Android 之蓝牙驱动开发总结

一 Bluetooth 基本概念

蓝牙是无线数据和语音传输的开放式标准，它将各种通信设备、计算机及其终端设备、各种数字数据系统、甚至家用电器采用无线方式联接起来。它的传输距离为 10cm～10m，如果增加功率或是加上某些外设便可达到 100m 的传输距离。它采用 2.4GHz ISM 频段和调频、跳频技术，使用权向纠错编码、ARQ、TDD 和基带协议。TDMA 每时隙为 0.625μs，基带符合速率为 1Mb/s。蓝牙支持 64kb/s 实时语音传输和数据传输，语音编码为 CVSD，发射功率分别为 1mW、2.5mW 和 100mW，并使用全球统一的 48 比特的设备识别码。由于蓝牙采用无线接口来代替有线电缆连接，具有很强的移植性，并且适用于多种场合，加上该技术功耗低、对人体危害小，而且应用简单、容易实现，所以易于推广。

蓝牙技术的系统结构分为三大部分：底层硬件模块、中间协议层和高层应用。底层硬件部分包括无线跳频（RF）、基带（BB）和链路管理（LM）。无线跳频层通过 2.4GHz 无需授权的 ISM 频段的微波，实现数据位流的过滤和传输，本层协议主要定义了蓝牙收发器在此频带正常工作所需要满足的条件。基带负责跳频以及蓝牙数据和信息帧的传输。链路管理负责连接、建立和拆除链路并进行安全控制。

二 Android Bluetooth 架构

2.1 Bluetooth 架构图

Android 蓝牙系统分为四个层次，内核层、BlueZ 库、BlueTooth 的适配库、BlueTooth 的 JNI 部分、Java 框架层、应用层。下面先来分析 Android 的蓝牙协议栈。

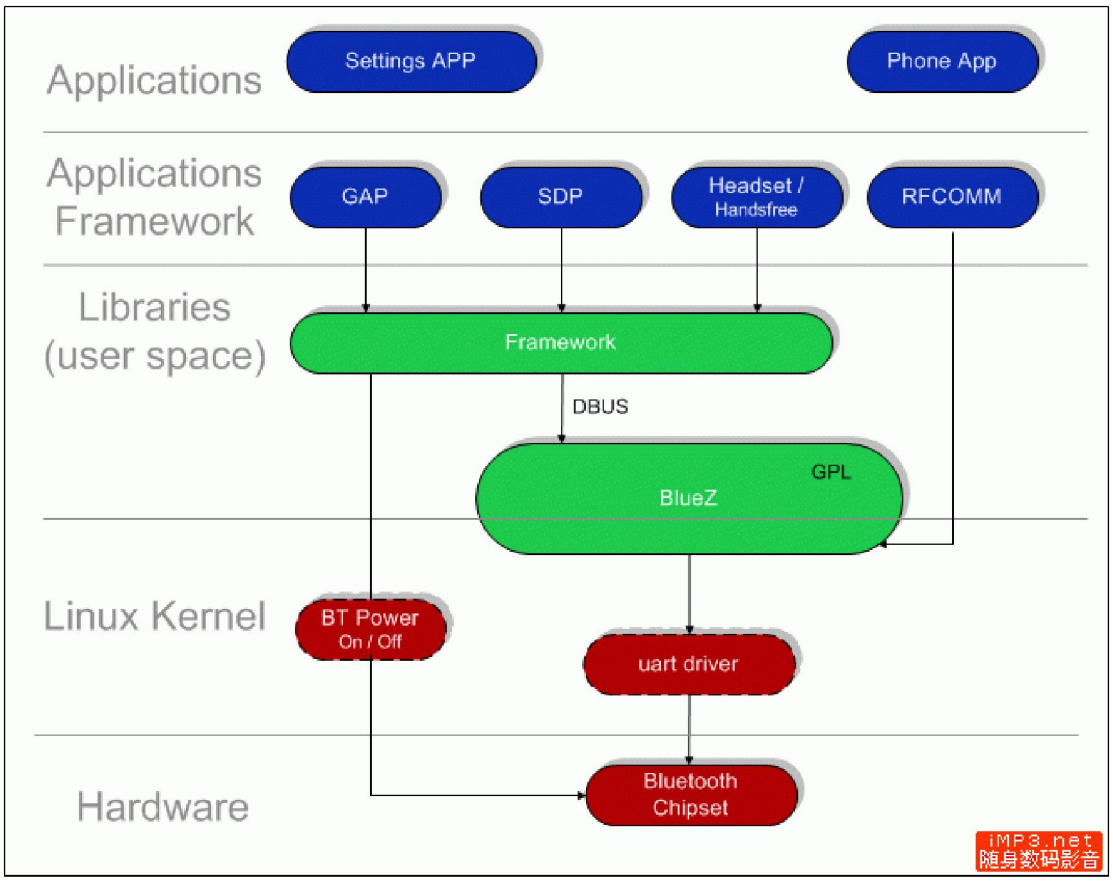

图 1 面向库的架构视图

Linuxkernel 层：

bluez 协议栈、uart 驱动, h4 协议, hci,l2cap, sco, rfcomm

bluez 层：

这是 bluez 用户空间的库，开源的 bluetooth 代码，包括很多协议，生成 libbluetooth.so。

library 层：

libbluedroid.so 等

framework 层：

实现了 Headset /Handsfree 和 A2DP/AVRCP profile，但其实现方式不同 Handset/Handfree 是直接 在 bluez 的 RFCOMM Socket 上开发的，没有利用 bluez 的 audio plugin，而 A2DP/AVRCP 是在 bluez 的 audio plugin 基础上开发的，大大降低了实现的难度。

Android 的蓝牙协议栈采用 BlueZ 来实现，BlueZ 分为两部分：内核代码和用户态程序及工具集。

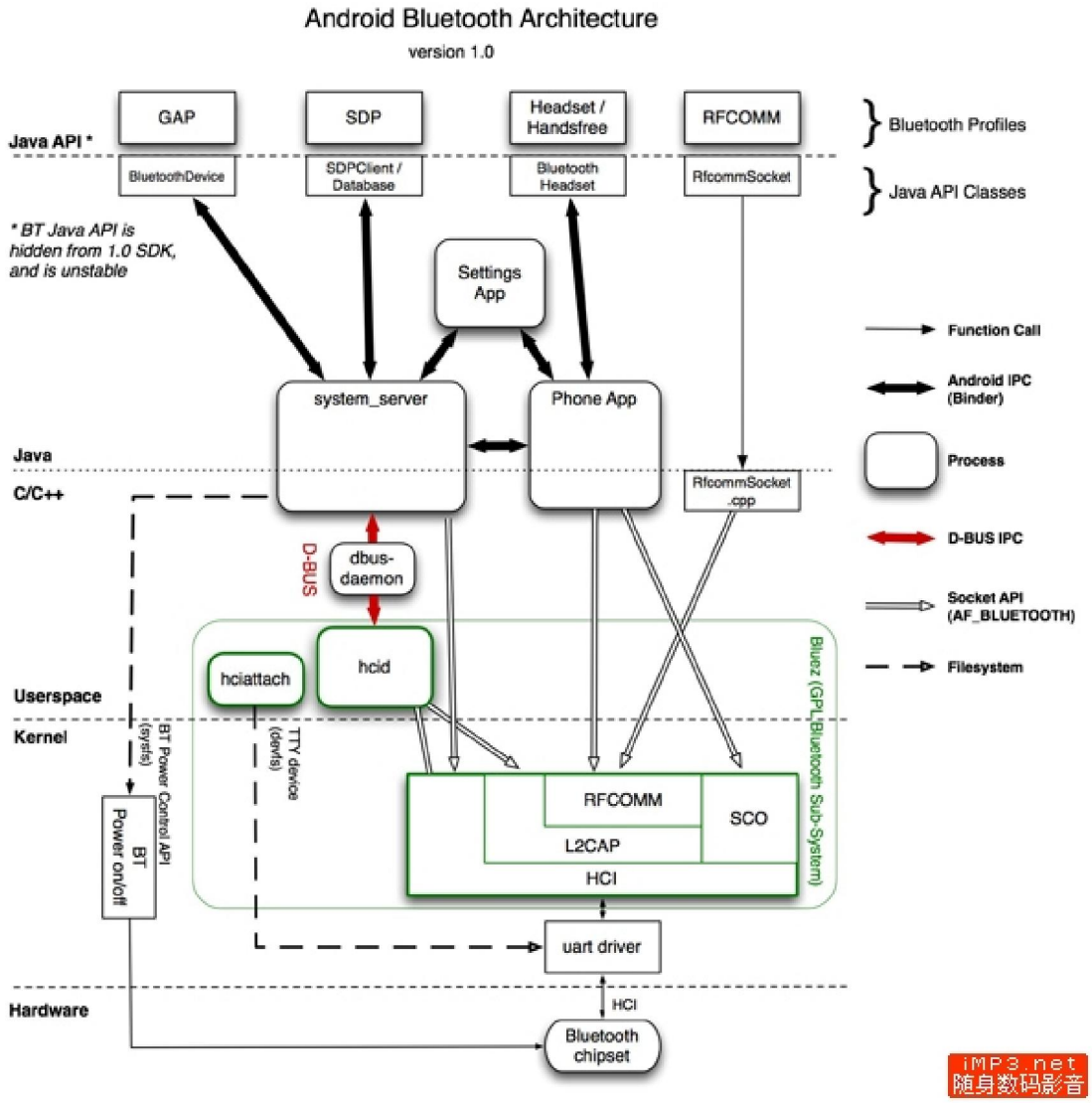

图 2 面向进程的架构视图

2.2 Bluetooth 代码层次结构

（1）JAVA 层
frameworks/base/core/java/android/bluetooth/
包含了 bluetooth 的 JAVA 类。
（2）JNI 层
frameworks/base/core/jni/android_bluetooth_开头的文件
定义了 bluez 通过 JNI 到上层的接口。
frameworks/base/core/jni/android_server_bluetoothservice.cpp
调用硬件适配层的接口 system/bluetooth/bluedroid/bluetooth.c
（3）bluez 库
external/bluez/
这是 bluez 用户空间的库，开源的 bluetooth 代码，包括很多协议，生成 libbluetooth.so。
（4）硬件适配层

system/bluetooth/bluedroid/bluetooth.c
包含了对硬件操作的接口
system/bluetooth/data/*一些配置文件，复制到/etc/bluetooth/。
还有其他一些测试代码和工具。

内核代码主要由 BlueZ 核心协议和驱动程序组成；蓝牙协议实现在内核源代码 net/bluetooth 中，驱动程序位于内核源代码目录 driver/bluetooth 中。用户态程序及工具集主要包括应用程序接口和 BlueZ 工具集，位于 Android 源代码目录 externel/bluetooth(注：Android 版本不一样，有的在 externel/bluez 目录下)中。

三 Bluetooth 协议栈分析

3.1 蓝牙协议栈

蓝牙协议栈的体系结构由底层硬件模块、中间协议层和高端应用层三部分组成。
（1）底层硬件模块
组成：
链路管理协议(Link ManagerProtocol，LMP)；
基带(Base Band，BB)；
射频(Radio Frequency，RF)。
功能：
射频(RF)通过 2.4GHz 的 ISM 频段实现数据流的过滤和传输。

基带(BB)提供两种不同的物理链路，即同步面向连接链路(Synchronous Connection Oriented，SCO)和异步无连接链路(AsynchronousConnection Less，ACL)，负责跳频和蓝牙数据，及信息帧的传输，且对所有类型的数据包提供不同层次的前向纠错码(Frequency Error Correction，FEC)或循环冗余度差错校验(CyclicRedundancy Check，CRC)。

链路管理协议(LMP)负责两个或多个设备链路的建立和拆除，及链路的安全和控制，如鉴权和加密、控制和协商基带包的大小等，它为上层软件模块提供了不同的访问入口。

主机控制器接口(HostController Interface，HCI)是蓝牙协议中软硬件之间的接口，提供了一个调用下层 BB、LMP、状态和控制寄存器等硬件的统一命令，上下两个模块接口之间的消息和数据的传递必须通过 HCI 的解释才能进行。
（2）中间协议层
组成：
逻辑链路控制和适配协议(LogicalLink Control and Adaptation Protocol，L2CAP)；
服务发现协议(ServiceDiscovery Protocol，SDP)；
串口仿真协议(或称线缆替换协议 RFCOMM)；
二进制电话控制协议(TelephonyControlprotocol Spectocol，TCS)。
功能：

L2CAP 位于基带(BB)之上，向上层提供面向连接的和无连接的数据服务，它主要完成数据的拆装、服务质量控制、协议的复用、分组的分割和重组，及组提取等功能。

SDP 是一个基于客户/服务器结构的协议，它工作在 L2CAP 层之上，为上层应用程序提供一种机制来发现可用的服务及其属性，服务的属性包括服务的类型及该服务所需的机制或协议信息。

RFCOMM 是一个仿真有线链路的无线数据仿真协议，符合 ETSI 标准的 TS07.10 串口仿真

协议，它在蓝牙基带上仿真 RS-232 的控制和数据信号，为原先使用串行连接的上层业务提供传送能力。

TCS 定义了用于蓝牙设备之间建立语音和数据呼叫的控制信令(Call Control Signalling)，并负责处理蓝牙设备组的移动管理过程。

（3）高端应用层

组成：

点对点协议(Point-to-PointProtocol，PPP);

传输控制协议/网络层协议(TCP/IP);

用户数据包协议(UserDatagram Protocol，UDP)；

对象交换协议(ObjectExchang Protocol，OBEX)；

无线应用协议(WirelessApplication Protocol，WAP);

无线应用环境(WirelessApplication Environment，WAE)；

功能：

PPP 定义了串行点对点链路应当如何传输因特网协议数据，主要用于 LAN 接入、拨号网络及传真等应用规范。

TCP/IP、UDP 定义了因特网与网络相关的通信及其他类型计算机设备和外围设备之间的通信。

OBEX 支持设备间的数据交换，采用客户/服务器模式提供与 HTTP(超文本传输协议)相同的基本功能。可用于交换的电子商务卡、个人日程表、消息和便条等格式。

WAP 用于在数字蜂窝电话和其他小型无线设备上实现因特网业务，支持移动电话浏览网页、收取电子邮件和其他基于因特网的协议。

WAE 提供用于 WAP 电话和个人数字助理(PersonalDigital Assistant，PDA)所需的各种应用软件。

3.2 Android 与蓝牙协议栈的关系

蓝牙系统的核心是 BlueZ，因此 JNI 和上层都围绕跟 BlueZ 的沟通进行。JNI 和 Android 应用层，跟 BlueZ 沟通的主要手段是 D-BUS，这是一套被广泛采用的 IPC 通信机制，跟 Android 框架使用的 Binder 类似。BlueZ 以 D-BUS 为基础，给其他部分提供主要接口。

四 Bluetooth 之 HCI 层分析

蓝牙系统的 HCI 层是位于蓝牙系统的 L2CAP（逻辑链路控制与适配协议）层和 LMP（链路管理协议）层之间的一层协议。HCI 为上层协议提供了进入 LM 的统一接口和进入基带的统一方式。在 HCI 的主机（Host）和 HCI 主机控制器（Host Controller）之间会存在若干传输层，这些传输层是透明的，只需完成传输数据的任务，不必清楚数据的具体格式。目前，蓝牙的 SIG 规定了四种与硬件连接的物理总线方式：USB、RS232、UART 和 PC 卡。其中通过 RS232 串口线方式进行连接具有差错校验。蓝牙系统的协议模型如图 3 所示。

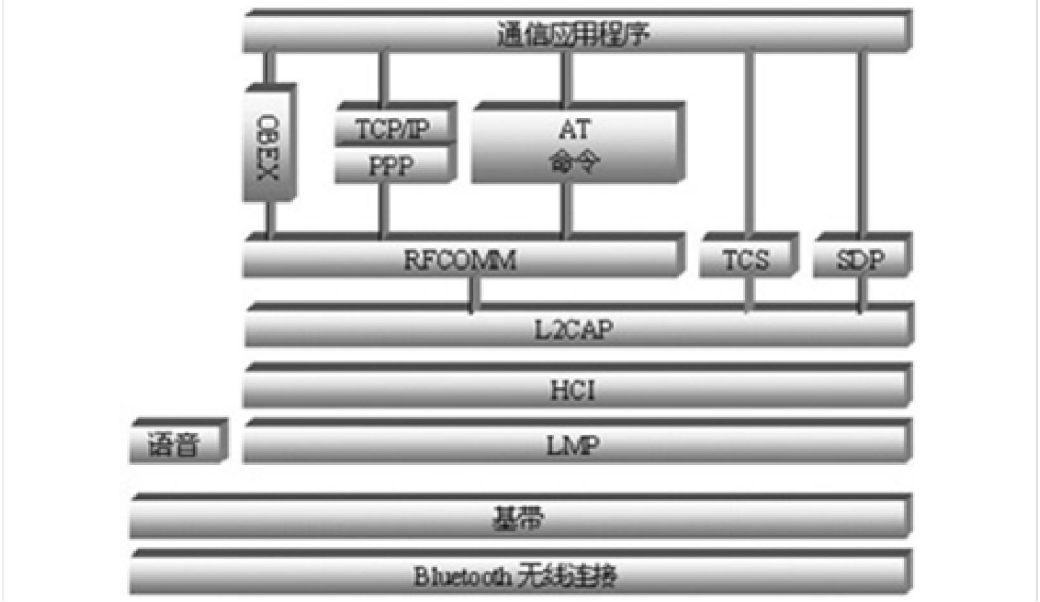

图 3 Bluetooth 协议模型

4.1 HCI 层与基带的通信方式

HCI 是通过包的方式来传送数据、命令和事件的，所有在主机和主机控制器之间的通信都以包的形式进行。包括每个命令的返回参数都通过特定的事件包来传输。HCI 有数据、命令和事件三种包，其中数据包是双向的，命令包只能从主机发往主机控制器，而事件包始终是主机控制器发向主机的。主机发出的大多数命令包都会触发主机控制器产生相应的事件包作为响应。

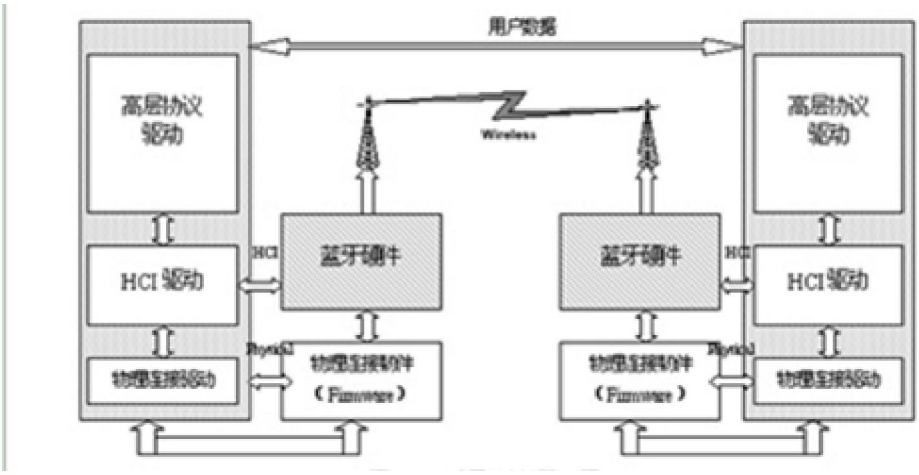

图 4 底层协议通信图

命令包分为六种类型：

* 链路控制命令；
* 链路政策和模式命令；
* 主机控制和基带命令；
* 信息命令；
* 状态命令；
* 测试命令。

事件包也可分为三种类型：
* 通用事件，包括命令完成包（Command Complete）和命令状态包（Command Status）；
* 测试事件；
* 出错时发生的事件，如产生丢失（Flush Occured）和数据缓冲区溢出（Data Buffer Overflow）。
数据包则可分为 ACL 和 SCO 的数据包。包的格式如图 5 所示。

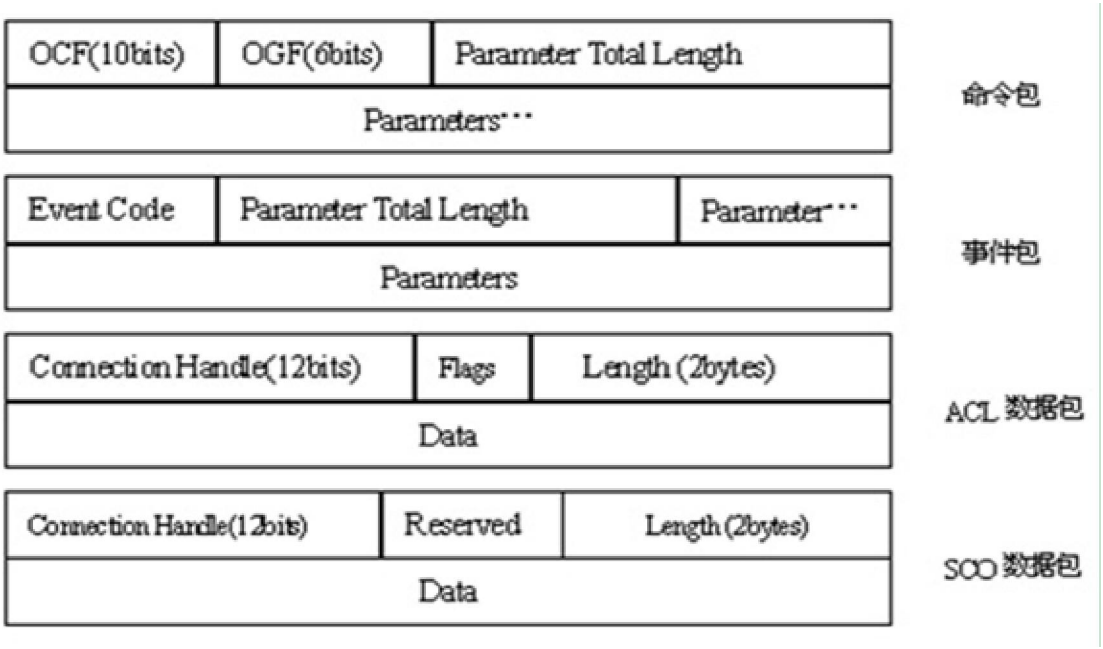

图 5 HCI 包格式

4.2 包的分析及研究

命令包：命令包中的 OCF（Opcode Command Field）和 OGF（Opcode Group Field）是用于区分命令种的。Parameter Length 表示所带参数的长度，以字节数为单位，随后就是所带的参数列表。下面以 Inquiry 命令为例对 HCI 的命令包做具体说明。

在 Inquiry 命令中，OGF=0x01 表示此命令属于链路控制命令，同时 OCF=0x0001 则表示此命令为链路控制命令中的 Inquiry 命令。OCF 与 OGF 共占 2 字节，又由于底位字节在前，则它们在命令包为 0x0104。在 Inquiry 命令中，参数 Parameter Length 为 5。Inquiry 命令带 3 个参数，第一个参数为 LAP(low address part)， 它将用来产生 Baseband 中查询命令包的包头中的 Access Code。第二个参数为 Inquiry_Length，它时表示在 Inquiry 命令停止前所定义的最大时间，超过此时间，Inquiry 命令将终止。第三个参数为 NUM_Response，它的值为 0X00 表示设备响应数不受限制，只为 0x00-0xff 则表示在 Inquiry 命令终止前最大的设备响应数。因此，若 LAP=0x9e8b00，Inquiry_Length=0x05，NUM_Response=0x05，则协议上层调用 Inquiry 命令是 HCI 向基带发的明令包将为：0x01 04 05 00 8b 9e 05 05。

事件包：事件包的 Event Code 用来区分不同的事件包，Parameter Length 表示所带参数的长度，以字节数为单位，随后就是所带的参数列表。以 Command Status Event 事件包为例对 HCI 的事件包进行具体说明。

当主机控制器收到主机发来的如上面所提到的 Inquiry 命令包并开始处理时，它就会向主机发送 Command Status Event 事件包，此事件包为：0x0f 04 00 0a 01 04。0xOf 表示此事件包为 Command Status Event 事件包，0x04 表示此事件包带 4 字节长度的参数，0x00 为此事件包的第一个参数即 Status，表示命令包正在处理。0x0a 为事件包的第二个参数

NUM_HCI_Command_Packets，表示主机最多可在向主机控制器发 10 个命令包。0x01 04 为第三个参数 Command_Opcode， 表示此事件包是对 Inquiry 命令包的响应。

数据包：ACL 和 SCO 数据包中的 Connection Handle 即连接句柄是一个 12 比特的标志符，用于唯一确认两台蓝牙设备间的数据或语音连接，可以看作是两台蓝牙设备间唯一的数据通道的标识。两台设备间只能有一条 ACL 连接，也就是只有一个 ACL 的连接句柄，相应 L2CAP 的信道都是建立在这个连接句柄表示的数据通道上；两台设备间可以有多个 SCO 的连接，则一对设备间会有多个 SCO 的连接句柄。连接句柄在两设备连接期间一直存在，不管设备处于什么状态。在 ACL 数据包中，Flags 分为 PB Flag 和 BC Flag，PB Flag 为包的界限标志，PB Flag=0x00 表示此数据包为上层协议包（如 L2CAP 包）的起始部分；PB Flag=0x01 表示此数据包为上层协议包（如 L2CAP 包）的后续部分。BC Flag 为广播发送的标志，BC Flag=0x00 表示无广播发送，只是点对点的发送；BC Flag=0x01 表示对所有处于激活状态的从设备进行广播发送，BC Flag=0x02 表示对所有的从设备包括处于休眠状态的从设备进行广播发送。ACL 和 SCO 数据包中的 Data Total Length 都表示所载荷的数据的长度，以字节位单位。

4.3 通信过程的研究与分析

当主机与基带之间用命令的方式进行通信时，主机向主机控制器发送命令包。主机控制器完成一个命令，大多数情况下，它会向主机发出一个命令完成事件包（Command Complete Packet），包中携带命令完成的信息。有些命令不会收到命令完成事件，而会收到命令状态事件包（Command Status Packet），当收到该事件则表示主机发出的命令已经被主机控制器接收并开始处理，过一段时间该命令被执行完毕时，主机控制器会向主机发出相应的事件包来通知主机。如果命令参数有误，则会在命令状态事件中给出相应错误码。假如错误出现在一个返回 Command Complete 事件包的命令中，则此 Command Complete 事件包不一定含有此命令所定义的所有参数。状态参数作为解释错误原因同时也是第一个返回的参数，总是要返回的。假如紧随状态参数之后是连接句柄或蓝牙的设备地址，则此参数也总是要返回，这样可判别出此 Command Complete 事件包属于那个实例的一个命令。在这种情况下，事件包中连接句柄或蓝牙的设备地址应与命令包种的相应参数一致。假如错误出现在一个不返回 Command Complete 事件包的命令中，则事件包包含的所有参数都不一定是有效的。主机必须根据于此命令相联系的事件包中的状态参数来决定它们的有效性。

五 Bluetooth 之编程实现

5.1 HCI 层编程

Host Controller Interface(HCI) 是用来沟通 Host 和 Module。Host 通常就是 PC，Module 则是以各种物理连接形式（USB,serial,pc-card 等）连接到 PC 上的 bluetooth Dongle。

在 Host 这一端：application,SDP,L2cap 等协议都是软件形式提出的（Bluez 中是以 kernel 层程序)。在 Module 这一端：Link Manager, BB, 等协议都是硬件中 firmware 提供的。
而 HCI 则比较特殊，它一部分在软件中实现，用来给上层协议和程序提供访问接口（Bluez 中,hci.c hci_usb.c，hci_sock.c 等）.另一部分也是在 Firmware 中实现，用来将软件部分的指令等用底层协议明白的方式传递给底层。

居于 PC 的上层程序与协议和居于 Modules 的下层协议之间通过 HCI 沟通，有 4 种不同形式的传输：Commands, Event, ACL Data, SCO/eSCO Data。

(1) 打开一个 HCI Socket---int hci_open_dev(int dev_id):

这个 function 用来打开一个 HCI Socket。它首先打开一个 HCI protocol 的 Socket（房间），并将此 Socket 与 device ID=参数 dev_id 的 Dongle 绑定起来。只有 bind 后，它才将 Socket 句柄与 Dongle 对应起来。

注意，所有的 HCI Command 发送之前，都需要使用 hci_open_dev 打开并绑定。

(2) 关闭一个 HCI Socket--- int hci_close_dev(int dd)

简单的关闭使用 hci_open_dev 打开的 Socket。

(3) 向 HCI Socket 发送 request---int hci_send_req(int dd, struct hci_request *r, int to)

BlueZ 提供这个 function 非常有用，它可以实现一切 Host 向 Modules 发送 Command 的功能。

参数 1：HCI Socket。

参数 2：Command 内容。

参数 3：以 milliseconds 为单位的 timeout.

下面详细解释此 function 和用法：

当应用程序需要向 Dongle(对应为一个 bind 后的 Socket)发送 Command 时，调用此 function.

参数一 dd 对应一个使用 hci_open_dev（）打开的 Socket（Dongle）。

参数三 to 则为等待 Dongle 执行并回复命令结果的 timeout.以毫秒为单位。

参数二 hci_request * r 最为重要,首先看它的结构：

```
struct hci_request {
  uint16_t ogf;       //Opcode Group
  uint16_t ocf;       //Opcode Command
  int          event;    //此 Command 产生的 Event 类型。
  void         *cparam; //Command 参数
  int          clen;       //Command 参数长度
  void         *rparam;    //Response 参数
  int          rlen;       //Response 参数长度
};
```

ogf,ocf 不用多说，对应前面的图就明白这是 Group Code 和 Command Code。这两项先确定下来，然后可以查 HCI Spec。察看输入参数（cparam）以及输出参数（rparam）含义。至于他们的结构以及参数长度，则在~/include/net/bluetooth/hci.h 中有定义。

至于 event.如果设置，它会被 setsockopt 设置于 Socket。

(4) 得到指定 Dongle BDAddr---int hci_read_bd_addr(int dd, bdaddr_t *bdaddr, int to)；

参数 1：HCI Socket,使用 hci_open_dev（）打开的 Socket（Dongle）。

参数 2：输出参数，其中会放置 bdaddr.

参数 3：以 milliseconds 为单位的 timeout.

(5) 读写 Dongle Name：

int hci_read_local_name(int dd, int len, char *name, int to)

int hci_write_local_name(int dd, const char *name, int to)

参数 1：HCI Socket,使用 hci_open_dev（）打开的 Socket（Dongle）。

参数 2：读取或设置 Name。

参数 3：以 milliseconds 为单位的 timeout.

注意：这里的 Name 与 IOCTL HCIGETDEVINFO 得到 hci_dev_info 中的 name 不同。

(6) 得到 HCI Version:

int hci_read_local_version(int dd, struct hci_version *ver, int to)

(7) 得到已经 UP 的 Dongle BDaddr--- int hci_devba(int dev_id, bdaddr_t *bdaddr)；

dev_id: Dongle Device ID.

bdaddr:输出参数，指定 Dongle 如果 UP， 则放置其 BDAddr。

(8) inquiry 远程 Bluetooth Device：

int hci_inquiry(int dev_id, int len, int nrsp, const uint8_t *lap, inquiry_info **ii, long flags)

hci_inquiry（）用来命令指定的 Dongle 去搜索周围所有 bluetooth device.并将搜索到的 Bluetooth Device bdaddr 传递回来。

参数 1：dev_id：指定 Dongle Device ID。如果此值小于 0，则会使用第一个可用的 Dongle。

参数 2：len: 此次 inquiry 的时间长度（每增加 1，则增加 1.25 秒时间）

参数 3：nrsp:此次搜索最大搜索数量，如果给 0。则此值会取 255。

参数 4：lap:BDADDR 中 LAP 部分，Inquiry 时这块值缺省为 0X9E8B33.通常使用 NULL。则自动设置。

参数 5：ii:存放搜索到 Bluetooth Device 的地方。给一个存放 inquiry_info 指针的地址，它会自动分配空间。并把那个空间头地址放到其中。

参数 6：flags:搜索 flags.使用 IREQ_CACHE_FLUSH，则会真正重新 inquiry。否则可能会传回上次的结果。

返回值是这次 Inquiry 到的 Bluetooth Device 数目。

注意：如果*ii 不是自己分配的，而是让 hci_inquiry()自己分配的，则需要调用 bt_free（）来帮它释放空间。

(9) 得到指定 BDAddr 的 reomte device Name:

int hci_read_remote_name(int dd, const bdaddr_t *bdaddr, int len, char *name, int to)

参数 1：使用 hci_open_dev（）打开的 Socket。

参数 2：对方 BDAddr.

参数 3：name 长度。

参数 4：(out)放置 name 的位置。

参数 5：等待时间。

(10) 读取连接的信号强度：

int hci_read_rssi(int dd, uint16_t handle, int8_t *rssi, int to)

注意，所有对连接的操作，都会有一个参数，handle.这个参数是连接的 Handle。前面讲过如何得到连接 Handle 的。

5.2 L2CAP 层编程

逻辑连接控制和适配协议 (L2CAP) 为上层协议提供面向连接和无连接的数据服务，并提供多协议功能和分割重组操作。L2CAP 充许上层协议和应用软件传输和接收最大长度为 64K 的 L2CAP 数据包。

L2CAP 基于 通道(channel) 的概念。 通道 (Channel) 是位于基带 (baseband) 连接之上的逻辑连接。每个通道以多对一的方式绑定一个单一协议 (single protocol)。多个通道可以绑定同一个协议，但一个通道不可以绑定多个协议。 每个在通道里接收到的 L2CAP 数据包被传到相应的上层协议。 多个通道可共享同一个基带连接。

L2CAP 处于 Bluetooth 协议栈的位置如下：

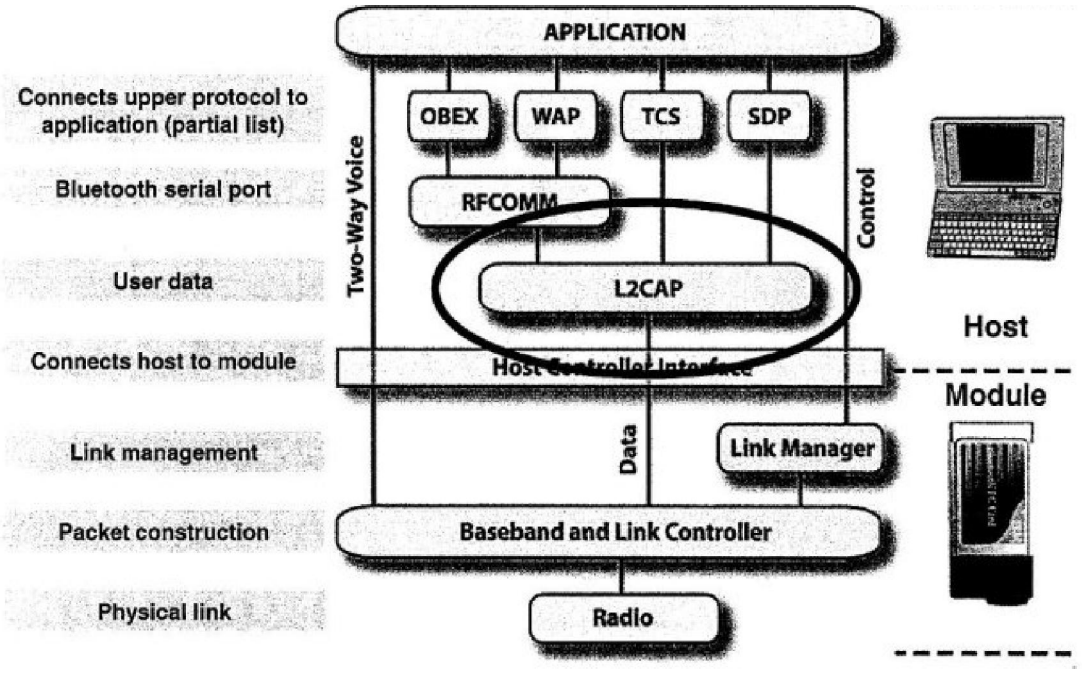

图 6 L2CAP 协议

L2CAP 使用 L2CAP 连接请求（Connection Request ）命令中的 PSM 字段实现协议复用。L2CAP 可以复用发给上层协议的连接请求，这些上层协议包括服务发现协议 SDP（PSM = 0x0001）、RFCOMM（PSM = 0x0003）和电话控制（PSM = 0x0005）等。

Protocol	PSM	Reference
SDP	0x0001	See *Bluetooth* Service Discovery Protocol (SDP), Bluetooth SIG.
RFCOMM	0x0003	See RFCOMM with TS 07.10, Bluetooth SIG.
TCS-BIN	0x0005	See *Bluetooth* Telephony Control Specification / TCS Binary, Bluetooth SIG.
TCS-BIN-CORDLESS	0x0007	See *Bluetooth* Telephony Control Specification / TCS Binary, Bluetooth SIG.
BNEP	0x000F	See *Bluetooth* Network Encapsulation Protocal, Bluetooth SIG.
HID_Control	0x0011	See Human Interface Device , Bluetooth SIG.
HID_Interrupt	0x0013	See Human Interface Device, Bluetooth SIG.
UPnP	0x0015	See [ESDP] , Bluetooth SIG.
AVCTP	0x0017	See Audio/Video Control Transport Protocol , Bluetooth SIG.
AVDTP	0x0019	See Audio/Video Distribution Transport Protocol , Bluetooth SIG.
AVCTP_Browsing	0x001B	See Audio/Video Remote Control Profile, Bluetooth SIG
UDI_C-Plane	0x001D	See the Unrestricted Digital Information Profile [UDI], Bluetooth SIG

图 7 PSM 协议字段

L2CAP 编程非常重要，它和 HCI 基本就是 Linux Bluetooth 编程的基础了。几乎所有协议的连接，断连，读写都是用 L2CAP 连接来做的

(1) 创建 L2CAP Socket：

socket(PF_BLUETOOTH, SOCK_RAW, BTPROTO_L2CAP);

domain=PF_BLUETOOTH, type 可以是多种类型。protocol=BTPROTO_L2CAP

(2) 绑定

```
memset(&addr, 0, sizeof(addr));
addr.l2_family = AF_BLUETOOTH;
bacpy(&addr.l2_bdaddr, &bdaddr); //bdaddr 为本地 Dongle BDAddr
if (bind(sk, (struct sockaddr *) &addr, sizeof(addr)) < 0)
{
  perror("Can't bind socket");
  goto error;
```

```
    }
```

(3) 连接

```
    memset(&addr, 0, sizeof(addr));
    addr.l2_family = AF_BLUETOOTH;
    bacpy(addr.l2_bdaddr, src);
    addr.l2_psm = xxx;
     if (connect(sk, (struct sockaddr *) &addr, sizeof(addr)) < 0)
    {
       perror("Can't connect");
       goto error;
    }
    注意：
    struct sockaddr_l2
    {
     sa_family_t l2_family;   //必须为 AF_BLUETOOTH
     unsigned short l2_psm;   //与前面 PSM 对应,这一项很重要
     bdaddr_t l2_bdaddr;        //Remote Device BDADDR
     unsigned short l2_cid;
    };
```

(4) 发送数据到 Remote Device：

send()或 write()都可以。

(5) 接收数据

revc() 或 read()都可以

5.3 SDP 层编程

服务发现协议(SDP 或 Bluetooth SDP)在蓝牙协议栈中对蓝牙环境中的应用程序有特殊的含意，发现哪个服务是可用的和确定这些可用服务的特征。SDP 定义了 bluetooth client 发现可用 bluetooth server 服务和它们的特征的方法。这个协议定义了客户如何能够寻找基于特定属性的服务而不让客户知道可用服务的任何知识。SDP 提供发现新服务的方法，在当客户登录到正在操作的蓝牙服务器的一个区域时是可用的时。

Service discovery 机制提供 client 应用程序侦测 server 应用程序提供的服务的能力，并且能够得到服务的特性。服务的品质包含服务 type 或服务 class.

SDP 也提供 SDP server 与 SDP client 之间的通讯。SDP server 维护着一个服务条目(service record)列表.每个服务条目描述一个单独的服务属性。SDP client 可以通过发送 SDP request 来得到服务条目。

如果一个 client 或者依附于 client 之上的应用程序决定使用某个 service. 它创建一个单独的连接到 service 提供者。 SDP 只提供侦测 Service 的机制，但不提供如何利用这些 Service 的机制。Sam 觉得，这里其实是说：SDP 只提供侦测 Service 的办法，但如何用，SDP 不管。

每个 Bluetooth Device 最多只能拥有一个 SDP Server。如果一个 Bluetooth Device 只担任 Client，那它不需要 SDP Server。但一个 Bluetooth Device 可以同时担当 SDP Server 和 SDP client.

（1）Service Record(Service 条目)：

一个 service 是一个实体为另一个实体提供信息，执行动作或控制资源。一个 service 可以由软件，硬件或软硬件结合提供。

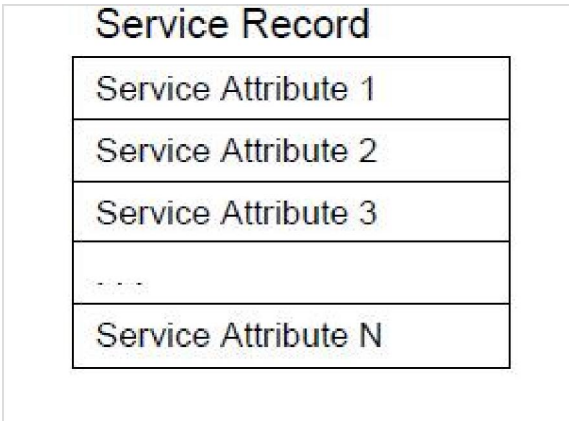

图 8 Service Record

所有的Service信息都包含于一个Service Record内。一个Service Record 包含一个Service attribute(Service 属性) list.

（2）Service Attribute(Service 属性)：

每个 Service 属性描述 servcie 的特性.一个 Service Attribute 由 2 部分：Attribute ID + Attribute Value。

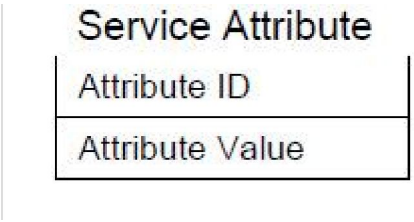

图 9 Service Attribute

（3）Service Class：

每个 Service 都是某个 Service Class 的实例. Service Class 定义了 Service Record 中包含的 Service 属性。属性 ID，属性值都被定义好了。

每个 Service Class 也有一个独特 ID。这个 Service Class 标识符包含在属性值 ServiceClassIDList 属性中。并描绘为 UUID。自从 Service Record 中的属性格式以及含义依赖于 Service Class 后，ServiceClassIDList 属性变得非常重要。

（4）Searching For Service:

Service Search transaction(事务？)允许 client 得到 Service Record Handle。一旦 SDP Client 得到 Service Record Handle，它就可以请求这个 Record 内具体属性的值。

如果某个属性值 UUID，则可以通过查找 UUID 查到这个属性。

UUID： universally unique identifier.(唯一性标识符)

总之，DP 协议栈使用 request/response 模式工作，每个传输过程包括一个 request protocol data unit(PDU)和一个 response PDU. SDP 使用 L2CAP 连接传输数据。在发送 Request PDU 但未收到 Response PDU 之前，不能向同一个 server 再发送 Request PDU。

六 Bluetooth 之启动过程实现

对于蓝牙无论最底层的硬件驱动如何实现，都会在 HCI 层进行统一。也就是说，HCI 在主机端的驱动主要是为上层提供统一接口，让上层协议不依赖于具体的硬件实现。HCI 在硬件中的固件与 HCI 在主机端的驱动通信方式有多种，比如 UART，USB 和 SDIO 等。

HCI 层在所有的设备面前都被抽象为一个 hci_dev 结构体，因此，无论实际的设备是哪

种蓝牙设备、通过什么方式连接到主机，都需要向 HCI 层和蓝牙核心层注册一个 hci_dev 设备，注册过程由 hci_registe_dev()函数来完成，同时也可以通过 hci_unregister_dev()函数卸载一个蓝牙设备。

具体的蓝牙驱动有很多，常用的在 linux 内核都自带有驱动。比如：hci_vhci.c 为蓝牙虚拟主控制器驱动程序，hci_uart.c(或者 hci_ldisc.c)为串口接口主控制器驱动程序，btusb.c 为 USB 接口主控制器驱动程序，btsdio.c 为 SDIO 主控制器驱动程序。

6.1 Bluetooth 启动步骤

（1）串口驱动必须要先就绪(uart 蓝牙而言)，这是 cpu 和蓝牙模块之间的桥梁。
（2）蓝牙初始化，模块上电和 PSKEY 的设置。
（3）通过 hciattach 建立串口和蓝牙协议层之间的数据连接通道。

6.2 Bluetooth 启动流程

（1） 打开蓝牙电源，通过 rfkill 来 enable;(system/bluetooth/bluedroid/bluetooth.c)
（2）启动 service hciattch -n -s 115200 /dev/ttyS2 bcm2035 115200；
（3）检测 HCI 是否成功（接受 HCIDEVUP socket 来判断或 hciconfig hci0 up）；
（4） hcid deamon start up。

6.3 Bluetooth 数据流向

（1）uart 口取得蓝牙模块的数据；
（2）uart 口通过 ldisc 传给 hci_uart;
（3）hci_uart 传给在其上的 bcsp；
（4）bcsp 传给 hci 层；
（5）hci 层传给 l2cap 层
（6）l2cap 层再传给 rfcomm；
蓝牙模块上电：

一般是通过一个 GPIO 来控制的,通常是先高再低再高；
PSKEY 的设置：

通过串口发送命令给蓝牙模块，对于串口必须要知道的是要能通讯，必须得设好波特率，另外一方面蓝牙模块的晶振频率也必须要设，否则它不知道该怎么跳了；当然不同的芯片可能初始化的过程也不一样，也许还要下载 firmware 等等，一般是通过 bccmd 来完成的。
经过上面的设置基本上蓝牙模块以及可以正常工作了；

但是还没有和上面的协议层建立纽带关系，也就是说从 uart 收到的数据还没有传给 hci 层；如何把 uart 也就是蓝牙模块传上来的数据交给 hci 层，在驱动里面是通过一个叫做 disc 的机制完成的，这个机制本意是用来做过滤或者限制收上来的字符的，但是在蓝牙驱动里面则直接把数据传给了蓝牙协议层，再也不回到串口的控制了；

6.4 Bluez 控制流程

class bluetoothsetting 是 UI 的入口，通过按 button scan 进入搜索状态，

applicaton 层调用 bluetoothdevice, 接着就是 bluetoothservice 的调用，
bluetoothservice 调用 native 方法，到此全部的 java 程序结束了。
下面的调用都是 JNI, cpp 实现的。android_server_bluetoothservice.cpp 里面实现了 native 方法，最终通过 dbus 封装，调用 HCID deamon 的 function DiscoverDevice。

6.5 Bluetooth 启动过程分析

1. 各协议层的注册

(1)在 af_bluetooth.c 中首先调用 bt_init()函数完成初始化，打印信息 Bluetooth: Core ver 2.16 随后调用函数 sock_register()注册 sock,打印信息 Bluetooth: HCI device and connection manager initialized

(2)接着调用函数 hci_sock_init(),l2cap_init(),sco_init()实现各个协议的初始化

(3)在 hci_sock.c 中，注册 bt_sock 协议，打印信息 Bluetooth: HCI socket layer initialized

(4)在 L2cap_core.c 中，调用 l2cap_init_socks()函数，完成初始化，打印信息 Bluetooth: L2CAP socket layer initialized

(5)在 sco.c 中，注册 BTPROTO_SCO 协议，完成初始化，打印信息 Bluetooth: SCO socket layer initialized

2. 各硬件的初始化

(1)在 Hci_ldisc.c 中，完成 hci_uart_init 的初始化，打印信息 Bluetooth: HCI UART driver ver 2.2

(2)在 Hci_h4.c 中，完成 h4_init 的初始化，打印信息 Bluetooth: HCI H4 protocol initialized

(3)在 Hci_ath.c 中，完成 ath_Init 的初始化，打印信息 Bluetooth:HCIATH3K protocol initialized

(4)在 hci_ibs.c 中，完成 ibs_init 的初始化，打印信息 Bluetooth:HCI_IBS protocol initialized

(5)在 Core.c 中，完成 rfcomm_init 的初始化，其中调用函数 rfcomm_init_ttys()、rfcomm_init_sockets()，实现 rfcomm 的初始化，打印信息 Bluetooth:RFCOMM ver 1.11

(6)在 TTY.C 中，完成 rfcomm_init_ttys 的初始化，实现 rfcomm_tty_driver 的注册，打印信息 Bluetooth:RFCOMM TTY layer initialized

(7)在 Sock.c 中，完成 rfcomm_init_sockets 的初始化，实现 BTPROTO_RFCOMM 的注册，打印信息 Bluetooth:RFCOMM socket layer initialized

(以上通过 rfcomm 完成 uart 串口的初始化，在 G:\M8960\M8960AAAAANLYA1023\kernel\net\bluetooth\rfcomm)

3. bnep-蓝牙网络封装协议

在 Core.c 中，完成 bnep_init 的初始化，实现 bnep 的初始化，打印信息 Bluetooth:BNEP (Ethernet Emulation) ver 1.3/Bluetooth:BNEP filters: protocol

(以上完成蓝牙网络封装协议的初始化 G:\M8960\M8960AAAAANLYA1023\kernel\net\bluetooth\bnep)

七 Bluetooth 之驱动移植

7.1 android 系统配置

build\target\board\generic 下面的 generic.mk 增加：

BOARD_HAVE_BLUETOOTH := true
这个是由于编译相关蓝牙代码时需要这个宏，请看：\system\bluetooth\android.mk
ifeq ($(BOARD_HAVE_BLUETOOTH),true)
  include $(all-subdir-makefiles)
endif
在 external\bluetooth 也同样存在此宏起作用

7.2 启动项修改

system\core\rootdir 下 init.rc 文件增加：
service hciattach /system/bin/hciattach -n -s 115200 /dev/ttyS2 bcm2035 115200
user bluetooth
group bluetooth net_bt_admin
disabled
oneshot
请放在 service bluetoothd /system/bin/bluetoothd -n 类似这种语句的后面任意位置即可。

7.3 电源管理 rfkill 驱动

Kernel/driver/bluetooth/bluetooth-power.c
高通的这个文件基本上不用动。
在 kernel\arch\arm\mach_msm7x27.c: static int bluetooth_power(int on)中
实现：上电：把 bt_reset pin 和 bt_reg_on pin 拉低
mdelay（10）；
把 bt_resetpin 和 bt_reg_on pin 拉高
mdelay（150）
下电：把 bt_reset pin 和 bt_reg_on pin 拉低

7.4 Rebuild Android image and reboot

命令行测试：
echo 0 >/sys/class/rfkill/rfkill0/state //BT 下电
echo 1 >/sys/class/rfkill/rfkill0/state //BT 上电
brcm_patchram_plus-d--patchram
/etc/firmware/BCM4329B1_002.002.023.0061.0062.hcd/dev/ttyHS0
hciattach -s115200 /dev/ttyHS0 any
没任何错误提示是可以用以下测试
hciconfig hci0up
hcitool scan

7.5 实现 BT 睡眠唤醒机制

Kernel\drivers\bluetooth\bluesleep.c 一般来说这个文件改动比较少，但可能逻辑上会有些问

题。需要小的改动。
在 kernel\arch\arm\mach_xxx/board_xxx.c：bluesleep_resources 中定义 gpio_host_wake（BT 唤醒 host 脚）、gpio_ext_wake（host 唤醒 BT 脚）、host_wake（BT 唤醒 host 的中断号）。
注：各个平台的 board_xxx.c 文件名字不同，请客户确认

7.6 系统集成

1）在 init.qcom.rc 中确认有下面的内容：

```
service hciattach/system/bin/sh /system/etc/init.qcom.bt.sh
    user bluetooth
    group qcom_oncrpc bluetooth net_bt_admin
    disabled
    oneshot
```

2）修改 init.qcom.bt.sh
确认有：

```
BLUETOOTH_SLEEP_PATH=/proc/bluetooth/sleep/proto
echo 1 >$BLUETOOTH_SLEEP_PATH
/system/bin/hciattach-n /dev/ttyHS0 any 3000000 flow &
```

改为：

```
./brcm_patchram_plus--enable_lpm –enable_hci --patchram /system/etc/wifi/BCM4329BT.hcd --baudrate3000000 /dev/ttyHS0 &
```

注掉：高通下载 firmware 的命令。
最后，重新编译 system。此时 BT 应该能运行了。

八 Bluetooth 之调试与编译

8.1 Bluetooth 驱动调试

调试你的蓝牙实现，可以通过读跟蓝牙相关的 logs(adb logcat)和查找 ERROR 和警告消息。Android 使用 Bluez，同时会带来一些有用的调式工具。下面的片段为了提供一个建议的例子：

```
1  hciconfig -a                 # print BT chipset address and features. Useful to check if you can communicate with your BT chipset.
2  hcidump -XVt                 # print live HCI UART traffic.
3  hcitool scan                 # scan for local devices. Useful to check if RX/TX works.
4  l2ping ADDRESS               # ping another BT device. Useful to check if RX/TX works.
5  sdptool records ADDRESS # request the SDP records of another BT device.
```

守护进程日志
hcid(STDOUT)和 hciattach(STDERR)的守护进程日志缺省是被写到/dev/null。编辑 init.rc 和 init.PLATFORM.rc 在 logwrapper 下运行这些守护进程，把它们输出到 logcat。
hciconfig -a 和 hcitool
如果你编译你自己的 system.img，除了 hcitool 扫描不行，hciconfig -a 是可以工作的，尝试安装固件到蓝牙芯片。

8.2 Bluetooth 调试工具

BlueZ 为调试和与蓝牙子系统通信提供很多设置命令行工具，包含下面这些：
（1）hciconfig
（2）hcitool
（3）hcidump
（4）sdptool
（5）dbus-send
（6） dbus-monitor

九 Bluetooth 之应用程序开发

9.1 Bluetooth 的 API 开发

Android 平台包含了对 Bluetooth 协议栈的支持，允许机器通过 Bluetooth 设备进行无线数据交换。应用框架通过 Android Bluetooth API 访问 Bluetooth 功能模块。这些 API 能让应用无线连接其他 Bluetooth 设备，实现点对点和多点之间的通信。
运用蓝牙 API，Android 应用程序可以完成如下操作：
（1）扫描其他 Bluetooth 设备。
（2）查询配对 Bluetooth 设备的本地 Bluetooth 适配器。
（3）建立 RFCOMM 通道。
（4）通过服务探索连接到其他设备。
（5）与其他设备进行数据传输。
（6）管理多个连接

9.2 The Basics 开发

本文描述如何使用 Android Bluetooth APIs 完成 Bluetooth 通讯的 4 个必要任务：设置 Bluetooth，搜寻本地配对或者可用的 Bluetooth 设备，连接 Bluetooth 设备，与 Bluetooth 设备进行数据传输。

所有可用的 Bluetooth APIs 都包含在 android.bluetooth 包中。下面是建立 Bluetooth 连接需要用到的类和接口的总结：

（1）BluetoothAdapter

描述本地 Bluetooth 适配器（Bluetooth 接收器）。BluetoothAdapter 是所有 Bluetooth 相关活动的入口。运用 BluetoothAdapter 可以发现其他 Bluetooth 设备，查询连接（或配对）的设备列表，用已知 MAC 地址实例化一个 BluetoothDevice 对象，创建一个 BluetoothServerSocket 对象侦听其他设备的通信。

（2）BluetoothDevice

描述一个远程 Bluetooth 设备。可以用它通过一个 BluetoothSocket 请求一个远程设备的连接，或者查询远程设备的名称、地址、类、连接状态等信息。

（3）BluetoothSocket

描述一个 Bluetooth Socket 接口（类似于 TCP Socket）。应用通过 InputStream 和

OutputStream 与另外一个 Bluetooth 设备交换数据，即它是应用与另外一个设备交换数据的连接点。

（4）BluetoothServerSocket

BluetoothServerSocket 是一个开放的 socket 服务器，用来侦听连接进来的请求（类似于 RCP ServerSocket）。为了连接两个 Android 设备，一个设备必须使用该类来开启一个 socket 做服务器，当另外一个设备对它发起连接请求时并且请求被接受时，BluetoothServerSocket 会返回一个连接的 BluetoothSocket 对象。

（5）BluetoothClass

BluetoothClass 是用来定义设备类和它的服务的只读属性集。然而，它并不是可靠的描述设备支持的所有 Bluetooth 配置和服务，而只是一些设备类型的有用特征。

（6）BluetoothProfile

Bluetooth Profile 是两个设备基于蓝牙通讯的无线接口描述。Profile 定义了设备如何实现一种连接或者应用，你可以把 Profile 理解为连接层或者应用层协议。 比如，如果一家公司希望它们的 Bluetooth 芯片支援所有的 Bluetooth 耳机，那么它只要支持 HeadSet Profile 即可，而无须考虑该芯片与其它 Bluetooth 设备的通讯与兼容性问题。如果你想购买 Bluetooth 产品，你应该了解你的应用需要哪些 Profile 来完成，并且确保你购买的 Bluetooth 产品支持这些 Profile。

（7）BluetoothHeadset

提供移动电话的 Bluetooth 耳机支持。包括 Bluetooth 耳机和 Hands-Free (v1.5) profiles。

（8）BluetoothA2dp

定义两个设备间如何通过 Bluetooth 连接进行高质量的音频传输。A2DP（Advanced Audio Distribution Profile）：高级音频传输模式。

（9）BluetoothProfile.ServiceListener

一个接口描述，在与服务连接或者断连接的时候通知 BluetoothProfile IPC（这是内部服务运行的一个特定的模式<profile>）。

9.3 Bluetooth Permissions 开发

要使用 Bluetooth 功能，至少需要 2 个 Bluetooth 权限：BLUETOOTH 和 BLUETOOTH_ADMIN. BLUETOOTH：用来授权任何 Bluetooth 通信，如请求连接，接受连接，传输数据等。

BLUETOOTH_ADMIN：用来授权初始化设备搜索或操作 Bluetooth 设置。大多数应用需要它的唯一场合是用来搜索本地 Bluetooth 设备。本授权的其他功能不应该被使用，除非是需要修改 Bluetooth 设置的“power manager（电源管理）”应用。

注意：需要 BLUETOOTH_ADMIN 权限的场合，BLUETOOTH 权限也是必需的。需要在 manifest 文件中声明 Bluetooth 权限，示例如下：

```
<manifest ... >
  <uses-permission android:name="android.permission.BLUETOOTH" />
  ...
</manifest>
```

9.4 Setting Up Bluetooth 服务

在用 Bluetooth 通讯之前，需要确认设备是否支持 Bluetooth，如果支持，还得确保 Bluetooth 是可用的。

如果设备不支持 Bluetooth，需要优雅的将 Bluetooth 置为不可用。如果支持 Bluetooth，但没有开启，可以在应用中请求开启 Bluetooth。该设置使用 BluetoothAdapter.通过两个步骤完成。

（1）获取 BluetoothAdapter

BluetoothAdapter 是每个 Bluetooth 的 Activity 都需要用到的。用静态方法 getDefaultAdapter() 获取 BluetoothAdapter，返回一个拥有 Bluetooth 适配器的 BluetoothAdapter 对象。如果返回 null，说明设备不支持 Bluetooth，关于 Bluetooth 的故事到此就结束了（因为你干不了什么了）。示例：

```
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
if (mBluetoothAdapter == null) {
    // Device does not support Bluetooth
}
```

（2）Enable Bluetooth

接下来，就是确保 Bluetooth 功能是开启的。调用 isEnabled()来检查 Bluetooth 当前是否是开启的。用 ACTION_REQUEST_ENABLE action Intent 调用 startActivityForResult()来请求开启 Bluetooth，这会通过系统设置发出一个 Bluetooth 使能请求（并且不会停止本应用程序）。示例：

```
if (!mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

用户请求使能 Bluetooth 时，会显示一个对话框。选择“Yes”，系统会使能 Bluetooth，并且焦点会返回你的应用程序。

如果使能 Bluetooth 成功，你的 Activity 会在 onActivityResult()回调函数中收到 RESULT_OK 的结果码。如果 Bluetooth 使能因发生错误（或用户选择了“No”）而失败，收到的结果码将是 RESULT_CANCELED。

作为可选项，应用也可以侦听 ACTION_STATE_CHANGED broadcast Intent，这样无论 Bluetooth 状态何时被改变系统都会发出 broadcast（广播）。该广播包含附加的字段信息 EXTRA_STATE 和 EXTRA_PREVIOUS_STATE 分别代表新的和旧的 Bluetooth 状态，该字段可能的值为 STATE_TURNING_ON, STATE_ON, STATE_TURNING_OFF, 和 STATE_OFF。应用运行时，侦听 ACTION_STATE_CHANGED 广播来检测 Bluetooth 状态的改变是很有用的。

提示：启用 Bluetooth 可被发现功能能够自动开启 Bluetooth。如果在完成 Activity 之前需要持续的使能 Bluetooth 可被发现功能，那么上面的第 2 步就可以忽略。

9.5 Finding Devices 服务

使用 BluetoothAdapter 可以通过设备搜索或查询配对设备找到远程 Bluetooth 设备。Device discovery（设备搜索）是一个扫描搜索本地已使能 Bluetooth 设备并且从搜索到的设备请求一些信息的过程（有时候会收到类似“discovering”，“inquiring”或“scanning”）。但是，搜索到的本地 Bluetooth 设备只有在打开被发现功能后才会响应一个 discovery 请求，响应的信息包括设备名，类，唯一的 MAC 地址。发起搜寻的设备可以使用这些信息来初始化跟被发现的设备的连接。

一旦与远程设备的第一次连接被建立，一个 pairing 请求就会自动提交给用户。如果设备已配对，配对设备的基本信息（名称，类，MAC 地址）就被保存下来了，能够使用 Bluetooth

API 来读取这些信息。使用已知的远程设备的 MAC 地址，连接可以在任何时候初始化而不必先完成搜索（当然这是假设远程设备是在可连接的空间范围内）。
需要记住，配对和连接是两个不同的概念：

配对意思是两个设备相互意识到对方的存在，共享一个用来鉴别身份的链路键（link-key），能够与对方建立一个加密的连接。连接意思是两个设备现在共享一个 RFCOMM 信道，能够相互传输数据。

目前 Android Bluetooth API's 要求设备在建立 RFCOMM 信道前必须配对（配对是在使用 Bluetooth API 初始化一个加密连接时自动完成的）。
下面描述如何查询已配对设备，搜索新设备。

注意：Android 的电源设备默认是不能被发现的。用户可以通过系统设置让它在有限的时间内可以被发现，或者可以在应用程序中要求用户使能被发现功能。
（1）Querying paired devices

在搜索设备前，查询配对设备看需要的设备是否已经是已经存在是很值得的，可以调用 getBondedDevices()来做到，该函数会返回一个描述配对设备 BluetoothDevice 的结果集。例如，可以使用 ArrayAdapter 查询所有配对设备然后显示所有设备名给用户：

```
Set<BluetoothDevice> pairedDevices = mBluetoothAdapter.getBondedDevices();
// If there are paired devices
if (pairedDevices.size() > 0) {
    // Loop through paired devices
    for (BluetoothDevice device : pairedDevices) {
        // Add the name and address to an array adapter to show in a ListView
        mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
    }
}
```

BluetoothDevice 对象中需要用来初始化一个连接唯一需要用到的信息就是 MAC 地址。
（2）Discovering devices

要开始搜索设备，只需简单的调用 startDiscovery()。该函数时异步的，调用后立即返回，返回值表示搜索是否成功开始。搜索处理通常包括一个 12 秒钟的查询扫描，然后跟随一个页面显示搜索到设备 Bluetooth 名称。
应用中可以注册一个带 CTION_FOUND Intent 的 BroadcastReceiver，搜索到每一个设备时都接收到消息。对于每一个设备，系统都会广播 ACTION_FOUND Intent，该 Intent 携带着而外的字段信息 EXTRA_DEVICE 和 EXTRA_CLASS，分别包含一个 BluetoothDevice 和一个 BluetoothClass。下面的示例显示如何注册和处理设备被发现后发出的广播：

```
// Create a BroadcastReceiver for ACTION_FOUND
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            // Add the name and address to an array adapter to show in a ListView
            mArrayAdapter.add(device.getName() + "\n" + device.getAddress());
```

```
            }
        }
};
// Register the BroadcastReceiver
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(mReceiver, filter); // Don't forget to unregister during onDestroy
```

警告：完成设备搜索对于 Bluetooth 适配器来说是一个重量级的处理，要消耗大量它的资源。一旦你已经找到一个设备来连接，请确保你在尝试连接前使用了 cancelDiscovery()来停止搜索。同样，如果已经保持了一个连接的时候，同时执行搜索设备将会显著的降低连接的带宽，所以在连接的时候不应该执行搜索发现。

（3）Enabling discoverability

如果想让本地设备被其他设备发现，可以带 ACTION_REQUEST_DISCOVERABLE action Intent 调用 startActivityForResult(Intent, int) 方法。该方法会提交一个请求通过系统刚设置使设备出于可以被发现的模式（而不影响应用程序）。默认情况下，设备在 120 秒后变为可以被发现的。可以通过额外增加 EXTRA_DISCOVERABLE_DURATION Intent 自定义一个值，最大值是 3600 秒，0 表示设备总是可以被发现的（小于 0 或者大于 3600 则会被自动设置为 120 秒）。下面示例设置时间为 300：

```
Intent discoverableIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
startActivity(discoverableIntent);
```

询问用户是否允许打开设备可以被发现功能时会显示一个对话框。如果用户选择“Yes”，设备会在指定时间过后变为可以被发现的。Activity 的 onActivityResult()回调函数被调用，结果码等于设备变为可以被发现所需时长。如果用户选择“No”或者有错误发生，结果码会是 Activity.RESULT_CANCELLED。

提示：如果 Bluetooth 没有启用，启用 Bluetooth 可被发现功能能够自动开启 Bluetooth。在规定的时间内，设备会静静的保持可以被发现模式。如果想在可以被发现模式被更改时受到通知，可以用 ACTION_SCAN_MODE_CHANGED Intent 注册一个 BroadcastReceiver，包含额外的字段信息 EXTRA_SCAN_MODE 和 EXTRA_PREVIOUS_SCAN_MODE 分别表示新旧扫描模式，其可能的值为 SCAN_MODE_CONNECTABLE_DISCOVERABLE （ discoverable mode ），SCAN_MODE_CONNECTABLE（not in discoverable mode but still able to receive connections），SCAN_MODE_NONE（not in discoverable mode and unable to receive connections）。

如果只需要连接远程设备就不需要打开设备的可以被发现功能。只在应用作为一个服务器 socket 的宿主用来接收进来的连接时才需要使能可以被发现功能，因为远程设备在初始化连接前必须先发现了你的设备。

9.6 Connecting Devices 服务

为了建立两个设备之间的应用的连接，需要完成服务器端和客户端，因为一个设备必须打开一个服务器 socket 而另外一个设备必须初始化连接（用服务器端的 MAC 地址）。服务器和客户端在各自获得一个基于同一个 RFCOMM 信道的已连接的 BluetoothSocket 对象后就被认为连接已经建立。这个时候，双方设备可以获取输入输出流，数据传输可以开始了。本节描述如何在两个设备之间初始化连接。

服务器设备和客户端设备用不同的方式获取各自需要的 BluetoothSocket 对象。服务器端

的在接收一个进来的连接时获取到，客户端的在打开一个与服务器端的 RFCOMM 信道的时候获取到。

一个实现技巧是自动把每个设备作为服务器，这样就拥有了一个打开的 socket 用来侦听连接。然后任一设备就能够发起与另一个设备的连接，并成为客户端。另外，一个设备也可以明确的成为“host”，并打开一个服务端 socket，另一个设备可以简单的发起连接。

注意：如果两个设备之前没有配对，那么在连接处理过程中 Android 应用框架会自动显示一个配对请求的通知或对话框给用户。因此，当尝试连接设备时，应用不需要关心设备是否已经配对。RFCOMM 连接会阻塞直到用户成功将设备配对(如果用户拒绝配对或者配对超时了连接会失败)。

（1）Connecting as a server

如果要连接两个设备，其中一个必须充当服务器，通过持有一个打开的 BluetoothServerSocket 对象。服务器 socket 的作用是侦听进来的连接，如果一个连接被接受，提供一个连接好的 BluetoothSocket 对象。从 BluetoothServerSocket 获取到 BluetoothSocket 对象之后，BluetoothServerSocket 就可以（也应该）丢弃了，除非你还要用它来接收更多的连接。

下面是建立服务器 socket 和接收一个连接的基本步骤：

① 通过调用 listenUsingRfcommWithServiceRecord(String, UUID) 得到一个 BluetoothServerSocket 对象。

该字符串为服务的识别名称，系统将自动写入到一个新的服务发现协议（SDP）数据库接入口到设备上的（名字是任意的，可以简单地是应用程序的名称）项。 UUID 也包括在 SDP 接入口中，将是客户端设备连接协议的基础。也就是说，当客户端试图连接本设备，它将携带一个 UUID 用来唯一标识它要连接的服务，UUID 必须匹配，连接才会被接受。

② 通过调用 accept()来侦听连接请求。

这是一个阻塞的调用，知道有连接进来或者产生异常才会返回。只有远程设备发送一个连接请求，并且携带的 UUID 与侦听它 socket 注册的 UUID 匹配，连接请求才会被接受。如果成功，accept()将返回一个连接好的 BluetoothSocket 对象。

③ 除非需要再接收另外的连接，否则的话调用 close()。

close()释放 server socket 和它的资源，但不会关闭连接 accept()返回的连接好的 BluetoothSocket 对象。与 TCP/IP 不同，RFCOMM 同一时刻一个信道只允许一个客户端连接，因此大多数情况下意味着在 BluetoothServerSocket 接受一个连接请求后应该立即调用 close()。accept()调用不应该在主 Activity UI 线程中进行，因为这是个阻塞的调用，会妨碍其他的交互。经常是在在一个新线程中做 BluetoothServerSocket 或 BluetoothSocket 的所有工作来避免 UI 线程阻塞。注意所有 BluetoothServerSocket 或 BluetoothSocket 的方法都是线程安全的。

示例：

下面是一个简单的接受连接的服务器组件代码示例：

```
private class AcceptThread extends Thread {
    private final BluetoothServerSocket mmServerSocket;

    public AcceptThread() {
        // Use a temporary object that is later assigned to mmServerSocket,
        // because mmServerSocket is final
        BluetoothServerSocket tmp = null;
        try {
            // MY_UUID is the app's UUID string, also used by the client code
```

```
            tmp = mBluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME,
MY_UUID);
        } catch (IOException e) { }
        mmServerSocket = tmp;
    }

    public void run() {
        BluetoothSocket socket = null;
        // Keep listening until exception occurs or a socket is returned
        while (true) {
            try {
                socket = mmServerSocket.accept();
            } catch (IOException e) {
                break;
            }
            // If a connection was accepted
            if (socket != null) {
                // Do work to manage the connection (in a separate thread)
                manageConnectedSocket(socket);
                mmServerSocket.close();
                break;
            }
        }
    }

    /** Will cancel the listening socket, and cause the thread to finish */
    public void cancel() {
        try {
            mmServerSocket.close();
        } catch (IOException e) { }
    }
}
```

本例中，仅仅只接受一个进来的连接，一旦连接被接受获取到 BluetoothSocket，就发送获取到的 BluetoothSocket 给一个单独的线程，然后关闭 BluetoothServerSocket 并跳出循环。注意：accept()返回 BluetoothSocket 后，socket 已经连接了，所以在客户端不应该呼叫 connnect()。

manageConnectedSocket()是一个虚方法，用来初始化线程好传输数据。

通常应该在处理完侦听到的连接后立即关闭 BluetoothServerSocket。在本例中，close()在得到 BluetoothSocket 后马上被调用。还需要在线程中提供一个公共的方法来关闭私有的 BluetoothSocket，停止服务端 socket 的侦听。

（2）Connecting as a client

为了实现与远程设备的连接，你必须首先获得一个代表远程设备 BluetoothDevice 对象。然后使用 BluetoothDevice 对象来获取一个 BluetoothSocket 来实现来接。

下面是基本的步骤：

① 用 BluetoothDevice 调用 createRfcommSocketToServiceRecord(UUID) 获取一个 BluetoothSocket 对象。

这个初始化的 BluetoothSocket 会连接到 BluetoothDevice。UUID 必须匹配服务器设备在打开 BluetoothServerSocket 时用到的 UUID(用 listenUsingRfcommWithServiceRecord(String, UUID))。可以简单的生成一个 UUID 串然后在服务器和客户端都使用该 UUID。

② 调用 connect()完成连接

当调用这个方法的时候，系统会在远程设备上完成一个 SDP 查找来匹配 UUID。如果查找成功并且远程设备接受连接，就共享 RFCOMM 信道，connect()会返回。这也是一个阻塞的调用，不管连接失败还是超时（12 秒）都会抛出异常。

注意：要确保在调用 connect()时没有同时做设备查找，如果在查找设备，该连接尝试会显著的变慢，慢得类似失败了。

实例：

下面是一个完成 Bluetooth 连接的样例线程：

```
private class ConnectThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final BluetoothDevice mmDevice;

    public ConnectThread(BluetoothDevice device) {
        // Use a temporary object that is later assigned to mmSocket,
        // because mmSocket is final
        BluetoothSocket tmp = null;
        mmDevice = device;

        // Get a BluetoothSocket to connect with the given BluetoothDevice
        try {
            // MY_UUID is the app's UUID string, also used by the server code
            tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
        } catch (IOException e) { }
        mmSocket = tmp;
    }

    public void run() {
        // Cancel discovery because it will slow down the connection
        mBluetoothAdapter.cancelDiscovery();

        try {
            // Connect the device through the socket. This will block
            // until it succeeds or throws an exception
            mmSocket.connect();
        } catch (IOException connectException) {
            // Unable to connect; close the socket and get out
            try {
                mmSocket.close();
            } catch (IOException closeException) { }
```

```
                return;
            }

            // Do work to manage the connection (in a separate thread)
            manageConnectedSocket(mmSocket);
        }

        /** Will cancel an in-progress connection, and close the socket */
        public void cancel() {
            try {
                mmSocket.close();
            } catch (IOException e) { }
        }
}
```

注意到 cancelDiscovery()在连接操作前被调用。在连接之前，不管搜索有没有进行，该调用都是安全的，不需要确认（当然如果有要确认的需求，可以调用 isDiscovering()）。
manageConnectedSocket()是一个虚方法，用来初始化线程好传输数据。
在对 BluetoothSocket 的处理完成后，记得调用 close()来关闭连接的 socket 和清理所有的内部资源。

9.7 Managing a Connection 服务

如果已经连接了两个设备，他们都已经拥有各自的连接好的 BluetoothSocket 对象。那就是一个有趣的开始，因为你可以在设备间共享数据了。使用 BluetoothSocket，传输任何数据通常来说都很容易了：
（1）通过 socket 获取输入输出流来处理传输（分别使用 getInputStream()和 getOutputStream()）。
（2）用 read(byte[])和 write(byte[])来实现读写。
仅此而已。

当然，还是有很多细节需要考虑的。首要的，需要用一个专门的线程来实现流的读写。只是很重要的，因为 read(byte[])和 write(byte[])都是阻塞的调用。read(byte[])会阻塞直到流中有数据可读。write(byte[])通常不会阻塞，但是如果远程设备调用 read(byte[])不够快导致中间缓冲区满，它也可能阻塞。所以线程中的主循环应该用于读取 InputStream。线程中也应该有单独的方法用来完成写 OutputStream。
示例：
下面是一个如上面描述那样的例子：

```
private class ConnectedThread extends Thread {
    private final BluetoothSocket mmSocket;
    private final InputStream mmInStream;
    private final OutputStream mmOutStream;

    public ConnectedThread(BluetoothSocket socket) {
        mmSocket = socket;
        InputStream tmpIn = null;
```

```
        OutputStream tmpOut = null;

        // Get the input and output streams, using temp objects because
        // member streams are final
        try {
            tmpIn = socket.getInputStream();
            tmpOut = socket.getOutputStream();
        } catch (IOException e) { }

        mmInStream = tmpIn;
        mmOutStream = tmpOut;
    }

    public void run() {
        byte[] buffer = new byte[1024];  // buffer store for the stream
        int bytes; // bytes returned from read()

        // Keep listening to the InputStream until an exception occurs
        while (true) {
            try {
                // Read from the InputStream
                bytes = mmInStream.read(buffer);
                // Send the obtained bytes to the UI Activity
                mHandler.obtainMessage(MESSAGE_READ, bytes, -1, buffer)
                        .sendToTarget();
            } catch (IOException e) {
                break;
            }
        }
    }

    /* Call this from the main Activity to send data to the remote device */
    public void write(byte[] bytes) {
        try {
            mmOutStream.write(bytes);
        } catch (IOException e) { }
    }

    /* Call this from the main Activity to shutdown the connection */
    public void cancel() {
        try {
            mmSocket.close();
        } catch (IOException e) { }
    }
```

}

构造函数中得到需要的流，一旦执行，线程会等待从 InputStream 来的数据。当 read(byte[]) 返回从流中读到的字节后，数据通过父类的成员 Handler 被送到主 Activity，然后继续等待读取流中的数据。

向外发送数据只需简单的调用线程的 write()方法。

线程的 cancel()方法时很重要的，以便连接可以在任何时候通过关闭 BluetoothSocket 来终止。它应该总在处理完 Bluetooth 连接后被调用。

9.8 Working with Profiles 服务

从 Android 3.0 开始，Bluetooth API 就包含了对 Bluetooth profiles 的支持。Bluetooth profile 是基于蓝牙的设备之间通信的无线接口规范。例如 Hands-Free profile（免提模式）。如果移动电话要连接一个无线耳机，他们都要支持 Hands-Free profile。

你在你的类里可以完成 BluetoothProfile 接口来支持某一 Bluetooth profiles。Android Bluetooth API 完成了下面的 Bluetooth profile：

Headset：Headset profile 提供了移动电话上的 Bluetooth 耳机支持。Android 提供了 BluetoothHeadset 类，它是一个协议，用来通过 IPC（interprocess communication）控制 Bluetooth Headset Service。BluetoothHeadset 既包含 Bluetooth Headset profile 也包含 Hands-Free profile，还包括对 AT 命令的支持。

A2DP：Advanced Audio Distribution Profile (A2DP) profile，高级音频传输模式。Android 提供了 BluetoothA2dp 类，这是一个通过 IPC 来控制 Bluetooth A2DP 的协议。

下面是使用 profile 的基本步骤：

① 获取默认的 Bluetooth 适配器。

② 使用 getProfileProxy()来建立一个与 profile 相关的 profile 协议对象的连接。在下面的例子中，profile 协议对象是 BluetoothHeadset 的一个实例。

③ 设置 BluetoothProfile.ServiceListener。该 listener 通知 BluetoothProfile IPC 客户端，当客户端连接或断连服务器的时候。

④ 在 onServiceConnected()内，得到一个 profile 协议对象的句柄。

⑤ 一旦拥有了 profile 协议对象，就可以用它来监控连接的状态，完成于该 profile 相关的其他操作。

例如，下面的代码片段显示如何连接到一个 BluetoothHeadset 协议对象，用来控制 Headset profile：

```
BluetoothHeadset mBluetoothHeadset;

// Get the default adapter
BluetoothAdapter mBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

// Establish connection to the proxy.
mBluetoothAdapter.getProfileProxy(context, mProfileListener, BluetoothProfile.HEADSET);

private BluetoothProfile.ServiceListener mProfileListener = new BluetoothProfile.ServiceListener() {
    public void onServiceConnected(int profile, BluetoothProfile proxy) {
        if (profile == BluetoothProfile.HEADSET) {
```

```
                mBluetoothHeadset = (BluetoothHeadset) proxy;
            }
        }
        public void onServiceDisconnected(int profile) {
            if (profile == BluetoothProfile.HEADSET) {
                mBluetoothHeadset = null;
            }
        }
};

// ... call functions on mBluetoothHeadset

// Close proxy connection after use.
mBluetoothAdapter.closeProfileProxy(mBluetoothHeadset);
```

十 总结与疑问

1. 蓝牙驱动的上下电是怎么完成的？是通过操作寄存器，还是其他的方式？
2. bccmd 命令主要用来初始化蓝牙，比如说上电，设置波特率，下载 firmware，它是如何实现的？
3. 如何设置 PSKEY 键值？
4. 蓝牙协议栈 bluez 是如何通过 dbus 总线与 framework 层进行数据交互的？