
32°C

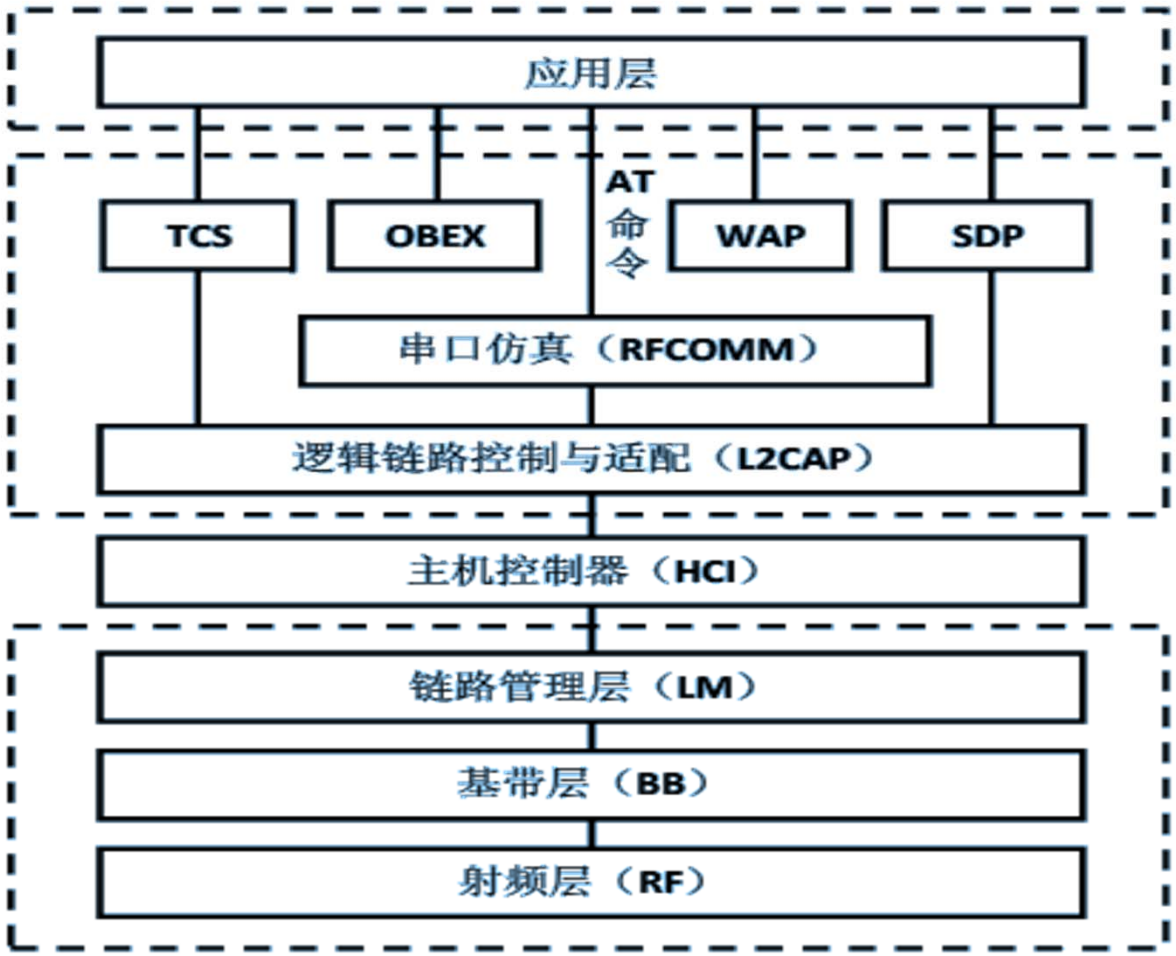
GARMIN
12:30
27°
14:00


03:11

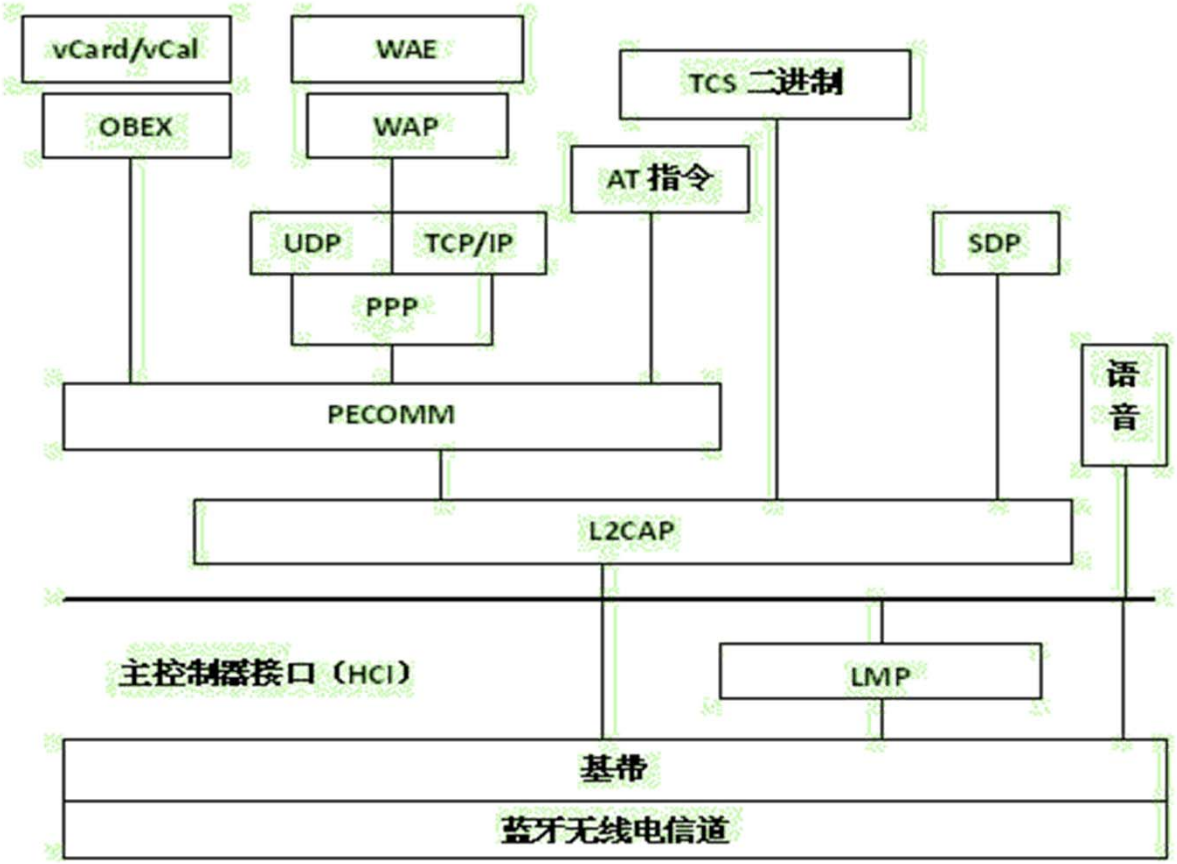
What is Bluetooth ?

所谓蓝牙(Bluetooth)技术，实际上是一种短距离无线电技术，利用"蓝牙"技术，能够有效地简化掌上电脑、笔记本电脑和移动电话手机等移动通信终端设备之间的通信，也能够成功地简化以上这些设备与因特网Internet之间的通信，从而使这些现代通信设备与因特网之间的数据传输变得更加迅速高效，为无线通信拓宽道路。蓝牙采用分散式网络结构以及快跳频和短包技术，支持点对点及点对多点通信，工作在全球通用的2.4GHz ISM（即工业、科学、医学）频段。其数据速率为1Mbps。采用时分双工传输方案实现全双工传输。

Structure


应用层
TCS
OBEX
AT
命
令
WAP
SDP
串口仿真（RFCOMM）
逻辑链路控制与适配（L2CAP）
主机控制器（HCI）
链路管理层（LM）
基带层（BB）
射频层（RF）

Bluetooth protocol stack



完整的蓝牙协议栈

蓝牙协议主要分类

- 核心协议：BaseBand、 LMP、 L2CAP、 SDP；
- 电缆替代协议：RFCOMM；
- 电话传送控制协议：TCS-Binary、 AT命令集；
- 选用协议：PPP、 UDP/TCP/IP、 OBEX、 WAP、 vCard、 vCal、 IrMC、 WAE

核心协议

基带协议（Baseband）

基带和链路控制层确保微微网内各蓝牙设备单元之间由射频构成的物理连接。基带部分将来自高层协议的数据进行信道编码，向下传给射频进行发送；接收数据时，射频部分经过解调恢复空中数据并上传给基带，基带再对数据进行信道解码，向高层传输

连接管理协议（LMP）

该协议负责各蓝牙设备间连接的建立。它通过连接的发起、交换、核实，进行身份认证和加密，通过协商确定基带数据分组大小。它还控制无线设备的电源模式和工作周期，以及微微网内设备单元的连接状态

逻辑链路控制和适配协议（L2CAP）

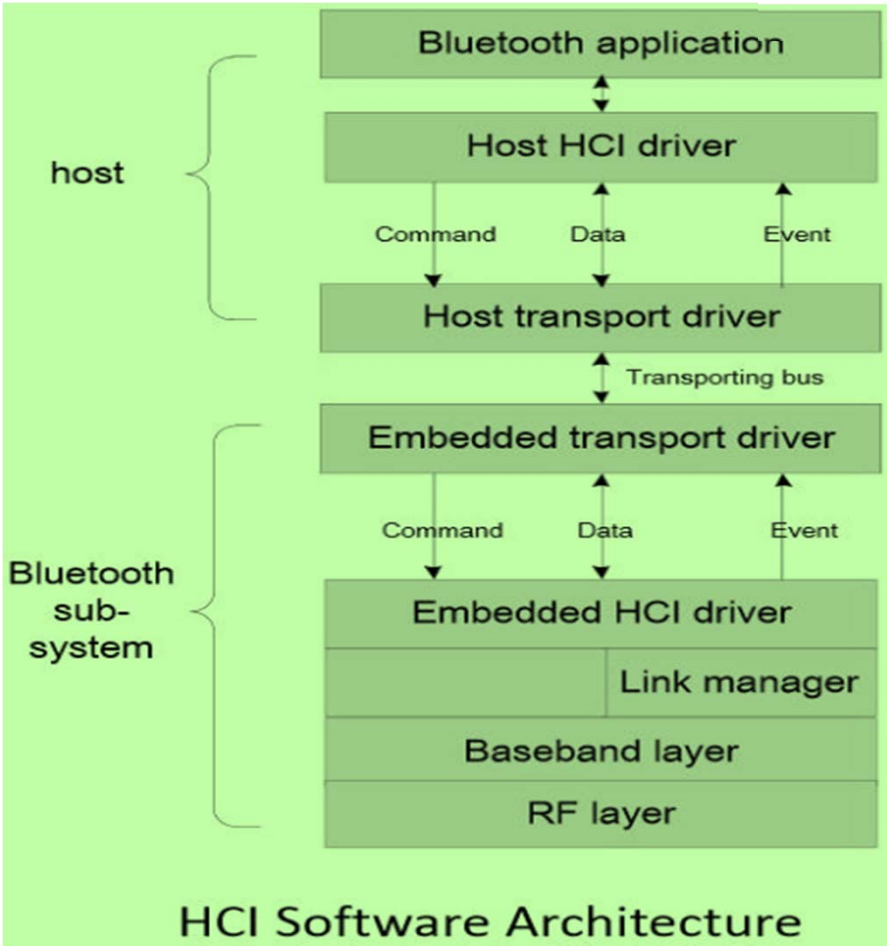
该协议是基带的上层协议，可以认为它与LMP并行工作，它们的区别在于，当业务数据不经过LMP时，L2CAP为上层提供服务。L2CAP向上层提供面向连接的和无连接的数据服务，它采用了多路技术、分割和重组技术、群提取技术

服务发现协议（SDP）

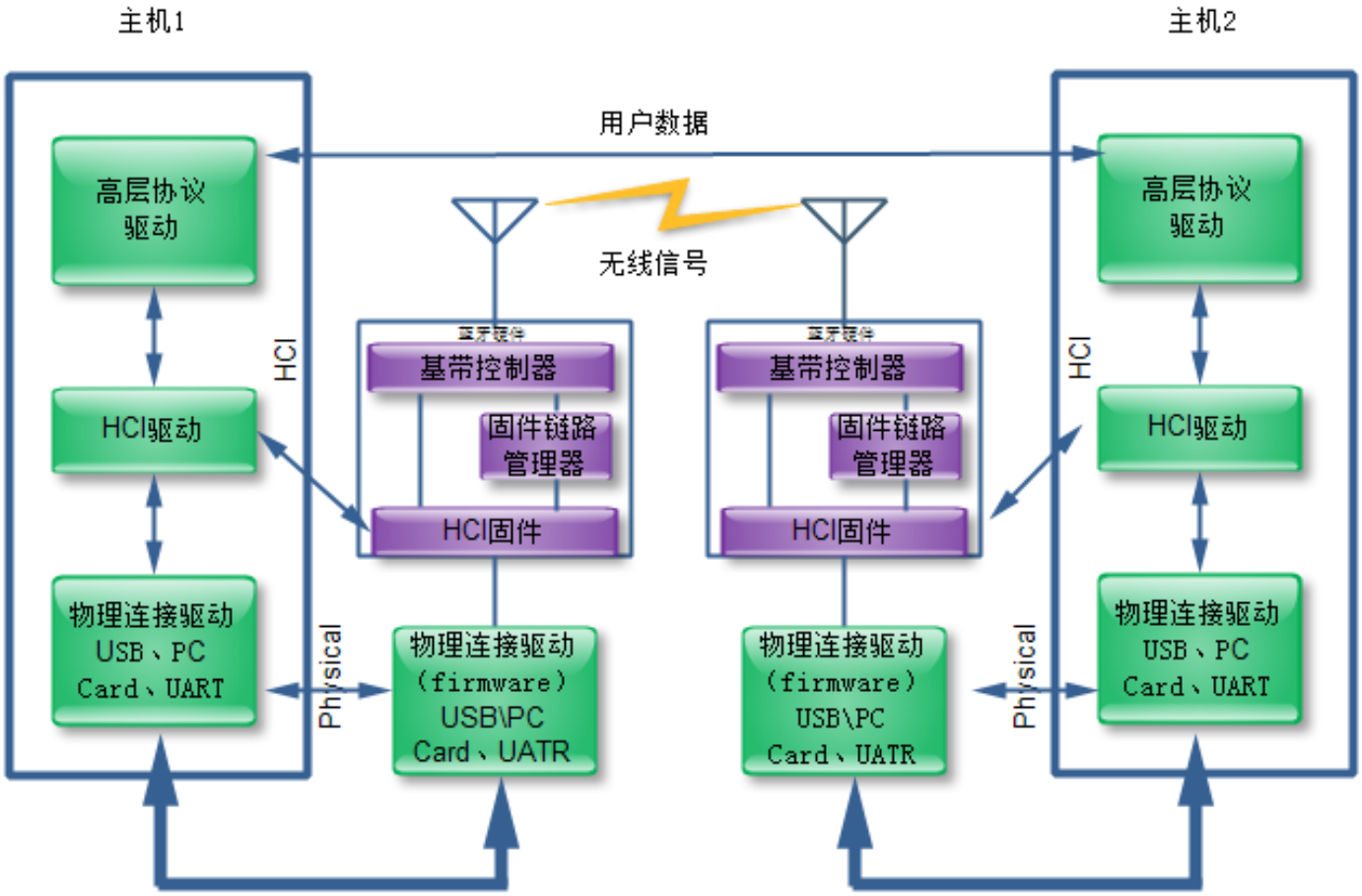
发现服务在蓝牙技术框架中起着至关紧要的作用，它是所有用户模式的基础。使用SDP可以查询到设备信息和服务类型，从而在蓝牙设备间建立相应的连接。

HIC：主机接口协议(Host Controller Interface)

为蓝牙协议层的上层提供了进入基带的统一接口。经过测试，所开发的接口能将上层的数据流匹配到基带，使基带能对之进行处理，并产生相应的事件。



蓝牙软件协议栈堆的数据传输过程



蓝牙规范协议版本1.0~3.0

蓝牙协议版本号	发布日期	支持的速度	
Bluetooth v1.0和v1.0B			• 强制发送BD_ADDR
Bluetooth v1.1			• 对应的IEEE标准是IEEE 802.15.1－2002 • RSSI(Received Signal Strength Indicator)
Bluetooth v1.2		实际：721Kb/s	• 对应的IEEE标准是IEEE 802.15.1－2005 • AFH(Adaptive Frequency-Hopping spread spectrum) • eSCO(Extended Synchronous Connections) • HCI(Host Controller Interface)
Bluetooth v2.0+EDR	2004年	• 理论：3Mb/s • 实际：2.1Mb/s	• EDR(π/4-DQPSK +8DPSK)
Bluetooth v2.1+EDR	2007-07-26	• 理论：24Mb/s -> High Speed	• SSP(Secure Simple Pairing) • EIR(Extended Inquiry Response)

Bluetooth v3.0+HS	2009-04-21		• AMP (Alternative MAC/PHY)
			  ◦ ->Bluetooth over 802.11 high-speed data transfer
			  ◦ ->Bluetooth High Speed based on Wi-Fi
			• L2CAP Enhanced modes
			• Unicast Connectionless Data
			• Enhanced Power Control

蓝牙 Android API

蓝牙API分类

- 开启设备
- 查找设备
- 连接设备

开启

- 1、获得BluetoothAdapter

```
BluetoothAdapter mBluetoothAdapter =
        BluetoothAdapter.getDefaultAdapter();
```

- 2、开启蓝牙
- if (!mBluetoothAdapter.isEnabled()) {
- Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
- startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);

查找设备

- 1、查询配对设备

```
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());
    }
}
```

查找设备

- 2、发现设备

```
// 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
```

查找设备

- 3、开启搜索

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

连接设备

- 服务器端
- 获取BluetoothServerSocket

  listenUsingRfcommWithServiceRecord(String, UUID)
- 接受连接请求

```
try {
        socket = mmServerSocket.accept();
    } catch (IOException e) {
     break;
  }
```

连接设备

- 关闭服务器端服务

```
if (socket != null) {
        // Do work to manage the connection (in a separate thread)
        manageConnectedSocket(socket);
        mmServerSocket.close();
        break;
    }
```

- 初始化传输线程

  manageConnectedSocket()

连接设备

- 客户端
- 远程服务器端查询UUID

```
try {
        // MY_UUID is the app's UUID string, also used by the server
code
        tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
    } catch (IOException e) { }
```

连接设备

- 初始化连接

```
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;
    }
```

连接设备

- 取消搜索

```
mBluetoothAdapter.cancelDiscovery();
```

连接管理

- 输入输出

getInputStream()

getOutputStream()

- 读写数据

read(byte[])

write(byte[])

Bluetooth v4.0

V4.0(2010):

- 蓝牙4.0协议版本是在蓝牙3.0高速版本基础上增加了低能耗协议部分
- 具有低成本、跨厂商互操作性、3毫秒低延迟、AES-128加密等特性
- 可以广泛应用于计步器、心律监视器、智能仪表、传感器物联网等众多领域
- 预计在2016年将有十亿的设备出货量
- 就单模而言和经典蓝牙设备不兼容


Bluetooth Classic
Bluetooth Smart Ready
Bluetooth Smart
蓝牙2.1BR/EDR
蓝牙4.0双模
BLE单模
SPP
SPP
Attribute Profile
Attribute Profile
RFCOM
RFCOM
Attribute Protocal
Attribute Protocal
L2CAP
L2CAP
L2CAP
Link Manager
Link Manager
Link Layer
Link Layer
BR/EDR PHY
BR/EDR + LE PHY
LE PHY

对比项	蓝牙BD/EDR	蓝牙BLE
射频物理上的通道	79个频段，1MHz的通道步长	40个频段，1MHz的通道步长
发现/连接	查询/传呼	广播
微微网从设备数目	活动的7个/共255个	无限制
设备地址隐私性	无	仅私有地址寻址有效
最大数据率	1-3Mbps	1Mbps（GFSK调制）
加密算法	E0/SAFER+	AES-CCM
典型覆盖范围	30米	50米
最大输出功率	100mW(20dBm)	10mW(10dBm)

Bluetooth Low Energy ==BLE

BLE有时候也被叫做Bluetooth Smart

BLE是标准的蓝牙（协议规范）的一个轻量级的子集

BLE是在蓝牙4.0的核心规范中引入的。

4.0以前的蓝牙，控制器和主机是分开的，而BLE中控制器和主机是在一起的。

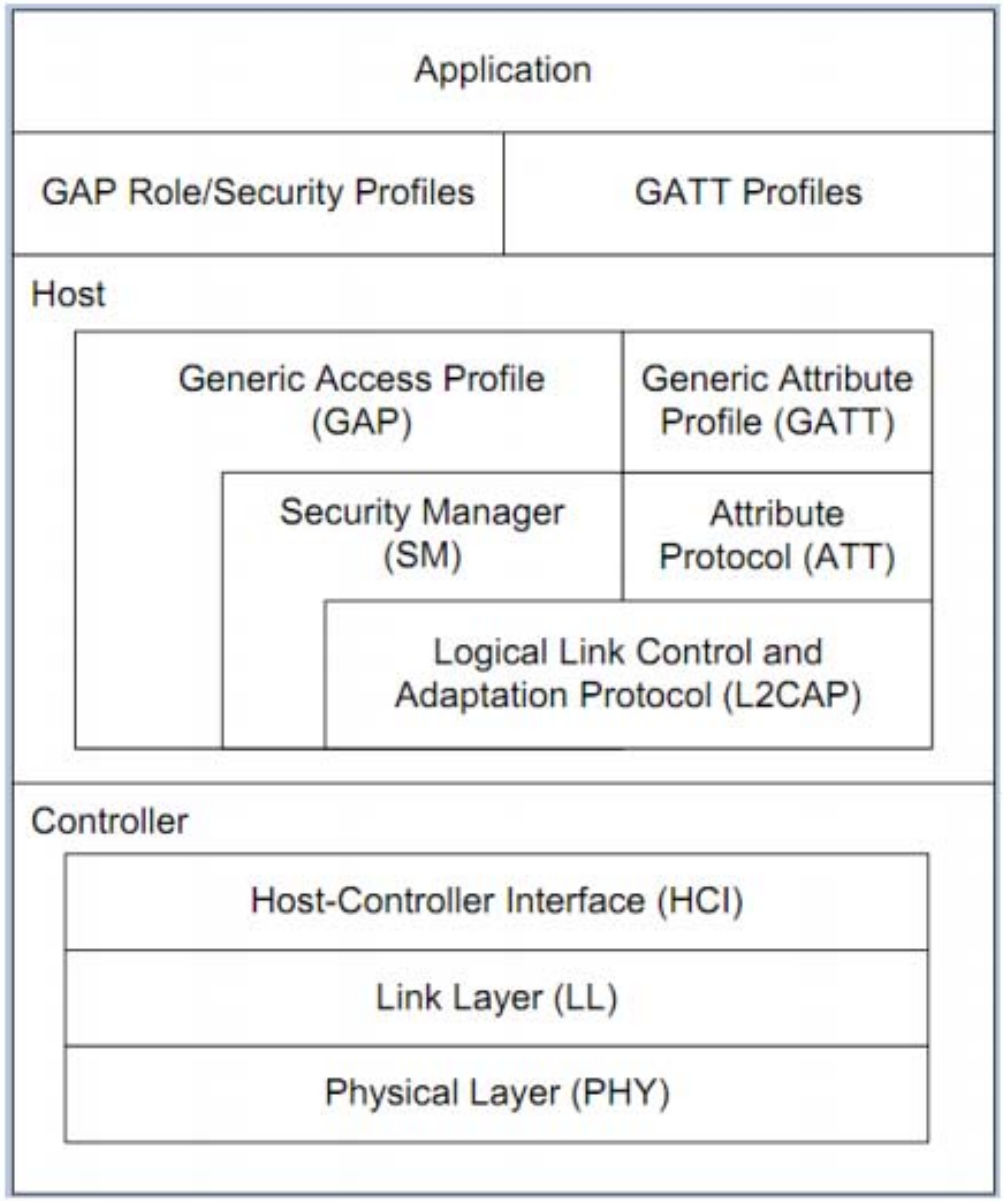
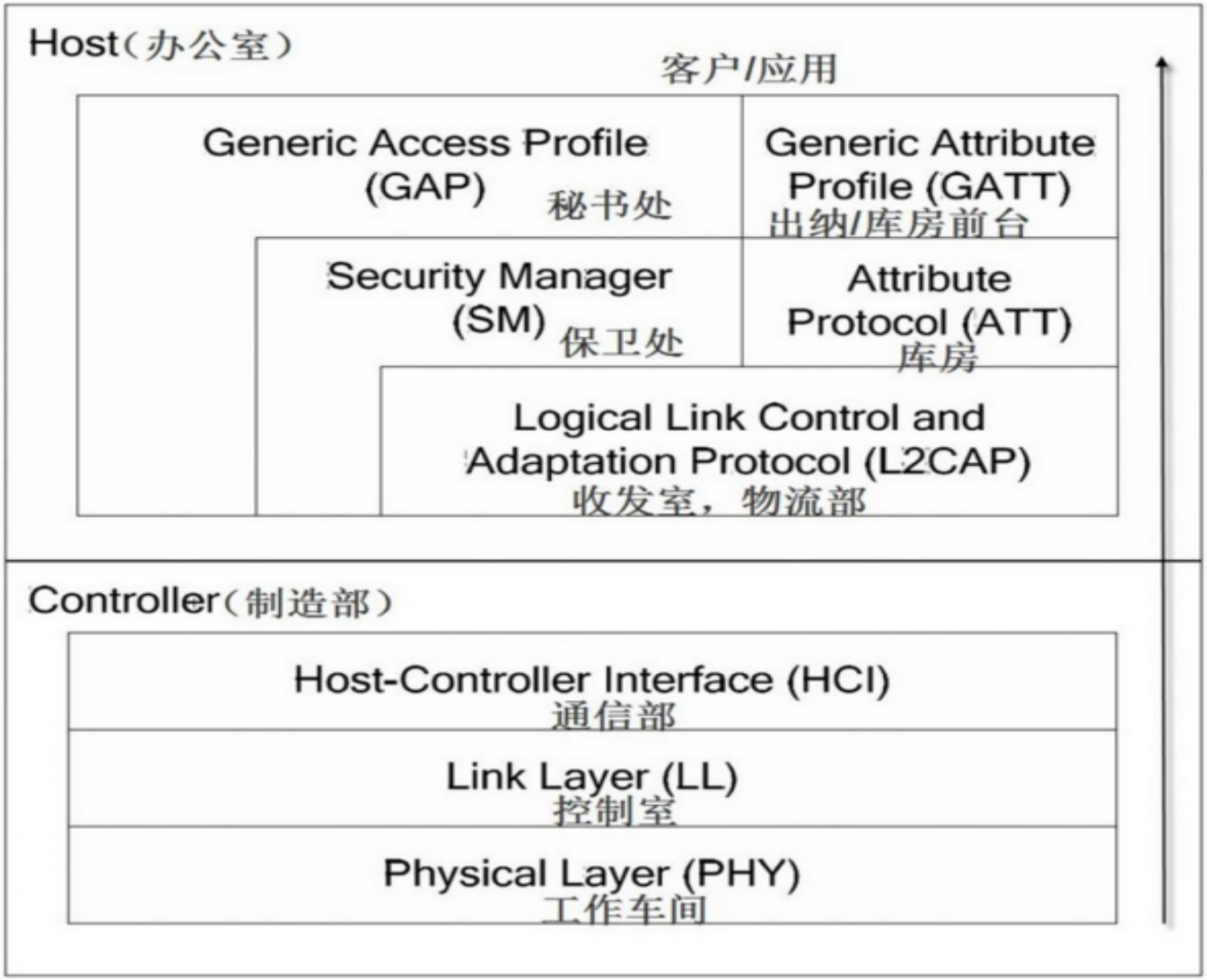
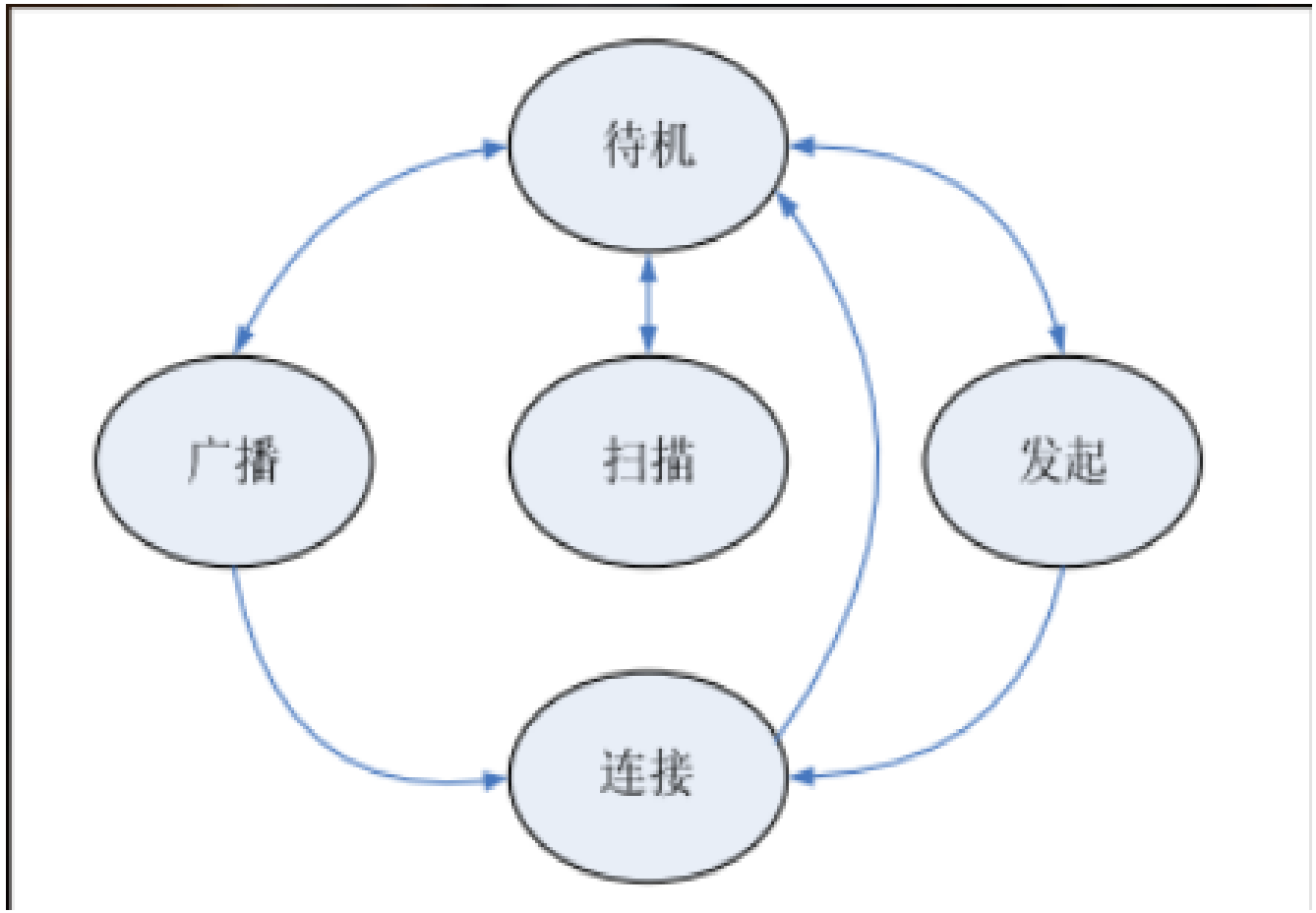

Application
GAP Role/Security Profiles
GATT Profiles
Host
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Security Manager (SM)
Attribute Protocol (ATT)
Logical Link Control and Adaptation Protocol (L2CAP)
Controller
Host-Controller Interface (HCI)
Link Layer (LL)
Physical Layer (PHY)

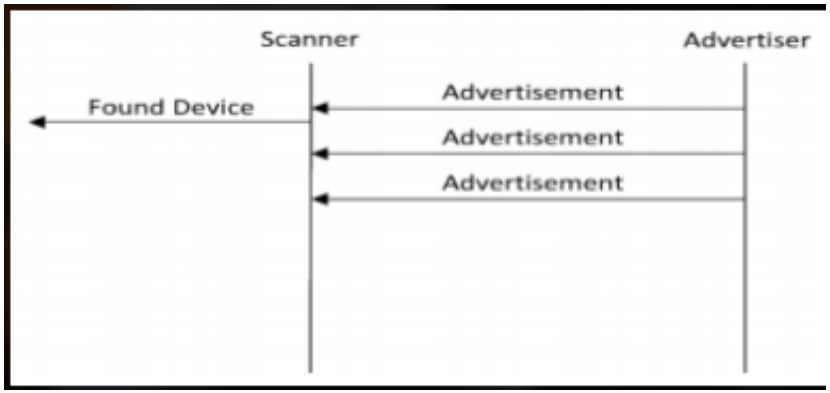
图 6.3. 以工厂为例来解释BLE的蓝牙协议栈架构

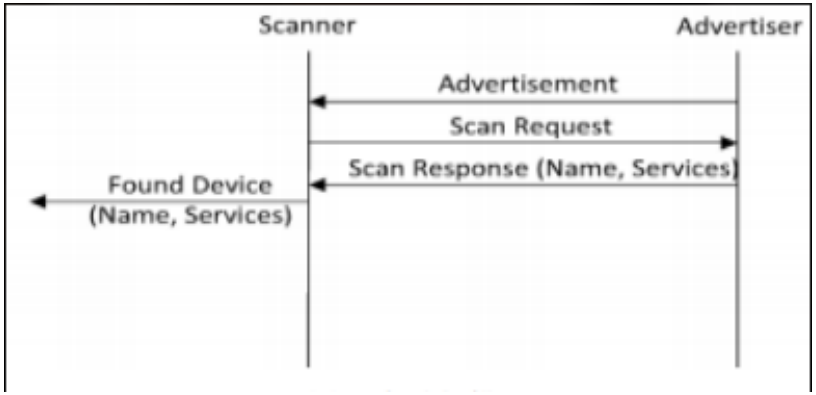



待机
广播
扫描
发起
连接

链路层事件操作

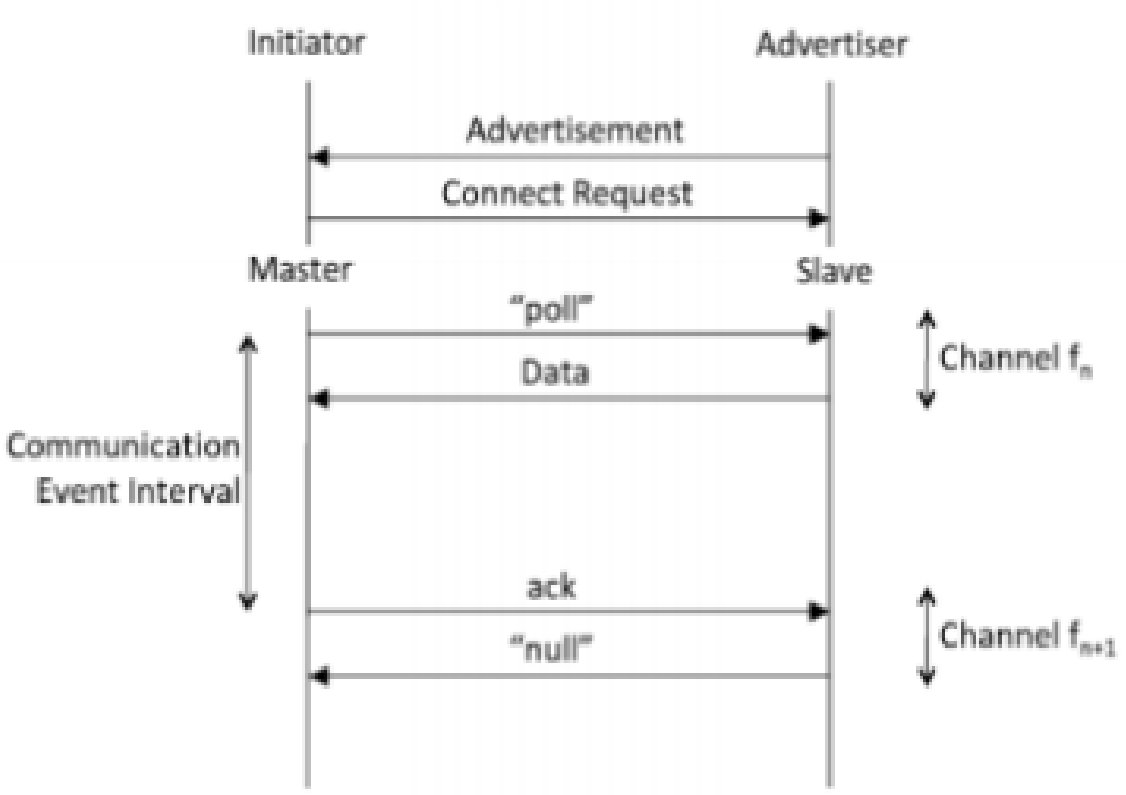
- 链路层主要有两种重要的事件操作：扫描与建立链接。
- 设备扫描有被动扫描和主动扫描：被动扫描是通过被动接收广播包得到设备信息；主动扫描是通过发送扫描请求得到扫描回应得到设备信息。





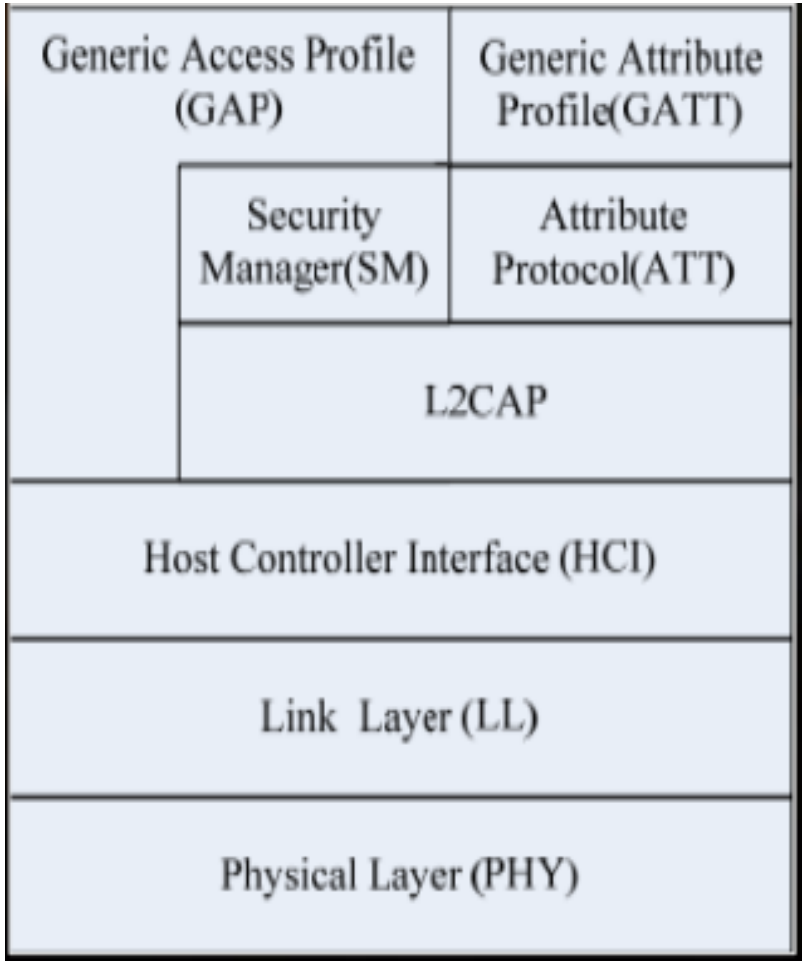
链路层事件操作

建立链接过程是通过发送链接请求包来建立设备链接

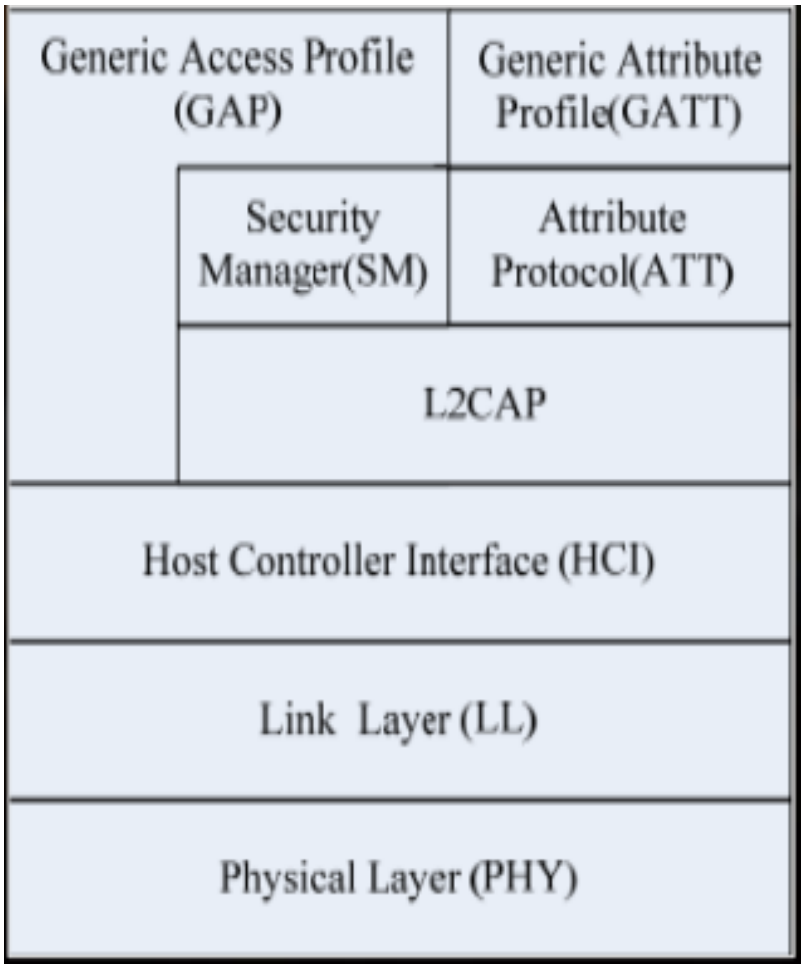


主机控制接口层 (HCILayer)

主机控制接口与标准蓝牙技术相同，
提供了主机与控制器层的通信方式与命令事件格式，
重用标准蓝牙传输层接口如UART,USB等。



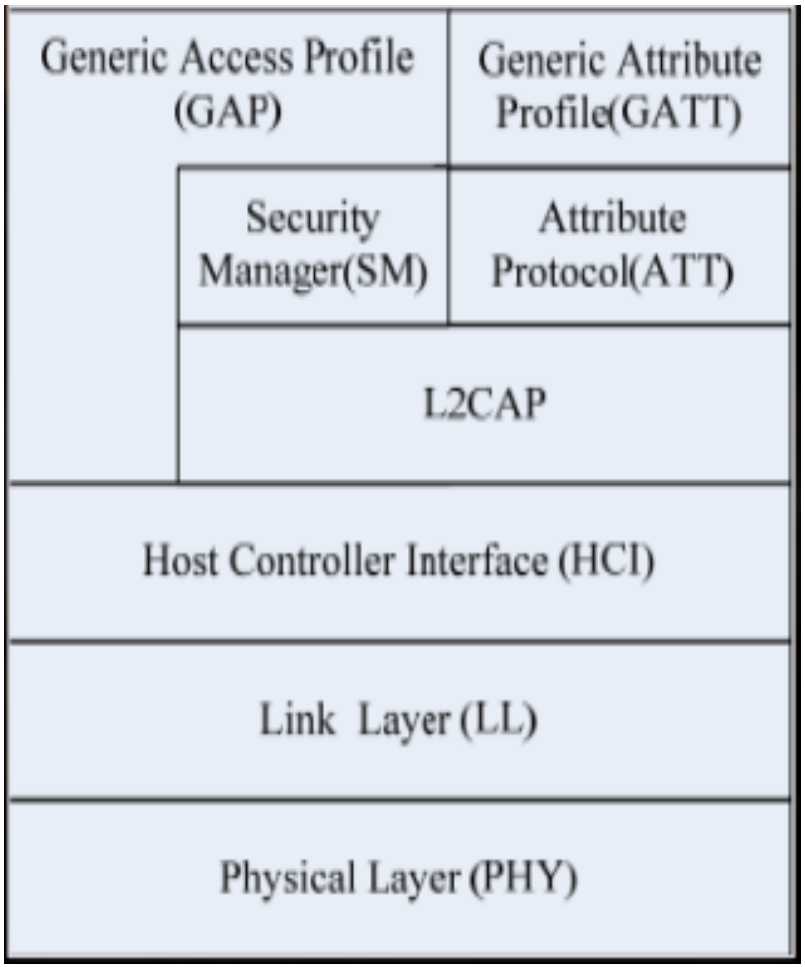
逻辑链路与适配协议层(L2CAPLayer)与标准蓝牙技术相同，为上层 提供了数据封装业务，提供端 到端的逻辑数据通信。



安全管理层(Security Manager Layer)

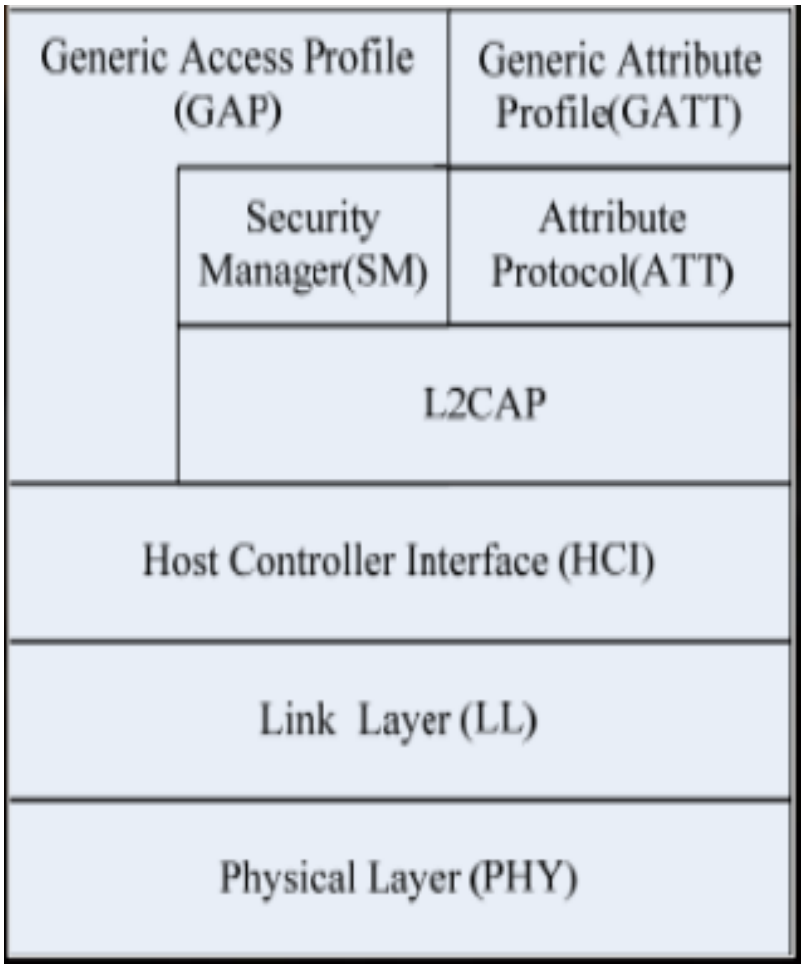
• 定义了配对和密钥分发方法，提供其它 层协议接口来安全的建立连接以及交换 数据。

• 安全管理层不涉及具体的BLE安全算法， 只是提供一些接口，为节省功耗以及降 低复杂性，具体安全算法可以通过在底 层硬件实现。



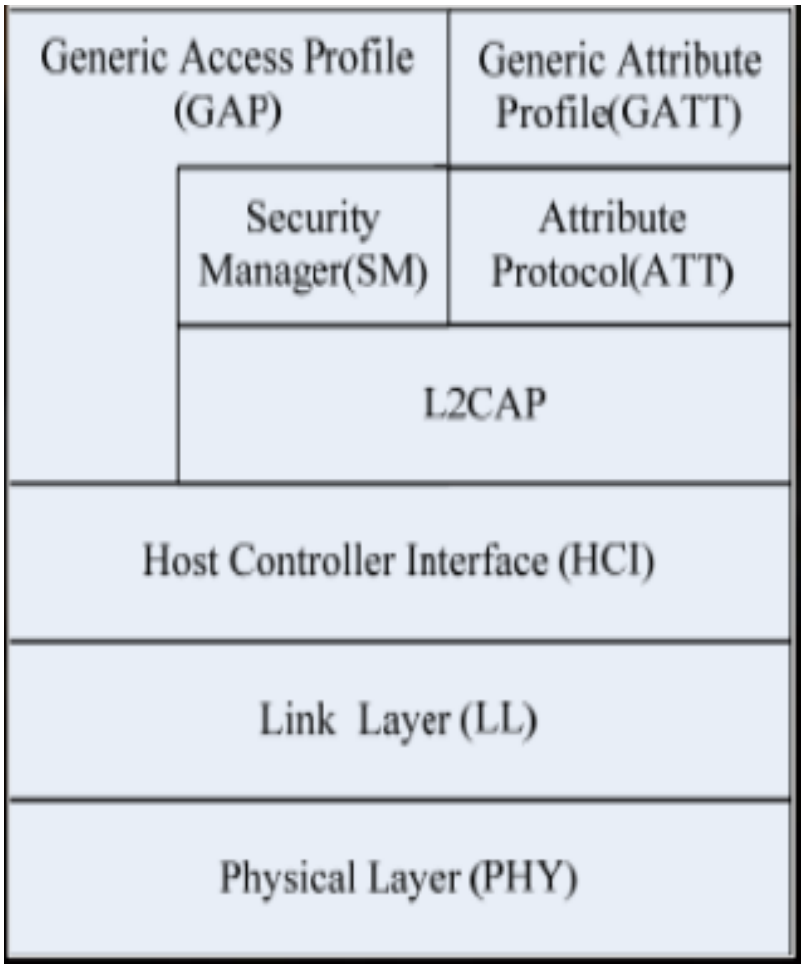
通用接入(GAP)

定义了通用的接口，供应用层 调用底层模块，比如设备发现，建立连接相关的业务，同时封 装了安全设置相关的API。



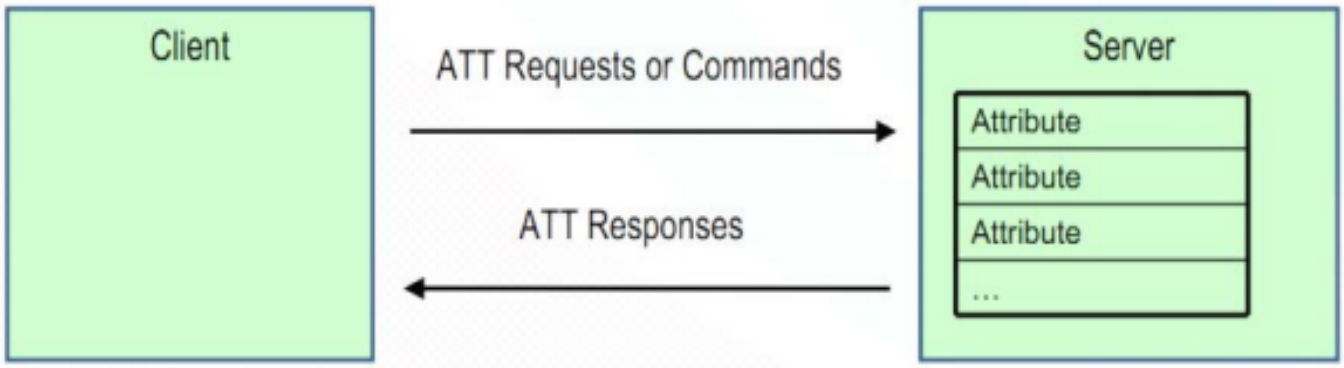
属性协议层(ATT Layer)

属性协议允许设备以“属性”的形式向另外的设备暴露他的 某些数据。在ATT协议里，暴 露属性的称为Server 端，另外 一端称为Client。

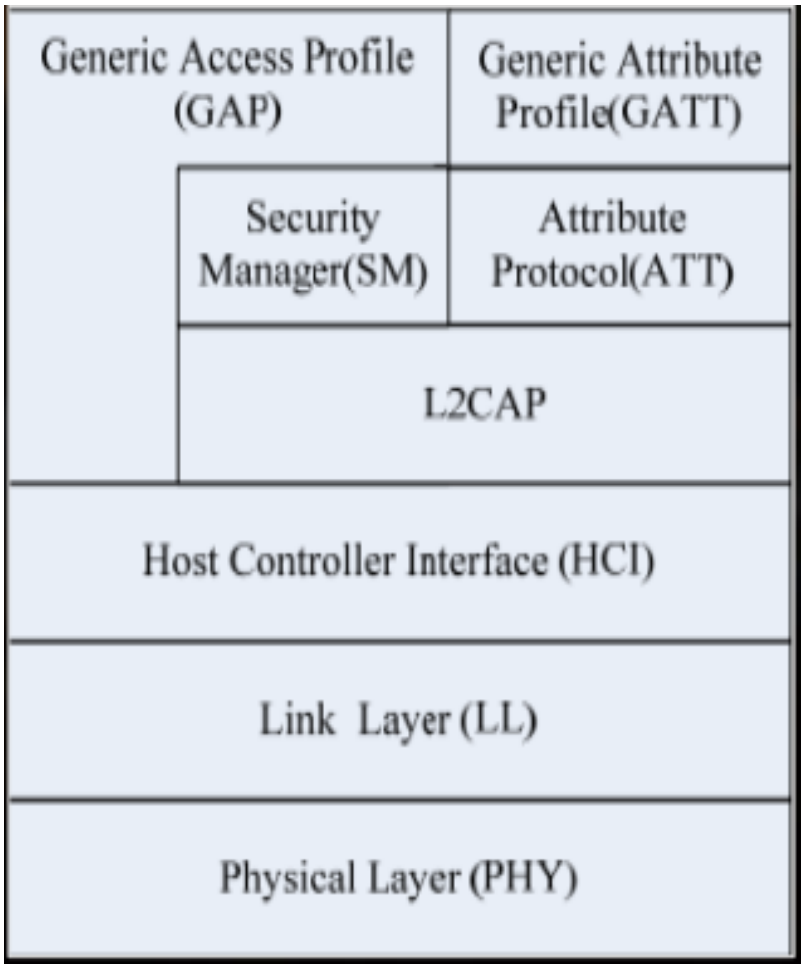


5.1.14 ATT 的 Client/Server 架构

服务设备提供数据，客户端使用这些数据
服务端通过操作属性的方式，提供数据访问服务
设备的服务/客户角色，不依赖于 GAP 层中心设备/外围设备角色，和 LL 层的 master/slave 角色定义
一个设备可能同时做为一个客户端和服务端，而两个设备上的属性不会相互影响。

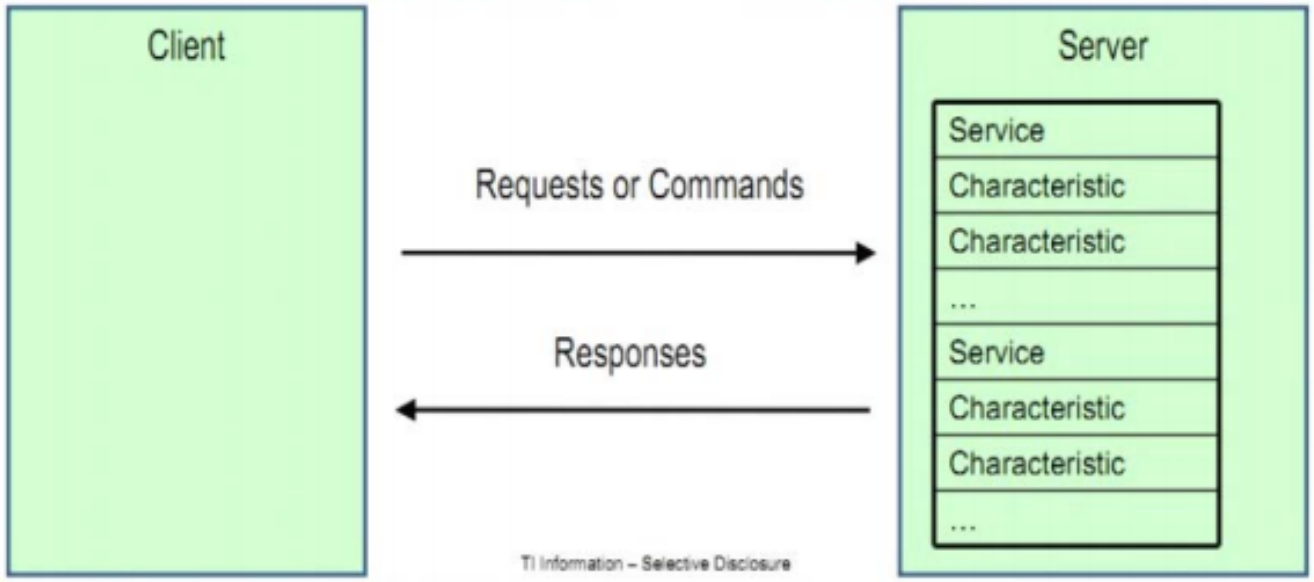


通用属性(GATT Profile)
GATT 层是一种具体使用属性 协议的应用框架。GATT 定义 了属性协议应用的架构。在 BLE协议中，应用中数据片段 被称为“特征”， 而BLE中两 个设备之间的数据通信就是通 过GATT子过程来处理的。

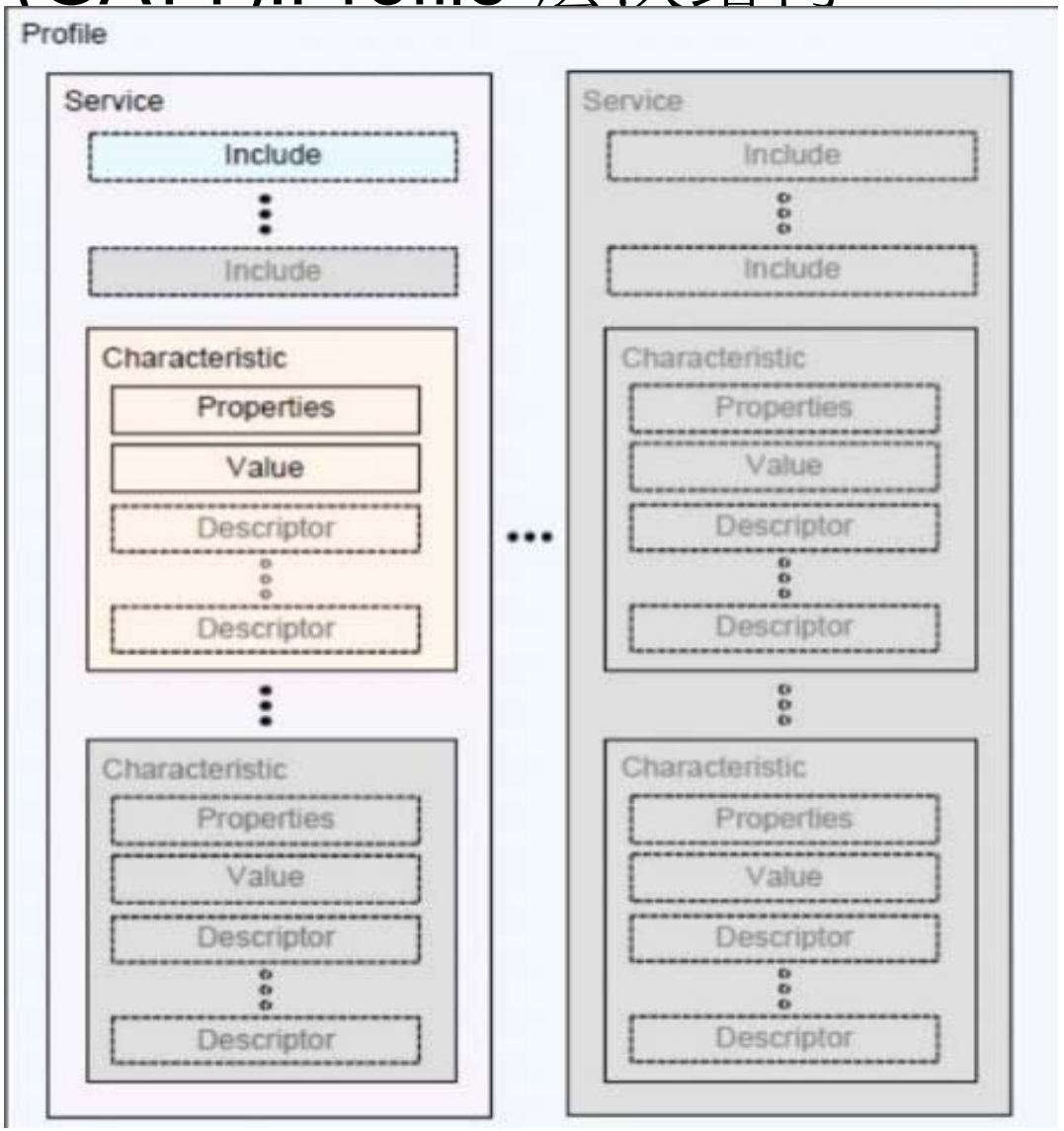


5.1.16 GATT 的 Client/Server 架构

GATT 指定了 profile 数据交换所在的结构
除了数据的封装方式不同，client server 和 Attribute 协议结构相同，数据封装在“Services”里，用“Characteristic”表示

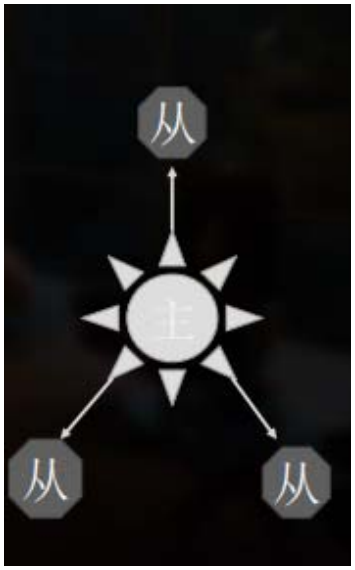


BLE (GATT):Profile 层次结构



BLE拓扑结构和设备状态

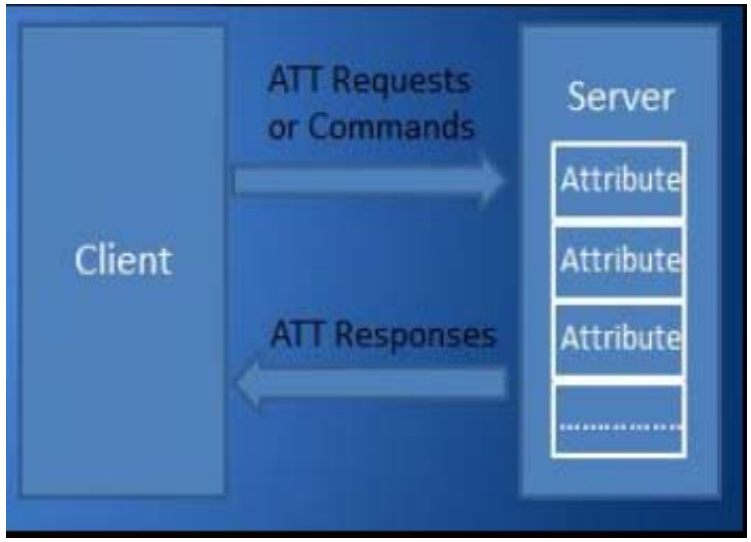
BLE 是一种星形拓扑结构:
主设备管理着连接，并且可以连接多个从设备 一个从设备只能连接一个主设备



做为一个BLE设备，有六种可能的状态:
待机状态(Standby)：设备没有传输和发送数据，并且没有连接 到任何设备
广播状态(Advertiser)：周期性广播状态
扫描状态(Scanner)：主动地寻找正在广播的设备
发起连接状态(Initiator): 主动向某个设备发起连接
主设备(Master)：作为主设备连接到其他设备
从设备(Slave) ：作为从设备连接到其他设备

BLE (ATT): Client / Server架构

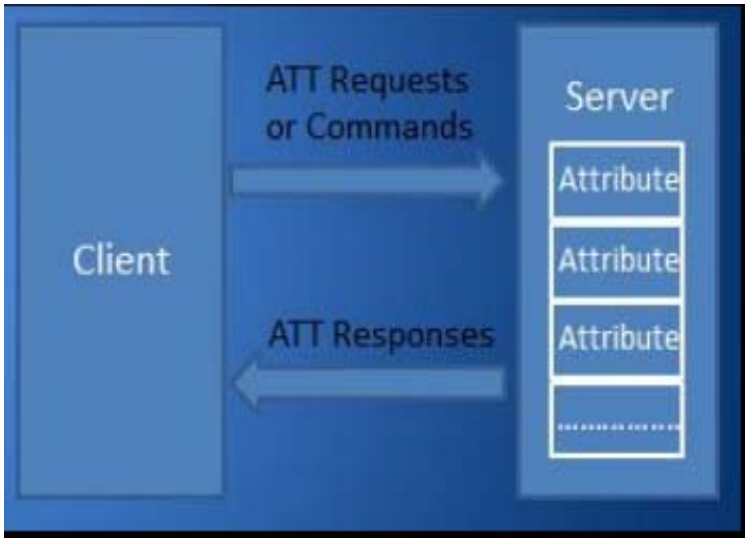
- 服务设备提供数据，客户端使用这些数据
- 服务端通过操作属性的方式，提供数据访 问服务
- 设备的服务/客户角色，不依赖于GAP层中 心设备 / 外围设备角色， 和 LL 层 master/slave角色定义
- 一个设备可能同时做为一个客户端和服务 端，而两个设备上的属性不会相互影响。



BLE (ATT): Client / Server 架构

GATT指定了profile数据交换所在的结构

除了数据的封装方式不同，client server和 Attribute 协议结构相同，数据封装在“Services”里，用“Characteristic”



蓝牙V4.0在Android上的应用

- JAVA层应用通过System Server和DBUS通信

- System Serve和BluetoothServe 的通信经Dbus Daemon中转

蓝牙V4.0典型应用

典型应用

- 无线因特网接入，因特网接入共享，Ad-hoc网络
- 无线打印机
- 高速文件传输
- 低功耗蓝牙连接
- 数字家庭应用
- 手机应用
- 运动健身器材、医疗保健

Android Ble API

BLE权限

- 为了在app中使用蓝牙功能，必须声明蓝牙权限BLUETOOTH。利用这个权限去执行蓝牙通信，例如请求连接、接受连接、和传输数据。
- 如果想让你的app启动设备发现或操纵蓝牙设置，必须声明BLUETOOTH_ADMIN权限。
- 注意：如果你使用BLUETOOTH_ADMIN权限，你也必须声明BLUETOOTH权限。
- 所以我觉得声明权限的时候应该是直接声明这两个就好了

在你的app manifest文件中声明蓝牙权限。

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

对应支持/不支持 BLE 的安卓设备

- 如果想声明你的app只为具有BLE的设备提供，在manifest文件中包括：

```
<uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
```

但显然为了让APP更具有普适性，我们应该对不支持BLE的设备页进行优化：

```
// 使用此检查确定BLE是否支持在设备上，然后你可以有选择性禁用BLE相关的功能
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
    Toast.makeText(this, R.string.ble_not_supported, Toast.LENGTH_SHORT).show();
    finish();
}
```

你的app能与BLE通信之前，你需要确认设备是否支持BLE，如果支持，确认已经启用。注意如果<uses-feature.../>设置为false，这个检查才是必需的。

如果不支持BLE，那么你应该适当地禁用部分BLE功能。如果支持BLE但被禁用，你可以无需离开应用程序而要求用户启动蓝牙。使用BluetoothAdapter两步完成该设置。

获取 BluetoothAdapter

- 所有的蓝牙活动都需要蓝牙适配器。BluetoothAdapter代表设备本身的蓝牙适配器(蓝牙无线）。整个系统只有一个蓝牙适配器，而且你的app使用它与系统交互。
- 下面的代码片段显示了如何得到适配器。注意该方法使用getSystemService（ ）]返回BluetoothManager，然后将其用于获取适配器的一个实例。
- Android 4.3（API 18）引入BluetoothManager。

```
// 初始化蓝牙适配器
final BluetoothManager bluetoothManager =
        (BluetoothManager) getSystemService(Context.BLUETOOTH_SERVICE);
mBluetoothAdapter = bluetoothManager.getAdapter();
```

开启蓝牙

- 接下来，你需要确认蓝牙是否开启。调用isEnabled())去检测蓝牙当前是否开启。如果该方法返回false,蓝牙被禁用。下面的代码检查蓝牙是否开启，如果没有开启，将显示错误提示用户去设置开启蓝牙。

```
// 确保蓝牙在设备上可以开启
if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

至此，BLE的准备工作已经完成，接下来就可以写入你想要BLE去完成的功能了。

发现BLE设备

- 为了发现BLE设备，使用startLeScan())方法。这个方法需要一个参数BluetoothAdapter.LeScanCallback。应当遵守以下准则：
- 只要找到所需的设备，停止扫描。
- 不要在循环里扫描，并且对扫描设置时间限制。以前可用的设备可能已经移出范围，继续扫描消耗电池电量。
- 接下来有一段示例的代码，来开始，或者停止一段扫描：

```
1
2   /**
3    * 扫描和显示可以提供的蓝牙设备.
4    */
5   public class DeviceScanActivity extends ListActivity {
6
7       private BluetoothAdapter mBluetoothAdapter;
8       private boolean mScanning;
9       private Handler mHandler;
10
11      // 10秒后停止寻找.
12      private static final long SCAN_PERIOD = 10000;
13      ...
14      private void scanLeDevice(final boolean enable) {
15          if (enable) {
16              // 经过预定扫描期后停止扫描
17              mHandler.postDelayed(new Runnable() {
18                  @Override
19                  public void run() {
20                      mScanning = false;
21                      mBluetoothAdapter.stopLeScan(mLeScanCallback);
22                  }
23              }, SCAN_PERIOD);
24
25              mScanning = true;
26              mBluetoothAdapter.startLeScan(mLeScanCallback);
27          } else {
28              mScanning = false;
29              mBluetoothAdapter.stopLeScan(mLeScanCallback);
30          }
31          ...
32      }
33  ...
34  }
```

连接到GATT服务端

- 与一个BLE设备交互的第一步就是连接它——更具体的，连接到BLE设备上的GATT服务端。为了连接到BLE设备上的GATT服务端，需要使用connectGatt( )方法。这个方法需要三个参数：一个Context对象，自动连接（boolean值,表示只要BLE设备可用是否自动连接到它），和BluetoothGattCallback调用。

```
1  mBluetoothGatt = device.connectGatt(this, false, mGattCallback);
```

连接到GATT服务端时，由BLE设备做主机，并返回一个BluetoothGatt实例，然后你可以使用这个实例来进行GATT客户端操作。请求方（Android app)是GATT客户端。BluetoothGattCallback用于传递结果给用户，例如连接状态，以及任何进一步GATT客户端操作。

```
1
2   //通过BLE API服务端与BLE设备交互
3   public class BluetoothLeService extends Service {
4       private final static String TAG = BluetoothLeService.class.getSimpleName();
5
6       private BluetoothManager mBluetoothManager; //蓝牙管理器
7       private BluetoothAdapter mBluetoothAdapter; //蓝牙适配器
8       private String mBluetoothDeviceAddress; //蓝牙设备地址
9       private BluetoothGatt mBluetoothGatt;
10      private int mConnectionState = STATE_DISCONNECTED;
11
12      private static final int STATE_DISCONNECTED = 0; //设备无法连接
13      private static final int STATE_CONNECTING = 1;  //设备正在连接状态
14      private static final int STATE_CONNECTED = 2;   //设备连接完毕
15
16      public final static String ACTION_GATT_CONNECTED =
17              "com.example.bluetooth.le.ACTION_GATT_CONNECTED";
18      public final static String ACTION_GATT_DISCONNECTED =
19              "com.example.bluetooth.le.ACTION_GATT_DISCONNECTED";
20      public final static String ACTION_GATT_SERVICES_DISCOVERED =
21              "com.example.bluetooth.le.ACTION_GATT_SERVICES_DISCOVERED";
```

.....

在这个例子中，这个BLE APP提供了一个activity（DeviceControlActivity)来连接，显示数据，显示该设备支持的GATT services和characteristics。根据用户的输入，这个activity与BluetoothLeService通信，通过Android BLE API实现与BLE设备交互。

当一个特定的回调被触发的时候，它会调用相应的broadcastUpdate()辅助方法并且传递给它一个action。

```
private void broadcastUpdate(final String action) {
    final Intent intent = new Intent(action);
    sendBroadcast(intent);
}

private void broadcastUpdate(final String action,
                             final BluetoothGattCharacteristic characteristic) {
    final Intent intent = new Intent(action);

    // 这是心率测量配置文件。
    if (UUID_HEART_RATE_MEASUREMENT.equals(characteristic.getUuid())) {
        int flag = characteristic.getProperties();
        int format = -1;
        if ((flag & 0x01) != 0) {
            format = BluetoothGattCharacteristic.FORMAT_UINT16;
            Log.d(TAG, "Heart rate format UINT16.");
        } else {
            format = BluetoothGattCharacteristic.FORMAT_UINT8;
            Log.d(TAG, "Heart rate format UINT8.");
        }
        final int heartRate = characteristic.getIntValue(format, 1);
        Log.d(TAG, String.format("Received heart rate: %d", heartRate));
        intent.putExtra(EXTRA_DATA, String.valueOf(heartRate));
    } else {
        // 对于所有其他的配置文件，用十六进制格式写数据
        final byte[] data = characteristic.getValue();
        if (data != null && data.length > 0) {
            final StringBuilder stringBuilder = new StringBuilder(data.length);
            for(byte byteChar : data)
                stringBuilder.append(String.format("%02X ", byteChar));
            intent.putExtra(EXTRA_DATA, new String(data) + "\n" +
                    stringBuilder.toString());
        }
    }
    sendBroadcast(intent);
}
```

返回DeviceControlActivity, 这些事件由一个BroadcastReceiver来处理:

```
// 通过服务控制不同的事件
// ACTION_GATT_CONNECTED: 连接到GATT服务端
// ACTION_GATT_DISCONNECTED: 未连接GATT服务端.
// ACTION_GATT_SERVICES_DISCOVERED: 未发现GATT服务.
// ACTION_DATA_AVAILABLE: 接受来自设备的数据，可以通过读或通知操作获得。
private final BroadcastReceiver mGattUpdateReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (BluetoothLeService.ACTION_GATT_CONNECTED.equals(action)) {
            mConnected = true;
            updateConnectionState(R.string.connected);
            invalidateOptionsMenu();
        } else if (BluetoothLeService.ACTION_GATT_DISCONNECTED.equals(action)) {
            mConnected = false;
            updateConnectionState(R.string.disconnected);
            invalidateOptionsMenu();
            clearUI();
        } else if (BluetoothLeService.
                ACTION_GATT_SERVICES_DISCOVERED.equals(action)) {
            // 在用户接口上展示所有的services and characteristics
            displayGattServices(mBluetoothLeService.getSupportedGattServices());
        } else if (BluetoothLeService.ACTION_DATA_AVAILABLE.equals(action)) {
            displayData(intent.getStringExtra(BluetoothLeService.EXTRA_DATA));
        }
    }
};
```

读取BLE变量

- 你的android app完成与GATT服务端连接和发现services后，就可以读写支持的属性。例如，这段代码通过服务端的services和 characteristics迭代，并且将它们显示在UI上。

```
public class DeviceControlActivity extends Activity {
    ...
    // 演示如何遍历支持GATT Services/Characteristics
    // 这个例子中，我们填充绑定到UI的ExpandableListView上的数据结构
    private void displayGattServices(List<BluetoothGattService> gattServices) {
        if (gattServices == null) return;
        String uuid = null;
        String unknownServiceString = getResources().
                getString(R.string.unknown_service);
        String unknownCharaString = getResources().
                getString(R.string.unknown_characteristic);
        ArrayList<HashMap<String, String>> gattServiceData =
                new ArrayList<HashMap<String, String>>();
        ArrayList<ArrayList<HashMap<String, String>>> gattCharacteristicData
                = new ArrayList<ArrayList<HashMap<String, String>>>();
        mGattCharacteristics =
                new ArrayList<ArrayList<BluetoothGattCharacteristic>>();

```

.....

接收GATT通知

- 当设备上的特性改变时会通知BLE应用程序。这段代码显示了如何使用setCharacteristicNotification( )给一个特性设置通知。

```
1
2   private BluetoothGatt mBluetoothGatt;
3   BluetoothGattCharacteristic characteristic;
4   boolean enabled;
5   ...
6   mBluetoothGatt.setCharacteristicNotification(characteristic, enabled);
7   ...
8   BluetoothGattDescriptor descriptor = characteristic.getDescriptor(
9           UUID.fromString(SampleGattAttributes.CLIENT_CHARACTERISTIC_CONFIG));
10  descriptor.setValue(BluetoothGattDescriptor.ENABLE_NOTIFICATION_VALUE);
11  mBluetoothGatt.writeDescriptor(descriptor);
```

如果对一个特性启用通知,当远程蓝牙设备特性发送变化，回调函数onCharacteristicChanged( ))被触发。

```
@Override
// 广播更新
public void onCharacteristicChanged(BluetoothGatt gatt,
        BluetoothGattCharacteristic characteristic) {
    broadcastUpdate(ACTION_DATA_AVAILABLE, characteristic);
}
```

关闭客户端App

- 当你的app完成BLE设备的使用后，应该调用close( )，系统可以合理释放占用资源。

```
public void close() {
    if (mBluetoothGatt == null) {
        return;
    }
    mBluetoothGatt.close();
    mBluetoothGatt = null;
}
```

那么到此，BLE的接入和使用的完整过程就算是结束了。