1.register\_root\_hub( )

root hub的设备编号设置为1；

set\_bit (devnum, usb\_dev->bus->devmap.devicemap)，这个函数的意思把devmap.devicemap的第一位设置为1，表明这一位已经有设备占了，没有占的用0表示。devmap.devicemap最多可以标记128个设备,将root hub设备的状态设置为USB\_STATE\_ADDRESS，表明这个设备已经有地址了。

1.1 usb\_set\_device\_state(usb\_dev, USB\_STATE\_ADDRESS)

将root hub端口0的最大字符长度设置为64

usb\_dev->ep0.desc.wMaxPacketSize = cpu\_to\_le16(64);cpu\_to\_le16这个函数是把一个数字转化成小端存放在内存中，因为usb的描述符都是按照小端存储的。

获取root hub的设备描述符，长度是18个字节

retval = usb\_get\_device\_descriptor(usb\_dev, USB\_DT\_DEVICE\_SIZE)

usb\_get\_device\_descriptor( )已分析🡪 usb\_control\_msg( )

1.2 usb\_control\_msg函数获取设备描述符，循环了三次，以防有的设备一次不能成功。

此函数获取到了八个参数：usb\_control\_msg函数获取设备描述符，循环了三次，以防有的设备一次不能成功。

dev：usb设备

pipe：管道

管道有8种类型：

usb\_sndctrlpipe(dev,endpoint)=((2 << 30) | (dev->devnum << 8) | (endpoint << 15)

usb\_rcvctrlpipe(dev,endpoint) =((2 << 30) | (dev->devnum << 8) | (endpoint << 15) | USB\_DIR\_IN)

usb\_sndisocpipe(dev,endpoint)=((0 << 30) | (dev->devnum << 8) | (endpoint << 15)

usb\_rcvisocpipe(dev,endpoint)=((0 << 30) | (dev->devnum << 8) | (endpoint << 15) | USB\_DIR\_IN)

usb\_sndbulkpipe(dev,endpoint)=((3 << 30) | (dev->devnum << 8) | (endpoint << 15)

usb\_rcvbulkpipe(dev,endpoint)=((3 << 30) |(dev->devnum << 8) | (endpoint << 15) | USB\_DIR\_IN)

usb\_sndintpipe(dev,endpoint) =((1 << 30) | (dev->devnum << 8) | (endpoint << 15)

usb\_rcvintpipe(dev,endpoint)=((1 << 30) | (dev->devnum << 8) | (endpoint << 15) | USB\_DIR\_IN)

request：

表示具体是哪个request，如下

#define USB\_REQ\_GET\_STATUS 0x00

#define USB\_REQ\_CLEAR\_FEATURE 0x01

#define USB\_REQ\_SET\_FEATURE 0x03

#define USB\_REQ\_SET\_ADDRESS 0x05

#define USB\_REQ\_GET\_DESCRIPTOR 0x06

#define USB\_REQ\_SET\_DESCRIPTOR 0x07

#define USB\_REQ\_GET\_CONFIGURATION 0x08

#define USB\_REQ\_SET\_CONFIGURATION 0x09

#define USB\_REQ\_GET\_INTERFACE 0x0A

#define USB\_REQ\_SET\_INTERFACE 0x0B

#define USB\_REQ\_SYNCH\_FRAME 0x0C

requesttype：

bit7表示了控制传输中DATA transaction阶段的方向

#define USB\_DIR\_OUT 0 /\* to device /

#define USB\_DIR\_IN 0x80 / to host \*/

bit5~6表示request的类型，是标准的，class-specific的还是vendor-specific的

#define USB\_TYPE\_STANDARD (0x00 << 5)

#define USB\_TYPE\_CLASS (0x01 << 5)

#define USB\_TYPE\_VENDOR (0x02 << 5)

#define USB\_TYPE\_RESERVED (0x03 << 5)

bit0~4表示了这个请求针对的是设备，接口，还是端点

#define USB\_RECIP\_DEVICE 0x00

#define USB\_RECIP\_INTERFACE 0x01

#define USB\_RECIP\_ENDPOINT 0x02

#define USB\_RECIP\_OTHER 0x03

value：这个字段是request的参数，request不同，wValue就不同

index：bRequestType指明request针对的是设备上的某个接口或端点的时候，wIndex就用来指明是哪个接口或端点

data：请求buffer

size：请求buffer的大小

timeout：请求超时时间

1.3 usb\_internal\_control\_msg函数

首先使用一个kmalloc( )函数为一个struct usb\_ctrlrequest结构体申请内存，如果申请失败，则返回错误码。

将requesttype的信息保存在结构体中的bRequestType

将request的信息保存在结构体中的bRequest

使用cpu\_to\_le16( )分别将value，index ，size保存在结构体的wValue，wIndex， wLength

最后使用usb\_internal\_control\_msg( )函数初始化root hub的urb

1.4 usb\_internal\_control\_msg( )

首先使用usb\_alloc\_urb( )创建一个urb,如果创建失败则返回错误码，并使用usb\_fill\_control\_urb( )函数对刚刚被创建的的urb初始化。最后使用usb\_start\_wait\_urb( )函数将urb提交给USB hcd，交给主机控制器驱动进行处理。

1.5 usb\_start\_wait\_urb( )函数的主要目的就是调用usb\_submit\_urb( )函数判断urb是否被成功提交到了hcd中。

1.6 int usb\_submit\_urb(struct urb \*urb, gfp\_t mem\_flags)

{

……

usb\_hcd\_submit\_urb(urb, mem\_flags)

//root hub调用rh\_urb\_enqueue进行USB通信

if (is\_root\_hub(urb->dev))

status = rh\_urb\_enqueue(hcd, urb);

//如果不是root hub，调用对应的控制器的urb\_enqueue函数

else

status = hcd->driver->urb\_enqueue(hcd, urb, mem\_flags);

1.7 static int rh\_urb\_enqueue (struct usb\_hcd \*hcd, struct urb \*urb)

{

//如果是中断传输

if (usb\_endpoint\_xfer\_int(&urb->ep->desc))

return rh\_queue\_status (hcd, urb);

//如果是控制传输

if (usb\_endpoint\_xfer\_control(&urb->ep->desc))

return rh\_call\_control (hcd, urb);

return -EINVAL;

}

rh\_urb\_enqueue 调用的又是rh\_call\_control 函数。

usb3\_rh\_dev\_descriptor，usb2\_rh\_dev\_descriptor，ss\_rh\_config\_descriptor等都是定义在hcd.c里面的全局变量。因为root hub是集成在主机控制器里面了，所以跟主机控制器通信获取root hub的信息实际上驱动软件里面写死了，而不是获取的硬件上的信息。

根据上面的描述信息，可以得到：

root hub只有一种配置，一个接口，一种设置，一个端口；

DeviceClass = 0x9；

InterfaceClass = 0x09；

配置描述符的总长度是25；

设备描述符的长度是18；

最终得到的设备描述符信息保存在dev->descriptor结构体中。

所有解析出来的配置描述符信息保存在usb\_device->usb\_host\_config[config\_num].desc结构体里面，i是该配置的编号；

usb\_device->usb\_host\_config[config\_num].intf\_cache[interface\_num].altsetting[setting\_num].desc保存每个接口描述符；

usb\_device->usb\_host\_config[config\_num].intf\_cache[interface\_num].altsetting[setting\_num].endpoint[point\_num].desc保存每个端口的描述符信息；

最后一步就是添加root hub设备

在hub初始化时，在hub\_configure( )函数中创建了一个中断urb请求。

hub->urb = usb\_alloc\_urb(0, GFP\_KERNEL);

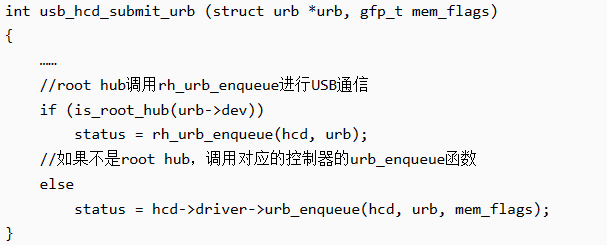
hub\_irq函数赋值给了urb->complete函数。

在hub\_activate函数中提交了urb请求。

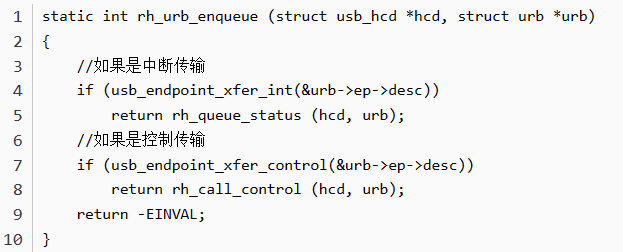
hub\_activate—>usb\_submit\_urb(hub->urb, GFP\_NOIO);

usb\_submit\_urb中调用usb\_hcd\_submit\_urb函数，将urb给HCD处理

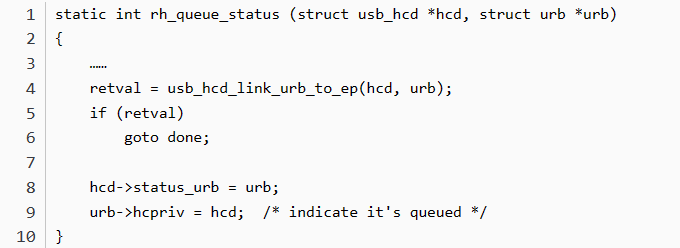
usb\_hcd\_submit\_urb函数里面，如果进来的设备是root hub，那么urb请求由rh\_urb\_enqueue函数处理，如果不是，调用主机控制器的urb\_enqueue函数。



我们这里还是root hub设备，所以调用rh\_urb\_enqueue函数。



我们这里是中断传输，所以调用rh\_queue\_status函数。



上面把urb放入urb端口的链表中。

把urb赋值给hcd->status\_urb保存起来。这样上面的hcd->status\_urb就有值了。

前面说到有一个定时器hcd->rh\_timer，执行函数是rh\_timer\_func

[发现新的usb设备(八)\_usb\_hcd\_poll\_rh\_status-CSDN博客](https://blog.csdn.net/weixin_38696651/article/details/89632477?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522170887174516800180624605%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fblog.%2522%257D&request_id=170887174516800180624605&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~first_rank_ecpm_v1~rank_v31_ecpm-2-89632477-null-null.nonecase&utm_term=usb&spm=1018.2226.3001.4450)

系统会调用kthread\_run(hub\_thread, NULL, "khubd")来启动守护进程的，其中的khubd，就是守护进程名称，这个进程几乎是个死循环，只有在执行kthread\_should\_stop时才会退出，而进程也不是时时都在执行的，当hub没有设备插入时，进程属于睡眠状态，只有当有设备插入时，才会唤醒进程，进行处理。

而对一个设备的插入的发现，hub通知主控制器后，由主控制器调用hub的探测函数，hub的探测函数才开始了一个检查设备的流程（跟hcd有关，还不了解）。因此，一个完整的流程是从hub的探测函数hub\_probe开始的，而前提条件是主控制器已经正常工作。

主控制器发现了root hub接口后，调用hub\_probe函数开始配置root hub。

1． 通过接口描述符获得当前接口描述符，通过分析当前的接口描述符的bInterfaceSubClass成员是不是hub类成员，是不是只有一个端口，这个端口是不是中断端口，来判断是否是一个hub设备，如果通不过这些验证，表示不是hub设备，退出

2． 通过1的验证，可以证明是hub设备了，才给hub设备分配控制(struct usb\_hub),并初始化hub结构体，即struct usb\_hub是在这里被初始化的。

3． 配置root hub，分配用于urb传输的缓存和DMA地址（两者是关联的），hub状态空间，hub描述符空间，分配完后，通过向设备发送请求，获得设备的描述符，通过描述符得到hub的各种参数，用这些参数配置usb\_hub中的usb\_device结构体。

4． 获取hub跟host连接的管道，及管道端点的大小，这个管道是中断传输的，端点大小不能大于刚刚分配的缓冲区的大小

5． 申请一个urb空间，用于准备传输urb请求数据。

6． 填充hub的urb数据，包括管道，缓存，缓存长度，中断处理函数等，这个是最重要的，经过在这里的配置，当hub端口有中断信号（即有设备插入或拔开）时，就会调用这里已经配置好的中断函数，来唤醒hub守护进程，进程得到执行，开始解析端口的状态---urb是需要请求的时候才会创建的，urb创建完，提交给主机后，就被系统释放了。

7． 通过设备接口属性，使能hub 上行端口的power-on

8． 把刚刚申请的urb发送给主机，如果一切正常，主机就会定时来询问hub是否有中断，如果就进行中断传输，发送的urb请求是把hub的各种数据报给主控制器，让主控制器知道。

9． 到此，hub就开始正常工作了，随时等候中断信号，并跟主控制器报告。

Hub正常工作后，主控制器就会定时询问hub是否有中断产生，当hub端口上有一个设备插入或拔除，hub就向主控制器发送urb请求，即把hub端口的变化状况告诉主控制器，这是通过urb请求来完成的，主机在处理完了这个urb后，就会调用urb所提供的完成函数，来调用hub的中断函数，即hub\_irq。

Hub\_irq是hub的中断处理函数，处理程序首先判断主控制器处理urb的结果状态，如果状态是OK的，则继续处理。

1．扫描hub的所有端口，确定是哪个端口发生了变化。端口是用位图来表示的，一个long型数据可以表示32个hub端口（每位表示一个端口），有多少个端口，就用多少位表示，而8位用一个字节表示，因此，最后的使用的位都是转化成了字节的的，比如一个hub有18个端口，则需要用18位来表示，但是一个字节只有8位，因此，需要用3个字节才能表示完。

2．调用kick\_khubd函数，把当前hub加入到hub驱动队列hub\_event\_list中，然后唤醒hub守护进程wake\_up(khubd\_wait)，开始解析hub发生的事情了。

3．hub\_thread被唤醒后，将得到执行，hub\_events也将得到执行

4．hub\_events是分析hub事件的主函数，hub分析的相关内容都在这里执行。这个函数是一个大的死循环。

5．Hub\_events处理流程

5．1先从hub\_event\_list中取出此次处理的hub节点，并把hub节点从原来的队列中删除，使之独立于任何链表，因为我们处理完事件后，hub结构体就要删除，因此不能保留在任何队列中。hub可以是根hub，也可以是接到根hub上的子hub，只是，不论是什么hub，都使用同样的守护进程了。

5．2．通过宏转换，找到hub节点对以的hub结构体，进而得到hub结构体对应的usb结构体，以及hub接口结构体usb\_intfdev，得到这三个结构体是这个函数处理的关键。

5．3．锁住当前的hub树，因为hub只有一个守护进程，所有的hub都使用这个守护进程，而守护进程每次只能服务一个hub，因此，只要有一个hub在使用这个守护进程，就需要锁住，以防止其他hub的使用。

5．4．按位检索hub每个端口是否正在执行reset或resume操作，注意是从1开始检索的，位0表示整个hub的情况，如果当前端口正在执行reset或resume操作，则跳过对这端口的检查，否则，测试这个端口是否有状态改变，如果没有状态改变，也跳过此端口。

5．5．如果端口有状态改变，则判断端口是发生了什么状态的改变，有以下的状态

#define USB\_PORT\_STAT\_C\_CONNECTION 0x0001

#define USB\_PORT\_STAT\_C\_ENABLE 0x0002

#define USB\_PORT\_STAT\_C\_SUSPEND 0x0004

#define USB\_PORT\_STAT\_C\_OVERCURRENT 0x0008

#define USB\_PORT\_STAT\_C\_RESET 0x0010

5．6．确定有状态改变后，就开始对有改变的端口的状态进行进一步的处理，此时的处理，就要根据刚刚分析得到的各种状态，分开进行处理。

5．7．首先要确定端口是否还有设备，如果有，则要把设备删除，原因是，hub端口上有两种状态发生转换，1为端口从无设备到有设备的状态，此时设备还在认证中，因此端口不应该有设备，2是端口从有到无设备的转换，此时检测到还有设备，表示设备还没有被移除，因此，需要直接把设备disable掉。

5．8．把设备disable掉的动作，需要进行反弹的检查，一个设备至少需要100ms的时间，才能表示此状态是稳定的，因此，等100ms后再判断端口是否是disable了，就可以判断了

5．9．判断设备状态为无设备插入时，需要检测是否是端口电源被disable掉了，如果是，则需要开启端口电源，如果电源是开启的，则表示设备有误，结束判断，返回上层。

5．10．经过上面的判断，能执行到这里，表示端口是有设备插入了，此时，就需要要分析端口的状态了，对端口的分析，可以尝试4次，主要是为了排除各种干扰。

5．11．为即将到来的设备分配空间(struct usb\_device),对这个即将到来的设备的设备结构体设置状态为连接状态，速度为未知，电源为hub分配给的，这些都是默认状态。

5．12．准备为设备分配地址，此时是子系统软件的操作，就是查看总线上128位位图中，哪位为0的，就选择出来，查找的方式是从上次记录的下一位开始查找，如果超过128了还没有找到，就接着从0开始查找，即如果频繁的拔插usb设备，即使只有一个设备，则每次插入，设备地址都会增加1，直到到达128后，从0接着开始，此时的设备地址，还不是设备的真实地址，因为还没有发送给设备。

5．13．复位设备，通过对设备的复位来达到使能设备的目的，此时，如果复位成功，则设备的状态将成为USB\_STATE\_DEFAULT状态，主控制器就可以通过控制端口获得设备的描述符符，通过对设备描述符的解析，可以得到设备的速度，从而可以根据速度猜测控制端口的空间大小。

5．14．得到控制端点的大小后，就开始准备往控制端点发送urb请求，获得设备的真正的设备描述符。

5．15．获得设备描述符时分两种模式，新模式和旧模式，每种模式最多试两次，每次可以最多读三回端口。对于新模式，需要分配一个64字节的空间，用于接收从设备返回的设备描述符。通过获得的设备描述符来判断端口的类别是不是确实是设备类别的，如果是，则只需要读取一次端口，就可以判断是设备描述符了，把描述符中记录的端口的空间大小给刚刚申请的空间的对应字节赋值了。

5．16．复位设备，准备给设备设置刚刚系统选择好，但还没有设置的那个地址，这个设置可以最多尝试设置两次，之间要停留200ms。

5．17．一旦设置成功，就要把设备的状态转变为地址状态（USB\_STATE\_ADDRESS），同时把控制端口0给disable掉，这样，这个端口上的urb链表都被清除了。

5．18．如果新的模式能走到这里，就表明新模式成功的获得了设备描述符了，因此旧模式就不要试了，而旧模式是分两次来获得设备的描述符的，第一次是发送要求获得8字节的urb请求，经过这个请求得到的数据，就可以知道设备的端口大小到达是多大，再按照获得的端口大小，获得端口的真实的设备描述符。

5．19．接下来，就要开始把这个设备加入设备模式中去了（usb\_new\_device）。

5．20．首先，要获得设备的配置描述符，注意，只有设备才有配置，接口，端口都是没有配置的，接口有设置。设备有多少个配置，就要分配多少个配置的空间，用于接收，而有多少个配置，已经在设备描述符中获得了，这时只需要从设备中读取就可以。

5．21．设置配置描述符指针，同样有多少个配置多少个这样的指针

5．22．一个配置描述符有9个字节，此时需要分配出这个空间，用于接收从设备中读回的配置描述符。

5．23．循环读取设备中的配置描述符，并把读回的数据放在设备结构体中的相应位置，之后，从配置中获得接口数，接口设置，以及接口下面的端点，通通都是在这里分析的，好长的几个函数。

5．24．接着，把设备添加到设备模型中去，这部分工作由设备模型完成，主要包括完成了设备链表的添加，和查找设备驱动程序的，此处是设备的驱动，usb系统只有一个设备驱动，而且是系统已经做好的，当设备里的接口被添加时，我们的编写的驱动，就是在这里被调用，用来判断驱动是否符合设备的了。

5．25．扫描设备的所有配置，选择最优的当前配置，再用选中的这个配置去设置设备。

5．26．到此，设备的配置，设置，什么都配置妥当了，系统就可以正常的使用设备了。