb_endpoint_descriptor（端点描述符，包含所有的 USB 特定数据）。

```
struct usb_host_endpoint {
struct usb_endpoint_descriptor desc; //端点描述符
struct list_head urb_list; //此端点的 URB 对列，由 USB 核心维护
void *hcpriv;
struct ep_device *ep_dev; /* For sysfs info */
unsigned char *extra; /* Extra descriptors */
int extralen;
int enabled;
};

/*-------------------------------------------------------------------------

/* USB_DT_ENDPOINT: Endpoint descriptor */
struct usb_endpoint_descriptor {
__u8 bLength;
__u8 bDescriptorType;

__u8 bEndpointAddress; /*这个特定端点的 USB 地址，这个 8 位数据包含端点的方向，结合位掩码
USB_DIR_OUT 和 USB_DIR_IN 使用，确定这个端点的数据方向。*/

__u8 bmAttributes; //这是端点的类型，位掩码如下

__le16 wMaxPacketSize; /*端点可以一次处理的最大字节数。驱动可以发送比这个值大的数据量到端点，
是当真正传送到设备时，数据会被分为 wMaxPakcetSize 大小的块。对于高速设备，通过使用高位部分几个额
位，可用来支持端点的高带宽模式。*/
__u8 bInterval; //如果端点是中断类型，该值是端点的间隔设置，即端点的中断请求间的间隔时间，以毫秒
单位

/* NOTE: these two are _only_ in audio endpoints. */
/* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
__u8 bRefresh;
__u8 bSynchAddress;
```

```
} __attribute__ ((packed));

#define USB_DT_ENDPOINT_SIZE 7
#define USB_DT_ENDPOINT_AUDIO_SIZE 9 /* Audio extension */

/*
* Endpoints
*/
#define USB_ENDPOINT_NUMBER_MASK 0x0f /* in bEndpointAddress 端点的 USB 地址掩
*/
#define USB_ENDPOINT_DIR_MASK 0x80 /* in bEndpointAddress 数据方向掩码 */

#define USB_DIR_OUT 0 /* to device */
#define USB_DIR_IN 0x80 /* to host */

#define USB_ENDPOINT_XFERTYPE_MASK 0x03 /* bmAttributes 的位掩码*/
#define USB_ENDPOINT_XFER_CONTROL 0
#define USB_ENDPOINT_XFER_ISOC 1
#define USB_ENDPOINT_XFER_BULK 2
#define USB_ENDPOINT_XFER_INT 3
#define USB_ENDPOINT_MAX_ADJUSTABLE 0x80
/*-------------------------------------------------------------------------
```

USB 和 sysfs

由于单个 USB 物理设备的复杂性，设备在 sysfs 中的表示也非常复杂。**物理 USB 设备(通过 struct usb_device 表示)和单个 USB 接口(由 struct usb_interface 表示)都作为单个设备出现在 sysfs 中，这是因为这两个结构都包含一个 struct device 结构。**以下内容是我的 USB 鼠标在 sysfs 中的目录树：

```
/sys/devices/pci0000:00/0000:00:1a.0/usb3/3-1 （表示 usb_device
结构）
.
|-- 3-1:1.0 （鼠标所对应的usb_interface）
| |-- 0003:046D:C018.0003
| | |-- driver ->
../../../../../../../bus/hid/drivers/generic-usb
| | |-- power
| | | `-- wakeup
| | |-- subsystem -> ../../../../../../../bus/hid
| | `-- uevent
| |-- bAlternateSetting
| |-- bInterfaceClass
| |-- bInterfaceNumber
| |-- bInterfaceProtocol
| |-- bInterfaceSubClass
| |-- bNumEndpoints
| |-- driver -> ../../../../../../bus/usb/drivers/usbhid
```

```
|  |-- ep_81 -> usb_endpoint/usbdev3.4_ep81
|  |-- input
|  |  `-- input6
|  |  |-- capabilities
|  |  |  |-- abs
|  |  |  |-- ev
|  |  |  |-- ff
|  |  |  |-- key
|  |  |  |-- led
|  |  |  |-- msc
|  |  |  |-- rel
|  |  |  |-- snd
|  |  |  `-- sw
|  |  |-- device -> ../../../3-1:1.0
|  |  |-- event3
|  |  |  |-- dev
|  |  |  |-- device -> ../../input6
|  |  |  |-- power
|  |  |  |  `-- wakeup
|  |  |  |-- subsystem -> ../../../../../../../../../class/input
|  |  |  `-- uevent
|  |  |-- id
|  |  |  |-- bustype
|  |  |  |-- product
|  |  |  |-- vendor
|  |  |  `-- version
|  |  |-- modalias
|  |  |-- mouse1
|  |  |  |-- dev
|  |  |  |-- device -> ../../input6
|  |  |  |-- power
|  |  |  |  `-- wakeup
|  |  |  |-- subsystem -> ../../../../../../../../../class/input
|  |  |  `-- uevent
|  |  |-- name
|  |  |-- phys
|  |  |-- power
|  |  |  `-- wakeup
|  |  |-- subsystem -> ../../../../../../../../../class/input
|  |  |-- uevent
|  |  `-- uniq
|  |-- modalias
|  |-- power
|  |  `-- wakeup
|  |-- subsystem -> ../../../../../../../bus/usb
|  |-- supports_autosuspend
|  |-- uevent
|  `-- usb_endpoint
|  `-- usbdev3.4_ep81
|  |-- bEndpointAddress
|  |-- bInterval
```

```
|  |-- bLength
|  |-- bmAttributes
|  |-- dev
|  |-- device -> ../../../3-1:1.0
|  |-- direction
|  |-- interval
|  |-- power
|  |  `-- wakeup
|  |-- subsystem -> ../../../../../../../../class/usb_endpoint
|  |-- type
|  |-- uevent
|  `-- wMaxPacketSize
|-- authorized
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPacketSize0
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- busnum
|-- configuration
|-- descriptors
|-- dev
|-- devnum
|-- driver -> ../../../../../bus/usb/drivers/usb
|-- ep_00 -> usb_endpoint/usbdev3.4_ep00
|-- idProduct
|-- idVendor
|-- manufacturer
|-- maxchild
|-- power
|  |-- active_duration
|  |-- autosuspend
|  |-- connected_duration
|  |-- level
|  |-- persist
|  `-- wakeup
|-- product
|-- quirks
|-- speed
|-- subsystem -> ../../../../../bus/usb
|-- uevent
|-- urbnum
|-- usb_endpoint
|  `-- usbdev3.4_ep00
|  |-- bEndpointAddress
|  |-- bInterval
```

```
|  |-- bLength
|  |-- bmAttributes
|  |-- dev
|  |-- device -> ../../../3-1
|  |-- direction
|  |-- interval
|  |-- power
|  |  `-- wakeup
|  |-- subsystem -> ../../../../../../../class/usb_endpoint
|  |-- type
|  |-- uevent
|  `-- wMaxPacketSize
`-- version

38 directories, 91 files
```

USB 鼠标驱动

USB 鼠标及 input 子系统

鼠标的返回的最大数据包为 4 个字节，

第 0 个字节：bit 0、1、2、3、4 分别代表左、右、中、SIDE、EXTRA 键的按下情况

第 1 个字节：表示鼠标的水平位移

第 2 个字节：表示鼠标的垂直位移

第 3 个字节：REL_WHEEL 位移

Linux 有自己的 input 子系统，可以统一管理鼠标和键盘事件。

基于输入子系统 实现的 uinput 可以方便的在用户空间模拟鼠标和键盘事件。

当然，也可以自己造轮子， 做一个字符设备接收用户输入，根据输入，投递 input 事件。

还有一种方式就是直接 往 evnent 里写入数据， 都可以达到控制鼠标键盘的功能。

linux/input.h 中有定义，这个文件还定义了标准按键的编码等

```
struct input_event {
    struct timeval time;  //按键时间
    __u16 type; //类型，在下面有定义
    __u16 code; //要模拟成什么按键
    __s32 value;//是按下还是释放
};
```

code：

事件的代码.如果事件的类型代码是 EV_KEY,该代码 code 为设备键盘代码.代码植 0~127 为键盘上的按键代码, 0x110~0x116 为鼠标上按键代码,其中 0x110(BTN_ LEFT) 为鼠标左键,0x111(BTN_RIGHT)为鼠标右键,0x112(BTN_ MIDDLE) 为鼠标中键.其它代码含义请参看 include/linux /input.h 文件. 如果事件的类型代码是 EV_REL,code 值表示轨迹的类型.如指示鼠标的 X 轴方向 REL_X (代码为 0x00),指示鼠标的 Y 轴方向 REL_Y(代码为 0x01),指示鼠标中轮子方向 REL_WHEEL(代码为 0x08).

type:

EV_KEY,键盘

EV_REL,相对坐标

EV_ABS,绝对坐标

value：

事件的值.如果事件的类型代码是 EV_KEY,当按键按下时值为 1,松开时值为 0;

如果事件的类型代码是 EV_ REL,value 的正数值和负数值分别代表两个不同方向的值.

==========

USB 总线引出两个重要的链表！

一个 USB 总线引出两个重要的链表，一个为 USB 设备链表，一个为 USB 驱动链表。设备链表包含各种系统中的 USB 设备以及这些设备的所有接口，驱动链表包含 USB 设备驱动程序（usb device driver）和 USB 驱动程序（usb driver）。

USB 设备驱动程序（usb device driver）和 USB 驱动程序（usb driver）的区别是什么？

USB 设备驱动程序包含 USB 设备的一些通用特性，将与所有 USB 设备相匹配。在 USB core 定义了：struct usb_device_driver usb_generic_driver。usb_generic_driver 是 USB 子系统中唯一的一个设备驱动程序对象。而 USB 驱动程序则是与接口相匹配，接口是一个完成特定功能的端点的集合。

设备是如何添加到设备链表上去的？

在设备插入 USB 控制器之后，USB core 即会将设备在系统中注册，添加到 USB 设备链表上去。

USB 设备驱动程序（usb device driver）是如何添加到驱动链表上去的？

在系统启动注册 USB core 时，USB 设备驱动程序即将被注册，也就添加到驱动链表上去了。

接口是如何添加到设备链表上去的？

在 USB 设备驱动程序和 USB 设备的匹配之后，USB core 会对设备进行配置，分析设备的结构之后会将设备所有接口都添加到设备链表上去。比如鼠标设备中有一个接口，USB core 对鼠标设备配置后，会将这个接口添加到设备链表上去。

USB 驱动程序（usb driver）是如何添加到驱动链表上去的？

在每个 USB 驱动程序的被注册时，USB 驱动程序即会添加到驱动链表上去。比如鼠标驱动程序，usb_mouse_init 函数将通过 usb_register(&usb_mouse_driver) 将鼠标驱动程序注册到 USB core 中，然后就添加到驱动链表中去了。其中 usb_mouse_driver 是描述鼠标驱动程序的结构体。

已配置状态（configured status）之后话

当鼠标的设备、接口都添加到设备链表，并且鼠标驱动程序也添加到驱动链表上去了， 系统就进入一种叫做已配置（configured）的状态。要达到已配置状态，将经历复杂的过程，USB core 为 USB 设备奉献着无怨无悔。在这个过程中，系统将会建立起该设备的的设备、配置、接口、设置、端点的描述信息，它们分别被 usb_device、usb_configuration、usb_interface、usb_host_interface、usb_host_endpoint 结构体描述。

设备达到已配置状态后，首先当然就要进行 USB 驱动程序和相应接口的配对，对于鼠标设备来说则是鼠标驱动程序和鼠标中的接口的配对。USB core 会调用 usb_device_match 函数，通过比较设备中的接口信息和 USB 驱动程序中的 id_table，来初步决定该 USB 驱动程序是不是跟相应接口相匹配。通过这一道关卡后，USB core 会认为这个设备应该由这个驱动程序负责。

然而，仅仅这一步是不够的，接着，将会调用 USB 驱动程序中的 probe 函数对相应接口进行进一步检查。如果该驱动程序确实适合设备接口，对设备做一些初始化工作，分配 urb 准备数据传输。

当鼠标设备在用户空间打开时，将提交 probe 函数构建的 urb 请求块，urb 将开始为传送数据而忙碌了。urb 请求块就像一个装东西的“袋子”，USB 驱动程序把“空袋子”提交给 USB core，然后再交给主控制器，主控制器把数据放入这个“袋子”后再将装满数据的“袋子”通过 USB core 交还给 USB 驱动程序，这样一次数据传输就完成了。

参考 2.6.14 版本中的 driver/usb/input/usbmouse.c。鼠标驱动可分为几个部分：驱动加载部分、probe 部分、open 部分、urb 回调函数处理部分。

```
/*
* $Id: usbmouse.c,v 1.15 2001/12/27 10:37:41 vojtech Exp $
*
*  Copyright (c) 1999-2001 Vojtech Pavlik
*
*  USB HIDBP Mouse support
*/
#include <linux/kernel.h>
```

```
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/usb/input.h>
#include <linux/hid.h>
/*
* Version Information
*/
#define DRIVER_VERSION "v1.6"
#define DRIVER_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"
#define DRIVER_DESC "USB HID Boot Protocol mouse driver"
#define DRIVER_LICENSE "GPL"
MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE(DRIVER_LICENSE);
/*
* 鼠标结构体，用于描述鼠标设备。
*/
struct usb_mouse
{
/* 鼠标设备的名称，包括生产厂商、产品类别、产品等信息 */
char name[128];
/* 设备节点名称 */
char phys[64];
/* USB 鼠标是一种 USB 设备，需要内嵌一个 USB 设备结构体来描述其 USB 属性 */
```

```
struct usb_device *usbdev;
/* USB 鼠标同时又是一种输入设备，需要内嵌一个输入设备结构体来描述其输入设备的属性 */
struct input_dev *dev;
/* URB 请求包结构体，用于传送数据 */
struct urb *irq;
/* 普通传输用的地址 */
signed char *data;
/* dma 传输用的地址 */
dma_addr_t data_dma;
};
/*
* urb 回调函数，在完成提交 urb 后，urb 回调函数将被调用。
* 此函数作为 usb_fill_int_urb 函数的形参，为构建的 urb 制定的回调函数。
*/
static void usb_mouse_irq(struct urb *urb)
{
/*
* urb 中的 context 指针用于为 USB 驱动程序保存一些数据。比如在这个回调函数的形参为 mouse 结构体分配的那块内存的地址指针，而又需要用到那块内存区域中的数据，context 指针则帮了大忙了！
* 在填充 urb 时将 context 指针指向 mouse 结构体数据区，在这又创建一个局部 mouse 指针指向在 probe 函数中为 mouse 申请的那块内存，那块内存保存着非常重要数据。
* 当 urb 通过 USB core 提交给 hc 之后，如果结果正常，mouse->data 指向的内存区域将保存着鼠标的按键和移动坐标信息，系统则依靠这些信息对鼠标的行为作出反应。
* mouse 中内嵌的 dev 指针，指向 input_dev 所属于的内存区域。
*/
```

```
struct usb_mouse *mouse = urb->context;
signed char *data = mouse->data;
struct input_dev *dev = mouse->dev;
int status;
/*
* status 值为 0 表示 urb 成功返回，直接跳出循环把鼠标事件报告给输入子系统。
* ECONNRESET 出错信息表示 urb 被 usb_unlink_urb 函数给 unlink 了，ENOENT 出错信息表示 urb 被 usb_kill_urb 函数给 kill 了。usb_kill_urb 表示彻底结束 urb 的生命周期，而 usb_unlink_urb 则是停止 urb，这个函数不等 urb 完全终止就会返回给回调函数。这在运行中断处理程序时或者等待某自旋锁时非常有用，在这两种情况下是不能睡眠的，而等待一个 urb 完全停止很可能会出现睡眠的情况。
* ESHUTDOWN 这种错误表示 USB 主控制器驱动程序发生了严重的错误，或者提交完 urb 的一瞬间设备被拔出。
* 遇见除了以上三种错误以外的错误，将申请重传 urb。
*/
switch (urb->status)
{
case 0:     /* success */
break;
case -ECONNRESET:    /* unlink */
case -ENOENT:
case -ESHUTDOWN:
return;
/* -EPIPE:  should clear the halt */
default:        /* error */
goto resubmit;
}
```

```
/*
* 向输入子系统汇报鼠标事件情况，以便作出反应。
* data 数组的第 0 个字节：bit 0、1、2、3、4 分别代表左、右、中、SIDE、EXTRA 键的按下情况；
* data 数组的第 1 个字节：表示鼠标的水平位移；
* data 数组的第 2 个字节：表示鼠标的垂直位移；
* data 数组的第 3 个字节：REL_WHEEL 位移。
*/
input_report_key(dev, BTN_LEFT,   data[0] & 0x01);
input_report_key(dev, BTN_RIGHT,  data[0] & 0x02);
input_report_key(dev, BTN_MIDDLE, data[0] & 0x04);
input_report_key(dev, BTN_SIDE,   data[0] & 0x08);
input_report_key(dev, BTN_EXTRA,  data[0] & 0x10);
input_report_rel(dev, REL_X,     data[1]);
input_report_rel(dev, REL_Y,     data[2]);
input_report_rel(dev, REL_WHEEL, data[3]);
/*
* 这里是用于事件同步。上面几行是一次完整的鼠标事件，包括按键信息、绝对坐标信息和滚轮信息，输入子系统正是通过这个同步信号来在多个完整事件报告中区分每一次完整事件报告。示意如下：
* 按键信息 坐标位移信息 滚轮信息 EV_SYC | 按键信息 坐标位移信息 滚轮信息 EV_SYC ...
*/
input_sync(dev);
/*
* 系统需要周期性不断地获取鼠标的事件信息，因此在 urb 回调函数的末尾再次提交 urb 请求块，这样又会调用新的回调函数，周而复始。
```

```
* 在回调函数中提交 urb 一定只能是 GFP_ATOMIC 优先级的，因为 urb 回调函数运行于中断上下文中，在提交 urb 过程中可能会需要申请内存、保持信号量，这些操作或许会导致 USB core 睡眠，一切导致睡眠的行为都是不允许的。
*/
resubmit:
status = usb_submit_urb (urb, GFP_ATOMIC);
if (status)
err ("can't resubmit intr, %s-%s/input0, status %d",
mouse->usbdev->bus->bus_name,
mouse->usbdev->devpath, status);
}
/*
* 打开鼠标设备时，开始提交在 probe 函数中构建的 urb，进入 urb 周期。
*/
static int usb_mouse_open(struct input_dev *dev)
{
struct usb_mouse *mouse = dev->private;
mouse->irq->dev = mouse->usbdev;
if (usb_submit_urb(mouse->irq, GFP_KERNEL))
return -EIO;
return 0;
}
/*
* 关闭鼠标设备时，结束 urb 生命周期。
*/
static void usb_mouse_close(struct input_dev *dev)
```

```
{
struct usb_mouse *mouse = dev->private;
usb_kill_urb(mouse->irq);
}
/*
* 驱动程序的探测函数
*/
static int usb_mouse_probe(struct usb_interface *intf, const struct usb_device_id *id)
{
/*
* 接口结构体包含于设备结构体中，interface_to_usbdev 是通过接口结构体获得它的设备结构体。(见源码)
* usb_host_interface 是用于描述接口设置的结构体，内嵌在接口结构体 usb_interface 中。
* usb_endpoint_descriptor 是端点描述符结构体，内嵌在端点结构体 usb_host_endpoint 中，而端点
* 结构体内嵌在接口设置结构体中。
*/
struct usb_device *dev = interface_to_usbdev(intf);
struct usb_host_interface *interface;
struct usb_endpoint_descriptor *endpoint;
struct usb_mouse *mouse;
struct input_dev *input_dev;
int pipe, maxp;
interface = intf->cur_altsetting;
/* cur_altsetting 就是表示的当前的这个 setting,或者说设置。
```

```
鼠标仅有一个 interrupt 类型的 in 端点，不满足此要求的设备均报错 */

if (interface->desc.bNumEndpoints != 1)

return -ENODEV;

endpoint = &interface->endpoint[0].desc;

if (!usb_endpoint_is_int_in(endpoint))

return -ENODEV;
```

```
endpoint = &interface->endpoint[0].desc;//端点 0 描述符，此处的 0 表示中断端点
        if (!(endpoint->bEndpointAddress & 0x80))
        return -ENODEV;


/*先看 bEndpointAddress,这个 struct usb_endpoint_descriptor 中的一个成员,是 8 个
bit，或者说 1 个 byte，其中 bit7 表示的是这个端点的方向，0 表示 OUT，1 表示 IN，OUT 与 IN
是对主机而言。OUT 就是从主机到设备，IN 就是从设备到主机。而宏*USB_DIR_IN 来自
        *include/linux/usb_ch9.h
        * USB directions
        * This bit flag is used in endpoint descriptors' bEndpointAddress field.
        * It's also one of three fields in control requests bRequestType.
        *#define USB_DIR_OUT 0 /* to device */
        *#define USB_DIR_IN 0x80 /* to host */
        */


if ((endpoint->bmAttributes & 3) != 3)? //判断是否是中断类型
        return -ENODEV;


/* bmAttributes 表示属性,总共 8 位,其中 bit1 和 bit0 共同称为 Transfer Type,即传输类型,即
00 表示控制,01 表示等时,10 表示批量,11 表示中断*/
```

```
/*

* 返回对应端点能够传输的最大的数据包，鼠标的返回的最大数据包为 4 个字节，数据包具体内容
在 urb

* 回调函数中有详细说明。

*/

pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);

maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
```

```
/* 为 mouse 设备结构体分配内存 */
mouse = kzalloc(sizeof(struct usb_mouse), GFP_KERNEL);
/* input_dev */
input_dev = input_allocate_device();
if (!mouse || !input_dev)
goto fail1;
/*
* 申请内存空间用于数据传输，data 为指向该空间的地址，data_dma 则是这块内存空间的 dma 映射，
* 即这块内存空间对应的 dma 地址。在使用 dma 传输的情况下，则使用 data_dma 指向的 dma 区域，
* 否则使用 data 指向的普通内存区域进行传输。
* GFP_ATOMIC 表示不等待，GFP_KERNEL 是普通的优先级，可以睡眠等待，由于鼠标使用中断传输方式，
* 不允许睡眠状态，data 又是周期性获取鼠标事件的存储区，因此使用 GFP_ATOMIC 优先级，如果不能
* 分配到内存则立即返回 0。
*/
mouse->data = usb_buffer_alloc(dev, 8, GFP_ATOMIC, &mouse->data_dma);
if (!mouse->data)
goto fail1;
/*
* 为 urb 结构体申请内存空间，第一个参数表示等时传输时需要传送包的数量，其它传输方式则为 0。
* 申请的内存将通过下面即将见到的 usb_fill_int_urb 函数进行填充。
*/
```

```
mouse->irq = usb_alloc_urb(0, GFP_KERNEL);
if (!mouse->irq)
goto fail2;
/* 填充 usb 设备结构体和输入设备结构体 */
mouse->usbdev = dev;
mouse->dev = input_dev;
/* 获取鼠标设备的名称 */
if (dev->manufacturer)
strlcpy(mouse->name, dev->manufacturer, sizeof(mouse->name));
if (dev->product)
{
if (dev->manufacturer)
strlcat(mouse->name, " ", sizeof(mouse->name));
strlcat(mouse->name, dev->product, sizeof(mouse->name));
}
if (!strlen(mouse->name))
snprintf(mouse->name, sizeof(mouse->name),
"USB HIDBP Mouse %04x:%04x",
le16_to_cpu(dev->descriptor.idVendor),
le16_to_cpu(dev->descriptor.idProduct));
/*
* 填充鼠标设备结构体中的节点名。usb_make_path 用来获取 USB 设备在 Sysfs 中的路径，格式为：usb-usb 总线号-路径名。
*/
usb_make_path(dev, mouse->phys, sizeof(mouse->phys));
```

```
    strlcat(mouse->phys, "/input0", sizeof(mouse->phys));
    /* 将鼠标设备的名称赋给鼠标设备内嵌的输入子系统结构体 */
    input_dev->name = mouse->name;
    /* 将鼠标设备的设备节点名赋给鼠标设备内嵌的输入子系统结构体 */
    input_dev->phys = mouse->phys;
    /*
    * input_dev 中的 input_id 结构体，用来存储厂商、设备类型和设备的编号，这个函数是将设备描述符
    * 中的编号赋给内嵌的输入子系统结构体
    */
    usb_to_input_id(dev, &input_dev->id);
    /* cdev 是设备所属类别（class device）*/
    input_dev->cdev.dev = &intf->dev;
    /* evbit 用来描述事件，EV_KEY 是按键事件，EV_REL 是相对坐标事件 */
    input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_REL);
    /* keybit 表示键值，包括左键、右键和中键 */
    input_dev->keybit[LONG(BTN_MOUSE)] = BIT(BTN_LEFT) | BIT(BTN_RIGHT) | BIT(BTN_MIDDLE);
    /* relbit 用于表示相对坐标值 */
    input_dev->relbit[0] = BIT(REL_X) | BIT(REL_Y);
    /* 有的鼠标还有其它按键 */
    input_dev->keybit[LONG(BTN_MOUSE)] |= BIT(BTN_SIDE) | BIT(BTN_EXTRA);
    /* 中键滚轮的滚动值 */
    input_dev->relbit[0] |= BIT(REL_WHEEL);
    /* input_dev 的 private 数据项用于表示当前输入设备的种类，这里将鼠标结构体对象赋给它 */
```

```
input_dev->private = mouse;
/* 填充输入设备打开函数指针 */
input_dev->open = usb_mouse_open;
/* 填充输入设备关闭函数指针 */
input_dev->close = usb_mouse_close;
/*
* 填充构建 urb，将刚才填充好的 mouse 结构体的数据填充进 urb 结构体中，在 open 中递交 urb。
* 当 urb 包含一个即将传输的 DMA 缓冲区时应该设置 URB_NO_TRANSFER_DMA_MAP。USB 核心使用 transfer_dma 变量所指向的缓冲区，而不是 transfer_buffer 变量所指向的。
* URB_NO_SETUP_DMA_MAP 用于 Setup 包，URB_NO_TRANSFER_DMA_MAP 用于所有 Data 包。
*/
usb_fill_int_urb(mouse->irq, dev, pipe, mouse->data,
(maxp > 8 ? 8 : maxp),
usb_mouse_irq, mouse, endpoint->bInterval);
mouse->irq->transfer_dma = mouse->data_dma;
mouse->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
/* 向系统注册输入设备 */
input_register_device(mouse->dev);
/*
* 一般在 probe 函数中，都需要将设备相关信息保存在一个 usb_interface 结构体中，以便以后通过 usb_get_intfdata 获取使用。这里鼠标设备结构体信息将保存在 intf 接口结构体内嵌的设备结构体中的 driver_data 数据成员中，即 intf->dev->dirver_data = mouse。
*/
usb_set_intfdata(intf, mouse);
```

```
return 0;
fail2:  usb_buffer_free(dev, 8, mouse->data, mouse->data_dma);
fail1:  input_free_device(input_dev);
kfree(mouse);
return -ENOMEM;
}
/*
* 鼠标设备拔出时的处理函数
*/
static void usb_mouse_disconnect(struct usb_interface *intf)
{
/* 获取鼠标设备结构体 */
struct usb_mouse *mouse = usb_get_intfdata (intf);
/* intf->dev->dirver_data = NULL，将接口结构体中的鼠标设备指针置空。*/
usb_set_intfdata(intf, NULL);
if (mouse)
{
/* 结束 urb 生命周期 */
usb_kill_urb(mouse->irq);
/* 将鼠标设备从输入子系统中注销 */
input_unregister_device(mouse->dev);
/* 释放 urb 存储空间 */
usb_free_urb(mouse->irq);
/* 释放存放鼠标事件的 data 存储空间 */
```

usb_buffer_free(interface_to_usbdev(intf), 8, mouse->data, mouse->data_dma);

/* 释放存放鼠标结构体的存储空间 */

kfree(mouse);

}

}

/*

* usb_device_id 结构体用于表示该驱动程序所支持的设备，USB_INTERFACE_INFO 可以用来匹配特定类型的接口，

* 这个宏的参数意思为 (类别，子类别，协议)。

* USB_INTERFACE_CLASS_HID 表示是一种 HID (Human Interface Device)，即人机交互设备类别；

* USB_INTERFACE_SUBCLASS_BOOT 是子类别，表示是一种 boot 阶段使用的 HID；

* USB_INTERFACE_PROTOCOL_MOUSE 表示是鼠标设备，遵循鼠标的协议。

鼠标设备遵循 USB 人机接口设备（HID），在 HID 规范中规定鼠标接口类码为：

接口类：0x03

接口子类：0x01

接口协议：0x02

*/

static struct usb_device_id usb_mouse_id_table [] = {

{ USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,

USB_INTERFACE_PROTOCOL_MOUSE) },

{ } /* Terminating entry */

};

/*

* 这个宏用来让运行在用户空间的程序知道这个驱动程序能够支持的设备，对于 USB 驱动程序来说，第一个参数必须是 usb。

*/

MODULE_DEVICE_TABLE (usb, usb_mouse_id_table);

/*

* 鼠标驱动程序结构体

*/

static struct usb_driver usb_mouse_driver = {

.name   = "usbmouse",

.probe   = usb_mouse_probe,

.disconnect = usb_mouse_disconnect,

.id_table   = usb_mouse_id_table,

};

/*

* 驱动程序生命周期的开始点，向 USB core 注册这个鼠标驱动程序。

*/

static int __init usb_mouse_init(void)

{

int retval = usb_register(&usb_mouse_driver);

if (retval == 0)

info(DRIVER_VERSION ":" DRIVER_DESC);

return retval;

}

/*

* 驱动程序生命周期的结束点，向 USB core 注销这个鼠标驱动程序。

```
*/

static void __exit usb_mouse_exit(void)

{

usb_deregister(&usb_mouse_driver);

}

module_init(usb_mouse_init);

module_exit(usb_mouse_exit);
```

++

在应用层编写测试鼠标的测试程序（见源码 testmouse.c）

++

下面是一个模拟鼠标和键盘输入的例子：

```
#include <string.h>
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/stat.h>
 #include <fcntl.h>
 #include <linux/input.h>
 #include <linux/uinput.h>
 #include <stdio.h>
 #include <sys/time.h>
 #include <sys/types.h>
 #include <unistd.h>

void simulate_key(int fd,int kval)
 {
   struct input_event event;
   event.type = EV_KEY;
   event.value = 1;
   event.code = kval;

  gettimeofday(&event.time,0);
   write(fd,&event,sizeof(event)) ;

     event.type = EV_SYN;
      event.code = SYN_REPORT;
      event.value = 0;
      write(fd, &event, sizeof(event));

     memset(&event, 0, sizeof(event));
      gettimeofday(&event.time, NULL);
      event.type = EV_KEY;
      event.code = kval;
```

```
    event.value = 0;
    write(fd, &event, sizeof(event));
    event.type = EV_SYN;
    event.code = SYN_REPORT;
    event.value = 0;
    write(fd, &event, sizeof(event));

}

void simulate_mouse(int fd)
 {
   struct input_event event;
    memset(&event, 0, sizeof(event));
    gettimeofday(&event.time, NULL);
    event.type = EV_REL;
    event.code = REL_X;
    event.value = 10;
    write(fd, &event, sizeof(event));

   event.type = EV_REL;
    event.code = REL_Y;
    event.value = 10;
    write(fd, &event, sizeof(event));

   event.type = EV_SYN;
    event.code = SYN_REPORT;
    event.value = 0;
    write(fd, &event, sizeof(event));
 }

int main()
 {
   int fd_kbd;
  int fd_mouse;
  fd_kbd = open("/dev/input/event1",O_RDWR);
   if(fd_kbd<=0){
    printf("error open keyboard:\n");
    return -1;

  }

  fd_mouse = open("/dev/input/event2",O_RDWR);
  if(fd_mouse<=0){
    printf("error open mouse\n");
    return -2;
   }

  int i = 0;
   for(i=0; i< 10; i++)
   {
    simulate_key(fd_kbd, KEY_A + i);
    simulate_mouse(fd_mouse);
    sleep(1);
```

```
    }

  close(fd_kbd);
}
```

模拟了鼠标和键盘的输入事件。

关于这里 open 哪个 event， 可以通过 cat /proc/bus/input/devices

```
I: Bus=0017 Vendor=0001 Product=0001 Version=0100
 N: Name="Macintosh mouse button emulation"
 P: Phys=
 S: Sysfs=/class/input/input0
 U: Uniq=
 H: Handlers=mouse0 event0
B: EV=7
 B: KEY=70000 0 0 0 0 0 0 0 0
 B: REL=3

I: Bus=0011 Vendor=0001 Product=0001 Version=ab41
 N: Name="AT Translated Set 2 keyboard"
 P: Phys=isa0060/serio0/input0
 S: Sysfs=/class/input/input1
 U: Uniq=
 H: Handlers=kbd event1
B: EV=120013
 B: KEY=4 2000000 3803078 f800d001 feffffdf ffefffff ffffffff fffffffe
 B: MSC=10
 B: LED=7

I: Bus=0019 Vendor=0000 Product=0002 Version=0000
 N: Name="Power Button (FF)"
 P: Phys=LNXPWRBN/button/input0
 S: Sysfs=/class/input/input3
 U: Uniq=
 H: Handlers=kbd event3
B: EV=3
 B: KEY=100000 0 0 0

I: Bus=0019 Vendor=0000 Product=0001 Version=0000
 N: Name="Power Button (CM)"
 P: Phys=PNP0C0C/button/input0
 S: Sysfs=/class/input/input4
 U: Uniq=
 H: Handlers=kbd event4
B: EV=3
 B: KEY=100000 0 0 0

I: Bus=0003 Vendor=046d Product=c018 Version=0111
 N: Name="Logitech USB Optical Mouse"
 P: Phys=usb-0000:00:1d.1-2/input0
 S: Sysfs=/class/input/input24
 U: Uniq=
 H: Handlers=mouse1 event2
B: EV=7
```

B: KEY=70000 0 0 0 0 0 0 0 0
B: REL=103

我的鼠标是 罗技 的 Logitech USB Optical Mouse， 所以 鼠标是 event2
下面是一个读取 鼠标和键盘事件的例子：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/input.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

static void show_event(struct input_event* event)
{
    printf("%d %d %d\n", event->type, event->code, event->value);

    return;
}

int main(int argc, char* argv[])
{
    struct input_event event = {{0}, 0};
    const char* file_name = argc == 2 ? argv[1] : "/dev/input/event2";

    int fd = open(file_name, O_RDWR);


    if(fd > 0)
    {

        while(1)
        {
            int ret = read(fd, &event, sizeof(event));
            if(ret == sizeof(event))
            {
                show_event(&event);
            }
            else
            {
                break;
            }
        }
        close(fd);
    }

    return 0;
}
```