

一、概述

1.1 概念

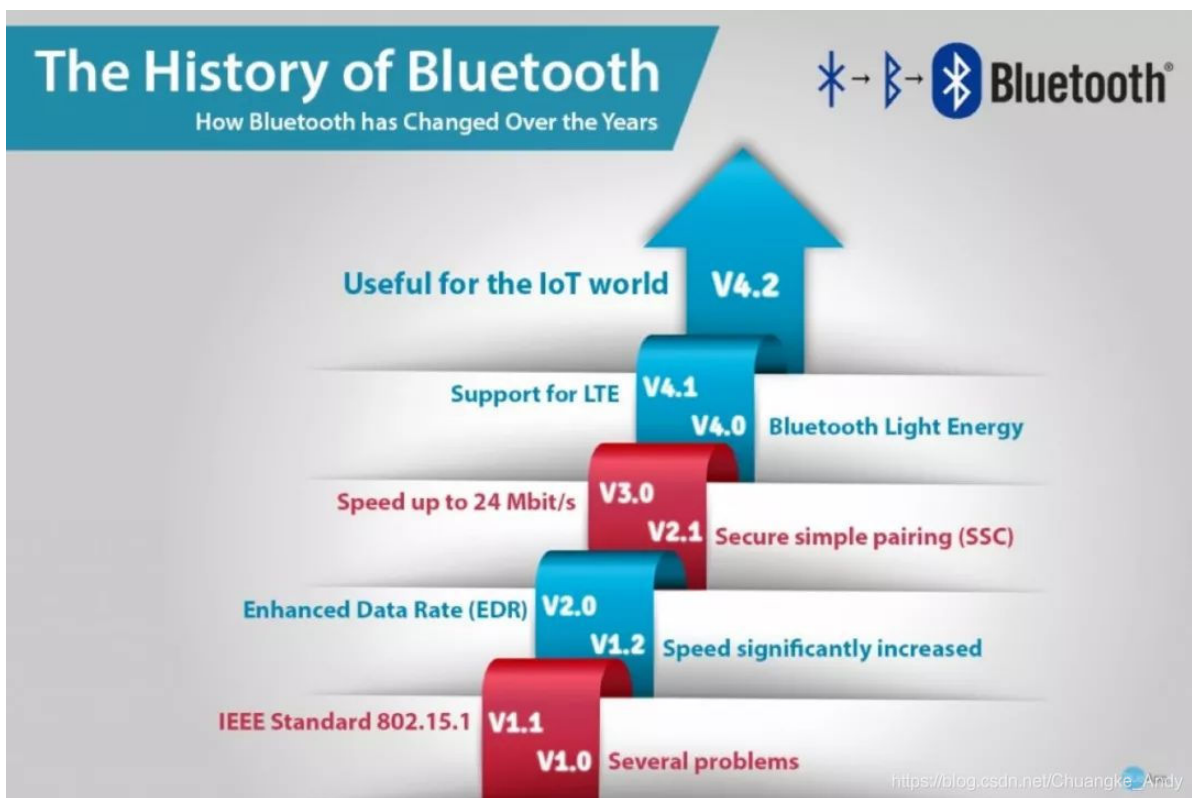
蓝牙，是一种支持设备短距离通信（一般10m内）的无线电技术，能在包括移动电话、PDA、无线耳机、笔记本电脑、相关外设等众多设备之间进行无线信息交换。利用“蓝牙”技术，能够有效地简化移动通信终端设备之间的通信，也能够成功地简化设备与因特网Internet之间的通信，从而数据传输变得更加迅速高效，为无线通信拓宽道路。

“蓝牙”这名称来自10世纪的丹麦国王哈拉尔德 (Harald Gormsson) 的外号。出身海盗家庭的哈拉尔德统一了北欧四分五裂的国家，成为维京王国的国王。由于他喜欢吃蓝莓，牙齿常常被染成蓝色，而获得“蓝牙”的绰号，当时蓝莓因为颜色怪异的缘故被认为是不适合食用的东西，因此这位爱尝新的国王也成为创新与勇于尝试的象征。1998年，爱立信公司希望无线通信技术能统一标准而取名“蓝牙”。

1.2 蓝牙和其他通信技术对比

无线名称	WiFi	BLE	ZigBee
传输速率	11~450Mbps	1Mbps	250Kbps
通信距离	200M	60~100M	100M
频段	2.4G/5G	2.4G	2.4G
安全性	AES128	AES128	AES128
国际标准	802.11b/g/n 802.11ac	802.15.4	802.15.4
功耗	10-50MA	0.3uA~7mA	0.4uA~20mA

1.3 蓝牙技术变迁史



第一代蓝牙：关于短距离通讯早期的探索

- 1999 年：蓝牙 1.0，早期的蓝牙存在多种问题，推出以后并未受到广泛的应用
- 2001 年：蓝牙 1.1，正式列入 IEEE 802.15.1 标准，但是容易受到同频率之间产品干扰，影响通讯质量
- 2003 年：蓝牙 1.2，完善了安全性问题，增加了AFH、eSCO、快连、支持Stereo 音效传输要求这四项新功能
- 代表作：爱立信第一台蓝牙手机 T39mc

第二代蓝牙：发力传输速率的 EDR 时

- 2004 年：蓝牙 2.0，使用了EDR技术，增加了连接设备的数量，提高了传输率，支持双工模式
- 2007 年：蓝牙 2.1，新增了省电功能以及SSP简易安全配对功能，改善了配对体验，提升了使用和安全强度，且支持 NFC 近场通信
- 代表作：以蓝牙与无线耳机沟通的 Sony Ericsson P910i PDA 手机

第三代蓝牙：High Speed，传输速率高达 24Mbps

- 2009 年：蓝牙 3.0，新增了可选技术 High Speed，可以使蓝牙调用 802.11 WiFi 用于实现高速数据传输，传输率高达 24Mbps，是蓝牙 2.0 的 8 倍，使用了AMP交替射频技术，允许蓝牙协议栈针对任一任务动态地选择正确射频；引入了 EPC 增强电源控制技术，再辅以 802.11，实际空闲功耗明显降低；还加入 UCD 单向广播无连接数据技术，提高了蓝牙设备的相应能力
- 代表作：蓝牙适配器

第四代蓝牙：主推“Low Energy”低功耗

- 2010 年：蓝牙 4.0，是第一个蓝牙综合协议规范，将三种规格集成在一起。其中最重要的变化就是 BLE（Bluetooth Low Energy）低功耗功能，提出了低功耗蓝牙、传统蓝牙和高速蓝牙三种模式：“高速蓝牙”主攻数据交换与传输；“传统蓝牙”则以信息沟通、设备连接为重点；“低功耗蓝牙”以不需占用太多带宽的设备连接为主，功耗较老版本降低了 90%。
- 2013 年：蓝牙 4.1，支持与 LTE 无缝协作，允许开发人员和制造商自定义蓝牙 4.1 设备的重新连接间隔，支持云同步，支持扩展设备与中心设备角色互换，支持蓝牙 4.1 标准的耳机、手表、键鼠，可以不用通过 PC、平板、手机等数据枢纽，实现自主收发数据

- 2014 年：蓝牙 4.2，改善了传输速率和隐私保护程度，蓝牙信号想要连接或者追踪用户设备，必须经过用户许可；支持 6LoWPAN，蓝牙 4.2 设备可以直接通过 IPv6 和 6LoWPAN 接入互联网，该技术允许多个蓝牙设备通过一个终端接入互联网或者局域网，这样大部分智能家居产品可以抛弃相对复杂的 WiFi 连接，改用蓝牙传输，让个人传感器和家庭间的互联更加便捷快速

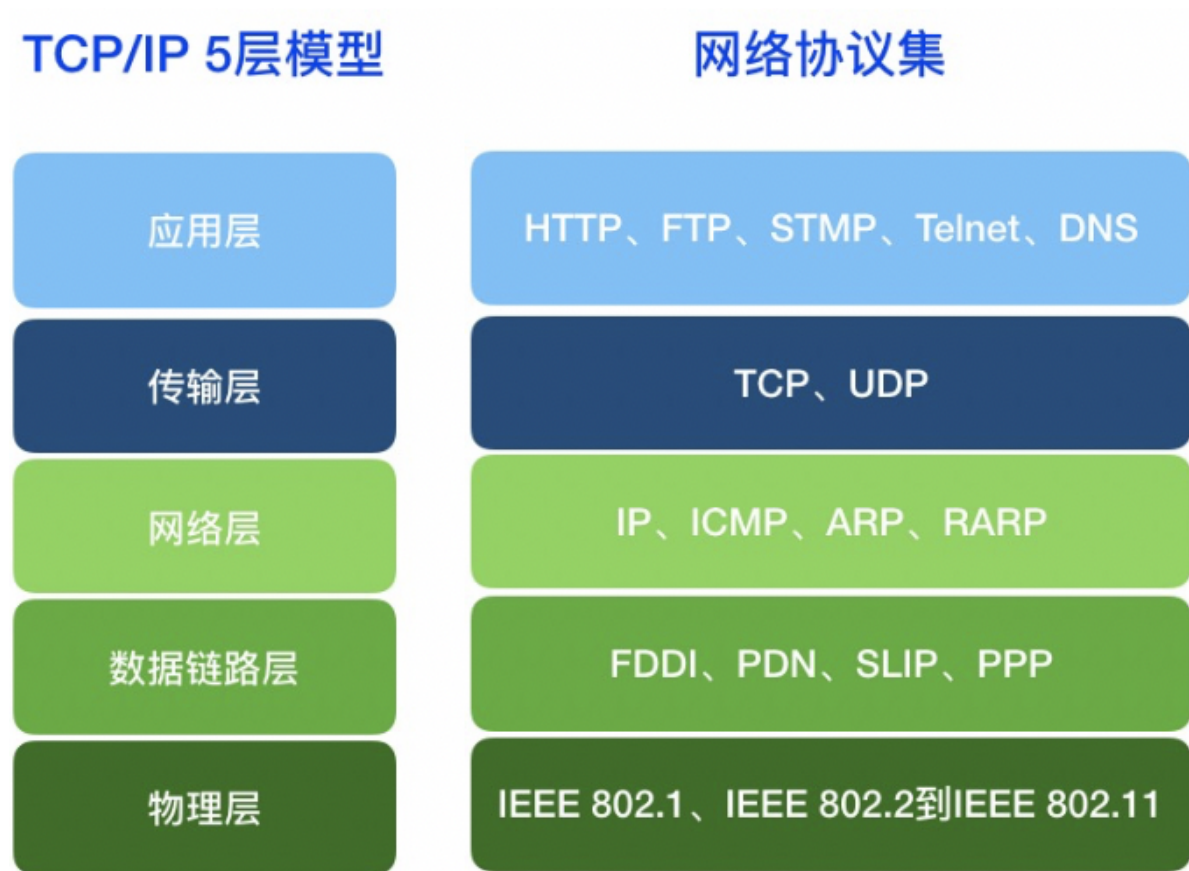
第五代蓝牙：开启物联网时代大门

- 2016 年：蓝牙 5.0，在低功耗模式下具备更快更远的传输能力，支持室内定位导航功能，针对 IoT 物联网进行底层优化，力求以更低的功耗和更高的性能为智能家居服务
- 2019 年：蓝牙 5.1，加入了测向功能和厘米级的定位服务，这项功能的加入使得室内的定位会变得更加精准，并且在小物体的位置上也能准确定位避免物品遗失

二、蓝牙技术

2.1 技术框架

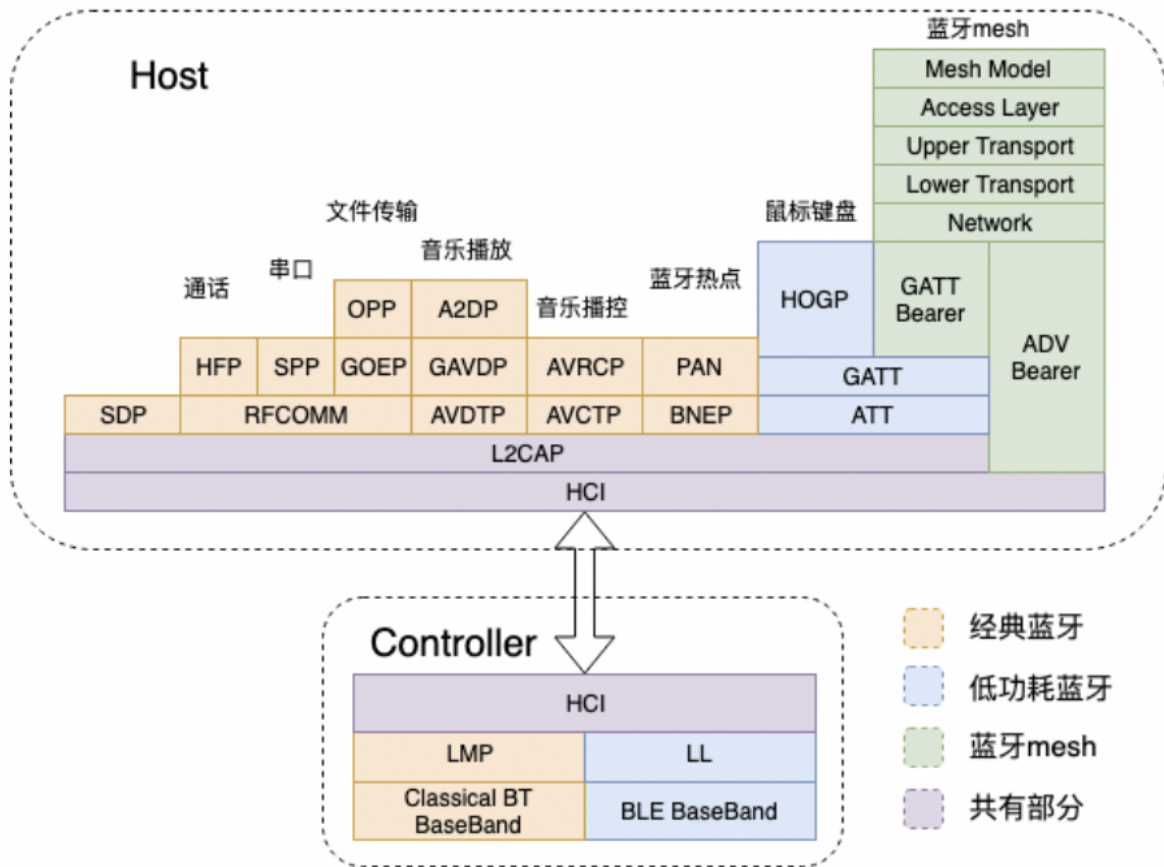
在看蓝牙技术整体框架之前，我们先回忆一下TCP/IP的5层网络模型和它所对应的网络协议集。



可能不同应用的同学只需要关注在不同层次上的网络协议就可以满足日常工作使用了，像应用开发同学，一般只关注应用层的协议，http，https 等等，网络的同学可能只需要关注 TCP 协议、IP 协议以及 ARP 协议等，WiFi 的同学主要focus在802.11协议上。

但蓝牙技术不同，它并不处于TCP/IP5层模型中的任何一层，而是覆盖了整个5层 TCP/IP 模型。

图1



在图1中我们可以看到，**蓝牙的架构分为 Host 和 Controller 两个模块**，

- Host 主要是各种业务场景需求的实现，
- Controller 部分主要负责的是蓝牙报文的收发以及蓝牙物理连接的管理这些基本功能。

所以通常绝大部分的开发工作都是在 Host 端进行，Controller 部分的工作大都是由专门的蓝牙芯片厂商来负责。

Host 和 Controller 分模块的最初设计理念是想让这两个模块单独运行在两颗不同的芯片甚至系统上，之间通过硬件通信端口(串口，USB)使用 HCI 协议进行连接和通信，这样可以方便替换和升级，例如对于不带蓝牙功能的电脑，我们可以买一个 USB 蓝牙接收器插到电脑上，就可以支持了蓝牙功能，这个场景下，HOST 模块就是运行在电脑系统上，Controller 模块就是运行在 USB 蓝牙接收器上。现在虽然有不少芯片把 Host 和 Controller 模块都放在了一颗芯片上，但是基本还是遵循这样的层次结构，只是将 HCI 协议从硬件通信端口换成了软件端口。

从应用场景来说，蓝牙规范针对了我们日常生活中会碰到的非常多的场景分别定义了不同的场景规范 (Profile)来支持这些场景下的需求，在图中我们可以看见，有 HFP(Hands Free Profile) 来支持蓝牙耳机通话场景，SPP(Serial Port Profile) 用于串口传输，OPP(Object Push Profile) 用于设备之间的文件传输场景，A2DP(Advanced Audio Distribution Profile) 用于蓝牙耳机收听音乐场景，AVRCP(A/V Remote Control Profile) 用于蓝牙耳机音乐播放控制场景，PAN(Personal Area Networking Profile) 可以让手机作为蓝牙热点提供上网服务。低功耗蓝牙鼠标键盘则是使用 HOGP(HID Over GATT Profile) 才能让蓝牙鼠标充一次电可以用三个月到半年。当然，图1中只是列出了我们最常会用到的一些蓝牙场景，蓝牙其实还有其他的像打印 (Basic Printing Profile)，心率(Heart Rate Profile)，寻物(Find Me Profile) 等等一系列场景规范(Profile) 来支持不同的应用场景。

2.2 经典蓝牙和低功耗蓝牙

蓝牙规范里分为经典蓝牙和低功耗蓝牙，经典蓝牙和低功耗蓝牙虽然都是蓝牙技术，但其实两种方案之间有非常大的差别，一个简单的区分就是看版本，**版本低于4.0的都是经典蓝牙，高于4.0的才能支持低功耗蓝牙**。在这里我们简单介绍一下两种蓝牙技术的概念和区别。

蓝牙按照支持协议划分，可分为以下两种：

- **经典蓝牙：Classic Bluetooth**，泛指支持蓝牙协议在4.0以下的模块，一般用于数据量比较大的传输，如：语音、音乐等较高数据量传输。

经典蓝牙模块可再细分为：传统蓝牙模块和高速蓝牙模块。

- 传统蓝牙模块在2004年推出，主要代表是支持蓝牙2.1协议的模块，在智能手机爆发的时期得到广泛支持。
- 高速蓝牙模块在2009年推出，速率提高到约24Mbps，是传统蓝牙模块的八倍，可以轻松用于录像机至高清电视、PC至PMP、UMPC至打印机之间的资料传输。
- **低功耗蓝牙：Bluetooth Low Energy**，是指支持蓝牙协议4.0或更高的模块，也称为BLE模块，最大的特点是成本和功耗的降低，应用于实时性要求比较高的产品中，比如：智能家居类（蓝牙锁、蓝牙灯）、传感设备的数据发送（血压计、温度传感器）、消费类电子（电子烟、遥控玩具）等。

蓝牙4.0以上的模块包含两个蓝牙标准，包含经典蓝牙部分和低功耗蓝牙部分。

蓝牙是工作在频率为 2400MHz 到 2483.5MHz 的无线通信协议，总共有 83.5MHz 的带宽资源。

- 在经典蓝牙的定义里，将这 83.5MHz 总共分为 80 个频道，每个频道是 1MHz 带宽，**蓝牙连接管理是用连接管理协议(Link Manager Protocol LMP)**。
- 而在低功耗蓝牙里，空中只有40个频道，每个频道是 2MHz 带宽，**连接管理使用的是连接层(Link Layer)**，空中数据包的结构也完全不同。

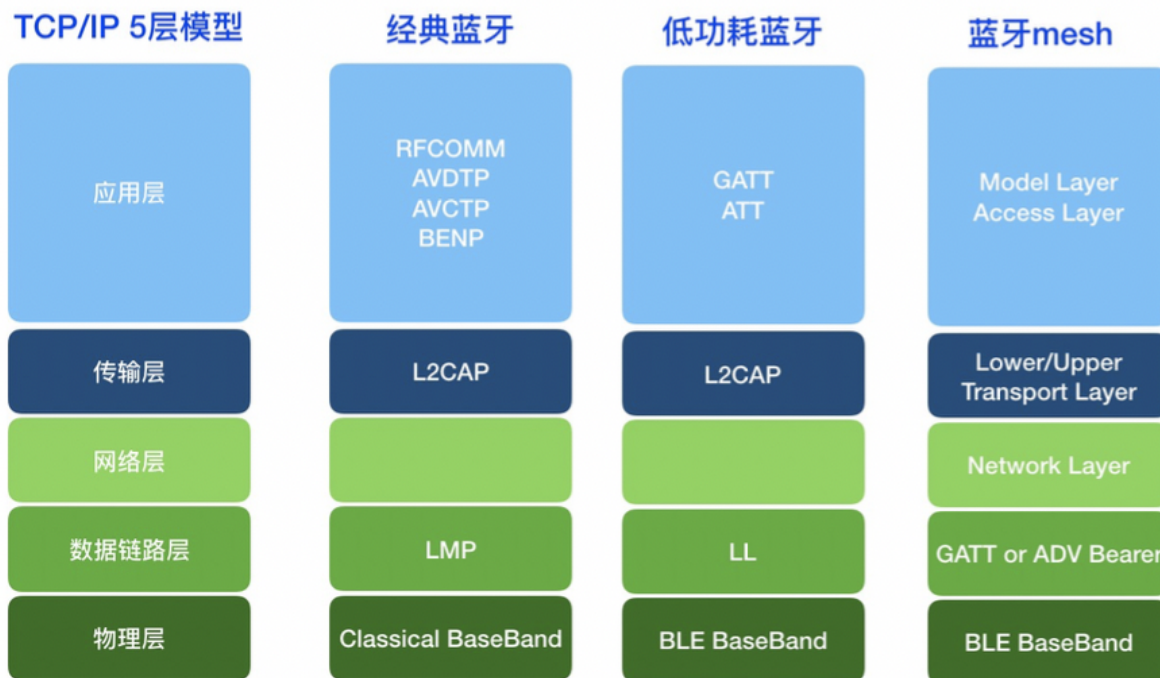
我们可以看到，**Controler 部分的功能完全是两个独立的路径，也就是说，经典蓝牙和低功耗蓝牙的 controller 可以独立存在，互相不依赖，所以目前市面上有只支持经典蓝牙的芯片，也有只支持低功耗蓝牙芯片，当然，也有两种蓝牙模式共存的芯片。**

在 Host 端，经典蓝牙和低功耗蓝牙的设计思路也不一样，

- 在经典蓝牙协议里，在逻辑链路控制与适配协议层(Logical Link Control and Adaptation Protocol L2CAP)之上，还根据不同的应用场景定义了不同的传输协议，例如 RFCOM，AVDTP，AVCTP，在不同的传输协议之上才定义了不同的 Profile，层次结构比较复杂，开发人员学习成本高。
- 而低功耗蓝牙就比较简单，只定义了一个属性协议(Attribute Protocol ATT)，基于属性协议定义了一个通用属性场景规范(Generic Attribute Profile GATT)和其他的针对特定业务的场景规范(Profile)，开发起来比较简单，整体框架也更容易实现私有场景的开发。
- 蓝牙mesh虽然使用了低功耗蓝牙的广播报文和 GATT，但可以算是一个半独立模块，蓝牙 mesh 构建了自己的一套网络寻址和数据传输机制，而且在 mesh 的协议文档里也强烈推荐使用广播承载 (ADV Bearer)的方式而不是 GATT 承载(GATT Bearer)。

2.3 蓝牙协议和规格文档

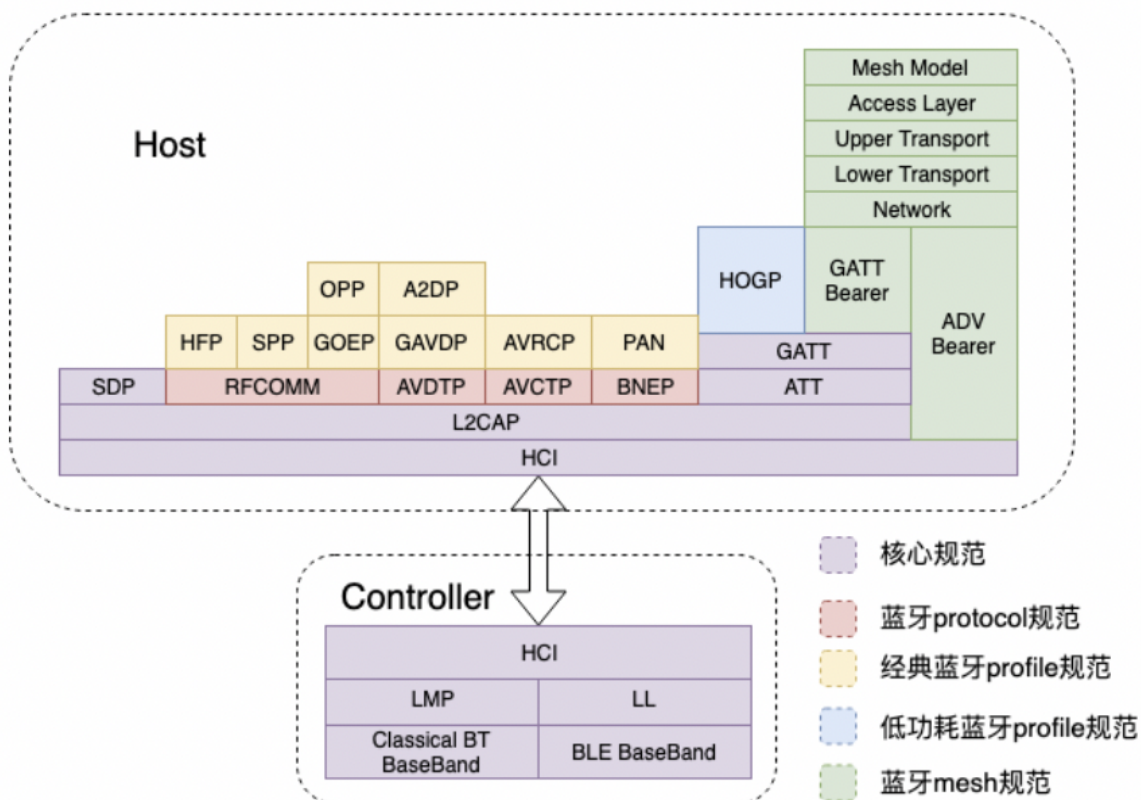
图2



由于经典蓝牙和低功耗蓝牙大都是点对点直连通信，完全不需要路由功能，所以没有特地定义网络层的协议，简单的寻址功能就在数据链路层的 LMP 层和 LL 层实现了，而蓝牙 mesh 由于是组建了 mesh 网络，有数据转发和寻址的需求，所以定义了一个Network Layer来做这件事情。

好的，了解到这些之后，那么我们去哪里可以看到这些协议更细节的内容呢？？我们打开蓝牙官网的协议页面<https://www.bluetooth.com/specifications/>，可以看到协议分为这些类型。

图3



可以看到，其实四大类主要的蓝牙规范就完全覆盖了整个蓝牙技术架构：










- 1、核心规范：Core Specifications，定义了蓝牙技术最核心的内容。覆盖了从物理层一直到传输层的内容。
- 2、Protocol规范：Protocol Specifications，在核心规范之上针对某一大类场景(例如音视频传输，线缆通信传输，网络通信传输)的数据通信需求来定义的传输协议，属于应用层协议，只在经典蓝牙中存在。
- 3、Profile规范：包含经典蓝牙的Traditional Profile Specifications和低功耗蓝牙的GATT Specifications。这类规范是针对某一个特定场景需求(例如听音乐，打电话)来对核心规范和protocol做出更细化的定义和对这些协议无法满足的一些细化需求做了补充协议。
- 4、蓝牙mesh规范：Mesh Networking Specifications，这是蓝牙最新的mesh规范。只使用了低功耗蓝牙的

还有一个需要注意的地方就是预分配序号：Assigned Numbers，蓝牙组织对某些字段的已经分配的数据定义，例如公司名，广播报文类型等等，都在这里可以查到。当然，还有一些协议修订案(Specification Errata)和蓝牙兼容性测试的文档，以及正在讨论开发中的文档草案，这些不是专门做蓝牙技术的同学可以不用关心。

<https://developer.aliyun.com/article/752526>

2.4 兼容性

蓝牙4.0、经典蓝牙和低功耗蓝牙之间的兼容性如下图示：

	If your product bears this logo...	It's compatible with products bearing any of these logos...
蓝牙4.0以上		  
经典蓝牙		 
低功耗蓝牙		

https://blog.csdn.net/Chuangke_Andy

三、Android 相关的类

BluetoothAdapter

代表本地设备蓝牙适配器。它用来执行一些基本的任务，

- 扫描蓝牙设备
- 查询已绑定的蓝牙列表、
- 使用已知 MAC 地址实例化 BluetoothDevice 对象、
- 创建一个 BluetoothServerSocket 来监听来自其他设备的连接请求
- 扫描蓝牙 LE 设备

比较常用的方法：

- public boolean enable()

打开本地蓝牙适配器。这将打开底层蓝牙硬件，并启动所有蓝牙系统服务。使用此方法需要获取 "Manifest.permission.BLUETOOTH_ADMIN" 权限。

调用此方法不会给用户任何提示。如果需要让用户来决定是否打开，使用以下方法：

```
ACTION_REQUEST_ENABLE
```

- public boolean disable()

关闭本地蓝牙适配器。这会优雅地关闭所有蓝牙连接，停止蓝牙系统服务，并关闭底层蓝牙硬件。

- public String getAddress()

返回本地蓝牙适配器的硬件地址。例如， "00:11:22:AA:BB:CC"。

- public String getName()

获取本地蓝牙适配器的蓝牙名称。

- public boolean startDiscovery()

开始扫描蓝牙设备。此方法要求权限 "Manifest.permission.BLUETOOTH_ADMIN"。

注册 ACTION_DISCOVERY_STARTED 和 ACTION_DISCOVERY_FINISHED 的 intent 可以获得扫描的开始和完成。

注册 BluetoothDevice.ACTION_FOUND 的 intent 可以在发现远程蓝牙设备时收到通知。

扫描蓝牙设备是一个重量级的过程，因此在扫描过程中，不应尝试与远程蓝牙设备建立新连接，现有连接将遇到带宽有限和高延迟的问题。可以通过 cancelDiscovery() 取消扫描。

- public boolean cancelDiscovery()

取消扫描蓝牙设备。因为扫描蓝牙设备是一个重量级的过程，所以我们在连接远程蓝牙（即 `android.bluetooth.BluetoothSocket#connect()`）之前要调用此方法。

- `public boolean isDiscovering()`

返回本地蓝牙适配器是否正在扫描。

- `public Set getBondedDevices()`

返回绑定（配对）到本地适配器的 `BluetoothDevice` 对象集。

- `public int getConnectionState()`

获取本地蓝牙适配器的当前连接状态。

- `public BluetoothServerSocket listenUsingRfcommWithServiceRecord(String name, UUID uuid)`

使用服务记录创建一个侦听、安全的 RFCOMM 蓝牙套接字。连接到此套接字的远程设备将被验证，并且此套接字上的通信将被加密。系统将分配一个未使用的 RFCOMM 频道来监听。

系统还将向本地 SDP 服务器注册一个服务发现协议 (Service Discovery Protocol, SDP) 记录，其中包含指定的 UUID、服务名称和自动分配的通道。远程蓝牙设备可以使用相同的 UUID 来查询我们的 SDP 服务器并发现要连接的通道。

当此套接字关闭或此应用程序意外关闭时，将删除此 SDP 记录。

使用 `BluetoothDevice#createRfcommSocketToServiceRecord` 从另一个使用相同 UUID 的设备连接到此套接字。

- `public BluetoothServerSocket listenUsingInsecureRfcommWithServiceRecord(String name, UUID uuid)`

使用服务记录创建一个侦听的、不安全的 RFCOMM 蓝牙套接字。不需要对链接密钥进行身份验证，通信可能容易受到中间人攻击。

对于蓝牙 2.1 设备，链接将被加密，因为加密是强制性的。对于旧设备（蓝牙 2.1 之前的设备），链接不会被加密。

如果需要加密和经过身份验证的通信通道，请使用 `listenUsingRfcommWithServiceRecord`。

- `public BluetoothServerSocket listenUsingEncryptedRfcommWithServiceRecord(String name, UUID uuid)`

使用服务记录创建一个侦听、加密的 RFCOMM 蓝牙套接字。链接将被加密，但不需要对链接密钥进行身份验证，即通信容易受到中间人攻击。使用 `listenUsingRfcommWithServiceRecord` 来确保经过身份验证的链接密钥。

如果无法验证链接密钥，请使用此套接字。例如，对于蓝牙 2.1 设备，如果任何设备没有输入和输出功能或仅具有显示数字键的能力，则无法进行安全套接字连接，可以使用此套接字。

如果不需要加密，请使用 `listenUsingInsecureRfcommWithServiceRecord`。对于蓝牙 2.1 设备，链接将被加密，因为加密是强制性的。

BluetoothServerSocket

一个监听蓝牙套接字。蓝牙套接字的接口类似于 TCP 套接字的接口：`java.net.Socket` 和 `java.net.ServerSocket`。

在服务器端，使用 `BluetoothServerSocket` 创建一个监听服务器套接字。当 `BluetoothServerSocket` 接受连接时，它将返回一个新的 `BluetoothSocket` 来管理连接。

在客户端，使用单个 `BluetoothSocket` 来启动、管理连接。

对于蓝牙 BR/EDR，最常见的套接字类型是 RFCOMM。RFCOMM 是基于蓝牙 BR/EDR 的面向连接的流式传输。它也称为串行端口配置文件 (Serial Port Profile, SPP)。要创建为传入蓝牙 BR/EDR 连接做好准备的监听 `BluetoothServerSocket`，请使用 `BluetoothAdapter#listenUsingRfcommWithServiceRecord()`。

对于蓝牙 LE，套接字使用 LE 面向连接的通道 (CoC)。LE CoC 是基于蓝牙 LE 的面向连接的流式传输，并具有基于信用的流量控制。相应地，使用 `BluetoothAdapter#listenUsingL2capChannel()` 创建一个侦听 `BluetoothServerSocket` 准备好接收蓝牙 LE CoC 连接。对于 LE CoC，可以使用 `getPsm()` 来获取对等方需要用来连接到您的套接字的协议服务多路复用器 (PSM) 值。

在 `BluetoothServerSocket` 创建后，调用 `accept()` 来监听传入的连接请求。此调用将阻塞，直到建立连接，此时，它将返回一个 {@link BluetoothSocket} 来管理连接。

{@link BluetoothServerSocket} 是线程安全的。特别是，{@link close} 将始终立即中止正在进行的操作并关闭服务器套接字

常用方法：

- `public BluetoothSocket accept()`

阻塞直到建立连接。

成功连接时返回已连接的 {@link BluetoothSocket}。

此调用返回后，可以再次调用它以接受后续传入连接。

{@link close} 可用于从另一个线程中止此调用

四、Android 蓝牙开发

4.1 蓝牙的开关、搜索

注册监听蓝牙广播

```

        private void registerBluetoothReceiver() {
            IntentFilter intentFilter = new IntentFilter();
            intentFilter.addAction(BluetoothDevice.ACTION_FOUND); // 发现新设备(未配对的设备)
            intentFilter.addAction(BluetoothAdapter.ACTION_DISCOVERY_STARTED); // 扫描开始
            intentFilter.addAction(BluetoothAdapter.ACTION_DISCOVERY_FINISHED); // 扫描结束
            intentFilter.addAction(BluetoothAdapter.ACTION_STATE_CHANGED); // 监听蓝牙状态改变，例如打开关闭

            intentFilter.addAction(BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED);

            //            intentFilter.addAction(BluetoothA2dp.ACTION_CONNECTION_STATE_CHANGED);
            //
            intentFilter.addAction(BluetoothHeadset.ACTION_CONNECTION_STATE_CHANGED);

            registerReceiver(mBluetoothReceiver, intentFilter);
        }

```

蓝牙广播的处理

```

class BluetoothReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        switch (intent.getAction()) {
            case BluetoothDevice.ACTION_FOUND:
                BluetoothDevice device =
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                Log.d(TAG, "发现设备 : " + device.getName() + ", " +
                    device.toString());
                mScanListAdapter.add(device);
                break;

            case BluetoothAdapter.ACTION_DISCOVERY_STARTED:
                Toast.makeText(BluetoothActivity.this, "蓝牙扫描开始",
                    Toast.LENGTH_SHORT).show();
                Log.d(TAG, "蓝牙扫描开始");
                break;

            case BluetoothAdapter.ACTION_DISCOVERY_FINISHED:
                Toast.makeText(BluetoothActivity.this, "蓝牙扫描结束",
                    Toast.LENGTH_SHORT).show();
                Log.d(TAG, "蓝牙扫描结束");
                break;

            case BluetoothAdapter.ACTION_STATE_CHANGED:
                int state = intent.getIntExtra(BluetoothAdapter.EXTRA_STATE,
                    -1);

                Log.d(TAG, "蓝牙状态改变 state : " + state);
                updateBluetoothState(state);
                break;

            case BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED:

```

```

        Toast.makeText(BluetoothActivity.this, "蓝牙连接状态改变",
            Toast.LENGTH_SHORT).show();
        break;
    }
}

```

查询已配对设备

```

Set<BluetoothDevice> pairedDevicesSet = mBluetoothAdapter.getBondedDevices();

```

4.2 文件传输服务端

```

class BluetoothServerService : Service() {
    companion object {
        private var listener: BluetoothListener? = null

        fun setListener(listener: BluetoothListener) {
            BluetoothServerService.listener = listener
        }
    }

    private val mExecutor = Executors.newFixedThreadPool(2)
    private var mAdapter = BluetoothAdapter.getDefaultAdapter()
    private var mSocket: BluetoothSocket? = null
    private var mServerTransferThread: BluetoothTransferThread? = null

    private val mHandler = Handler(Looper.getMainLooper()) {
        listener?.onTip(it.obj as String)
        true
    }

    private val mActionReceiver = object : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            when (intent.action) {
                BluetoothTool.ACTION_SEND_MSG -> {
                    val msg = intent.getStringExtra(BluetoothTool.EXTRA) ?: ""
                    mServerTransferThread?.sendMsg(msg) ?: notifyUI("设备未连接!")

                }

                BluetoothTool.ACTION_SEND_FILE -> {
                    val path = intent.getStringExtra(BluetoothTool.EXTRA) ?: ""
                    mServerTransferThread?.sendFile(path) ?: notifyUI("设备未连接!")

                }
            }
        }
    }

    override fun onCreate() {

```

```

        super.onCreate()
        "BluetoothServerService onCreate".log()
        mAdapter.enable() //打开蓝牙

//        //开启蓝牙发现功能（300秒）
//        val discoverableIntent =
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE)
//
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300)
//        discoverableIntent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
//        startActivity(discoverableIntent)


// 监听来自UI页面的动作
val intentFilter = IntentFilter().apply {
    addAction(BluetoothTool.ACTION_SEND_MSG)
    addAction(BluetoothTool.ACTION_SEND_FILE)
}
registerReceiver(mActionReceiver, intentFilter)


//开启服务端监听线程
startListen()
}


override fun onDestroy() {
    "BluetoothServerService onDestroy".log()
    unregisterReceiver(mActionReceiver)
    super.onDestroy()
}


override fun onBind(intent: Intent?): IBinder? {
    return null
}


/**
 * 开启服务端监听
 */
private fun startListen() {
    mExecutor.execute(ServerThread())
}


fun close() {
    try {
        mSocket?.close()
    } catch (e: Exception) {
        e.printStackTrace()
    }
}


fun notifyUI(msg: String) {
    msg.log()
    val message = Message.obtain().apply {
        obj = msg
    }
}

```



```

        mHandler.sendMessage(message)
    }

    /**
     * 服务端监听线程
     */
    inner class ServerThread : Runnable {
        private var mServerSocket: BluetoothServerSocket? = null
        var shouldLoop = true

        override fun run() {
            // Keep listening until exception occurs or a socket is returned.
            mServerSocket =
                mAdapter.listenUsingRfcommWithServiceRecord(
                    "BLUETOOTH_SERVER",
                    BluetoothTool.RFCOMM_UUID
                )
            notifyUI("服务端初始化完成，开始监听")

            while (shouldLoop) {
                try {
                    mSocket = mServerSocket?.accept()
                    mSocket?.also {
                        notifyUI("服务端与客户端${it.remoteDevice.name}建立连接")

                        // 开启数据传输线程
                        mServerTransferThread =
                            BluetoothTransferThread(it, filesDir.absolutePath,
                                mHandler)

                        mExecutor.execute(mServerTransferThread)

                        close() // RFCOMM 一次只允许每个通道有一个已连接
                                的客户端，因此大多数情况下，在接受已连接的套接字后，您可以立即在 BluetoothServerSocket 上调
                                用 close()。

                        shouldLoop = false // 只支持一对一，所以在连接成功后停止监听
                    }
                } catch (e: Exception) {
                    e.printStackTrace()
                    notifyUI("服务端监听异常! ${e.message}")
                }
            }
            notifyUI("服务端与客户端连接完成，监听线程结束")
        }

        fun close() {
            try {
                shouldLoop = false
                mServerSocket?.close()
                notifyUI("服务端监听线程关闭")
            } catch (e: Exception) {
                e.printStackTrace()
                notifyUI("服务端监听线程关闭异常! ${e.message}")
            }
        }
    }
}

```

4.3 文件传输客户端

```
class BluetoothClientService : Service() {
    companion object {
        private var listener: BluetoothListener? = null

        fun setListener(listener: BluetoothListener) {
            BluetoothClientService.listener = listener
        }
    }

    private val mExecutor = Executors.newFixedThreadPool(3)
    private var mAdapter = BluetoothAdapter.getDefaultAdapter()
    private var mClientTransferThread: BluetoothTransferThread? = null
    private var mSocket: BluetoothSocket? = null

    private val mHandler = Handler(Looper.getMainLooper()) {
        listener?.onTip(it.obj as String)
        true
    }

    private val mActionReceiver = object : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            when (intent.action) {
                BluetoothTool.ACTION_SEND_MSG -> {
                    val msg = intent.getStringExtra(BluetoothTool.EXTRA) ?: ""
                    mClientTransferThread?.sendMsg(msg) ?: notifyUI("设备未连接")
                }

                BluetoothTool.ACTION_SEND_FILE -> {
                    val path = intent.getStringExtra(BluetoothTool.EXTRA) ?: ""
                    mClientTransferThread?.sendFile(path) ?: notifyUI("设备未连接")
                }

                BluetoothTool.ACTION_CONNECT -> {
                    val device = intent.getParcelableExtra<BluetoothDevice>(BluetoothTool.EXTRA)
                    device?.also {
                        connect(it)
                    }
                }
            }
        }
    }

    override fun onCreate() {
        super.onCreate()
        "BluetoothClientService onCreate".log()

        val intentFilter = IntentFilter().apply {
            addAction(BluetoothTool.ACTION_SEND_MSG)
            addAction(BluetoothTool.ACTION_SEND_FILE)
            addAction(BluetoothTool.ACTION_CONNECT)
        }
    }
}
```

```

    }
    registerReceiver(mActionReceiver, intentFilter)
}

```

```

override fun onDestroy() {
    "BluetoothClientService onDestroy".log()
    unregisterReceiver(mActionReceiver)
    super.onDestroy()
}

```

```

override fun onBind(intent: Intent?): IBinder? {
    return null
}

```

```

fun close() {
    try {
        mSocket?.close()
    } catch (e: Exception) {
        e.printStackTrace()
    }
}

```

```

fun notifyUI(msg: String) {
    msg.log()
    val message = Message.obtain().apply {
        obj = msg
    }
    mHandler.sendMessage(message)
}

```

```

/**
 * 连接指定蓝牙设备[device]
 */
fun connect(device: BluetoothDevice) {
    // Cancel discovery because it otherwise slows down the connection.
    if (mAdapter.isDiscovering) {
        mAdapter.cancelDiscovery()
    }

    notifyUI("开始连接远程设备 ${device.name}")
    mExecutor.execute(ClientThread(device))
}

```

```

/**
 * 客户端连接远程
 */
inner class ClientThread(private val device: BluetoothDevice) : Runnable {

    override fun run() {
        try {
            mSocket =

```

device.createRfcommSocketToServiceRecord(BluetoothTool.RFCOMM_UUID)
加密传输, Android强制执行配对, 弹窗显示配对码

//

个线程

```
        mSocket?.let {
            mExecutor.execute(ConnectThread(it))    //执行连接操作需要另开启一
        }

    } catch (e: Exception) {
        e.printStackTrace()
        notifyUI("连接远程设备 ${device.name} 异常! ${e.message}")
    }
}

fun close() {
    try {
        mSocket?.close()
    } catch (e: Exception) {
        e.printStackTrace()
    }
}

inner class ConnectThread(private val socket: BluetoothSocket) : Runnable {

    override fun run() {
        try {
            socket.connect()
            notifyUI("连接远程设备成功")

            // 开启数据传输线程
            mClientTransferThread =
                BluetoothTransferThread(socket, filesDir.absolutePath,
mHandler)

            mExecutor.execute(mClientTransferThread)
        } catch (e: Exception) {
            e.printStackTrace()
            notifyUI("连接远程设备异常! ${e.message}")
        }
    }
}
}
```