1. 杂记

BLE广播数据包简介：

\* 有效广播数据最大长度： 37 字节

\* 广播频道：37，38，39 （跳频）

\* 功率：一般是0dBm, 最大10dBm

存取地址 （Access Address) 对于广播包，这是一个固定长度(4个字节）及固定内容（0x8e89bed6)的主从设备识别广播包的存取地址。

头信息 (Header Info) 固定为2个字节：分解为16个位来使用

位[0:3]：广播包（PDU) 类型，总共有6个类型：

PDU Type

b3b2b1b0 Packet Name

0000 ADV\_IND： 可连接通用连接广播

0001 ADV\_DIRECT\_IND：可连接定向连接（指定设备）广播

0010 ADV\_NONCONN\_IND：不可连接通用广播

0011 SCAN\_REQ：扫描请求

0100 SCAN\_RSP：扫描响应

0101 CONNECT\_REQ：连接请求

0110 ADV\_SCAN\_IND：可扫描通用广播

0111-1111 Reserved

位[4:5]：保留

位[6]：RxAdd

位[7]：TxAdd

位[8:13]：广播数据长度 （最大为37 字节）

位[14:15]：保留

MAC 地址 固定为6个字节

广播数据：标志域 02 01 06

02：此域长度为２个字节

01: 为标志域

06：BIT1,BIT2 = 1， 根据以下定义，所以为：

0 LE Limited Discoverable Mode

1 LE General Discoverable Mode

2 BR/EDR Not Supported (i.e. bit 37 of LMP Extended

Feature bits Page 0)

3 Simultaneous LE and BR/EDR to Same Device Capa-

ble (Controller) (i.e. bit 49 of LMP Extended Feature

bits Page 0)

4 Simultaneous LE and BR/EDR to Same Device Capa-

ble (Host) (i.e. bit 66 of LMP Extended Feature bits

Page 1)

1. .7 Reserved

服务标识域 03 03 02 18

03：此域长度为２个字节

03: 为服务标识域

02 18：-> 18 02 => Immediate Alert Service (防丢报警服务）

设备名域 0e 09 45 .... 6d 65

0e: 此域长度为14个字节

09：为设备名域

1. ...6d 65: 设备名

BLE 连接过程中有三个重要的数据包：SCAN\_REQ, SCAN\_RSP 和 CONNECT\_REQ。

SCAN\_REQ： 扫描请求，由主设备（MASTER DEVICE）向从设备（SLAVE DEVICE）发出，目的是为了获得从设备的响应以得到更多的从设备 广播数据信息（包括设备名字，或者服务UUID，及其它如厂家特定格式的信息（如硬件版本，软件版本号，设备系列号等等）

SCAN\_RSP: 从设备对就主设备发起的SCAN\_REQ的响应，作为广播包的补充，从设备可以给主设备更多的广播数据，比如说，有些设备在广播 包里面没有设备名字，这个时候就可以把设备名字放在这个包里面发给主设备。

CONNECT\_REQ：主设备向从设备发出连接请求。至此连接建立完成（从设备不会响应这个请求），如果从设备没有连接上面的问题的话，以后 主从双方会开始相互交换有效数据（基于GAP,GATT及SMP协议）或者交换空包。

1. 简介

BLE：低功耗蓝牙是 2.4 GHz 蓝牙无线技术的子集，专门为连接的设备提供低功耗和不频繁的数据传输。

Central/Client：扫描并连接到 BLE 外围设备以执行某些操作的设备。在应用程序开发的上下文中，这通常是 Android 设备。

Peripheral/Server：宣传其存在并通过中心连接以完成某些任务的设备。在应用程序开发的上下文中，这通常是您正在使用的 BLE 设备，例如心率监测器。

GATT Service：描述设备特性的集合（characteristics数据字段），例如设备信息服务可以包含一个表示设备序列号的特征，以及另一个表示设备电池电量的特征。

GATT Characteristic：包含通常可以读取或写入的有意义数据的实体，例如序列号字符串特征。

GATT Descriptor：一个定义的属性，描述它所绑定的Characteristic，例如Client Characteristic Configuration描述符显示Client当前是否订阅了Characteristic的值更改。

Notifications：当Characteristic值发生变化时，BLE Server设备通知Client的一种方法。Client不需要确认它已收到数据包。

Indictations：每个数据包都由Client确认。这以吞吐量为代价保证了它们的交付。

UUID：通用唯一标识符，用于标识Server、Characteristic和GATT Descriptor的 128 位数字。

BLE中的Client-Server关系：

作为Client的 Android 设备可以同时连接到多个Server设备（外部 BLE 设备），但作为Server设备的每个 BLE 设备通常一次只能与一个Client交互。最常见的行为是，当 BLE 设备连接到Client时，它将停止作为Server设备进行广播，因为它不再能够连接到。

Server托管一个 GATT 数据库，该数据库提供Client通过 BLE 访问的信息。

BLE 通信的要点可以概括为 3 种常见的 BLE 操作：

写入：Client（应用程序）将一些字节写入Server（BLE 设备）上的Characteristic 或Descriptor。Server的固件处理写入并执行一些Server端操作以响应它。例如，智能恒温器可能具有在写入时改变目标温度的特性。

读取：Client（应用程序）读取Server（BLE 设备）上的Characteristic 或Descriptor的值，并根据事先建立的协议对其进行解释。智能恒温器可能具有其值代表当前目标温度的特性。

Notify/Indicate：Client（app）订阅一个Characteristic 用于通知或指示，当Characteristic 值发生变化时由Server通知。智能恒温器可能具有可通知的特性，在订阅时会报告环境温度的变化。

BLE API：

BluetoothAdapter：Android 设备的蓝牙硬件的表示。此类的一个实例由 BluetoothManager 类提供。 BluetoothAdapter 提供有关蓝牙硬件的开/关状态的信息，允许我们查询绑定到 Android 的蓝牙设备，还为我们提供启动 BLE 扫描的能力。

BluetoothLeScanner：由 BluetoothAdapter 类提供，该类允许我们启动 BLE 扫描。注意：从 Android M (6.0) 及以上版本开始，BLE 扫描需要 ACCESS\_COARSE\_LOCATION 或 ACCESS\_FINE\_LOCATION，而 Android 10 及以上版本需要 ACCESS\_FINE\_LOCATION。

ScanFilter：允许我们将扫描结果范围缩小到我们在 BLE 扫描期间寻找的特定设备。应用程序的一个典型用例是根据 BLE 设备的广告服务 UUID 过滤 BLE 扫描结果。

ScanResult：表示通过BLE扫描得到的BLE扫描结果，包含BLE设备的MAC地址、RSSI（信号强度）、广告数据等信息。 getDevice() 方法公开了 BluetoothDevice 句柄，该句柄可能包含 BLE 设备的名称，并允许应用程序连接到它。

BluetoothDevice：表示应用程序可以连接、绑定（配对）或两者兼有的物理蓝牙（不是特别是 BLE）设备。此类提供的关键信息包括：设备名称（如果可用）、设备的 MAC 地址和设备的当前绑定状态。

BluetoothGatt：BLE 设备 GATT 配置文件的入口点。允许我们执行服务发现、断开连接、请求 MTU 更新，以及访问 BLE 设备上存在的服务和特性。我们可以将其视为已建立的 BLE 连接的句柄。

BluetoothGattService、BluetoothGattCharacteristic 和 BluetoothGattDescriptor：代表 GATT 服务、特征和描述符的包装类。

BluetoothGattCallback：应用程序必须实现的接口，以便接收大多数与 BluetoothGatt 相关的操作的回调，例如读取、写入或收到有关传入通知或指示的通知。

BLE扫描：

除非BluetoothDevice句柄是从最近的扫描中缓存的，否则我们需要在连接到 BLE 设备之前执行 BLE 扫描。典型的 BLE 应用程序连接设置流程如下所示：

1. 使用可选的扫描结果过滤执行 BLE 扫描。

扫描结果过滤

在我们扫描周围的设备之前，询问我们感兴趣的设备是否具有独特的特征会有助于 Android BLE 扫描仪将扫描结果缩小到我们关心的那些。 ScanFilter 类允许我们根据以下条件过滤传入的 ScanResults：

1.广告服务 UUID

2.广告服务的服务数据

3.广告 BLE 设备的名称

4.广播 BLE 设备的 MAC 地址

5.其他制造商特定数据

虽然有些应用程序确实有在没有 ScanFilter 的情况下进行扫描的用例（我们自己的 LightBlue 应用程序就是其中之一），但大多数应用程序使用 BLE 连接到特定类型的设备，因为它们旨在执行某些有意义的任务，仅使用该类型的设备。例如，显示当前温度、湿度和气压的应用程序只会尝试连接到具有这些功能的 BLE 设备.使用 ScanFilter.Builder 类创建一个 ScanFilter，并在最终调用 build() 之前调用其方法来设置过滤条件，例如：

val filter = ScanFilter.Builder().setServiceUuid(

ParcelUuid.fromString(ENVIRONMENTAL\_SERVICE\_UUID.toString())

).build()

确保应用程序仅获取运行所述自定义固件的设备的最简单方法是生成随机 UUID，并拥有固件宣传这个 UUID。然后，该应用程序会根据 UUID 执行扫描过滤，并且由于 UUID 对于实际用途而言足够独特，因此我们可以预期扫描只会拾取我们感兴趣的设备。

除了扫描结果过滤之外，Android 还允许我们指定扫描期间使用的扫描设置，由 ScanSettings 类表示，该类也带有自己的 ScanSettings.Builder 构建器类。以下是 Android 允许我们调整的一些实用且常用的扫描设置：

1.指定所需的 BLE 扫描模式，从低功率高延迟扫描到高功率低延迟扫描。

大多数在前台进行扫描的应用程序应该使用 SCAN\_MODE\_BALANCED 来进行超过大约 30 秒的扫描。

如果应用程序只会扫描一小段时间，则建议使用 SCAN\_MODE\_LOW\_LATENCY，通常是为了查找非常特定类型的设备。

SCAN\_MODE\_LOW\_POWER 用于超长时间扫描，或用于在后台进行的扫描（在用户许可的情况下）。请注意，如果正在扫描的设备具有足够高的广告间隔，它不会与应用程序的扫描频率重叠，那么这种低功耗扫描模式的高延迟特性可能会导致丢失广告包。

2.指定遇到的 BLE 广告数据包的回调类型。

像 LightBlue 这样需要不断更新传入广告数据包的应用程序应该使用 CALLBACK\_TYPE\_ALL\_MATCHES 来获得有关所有传入数据包的通知。如果应用未指定所需的回调类型，则这是默认设置。

CALLBACK\_TYPE\_FIRST\_MATCH 用于如果应用只关心为每个与 ScanFilter 指定的过滤条件匹配的设备（或附近的所有设备，如果未指定 ScanFilter）获取单个回调。

CALLBACK\_TYPE\_MATCH\_LOST 有点奇怪——我们在使用它时有不同的经验，一般不推荐它。通常最好自己实现一个计时器，该计时器定期检查的 ScanResults 列表并根据 ScanResult 的 getTimestampNanos() 方法删除过时的（例如，在过去 10 秒左右没有遇到过的），但请注意此方法提供自 Android 系统启动时间以来的时间戳，而不是自 Epoch 时间以来的时间。

3.指定应将广告包目击作为扫描结果显示的阈值。

MATCH\_MODE\_STICKY 在过滤掉距离 Android 设备太远的广告 BLE 设备时很有用，因为它需要更高的信号强度阈值和目击次数才能将该 BLE 设备作为扫描结果显示给我们的应用程序。

MATCH\_MODE\_AGGRESSIVE 与 MATCH\_MODE\_STICKY 相反，它将显示在 BLE 扫描仪范围内广播的每个设备，无论远近。

执行 BLE 扫描时最常见的错误是未记录的“应用程序扫描太频繁”错误。 Android 对 BluetoothLeScanner 对象的每个应用程序每 30 秒调用 5 次 startScan(...) 方法的内部限制，超出此限制不会触发任何错误回调——只是 Logcat 中的一个模糊条目表明您的应用程序扫描过于频繁。更奇怪的是，除非您的 Logcat 窗口中没有任何过滤活动，否则您通常不会看到这个 Logcat 条目，因为该条目有一个未知的包名称“？”与之相关的.

如果应用程序超出此限制，则扫描似乎已经开始，但不会将扫描结果传递到回调正文。 30 秒后，应用应调用 stopScan()，然后再次调用 startScan(...) 以开始接收扫描结果 - 导致错误的旧 startScan(...) 方法调用不会自动启动30 秒冷却时间结束后再次接收扫描结果。

可能遇到的另一个模糊错误是 SCAN\_FAILED\_APPLICATION\_REGISTRATION\_FAILED 错误。此错误由 ScanCallback 的 onScanFailed(...) 方法返回，并被模糊地描述为应用程序未能向 BLE 扫描仪注册。未记录的解决方案是请求用户在其设备上禁用和重新启用蓝牙。如果这失败了（通常是这样），接下来也是唯一要做的就是要求用户执行 Android 重启。

为了停止正在进行的 BLE 扫描，BluetoothLeScanner 提供了一个 stopScan() 方法。

我们可以添加一个 isScanning 属性，该属性将通知我们 BluetoothLeScanner 的状态，因为扫描仪的状态并未公开。

1. 获取 ScanResult 对象。

ScanResult 对象作为 ScanCallback 的 onScanResult(...) 方法的一部分出现，通常我们在 ScanResult 中关心的事情是：

1.标识广告扫描结果的设备 MAC 地址。

通过 getDevice() 后跟 getAddress() 获得，或者在 Kotlin 中简单地通过 device.address 获得。

警告：实现蓝牙 4.2 的 LE 隐私功能的设备会定期随机化其公共 MAC 地址，因此通常不应将通过扫描获得的 MAC 地址用作识别设备的长期手段——除非固件保证它不是轮换 MAC 地址，或者如果它有一种带外方式来传达它当前的公共 MAC 地址。

2.我们可以向用户显示的设备名称。

通过 getDevice() 后跟 getName() 获得，或者在 Kotlin 中简称为 device.name。并非所有 BLE 设备都会公布设备名称，因此某些 BLE 设备的名称可能为空。

3.广告 BLE 设备的 RSSI 或信号强度，以 dBm 为单位。

通过 getRssi() 获得，或者在 Kotlin 中简单地通过 rssi 获得。

按信号强度降序排列扫描结果是找到最接近 Android 设备的外设的好方法，但它不是 100% 保证，因为 RSSI 会受到广告设备天线的传输功率和其他物理因素的影响，例如 Android 或 BLE 设备周围存在金属物体。

分贝值通常是相对的，而不是基于绝对比例。这意味着 -42 dBm 的 RSSI 读数对于一部 Android 手机来说可能是“近距离”，而对于另一部手机来说可能是“中等范围”。通常不建议将 RSSI 读数普遍映射到真实世界的物理距离。

1. 我们需要连接到设备的 BluetoothDevice 句柄，通过 getDevice() 方法访问。

ScanResult 的 ScanRecord 中的额外广告数据，通过 getScanRecord() 访问。

ScanRecord 可以方便地从扫描记录中解析出任何制造商特定数据和服务数据，并且可以使用 getManufacturerSpecificData(...) 和 getServiceData(...) 方法访问这些数据。

可以使用 getBytes() 方法访问原始扫描记录字节。

3、调用 ScanResult 对象的 getDevice() 方法获取 BluetoothDevice 对象。

4、调用 BluetoothDevice 对象的 connectGatt() 方法来启动 BLE 连接。

在应用程序中启动 BLE 连接的典型流程大致可以分为两种类型：

1.自动连接。该应用程序根据特定的启发式方法从返回的扫描结果中自动连接到设备，例如我们正在扫描广告某些私有服务 UUID 的设备，并且在低延迟模式下扫描几秒钟后只有一个这样的设备。

2.手动连接。用户仔细阅读扫描结果列表并手动选择要连接的设备。

无论应用属于哪种用例，都应始终在连接到 BLE 设备之前停止 BLE 扫描。这样做可以节省电力，更重要的是——根据我们的经验——还有助于提高连接过程的可靠性。

BluetoothDevice 句柄上的 connectGatt() 方法将用于启动与 BLE 设备的连接。 connectGatt() 有多个重载，但我们将使用的重载具有以下签名：

public BluetoothGatt connectGatt(

Context context,

boolean autoConnect,

BluetoothGattCallback callback

)

原因：这个版本的 connectGatt(...) 早在 API 级别 18 就可用，我们打算使用最低 API 级别 21（Android 5.0 Lollipop），如本博文中所述。其他重载仅在 API 级别 23 或 26 中可用。

此方法的所有变体都在其参数中包含一个 autoConnect 布尔值，这可能会误导新手——将此标志设置为 true 不会导致 Android 在连接时自动尝试重新连接到 BluetoothDevice！相反，将标志设置为 true 只会导致连接操作不会超时，如果尝试连接到之前缓存的蓝牙设备，这可能会很有帮助。

根据我们的经验，如果设备已经在附近被发现，则将标志设置为 true 可能会导致连接过程比平常慢。我们在开发 BLE 应用程序时的偏好是将标志设置为 false，而是依靠 onConnectionStateChange 回调来通知我们连接过程是否成功。如果连接失败，我们可以简单地尝试再次连接，并将 autoConnect 设置为 false。

connectGatt(...) 的所有变体都返回一个 BluetoothGatt 对象，可以将其视为我们正在建立的 BLE 连接的句柄，它允许我们启动读取和写入操作。但是，我们通常不保留对返回的 BluetoothGatt 的引用，而只保留对作为 BluetoothGattCallback 回调方法参数提供给我们的引用。

我们在这个阶段需要实现的主要回调方法是 onConnectionStateChange()，它提供了有关 BLE 连接状态的关键信息。这是与给定 BLE 连接相关的连接和断开事件的真实来源。

成功的连接尝试将看到 onConnectionStateChange() 回调发生，其status参数设置为 GATT\_SUCCESS，newState 参数设置为 BluetoothProfile.STATE\_CONNECTED。此时，存储对同样由该回调提供的 BluetoothGatt 对象的引用至关重要。这将是我们向 BLE 设备发出命令的主界面。

建立 BLE 连接时的错误也会通过 BluetoothGattCallback 的 onConnectionStateChange() 方法浮出水面。典型的错误检查流程检查 onConnectionStateChange() 的状态参数是否为 GATT\_SUCCESS。如果不是，我们手上有一个由 status 参数表示的错误。

一些错误状态代码被记录为 BluetoothGatt 的公共常量，但我们在尝试连接到 BluetoothDevice 时遇到的最常见的错误必须是臭名昭著的状态 133 错误。这个错误不仅没有在 BluetoothGatt 常量列表下记录，而且查看 Android 源代码会发现它的名称只是 GATT\_ERROR（0x85，133 的十六进制）。

我们通常看到错误 133 在以下两种情况之一中最常发生：

1.我们尝试连接的 BluetoothDevice 不再是广告或在蓝牙范围内，并且在 autoConnect 参数设置为 false 的情况下尝试连接大约 30 秒后，connectGatt() 调用已超时。

2.BLE 设备上运行的固件拒绝了 Android 的连接尝试。

无论在 onConnectionStateChange() 回调中获得哪个错误状态代码，恢复流程通常如下所示：

1.在 BluetoothGatt 对象上调用 close() 以指示我们已完成此 BluetoothGatt 实例，并且系统可以释放任何挂起的资源。

2.清空对此 BluetoothGatt 对象的任何引用。

3.如果状态码是 GATT\_INSUFFICIENT\_ENCRYPTION 或 GATT\_INSUFFICIENT\_AUTHENTICATION：首先调用 createBond() 并等待绑定过程首先成功，然后再调用 connectGatt()。

4.对于其他状态代码：要么向用户显示错误，要么尝试再次静默重新连接几次，然后放弃。考虑到错误 133 有时可能非常随机并且在连接尝试最终成功之前连续发生几次，后者实际上是一种可行的策略。

BLE发现服务：

与设备建立 BLE 连接后，服务发现至关重要。它使我们能够探索设备的功能，这有点像第一次去商场并查看它的楼层目录。由于服务是特征的集合，并且每个特征都可以具有定义其属性的描述符，因此我们可以将每个服务视为商场中的楼层，将特征视为给定楼层上的单个商店，而将描述符视为特定的给定商店中的过道或部分。

Service -> Characteristics -> Descriptors

执行服务发现很简单。使用我们从成功的连接尝试中保存的 BluetoothGatt 对象，我们只需在其上调用 discoverServices() ：

gatt.discoverServices()

建议从主/UI 线程调用 discoverServices() 以防止罕见的线程问题导致死锁情况，即应用程序可能会等待以某种方式被丢弃的 onServicesDiscovered() 回调。

在调用 discoverServices() 之后，服务发现的结果将通过 BluetoothGattCallback 的 onServicesDiscovered() 方法传递，我们应该在我们的 BluetoothGattCallback 回调主体中覆盖该方法以确保我们收到事件通知。在传递 onServicesDiscovered() 回调后，我们可以使用 BluetoothGatt 的 getServices() 方法安全地浏览可用的服务和特征。

蓝牙官方网站将 GATT Service 和Characteristics 的 UUID 列为 16 位（2 字节）值，例如0x180F 用于电池服务。如果我们尝试仅使用 16 位值创建 java.util.UUID 的实例：

val batteryServiceUuid = UUID.fromString("180F")

当 batteryServiceUuid 被实例化时，我们很快就会看到崩溃：

java.lang.IllegalArgumentException: Invalid UUID string: 180F

在 Android 上处理 UUID 时，重要的是要知道 16 位值应该与 00000000-0000-1000-8000-00805f9b34fb 的基本 UUID 一起使用，正如服务上的详细说明（我们认为相当模糊）官方蓝牙网站的发现页面，因此电池服务的完全合格的 UUID 是：

val batteryServiceUuid = UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb")

请求更大的 ATT MTU:

在我们深入执行读写操作之前，我们应该确保我们发送或接收的数据能够很好地适应连接的 ATT 最大传输单元 (MTU)。 ATT 数据包的最大长度由 ATT MTU 确定，通常作为连接过程的一部分在 Android 和 BLE 设备之间协商。

BLE 连接开始时协商的 ATT MTU 在 Android 设备和 BLE 设备的组合之间是不同的，因此不要对此做出任何假设是谨慎的。但是，如果我们打算与 BLE 设备交换更大的数据块（通常超过 20 字节的 Android 应用程序数据），我们应该请求更大的 ATT MTU，以便每次传输可以交换更多信息。

这可以在 Android 端使用 BluetoothGatt 的 requestMtu() 方法轻松完成。一个可行的策略是在连接到设备时尝试最大可能的 ATT MTU 值，Android 源代码中的 gatt\_api.h 头文件显示，Android 可以请求的最大可能 MTU 大小由常量 GATT\_MAX\_MTU\_SIZE 详细说明是517.

在请求某个 ATT MTU 值之后，应该传递一个回调 onMtuChanged()（不出所料，BluetoothGattCallback 的一部分），通知应用程序请求是否成功，以及当前的 ATT MTU 是什么。获得与最初请求不同的最终 ATT MTU 值是很常见的，因为必须在 Android 和 BLE 设备的固件之间协商请求。

override fun onMtuChanged(gatt: BluetoothGatt, mtu: Int, status: Int) {

Timber.w("ATT MTU changed to $mtu, success: ${status == BluetoothGatt.GATT\_SUCCESS}")

}

不幸的是，我们还看到 onMtuChanged() 回调在使用闭源固件时有时不会被传递，如果应用程序在继续执行某些操作之前依赖于传递的回调，这可能会阻碍事情的发展。我们的建议是始终假设最坏的情况——ATT MTU 的最小值为 23——并在使用闭源固件时围绕它进行计划，任何成功的 onMtuChanged() 调用都被视为额外的奖励。

在不支持更高 ATT MTU 的 BLE 设备固件上，每次 requestMtu(...) 调用后协商的 ATT MTU 可能会保持不变，最小值为 23。考虑到大多数读取、写入的 ATT 标头的 3 个字节，以及指示/通知操作（操作码和属性句柄），在这种情况下我们可以发送的最大应用程序字节数是 20 字节。如果我们的应用执行超过连接的 ATT MTU 的写入，则写入将失败，并且我们的 onCharacteristicWrite(...) 回调将呈现 GATT\_INVALID\_ATTRIBUTE\_LENGTH 的状态参数。

BLE读写：

解析 BluetoothGattCharacteristic 属性：

对 BluetoothGattCharacteristic 执行读取或写入操作之前，我们需要确保它们是可读或可写的。否则，操作将失败并出现 GATT\_READ\_NOT\_PERMITTED 或 GATT\_WRITE\_NOT\_PERMITTED 错误（这两个常量都是 BluetoothGatt 的一部分）。

BluetoothGattCharacteristic 包含一个 getProperties() 方法，该方法是其属性的位掩码，由 BluetoothGattCharacteristic.PROPERTY\_\* 常量表示。然后，我们可以在 getProperties() 和给定的 PROPERTY\_\* 常量之间执行按位与运算，以确定特征上是否存在某个属性。

我们关心的常量是 PROPERTY\_READ、PROPERTY\_WRITE 和 PROPERTY\_WRITE\_NO\_RESPONSE。我们将使用以下扩展函数来确定特征是否可读、可写或两者兼而有之：

fun BluetoothGattCharacteristic.isReadable(): Boolean =

containsProperty(BluetoothGattCharacteristic.PROPERTY\_READ)

un BluetoothGattCharacteristic.isWritable(): Boolean =

containsProperty(BluetoothGattCharacteristic.PROPERTY\_WRITE)

fun BluetoothGattCharacteristic.isWritableWithoutResponse(): Boolean =

containsProperty(BluetoothGattCharacteristic.PROPERTY\_WRITE\_NO\_RESPONSE)

fun BluetoothGattCharacteristic.containsProperty(property: Int): Boolean {

return properties and property != 0

}

从Characteristic中读取：

它允许我们利用 BLE 设备必须提供的信息，这符合我们对Service（BLE 外围设备）执行一些工作并向Client（BLE 中心）提供信息的Service-Client交互的期望。例如，测量环境读数的物联网传感器也可以将其电池电量作为电池服务的一部分提供，其中包含我们可以读取的电池电量特征。

在对可读Characteristic执行读取后，我们将看到一个字节数组（Kotlin 中的 ByteArray），它表示执行读取期间给定Characteristic的值。

当使用具有由蓝牙 SIG 标准化的特殊 UUID（分配编号）的Characteristic时，我们可以在这里通读与每个Characteristic相关的 XML，以了解如何理解我们在阅读后将遇到的原始字节。对于自定义和专有服务或特性，应解析字节的格式将取决于 BLE 固件对某种数据传输协议的实现，其细节可能会或可能不会公开。

BluetoothGatt 对象现在包含所有发现的Service和Characteristic，可以通过 getService() 和随后的 getCharacteristics() 访问。另外，请记住，作为蓝牙规范一部分的 GATT 服务和特性的 UUID 需要与基本 UUID 00000000-0000-1000-8000-00805f9b34fb 组合后才能使用。有了这些知识，我们可以通过执行以下操作来读取 BLE 设备的电池电量特性：

private fun readBatteryLevel() {

val batteryServiceUuid = UUID.fromString("0000180f-0000-1000-8000-00805f9b34fb")

val batteryLevelCharUuid = UUID.fromString("00002a19-0000-1000-8000-00805f9b34fb")

val batteryLevelChar = gatt

.getService(batteryServiceUuid)?.getCharacteristic(batteryLevelChar)

if (batteryLevelChar?.isReadable() == true) {

gatt.readCharacteristic(batteryLevelChar)

}

}

在调用 BluetoothGatt 的 readCharacteristic() 后不久，无论读取操作是否成功，都应将回调传递到您的 BluetoothGattCallback 的 onCharacteristicRead()。

override fun onCharacteristicRead(

gatt: BluetoothGatt,

characteristic: BluetoothGattCharacteristic,

status: Int

) {

with(characteristic) {

when (status) {

BluetoothGatt.GATT\_SUCCESS -> {

Log.i("BluetoothGattCallback", "Read characteristic $uuid:\n${value.toHexString()}")

}

BluetoothGatt.GATT\_READ\_NOT\_PERMITTED -> {

Log.e("BluetoothGattCallback", "Read not permitted for $uuid!")

}

else -> {

Log.e("BluetoothGattCallback", "Characteristic read failed for $uuid, error: $status")

}

}

}

}

在Characteristic读取操作的情况下，如果状态参数不是 GATT\_SUCCESS，则很可能是 GATT\_READ\_NOT\_PERMITTED，当我们在尝试执行读取之前检查Characteristic是否包含 PROPERTY\_READ 标志时，我们的代码有望缓解这种情况。如果读取成功，则可以通过调用 BluetoothGattCharacteristic 的 getValue() 方法（或在 Kotlin 中通过简单地使用 value 属性）来访问读取的数据。

大多数 BLE 固件使用 little-endian 字节顺序，因此确保以正确的顺序解析字节非常重要。例如，给定一个大小为 4（32 字节）的小端字节数组，最终的 Int 值可以这样计算：

val fourBytes: ByteArray // ... obtained from onCharacteristicRead()

val numericalValue = (fourBytes[3].toInt() and 0xFF shl 24) +

(fourBytes[2].toInt() and 0xFF shl 16) +

(fourBytes[1].toInt() and 0xFF shl 8) +

(fourBytes[0].toInt() and 0xFF)

写入BluetoothGattCharacteristic ：

有两种主要的写入类型，支持写入操作的特性通常支持其中一种写入类型，但也可以同时支持两种写入类型：

1. 写有回应。也称为写入请求，这种写入类型需要 BLE 设备固件确认它已处理写入或在写入失败时发送错误数据包。由 WRITE\_TYPE\_DEFAULT 表示。
2. 写无回应。也称为写入命令，这种写入类型不需要 BLE 设备的确认。由于缺乏带宽或处理能力来处理它，写入被忽略或丢弃是可能的。由 WRITE\_TYPE\_NO\_RESPONSE 表示。

大多数简单的 BLE 用例只使用响应写入，因为吞吐量通常不是这些用例的优先级，并且通常最好通知您的写入状态而不是在黑暗中。

保证使用无响应写入的用例通常是高吞吐量或低延迟短突发用例。例如，具有从应用程序接收大文件的能力的 BLE 设备可能希望使用高 MTU、数据长度扩展（请参阅我们关于 DLE 的博客文章）和某种重试机制的组合使用无响应写入具有 CRC 支持，以防 ATT 有效负载被丢弃。连接间隔、从属延迟和监督超时等连接参数也可能需要调整以针对此类用例进行优化，从而使无响应写入更适合团队（或个人）可以访问的高级系统实现移动和BLE固件源代码。

与读取特性类似，在执行写入操作之前，我们应该确保要写入的特性支持我们打算使用的写入类型。

BluetoothGatt 的 writeCharacteristic() 方法将我们正在写入的 BluetoothGattCharacteristic 作为其唯一参数。那么，我们如何指定写入类型和我们正在写入的有效负载？答案是我们必须使用 setWriteType() 和 setValue() 在 BluetoothGattCharacteristic 本身上设置这些字段。

fun writeCharacteristic(characteristic: BluetoothGattCharacteristic, payload: ByteArray) {

val writeType = when {

characteristic.isWritable() -> BluetoothGattCharacteristic.WRITE\_TYPE\_DEFAULT

characteristic.isWritableWithoutResponse() -> {

BluetoothGattCharacteristic.WRITE\_TYPE\_NO\_RESPONSE

}

else -> error("Characteristic ${characteristic.uuid} cannot be written to")

}

bluetoothGatt?.let { gatt ->

characteristic.writeType = writeType

characteristic.value = payload

gatt.writeCharacteristic(characteristic)

} ?: error("Not connected to a BLE device!")

}

此函数根据特征的属性选择最合适的写入类型。请注意，如果特性碰巧支持两种写入类型，我们如何优先使用带响应的写入 - 如果不带响应的写入是该特性支持的唯一写入类型，则只会将其用作写入操作的写入类型。

根据写入类型，您可能会或可能不会收到 BluetoothGattCallback 的 onCharacteristicWrite() 回调。对于 WRITE\_TYPE\_DEFAULT（写入响应），此回调是有保证的，但对于 WRITE\_TYPE\_NO\_RESPONSE（不响应写入），我们希望回调不会被传递，它仍然会被传递——至少对于我们一直在测试的 Pixel 设备。由于此行为的未记录性质，如果特性支持两种写入类型，我们仍然建议您使用 WRITE\_TYPE\_DEFAULT 而不是 WRITE\_TYPE\_NO\_RESPONSE。

我们的 onCharacteristicWrite() 回调需要在处理成功案例之前检查状态是否为 GATT\_SUCCESS。如果应用程序尝试写入大于连接当前 ATT MTU 的字节数，则可能会获得 GATT\_INVALID\_ATTRIBUTE\_LENGTH 状态，因此我们应该适当地处理这个问题.最后，我们还应该检查 GATT\_WRITE\_NOT\_PERMITTED，我们之前的代码片段应该可以缓解这种情况，因为我们要确保在执行写入之前可以写入特征。

读写Descriptor：

Descriptor嵌套在Characteristic下，类似于Characteristic嵌套在Service下的方式。Descriptor可以预见地由 BluetoothGattDescriptor 对象表示，我们可以通过 getPermissions() 检查它的权限，就像 BluetoothGattCharacteristic 的 getProperties()。

我们强烈建议在对其执行读取或写入操作时不要检查 BluetoothGattDescriptor 权限。根据我们的经验，权限标志通常不能反映Descriptor的实际可读性或可写性。

例如，我们将使用的两个最常见的Descriptor是Client Characteristic Configuration Descriptor (CCCD) 和Characteristic User Description Descriptor (CUDD)。 CCCD 可以写入和读取，因为它控制是否为某个Characteristic 启用通知或指示。 CUDD 为自定义Characteristic 提供了一个人类可读的名称，因此是可读的。但是，Android SDK 的 getPermissions() 方法在检查这些Characteristic 的可读性和可写性时总是返回 false。

fun BluetoothGattDescriptor.isReadable(): Boolean =

containsPermission(BluetoothGattDescriptor.PERMISSION\_READ) ||

containsPermission(BluetoothGattDescriptor.PERMISSION\_READ\_ENCRYPTED) ||

containsPermission(BluetoothGattDescriptor.PERMISSION\_READ\_ENCRYPTED\_MITM)

fun BluetoothGattDescriptor.isWritable(): Boolean =

containsPermission(BluetoothGattDescriptor.PERMISSION\_WRITE)|| containsPermission(BluetoothGattDescriptor.PERMISSION\_WRITE\_ENCRYPTED) ||

containsPermission(BluetoothGattDescriptor.PERMISSION\_WRITE\_ENCRYPTED\_MITM) || containsPermission(BluetoothGattDescriptor.PERMISSION\_WRITE\_SIGNED) ||

containsPermission(BluetoothGattDescriptor.PERMISSION\_WRITE\_SIGNED\_MITM)

fun BluetoothGattDescriptor.containsPermission(permission: Int): Boolean =

permissions and permission != 0

如何对Descriptor 执行读取和写入的其余部分与对Characteristic 执行的操作非常相似——使用类似命名的方法——唯一的例外是不需要为Descriptor 写入指定写入类型。



在 Android 上执行任何 BLE 操作时，我们强烈建议您等待回调进入，然后再执行下一个操作。在执行读取和写入的上下文中，这意味着在继续其他操作之前等待 onCharacteristicRead() 和 onCharacteristicWrite() 回调。

订阅notifications或indications：

notifications和indications是 BLE 设备（外围设备）在Characteristic 发生更改时通知 Android 设备（中央设备）的一种方式，实际上是外围设备与中央设备通信的主要机制。

它有助于将notifications和indications分别视为没有响应和有响应的写入的镜像版本，现在只有数据来自Service（外围）而不是Client（中央）。从移动开发的角度来看，notifications和indications之间的唯一区别是notifications在收到数据包时不需要客户端确认，而indications确实需要这种确认，因此速度较慢。然而，这个确认是由 Android 蓝牙堆栈处理的，所以我们的应用程序在两者之间没有什么特别的需要做的。

就像我们在确保可以写入或读取Characteristic 时检查的其他属性一样，我们可以使用 BluetoothGattCharacteristic 的 PROPERTY\_INDICATE 和 PROPERTY\_NOTIFY 来确保Characteristic 支持在订阅其值更改之前发送notifications或indications。

fun BluetoothGattCharacteristic.isIndicatable(): Boolean =

containsProperty(BluetoothGattCharacteristic.PROPERTY\_INDICATE)

fun BluetoothGattCharacteristic.isNotifiable(): Boolean =

containsProperty(BluetoothGattCharacteristic.PROPERTY\_NOTIFY)

fun BluetoothGattCharacteristic.containsProperty(property: Int): Boolean =

properties and property != 0

一旦我们确定某个Characteristic 支持notifications或indications（或两者），就需要通知 Android 蓝牙堆栈应用打算选择加入或退出有关Characteristic 更改的通知。我们通过在我们的 BluetoothGatt 对象上调用 setCharacteristicNotification() 并确保它返回 true 来做到这一点。

CCCD（客户端特征配置描述符）特殊描述符的名称很难记住，但它对于订阅通知或指示的过程至关重要。任何支持发送通知或指示的特性都将具有此描述符。写入 CCCD 可启用或禁用 BLE 设备上的notifications或indications。

在与 CCCD 交互时，应该期望这个特殊的描述符可以被写入和读取，即使它不包含来自 BluetoothGattDescriptor 的 getPermissions() 的 PERMISSION\_READ 和 PERMISSION\_WRITE 权限。

根据我们是否启用notifications或indications，我们需要将 CCCD 的值设置为 ENABLE\_INDICATION\_VALUE 或 ENABLE\_NOTIFICATION\_VALUE（均在 BluetoothGattDescriptor 中定义），并将该值写入 BLE 设备上的远程描述符。要禁用通知、指示或两者，我们只需将 DISABLE\_NOTIFICATION\_VALUE 写入 CCCD。

我们正在短路属性检查描述符的 UUID 是否与 CCCD UUID 匹配：

fun writeDescriptor(descriptor: BluetoothGattDescriptor, payload: ByteArray) {

bluetoothGatt?.let { gatt ->

descriptor.value = payload

gatt.writeDescriptor(descriptor)

} ?: error("Not connected to a BLE device!")

}

下面的代码片段显示了我们通常如何实现启用和禁用通知或指示的功能。我们试图使我们的函数尽可能易于使用，这意味着该函数不接受指定我们是否要启用通知或指示的参数。它试图通过检查特征的属性来找出我们启用的适当通知类型。

fun enableNotifications(characteristic: BluetoothGattCharacteristic) {

val cccdUuid = UUID.fromString(CCC\_DESCRIPTOR\_UUID)

val payload = when {

characteristic.isIndicatable() -> BluetoothGattDescriptor.ENABLE\_INDICATION\_VALUE

characteristic.isNotifiable() -> BluetoothGattDescriptor.ENABLE\_NOTIFICATION\_VALUE

else -> {

Log.e("ConnectionManager", "${characteristic.uuid} doesn't support notifications/indications")

return

}

}

characteristic.getDescriptor(cccdUuid)?.let { cccDescriptor ->

if (bluetoothGatt?.setCharacteristicNotification(characteristic, true) == false) {

Log.e("ConnectionManager", "setCharacteristicNotification failed for ${characteristic.uuid}")

return

}

writeDescriptor(cccDescriptor, payload)

} ?: Log.e("ConnectionManager", "${characteristic.uuid} doesn't contain the CCC descriptor!")

}

fun disableNotifications(characteristic: BluetoothGattCharacteristic) {

if (!characteristic.isNotifiable() && !characteristic.isIndicatable()) {

Log.e("ConnectionManager", "${characteristic.uuid} doesn't support indications/notifications")

return

}

val cccdUuid = UUID.fromString(CCC\_DESCRIPTOR\_UUID)

characteristic.getDescriptor(cccdUuid)?.let { cccDescriptor ->

if (bluetoothGatt?.setCharacteristicNotification(characteristic, false) == false) {

Log.e("ConnectionManager", "setCharacteristicNotification failed for ${characteristic.uuid}")

return

}

writeDescriptor(cccDescriptor, BluetoothGattDescriptor.DISABLE\_NOTIFICATION\_VALUE)

} ?: Log.e("ConnectionManager", "${characteristic.uuid} doesn't contain the CCC descriptor!")

}

检查 BluetoothGattCallback 的 onDescriptorWrite() 回调中的 GATT\_SUCCESS 状态参数，然后才能断定通知或指示的启用或禁用是否成功。

一旦我们在特性上启用了通知或指示，任何传入的通知或指示都会通过 BluetoothGattCallback 的 onCharacteristicChanged() 回调传递：

override fun onCharacteristicChanged(

gatt: BluetoothGatt,

characteristic: BluetoothGattCharacteristic

) {

with(characteristic) {

Log.i("BluetoothGattCallback", "Characteristic $uuid changed | value: ${value.toHexString()}")

}

}

值得注意的是，我们通常设置通知或指示的流程是我们连接设置过程的一部分，通常如下所示：

1.扫描设备。

2.连接到设备。

3.[可选] 请求最大 ATT MTU。

4.发现服务（和特征）。

5.启用有关感兴趣特征的通知或指示。

6.向更高级别的调用者（通常是 UI 或 ViewModel 层）发出信号，表明连接设置已完成，BLE 设备已准备好与之交互（即现在可以执行读取和写入操作）。

绑定BLE：

配对是临时加密密钥的交换，允许交换长期加密密钥——绑定过程——发生。

想象一下，当你在公共场所时，你想和朋友秘密握手，但你不想让别人看到，所以你搭了一个帐篷，在你们俩都在帐篷里的时候秘密握手.搭建帐篷的过程将与配对同义，而绑定则由在帐篷内交换秘密握手表示（配对过程建立的临时加密通道）。

作为 Android 开发者，我们唯一关心的就是绑定 BLE 设备；双方的蓝牙堆栈为我们处理配对过程。

与智能恒温器或其他物联网设备等简单设备交互时，通常不需要绑定，因为它们提供了应用程序可以使用的信息。但创建绑定背后的主要动机是在 BLE 中央和外围设备之间建立加密通信通道。

有数据包分析器可以“嗅探” BLE 数据包传输，因为它们发生在空中，任何未加密的流量都是公平的游戏。想象一个怀有恶意的人坐在咖啡店里，分析正在空中传输的 BLE 数据包。例如，如果某人的智能手表没有使用 BLE 绑定来加密其与智能手机的通信通道，那么在智能手表和用户的智能手机之间同步时，任何短信或联系人都会被暴露。

根据系统设计，绑定请求可以来自Client或Service设备。有多种方法可以启动绑定过程，这里按我们偏好的降序排列：

1.安卓发起。当 BLE 设备由于未经授权的 ATT 请求而发出身份验证不足错误时，让 Android 自动启动绑定过程。

2.开发者发起。通过调用 BluetoothDevice 的 createBond() 方法主动启动绑定过程。

3.Service发起。如果可以访问 BLE 设备上运行的固件的源代码，可以让它主动发送一个安全请求，一旦它连接到Client，它将启动绑定过程。

第一种方法涉及 Android 自行启动绑定，开发人员不需要任何操作。这通常发生在由于 Android 尝试执行未经授权的 ATT 请求而收到 Insufficient Authentication 错误代码时——例如读取在Service设备上加密的特征值。

对于大多数 Android 设备而言，使用此方法启动绑定非常可靠。以这种方式触发绑定过程也恰好是 Apple 在其附件设计指南文档中推荐的技术，并且实际上是在 iOS 设备的幕后发生的事情，因为开发人员没有明确的方法来启动 iOS 上的绑定过程。

为了在移动平台之间实现平等，我们建议同时控制其应用程序和 BLE 设备固件的读者尽可能使用此技术，方法是包含应用程序随后在连接过程中读取的加密特征（在事先执行服务发现）。

对于我们测试过的大多数 Android 设备，这足以让 Android 代表我们自动启动与 BLE 设备的绑定过程。与其他两种方法相比，以这种方式做事似乎也有更高的成功机会。

第二种方法，由于它实际上是一个公共 API，它可能比第一种更知名，我们看到与 iOS 不同，Android SDK 提供了一个公共 createBond() 方法作为 BluetoothDevice 对象的一部分我们可以使用它来主动让 Android 启动与外围设备的绑定过程。当您导航到 Android 的蓝牙设置并点击附近正在投放广告的设备以与之绑定时，这就是所谓的幕后花絮。

一些说明过时的网站可能会建议您使用 Java 反射来访问 createBond() 方法的私有重载，但正如我们的 Android BLE 开发技巧博客文章中所述，Google 已开始限制私有 API 的使用并使用私有 API不再是我们建议客户做的事情，即使讨论的私有 API 尚未在灰名单上。

当前面提到的第一种方法失败时，这种启动绑定过程的 createBond() 方法很有帮助。我们看到小米和三星的某些设备在使用第一种方法时不会启动绑定，而 createBond() 对这些设备的效果很好。

createBond() 调用的时机非常重要。如果 BLE 设备上运行的固件不希望绑定，或者它根本不支持绑定，则它可能会拒绝绑定请求。我们还看到了 createBond() 调用的成功概率取决于 OEM 和 Android 版本的情况，以及在发出绑定请求时应用程序是否已经连接到 BLE 设备的情况。

我们的建议是在尝试使用 createBond() 之前尝试第一种 Android 发起的绑定方法。如果您的 Android 设备上的 createBond() 也始终失败，请先尝试连接到 BLE 设备，然后再调用 createBond() 以查看是否有帮助，或者如果您在 createBond() 失败时已经连接，则在 connectGatt() 之前调用 createBond()。

我们很少建议外围设备启动绑定过程。但是，如果上述两种方法都失败了，并且您可以访问在 BLE 设备上运行的固件的源代码，您可以让 BLE 固件主动发送安全请求，这将在您的时间启动绑定过程选择，通常只要它连接到中央。但是，请注意：我们在某些 OEM 的 Android 设备上看到了一些奇怪的行为，这些行为会直接拒绝这个绑定请求。我们还没有看到太多以这种方式结合的成功案例，但这是你可以尝试的最后手段。

绑定状态更改更新通过具有 ACTION\_BOND\_STATE\_CHANGED 操作的广播传递，并使用广播接收器进行处理。绑定状态更改不是通过 BluetoothGattCallback 传递的，因为应用程序不一定需要连接到 BLE 设备才能更改其绑定状态。但是，由于广播的多播性质，您还会收到有关您可能不感兴趣的其他设备的绑定状态更改的通知，因此在执行任何操作之前检查广播所针对的蓝牙设备的地址非常重要。

广播将始终包含以下附加内容：

1.EXTRA\_DEVICE。包含绑定状态已更改的 BluetoothDevice。大多数应用程序都希望检查设备的地址属性，以确保传递的广播涉及感兴趣的设备。

2.EXTRA\_BOND\_STATE。表示 BluetoothDevice 的当前绑定状态。

3.EXTRA\_PREVIOUS\_BOND\_STATE。表示 BluetoothDevice 的先前绑定状态。基本上，BluetoothDevice 正在从这个状态转换到 EXTRA\_BOND\_STATE 中表示的状态。

虽然 EXTRA\_DEVICE 是 Parcelable 额外字段，但 EXTRA\_BOND\_STATE 和 EXTRA\_PREVIOUS\_BOND\_STATE 是 Int 额外字段，可能值为 BOND\_NONE、BOND\_BONDING、BOND\_BONDED。通过检查两个 Int 额外字段，我们可以了解给定 BluetoothDevice 的键状态转换，如下所示：

fun listenToBondStateChanges(context: Context) {

context.applicationContext.registerReceiver(

broadcastReceiver,

IntentFilter(BluetoothDevice.ACTION\_BOND\_STATE\_CHANGED)

)

}

private val broadcastReceiver = object : BroadcastReceiver() {

override fun onReceive(context: Context, intent: Intent) {

with(intent) {

if (action == BluetoothDevice.ACTION\_BOND\_STATE\_CHANGED) {

val device = getParcelableExtra<BluetoothDevice>(BluetoothDevice.EXTRA\_DEVICE)

val previousBondState = getIntExtra(BluetoothDevice.EXTRA\_PREVIOUS\_BOND\_STATE, -1)

val bondState = getIntExtra(BluetoothDevice.EXTRA\_BOND\_STATE, -1)

val bondTransition = "${previousBondState.toBondStateDescription()} to " +

bondState.toBondStateDescription()

Log.w("Bond state change", "${device?.address} bond state changed | $bondTransition")

}

}

}

private fun Int.toBondStateDescription() = when(this) {

BluetoothDevice.BOND\_BONDED -> "BONDED"

BluetoothDevice.BOND\_BONDING -> "BONDING"

BluetoothDevice.BOND\_NONE -> "NOT BONDED"

else -> "ERROR: $this"

}

}

不幸的是，没有公共方法可以删除与 BluetoothDevice 的绑定。虽然可以使用 Java 反射调用 BluetoothDevice 中的私有 removeBond() 方法，但我们不再建议您使用 Google 对非 SDK 接口的最新立场（一种更奇特的私有 API 说法）——尽管你您可以自行承担风险，因为在撰写本文时 removeBond() 尚未出现在 Android 的私有 API 灰名单中。

除了编程方式，您可以通过指示用户导航到蓝牙设置并从那里忘记绑定设备来实现相同的效果。

最佳实践：

在 BLE 应用程序开发方面，以快速方式连续执行 BLE 操作是 Android 平台出现意外行为的最大原因。如果您一直在遵循并实现自己的特征和描述符的读写函数，并且您打算将这些操作链接起来，您会很快意识到，通常只有第一个操作会成功，而所有其他操作似乎都陷入了一个黑洞，再也听不见。

所有 BLE 操作（描述符读/写、特征读/写、连接、断开连接、执行 MTU 更新请求、服务发现等）都必须在理想情况下等待之前的 BLE 操作在执行之前先成功或失败。

对于 Android 上的大多数 BLE 用例，实现基本的队列机制可以解决与并发相关的 BLE 问题的许多痛点。

在后台保持连接（非绑定用例）：

对于未绑定设备，Android 上 BLE 的默认行为是应用拥有与 BLE 设备的连接。相比之下，Android 操作系统拥有绑定设备的连接。这意味着对于非绑定用例，如果您的应用程序由于资源限制在后台被操作系统终止，或者如果您的用户将您的应用程序滑开（有效地立即终止它），BLE 连接将丢失，应该会看到大多数 BLE 设备再次开始广告。

尝试使您的应用程序进程尽可能长时间保持活动状态的最直接方法是使用前台服务。前台服务本质上是一个在通知抽屉中以持久横幅运行的服务，让用户了解您的应用程序正在后台执行某些操作。由于内存限制而被系统终止的概率非常低，但并非不可能。

值得注意的是，当用户在前台服务运行时滑动您的应用程序时，您的应用程序进程仍然存在，但您的 Activity 堆栈和任何其他面向用户的元素都会被吹走。因此，必须格外小心以确保您的 BLE 逻辑位于这些实体之外，就像在一个名为 ConnectionManager 的单例中处理应用程序的所有 BLE 需求的情况一样。

通常，您的活动和片段不应该对 BLE 连接的状态做出任何假设。相反，他们应该依赖 ConnectionManager（或任何管理应用程序 BLE 需求的实体）作为唯一的事实来源，并根据 UI 的需要在 onCreate() 和 onResume() 中相应地更新 UI。

使用前台服务也不是完全安全的，因为 Android P 及更高版本具有自适应电池功能，有时会限制运行前台服务的应用程序进程的正常运行时间，但根据我们的经验，这似乎是随机发生的。到目前为止，解决此限制的唯一方法是要求您的用户关闭应用程序的电池优化，这是隐藏在“设置”应用程序中的一个选项。

其他 Android 后台处理技术，如依赖 AlarmManager 和 WorkManager 也是可行的，但连接事件不会是即时的，并且可能会丢失一些 BLE 通知或指示，如果这些事件发生时应用程序碰巧没有运行。长话短说，如果您的应用程序不需要在事件发生时知道事件，这些其他技术很好，而是可以定期唤醒，再次连接到 BLE 设备（如果它在附近），并查询它是否有新的数据。

三、MTU

MTU: 最大传输单元（MAXIMUM TRANSMISSION UNIT）， 指在一个PDU （Protocol Data Unit： 协议数据单元，在一个传输单元中的有效传输数据）能够传输的最大数据量（多少字节可以一次性传输到对方）。

MTU 交换是为了在主从双方设置一个PDU中最大能够交换的数据量，通过MTU的交换和双方确认（注意这个MTU是不可以协商的，只是通知对方，双方在知道对方的极限后会选择一个较小的值作为以后的MTU，比如说，主设备发出一个150个字节的MTU请求，但是从设备回应MTU是23字节，那么今后双方要以较小的值23字节作为以后的MTU），主从双方约定每次在做数据传输时不超过这个最大数据单元。

MTU交换通常发生在主从双方建立连接关系后。

以太网：1500

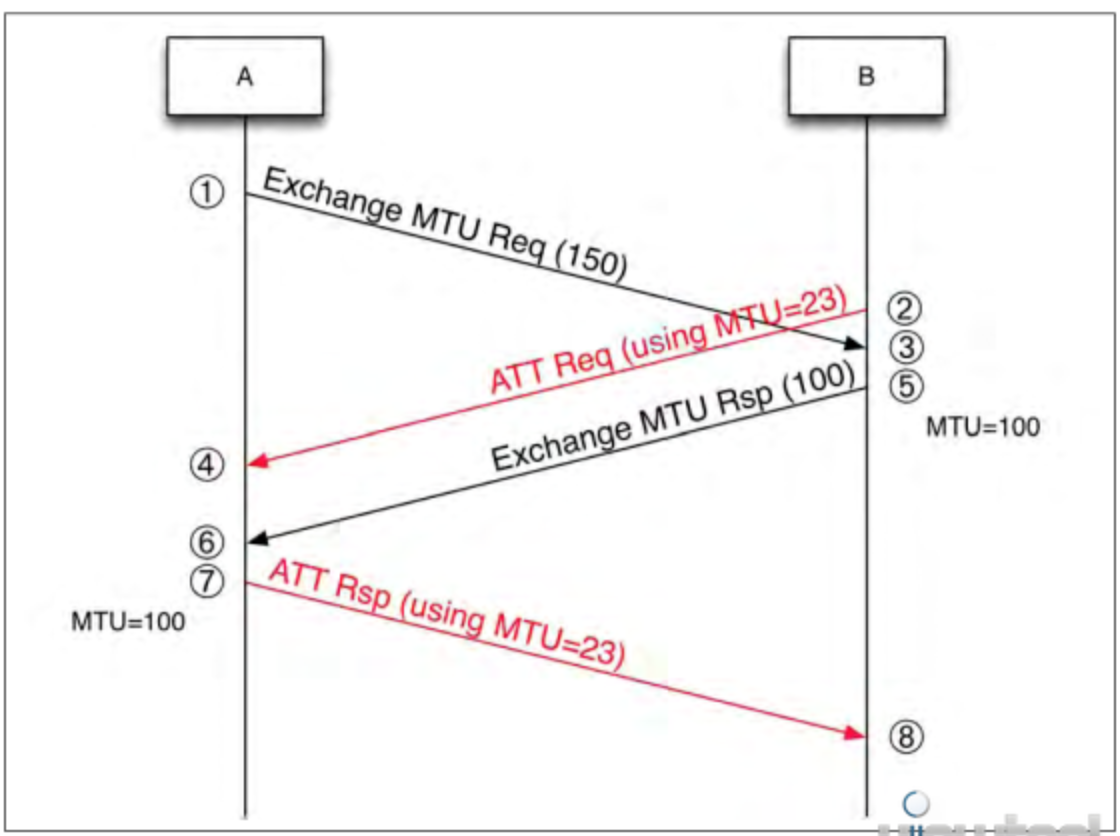
IEEE 802.3/802.2:　1492

X.25: 576

BLE: 23 => 这就是为什么WIFI 可以用于传输视频，传统蓝牙(BT)可以传输音频，而低功耗蓝牙（BTLE 或者BLE）只能够传输控制数据的原因了。

MTU 交换命令：属于ATT 命令

MTU 交换过程：如下图



MTU 两个命令（“MTU 请求”及“MTU 响应”）