蓝牙涉及key

*CoreBluetooth1* 初识
*CoreBluetooth2* 作为 *Central* 时的数据读写
*CoreBluetooth3* 作为 *Central* 时的数据读写（补充）
*CoreBluetooth4* 作为 *Central* 时的数据读写（最佳实践）
*CoreBluetooth5* 作为 *Central* 时的数据读写（*OTA* 固件升级与文件传输）
*CoreBluetooth6* 作为 *Peripheral* 时的请求响应
*CoreBluetooth7* 作为 *Peripheral* 时的请求响应（最佳实践）

对于 iOS app 来说，知道现在是运行在前台和后台是至关重要的。因为当程序挂起后，对资源的使用是相当有限的。关于多任务的介绍，可以看 app 开发手册。

默认情况下，Core Bluetooth 是不会在后台运行的（无论是 central 还是 peripheral）。但你也可以配置在 app 收到事件后，从挂起状态唤醒。即使程序不是完全的支持后台模式，也可以要求在有重要事件时接收系统通知。

即使在以上两种情况下（完全允许后台和部分允许后台），程序也有可能不会永远挂起。在前台程序需要更多内存时，被挂起的程序很有可能会被强制退出，那样会断开所有的连接。从 iOS 7 开始，能够先保存状态（无论是 central 还是 peripheral），并在重新打开 app 时还原这些状态。通过这一特性，就可以做长时间操作了。

**运行在前台的 app （Foreground-Only）**

除非去申请后台权限，否则 app 都是只在前台运行的，程序在进入后台不久便会切换到挂起状态。挂起后，程序将无法再接收任何蓝牙事件。

对于 central 来说，挂起将无法再进行扫描和搜索 peripheral。对于 peripheral 来说，将无法再发起广播，central 也无法再访问动态变化的

characteristic 数据，访问将返回 error。

根据不同情况，这种机制会影响程序在以下几个方面的运用。你正在读取 peripheral 的数据，结果程序被挂起了（可能是用户切换到了另外一个 app），此时连接会被断开，但是要直到程序重新唤醒时，你才知道被断开了。

**利用连接 Peripheral 时的选项**

Foreground-Only app 在挂起的时候，便会加入到系统的一个队列中，当程序重新唤醒时，系统便会通知程序。Core Bluetooth 会在程序中包含 central 时，给用户以提示。用户可根据提示来判断是否要唤醒该 app。

你可以利用 central 在连接 peripheral 时的方法 `connectPeripheral:options:` 中的 `options` 来触发提示：

- `CBConnectPeripheralOptionNotifyOnConnectionKey` —— 在连接成功后，程序被挂起，给出系统提示。
- `CBConnectPeripheralOptionNotifyOnDisconnectionKey` —— 在程序挂起，蓝牙连接断开时，给出系统提示。
- `CBConnectPeripheralOptionNotifyOnNotificationKey` —— 在程序挂起后，收到 peripheral 数据时，给出系统提示。

**Core Bluetooth 后台模式**

如果你想让你的 app 能在后台运行蓝牙，那么必须在 `info.plist` 中打开蓝牙的后台运行模式。当配置之后，收到相关事件便会从后台唤醒。这一机制对定期接收数据的 app 很有用，比如心率监测器。

下面会介绍两种后台模式，一种是作为 central 的，一种是作为 peripheral 的，如果 app 两种角色都有，那则需要开启两种模式。配置即是在 `info.plist` 中添加 `UIBackgroundModes` key，类型为数组，value 则根据你当前角色来选择：

- `bluetooth-central` —— 即 Central。

- `bluetooth-peripheral` —— 即 Peripheral。

这个配置在 *Xcode* 中，可以在 *Capabilities* 中进行配置，而不用直接面对 *key-value*。如果要看到 *key-value*，可以在 `info.plist` 中打开查看。

**作为 Central 的后台模式**

如果在 `info.plist` 中配置了 `UIBackgroundModes` – `bluetooth-central`，那么系统则允许程序在后台处理蓝牙相关事件。在程序进入后台后，依然能扫描、搜索 peripheral，并且还能进行数据交互。

当 `CBCentralManagerDelegate` 和 `CBPeripheralDelegate` 的代理方法被调用时，系统将会唤醒程序。此时允许你去处理重要的事件，比如：连接的建立或断开，peripheral 发送了数据，central manager 的状态改变。

虽然此时程序能在后台运行，但是对 peripheral 的扫描和在前台时是不一样的。实际情况是这样的：

- 设置的 `CBCentralManagerScanOptionAllowDuplicatesKey` 将失效，并将发现的多个 peripheral 广播的事件合并为一个。
- 如果全部的 app 都在后台搜索 peripheral，那么每次搜索的时间间隔会更大。这会导致搜索到 peripheral 的时间变长。

这些相应的调整会减少无线电使用，并提升续航能力。

**作为 peripheral 的后台模式**

作为 peripheral 时，如果需要支持后台模式，则在 `info.plist` 中配置 `UIBackgroundModes` – `bluetooth-peripheral`。配置后，系统会在有读写请求和订阅事件时，唤醒程序。

在后台，除了允许处理读写请求和订阅事件外，Core Bluetooth 框架还允许 peripheral 发出广播。同样，广播事件也有前后台区别。在后台发起时是这样的：

- `CBAdvertisementDataLocalNameKey` 将失效，在广播时，广

播数据将不再包含 peripheral 的名字。
- 被 `CBAdvertisementDataServiceUUIDsKey` 修饰的 UUID 数组将会被放到 overflow 区域中，意味着只能被明确标识了搜索 service UUID 的 iOS 设备找到。
- 如果所有 app 都在后台发起广播，那么发起频率会降低。

**巧妙的使用后台模式**

虽然程序支持一个或多个 Core Bluetooth 服务在后台运行，但也不要滥用。因为蓝牙服务会占用 iOS 设备的无线电资源，这也会间接影响到续航能力，所以尽可能少的去使用后台模式。app 会唤醒程序并处理相关事务，完成后又会快速回到挂起状态。

无论是 central 还是 peripheral，要支持后台模式都应该遵循以下几点：

- 程序应该提供 UI，让用户决定是否要在后台运行。
- 一旦程序在后台被唤醒，程序只有 10s 的时间来处理相关事务。所以应该在程序再次挂起前处理完事件。后台运行的太耗时的程序会被系统强制关闭进程。
- 处理无关的事件不应该唤醒程序。

和后台运行的更多介绍，可以查看 App Programming Guide for iOS。

**处理常驻后台任务**

某些 app 可能需要 Core Bluetooth 常驻后台，比如，一款用 BLE 技术和门锁通信的 app。当用户离开时，自动上锁，回来时，自动开锁（即使程序运行在后台）。当用户离开时，可能已超出蓝牙连接范围，所以没办法给锁通信。此时可以调用 `CBCentralManager` 的 `connectPeripheral:options:` 方法，因为该方法没有超时设置，所以，在用户返回时，可以重新连接到锁。

但是还有这样的情形：用户可能离开家好几天，并且在这期间，程序已经被完全退出了。那么用户再次回家时，就不能自动开锁。对于这类 app 来说，常驻后台操作就显得尤为重要。

## 状态保存与恢复

因为状态的保存和恢复 Core Bluetooth 都为我们封装好了，所以我们只需要选择是否需要这个特性即可。系统会保存当前 central manager 或 peripheral manager，并且继续执行蓝牙相关事件（及时程序已经不再运行）。一旦事件执行完毕，系统会在后台重启 app，这是你有机会去存储当前状态，并且处理一些事物。在之前提到的 “门锁” 的例子中，系统会监视连接请求，并在 `centralManager:didConnectPeripheral:` 回调时，重启 app，在用户回家后，连接操作结束。

Core Bluetooth 的状态保存于恢复在设备作为 central、peripheral 或者这两种角色时，都可用。在设备作为 central 并添加了状态保存与恢复支持后，如果 app 被强行关闭进程，系统会自动保存 central manager 的状态（如果 app 有多个 central manager，你可以选择哪一个需要系统保存）。对于 `CBCentralManager`，系统会保存以下信息：

- central 需要扫描的 service（包括扫描时，配置的 options）
- central 准备连接或已经连接的 peripheral
- central 订阅的 characteristic

对于 peripheral 来说，情况也差不多。系统对 `CBPeripheralManager` 的处理方式如下：

- peripheral 在广播的数据
- peripheral 存入的 service 和 characteristic 的树形结构
- 已经被 central 订阅了的 characteristic 的值

当系统在后台重新加载程序后（可能是因为找到了要找的 peripheral），你可以重新实例化 central manager 或 peripheral 并恢复他们的状态。接下来会详细介绍如何存储和恢复状态。

## 添加状态存储和恢复支持

状态的存储和恢复功能在 Core Bluetooth 中是可选的，添加支持可以通过以下几个步骤：

1. （必须）在初始化 central manager 或 peripheral manager 时，要选择是否需要支持。会在文后的【选择支持存储和恢复】中介绍。
2. （必须）在系统从后台重新加载程序时，重新初始化 central manager 或 peripheral manager。会在文后的【重新初始化 central manager 和 peripheral manager】中介绍。
3. （必须）实现恢复状态相关的代理方法。会在文后的【 实现恢复状态的代理方法】中介绍。
4. （可选）更新 central manager 或 peripheral manager 的初始化过程。会在文后的【更新 manager 初始化过程】中介绍。

**选择支持存储和恢复**

如果要支持存储和恢复，则需要在初始化 manager 的时候给一个 restoration identifier。restoration identifier 是 string 类型，并标识了 app 中的 central manager 或 peripheral manager。这个 string 很重要，它将会告诉 Core Bluetooth 需要存储状态，毕竟 Core Bluetooth 恢复有 identifier 的对象。

例如，在 central 端，要想支持该特性，可以在调用 `CBCentralManager` 的初始化方法时，配置 `CBCentralManagerOptionRestoreIdentifierKey`:

```
myCentralManager = [[CBCentralManager alloc]
initWithDelegate:self queue:nil

options:@{CBCentralManagerOptionRestoreIdentifierKey
: @"myCentralManagerIdentifier"}];
```

虽然以上代码没有展示出来，其实在 peripheral manager 中要设置 identifier 也是这样的。只是在初始化时，将 key 改成了 `CBPeripheralManagerOptionRestoreIdentifierKey`。

因为程序可以有多个 `CBCentralManager` 和 `CBPeripheralManager`，所以要确保每个 identifier 都是唯一的。

**重新初始化 central manager 和 peripheral manager**

当系统重新在后台加载程序时，首先需要做的即根据存储的 identifier，重新初始化 central manager 或 peripheral manager。如果你只有一个 manager，并且 manager 存在于 app 生命周期中，那这个步骤就不需要做什么了。

如果 app 中包含多个 manager，或者 manager 不是在整个 app 生命周期中都存在的，那 app 就必须要区分你要重新初始化哪个 manager 了。你可以通过从 app delegate 中的 `application:didFinishLaunchingWithOptions:` 中取出 key（`UIApplicationLaunchOptionsBluetoothCentralsKey` 或 `UIApplicationLaunchOptionsBluetoothPeripheralsKey`）中的 value（数组类型）来得到程序退出之前存储的 manager identifier 列表：

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    NSArray *centralManagerIdentifiers =
launchOptions[UIApplicationLaunchOptionsBluetoothCent
ralsKey];
    return YES;
}
```

拿到这个列表后，就可以通过循环来重新初始化所有的 manager 了。

### 实现恢复状态的代理方法

在重新初始化 manager 之后，接下来需要同步 Core Bluetooth 存储的他们的状态。要想弄清楚在程序被退出时都在做些什么，就需要正确的实现代理方法。对于 central manager 来说，需要实现 `centralManager:willRestoreState:`；对于 peripheral manager 来说，需要实现 `peripheralManager:willRestoreState:`。

注意：如果选择存储和恢复状态，当系统在后台重新加载程序时，首先调用的方法是 *`centralManager:willRestoreState:`* 或 *`peripheralManager:willRestoreState:`*。如果没有选择存储

的恢复状态（或者唤醒时没有什么内容需要恢复），那么首先调用的方法是 *centralManagerDidUpdateState:*
或 *peripheralManagerDidUpdateState:*。
无论是以上哪种代理方法，最后一个参数都是一个包含程序退出前状态的字典。字典中，可用的 key ，central 端有：

```
NSString *const
CBCentralManagerRestoredStatePeripheralsKey;
NSString *const
CBCentralManagerRestoredStateScanServicesKey;
NSString *const
CBCentralManagerRestoredStateScanOptionsKey;
```

peripheral 端有：

```
NSString *const
CBPeripheralManagerRestoredStateServicesKey;
NSString *const
CBPeripheralManagerRestoredStateAdvertisementDataKey;
```

要恢复 central manager 的状态，可以用 `centralManager:willRestoreState:` 返回字典中的 key 来得到。假如说 central manager 有想要或者已经连接的 peripheral，那么可以通过 `CBCentralManagerRestoredStatePeripheralsKey` 对应得到的 peripheral （`CBPeripheral` 对象）数组来得到。

```
- (void)centralManager:(CBCentralManager *)central
      willRestoreState:(NSDictionary *)state {

    NSArray *peripherals =
state[CBCentralManagerRestoredStatePeripheralsKey];
}
```

具体要对拿到的 peripheral 数组做什么就要根据需求来了。如果这是个 central manager 搜索到的 peripheral 数组，那就可以存储这个数组的引用，并且开始建立连接了（注意给这些 peripheral 设置代理，否则连接后不会走 peripheral 的代理方法）。

恢复 peripheral manager 的状态和 central manager 的方式类似，就只是把代理方法换成了 `peripheralManager:willRestoreState:`，并且使

用对应的 key 即可。

### 更新 manager 初始化过程

在实现了全部的必须步骤后，你可能想要更新 manager 的初始化过程。虽然这是个可选的操作，但是它对确保各种操作能正常进行尤为重要。假如，你的应用在 central 和 peripheral 做数据交互时，被强制退出了。即使 app 最后恢复状态时，找到了这个 peripheral，那你也不知道 central 和这个 peripheral 当时的具体状态。但其实我们在恢复时，是想恢复到程序被强制退出前的那一步。

这个需求，可以在代理方法 `centralManagerDidUpdateState:` 中，通过发现恢复的 peripheral 是否之前已经成功连接来实现：

```
NSUInteger serviceUUIDIndex = [peripheral.services
indexOfObjectPassingTest:^BOOL(CBService *obj,
NSUInteger index, BOOL *stop) {
    return [obj.UUID isEqual:myServiceUUIDString];
}];

if (serviceUUIDIndex == NSNotFound) {
    [peripheral
discoverServices:@[myServiceUUIDString]];
}
```

上面的代码描述了，当系统在完成搜索 service 之后才退出的程序，可以通过调用 `discoverServices:` 方法来恢复 peripheral 的数据。如果 app 成功搜索到 service，你可以是否能搜索到需要的 characteristic（或者已经订阅过）。通过更新初始化过程，可以确保在正确的时间点，调用正确的方法。

### 最后

到这里，对 Core Bluetooth 的理解就暂告一段落，如果有什么问题或建议，欢迎评论。

发布于

2016年3月23日