Gabeldorsche（叉尾鱼）

原文链接

Gabeldorsche：Android Bluetooth stack implementation

Gabeldorsche Architecture

开发Gabeldorsche（GD）蓝牙栈时所考虑的一些架构问题。

1. 线程模型（Threading model）

GD栈并不是基于线程的概念，它使用 Handlers 进行线程间通信。但是 GD 最终是在操作系统上运行的，因此在实现完全抽象之前仍然需要与进程和线程进行交互。

1.1 Processes

GD运行环境错在三种类型的进程：

- **Application processes**

  包括第三方应用程序、其他系统组件（如音频和电信服务），它们与蓝牙协议栈进行交互，使用 AIDL 或 Protobuf 等语言，处理通过各种 RPC/IPC 方法（如 Binder、Socket IPC、gRPC、DBUS 等）定义的 API。

  > **Application processes** : include third-party apps, other system components such as audio and telecom services that interact with the *Bluetooth stack process* APIs defined through various RPC/IPC methods such as Binder, Socket IPC, gRPC, DBUS. and so on, using languages such as AIDL or Protobuf. For Android applications, although APIs are defined in AIDL, some boiler plate code is wrapped in Java libraries exposed through code in `frameworks/base/core/java/android/bluetooth` that is released to developers as Android SDK.

- **Hardware abstraction layer（HAL） processes**

  > **Hardware abstraction layer (HAL) processes** : one or many processes from the vendor partition, and hence is hardware depenedent. They interact with the *Bluetooth stack process* via a set of hardware abstraction APIs defined through RPC/IPC methods such as Binder, Socket IPC, DBUS, and so on, using languages such as HIDL. On Android, this would

> be HAL processes that implement HIDL APIs such as IBluetoothHci and IBluetoothAudioProvider.

- **Bluetooth stack process**

  一个单一进程，它在 Host Controller Interface（HCI） 之上和 Bluetooth SDK API 之下实现各种蓝牙协议和配置文件。一方面，它为应用程序进程提供服务；另一方面，它通过与硬件抽象层（HAL）进程的交互来转发这些请求。Android 中，该进程以 AID_BLUETOOTH(1002) 运行，名为 “com.android.bluetooth”。此进程在 Java 启动，并通过 JNI 来加载 native library。其他不使用 JVM 的系统会有一个 pure native process。该进程由于各种原因，存在多个进程。GD stack 完全运行在此进程中。

1.2 Thread in Bluetooth stack process

Bluetooth stack 中线程优化目标如下：

- 尽可能减少线程数量以简化同步操作
- 在单独线程中进行阻塞 I/O 操作
- 尝试将 I/O 操作切换到轮询模式（polling mode），这样我们可以在主线程上使用事件驱动的方法与之交互
- 将报警和计时器机制（alarm and timer mechanisms）移到其调用线程中，以避免创建单独的报警线程
- 将各个组件隔离开来，以便每个组件都可以独立启动和停止，而不会终止主线程
- 尽量采用数据传递而非多线程间的数据共享，以减少锁和竞态条件的发生

在上述优化之后，剩下五种原生代码线程类型：

1. **Main thread**

   Bluetooth stack 的主要工作线程。该线程的执行上下文进一步划分为驻留在各个 Modules 中的 Handlers。如果运行平台的性能受到限制，该线程还可以进一步划分为更小的线程。部署者只需要绑定 handlers 到不同的线程即可，这不会影响整体操作。

2. **JNI thread**

   在 native 线程中，我们把 Java 层视为一个独立的应用程序，因为它的线程模块完全不同。因此，在两层之间添加一个线程，以缓冲任何阻塞操作。

3. **HCI thread（or other HW I/O thread）**

   该线程负责处理硬件 I/O 操作，可能会阻塞。因此需要单独的线程来避免阻塞主线程。

4. **Audio worker thread**

   负责需要更高精度执行时间的音频编码和解码操作。该线程独立运行，以避免受到主线程影响。

5. **Socket I/O thread**

   与使用 BluetoothSocket 接口的各种应用程序进行通信。由于存在潜在的 I/O 延迟，因此需要单独的线程。

1.3 Data flow diagram

不同组件间的函数调用被抽象为通过队列传递的控制包 control packet（函数闭包，function closure）。组件之间的数据流是通过队列发送的数据包，用 Reactor 进行信号指示。它们将合并为每个组件的输入队列。我们定义了三种类型的队列：

1. **Non-blocking queue**

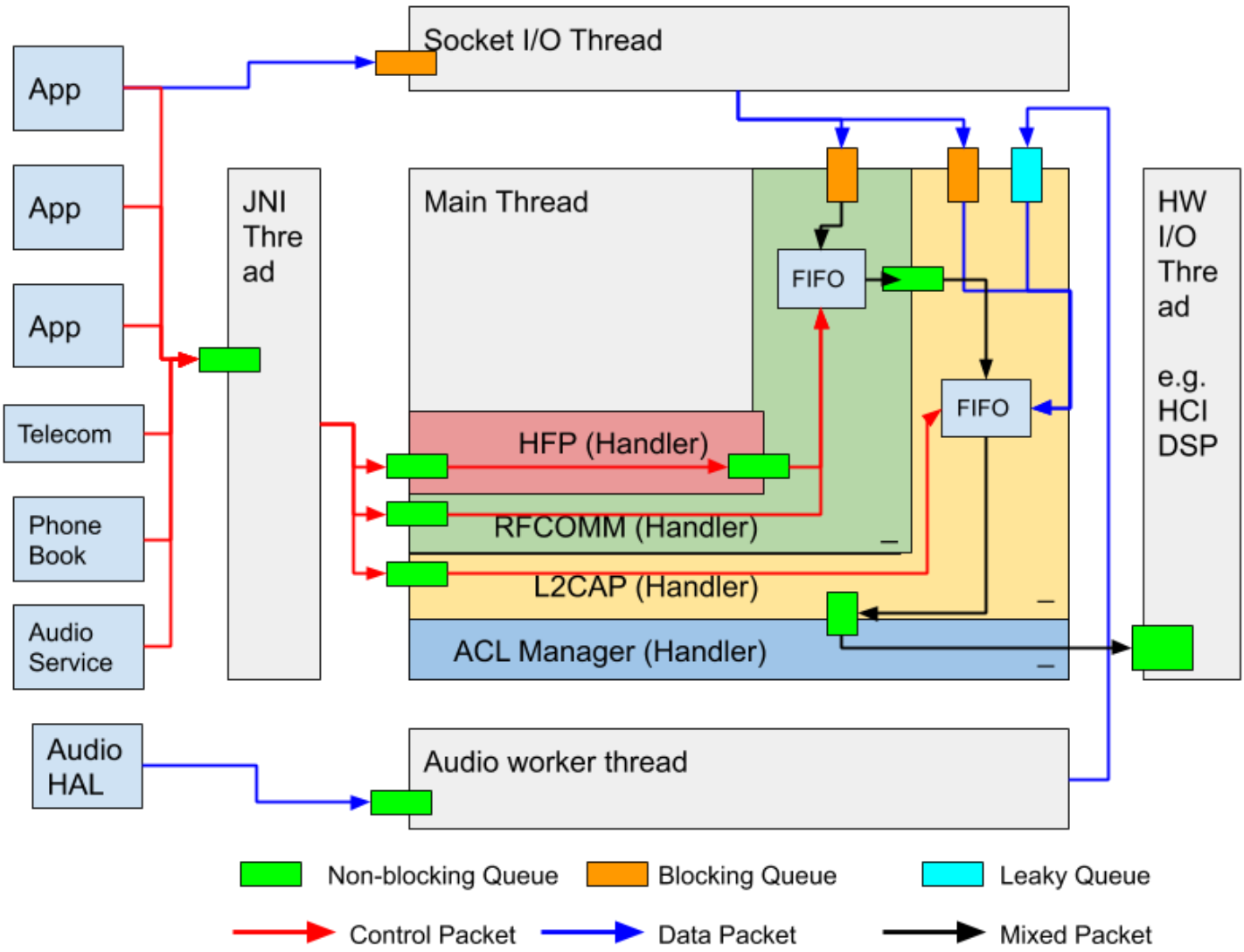
   当用户在队列为空时尝试 dequeue，或者队列为满时 enqueue，会立即 return。在一个线程中进行的所有队列操作都必须是非阻塞的，否则可能会导致死锁。

2. **Blocking queue**

   当用户在队列为空时尝试 dequeue，或者队列为满时 enqueue，就会 block，直到其他线程队列变为可读写状态。它可以作为一种流量控制机制，避免用户线程发送过多的数据包。

3. **Leaky queue**

   与非阻塞队列类似，但当队列已满且用户试图 enqueue 时，它会刷新。这对音频编码很有用。



2. 构建块（Building blocks）

2.1 Module

GD 中的代码被打包成名为 Module 的 C++ objects。

- **Dependencies**

  模块通过实现 ListDependencies() 来为其他模块提供自身的依赖关系

- **Life Cycle**

  模块必须实现 Start() 和 Stop() 生命周期方法

- **Threading Module**

  Module 基类通过 GetHandler() 提供了一个用于代码执行上下文的 Handler

- **Metrics**

  模块可通过 DumpState() 为 dumpsys 转储其状态信息

2.2 Handler

与 android.os.Handler 类似，bluetooth::os::Handler 提供了一个顺序执行上下文，同时隐藏了线程的概念，使执行代码无法感知线程的存在。

通过将执行上下文划分为更小的区域，Handler 在以下方面有利于开发：

- 由于顺序执行上下文，不需要太多的 locking
- 较小的 context 使得代码流的管理更加容易
- 线程分离让系统部署者有更多的自由来调整底层的线程分配。例如，对于没有完全线程实现的实时操作系统，可以使用 Handler 来提供接近线程的执行上下文。

当然，使用 Handler 也有一些缺点：

- 虽然多个 Handler 可以绑定到同一个线程，但 Handler 并不能保证不同 Handler（即使它们在同一线程上）之间执行代码的顺序。
- 绑定到同一线程的 Handlers 之间的锁可能会导致死锁。
- 必须在 Handler 之间复制数据，以避免死锁和竞态条件。

2.3 Reactor

bluetooth::os:Reactor 实现了 Reactor 设计模式，其中并发事件通过同步事件分发器被分解为 a list of Request Handlers，它们是通过 Dispatcher 注册的。

Android 中，我们使用文件描述符 file descriptors（如 eventfd for `Handler`, timerfd for `Alarm`, and socketfd for data processing pipelines）来实现它。在文件描述符的上下文中，事件被分为两种类型：

- **OnReadReady**

  表示多路分配器（demultiplexer）为 handler 提供了一些事件，并且 handler 至少可以从底层的事件队列中读取一个事件。这通常与 EPOLLIN、EPOLLHUP、EPOLLRDHUP 和 EPOLLERR 相关联。

- **OnWriteReady**

  表示分发器已准备好从该 handler 中接收更多事件，并且 handler 至少可以向底层队列写入一个事件，这通常与 EPOLLOUT 相关联。

这种模式会在一个队列和另一个队列之间产生反向压力。而无需额外的信号机制。在网络栈中使用这种模式，可以简化信号代码流。

3. 包定义语言（Packet Definition Language，PDL）

包（Packet）解析和序列化一直是任何网络栈的重要组成部分。通常它是第一个与远程设备交互的代码片段。过去这是通过使用 STREAM_TO_UNIT8 或 UNIT8_TO_STREAM 之类的宏手动实现的。这种手动方法既冗长又容易出错。为了解决这个问题，我们创建了一种 Pakcet Definition Language，它将网络包结构定义到位级（bit level）。其代码生成器将自动生成 C++ 头文件和 Python 绑定，并且对代码生成器的任何更改都将系统地应用于所有生成的 packet code。

PDL 示例：

```
// Comments
little_endian_packets // Whether this packet is big or small endian

// Include header from other C++ header files
custom_field SixBytes : 48 "packet/parser/test/" // expect six_bytes.h
custom_field Variable "packet/parser/test/" // expect variable.h

// A packet
packet Parent {
  _fixed_ = 0x12 : 8, // fixed field 0x12 that takes 8 bits
  _size_(_payload_) : 8, // Size field that takes 8 bits
  _payload_, // special payload field of variable size
  footer : 8, // fiexed size footer of 8 bits
}

packet Child : Parent {
  field_name : 16, // addition field append after Parent
}

// an enum of 4 bits
enum FourBits : 4 {
  ONE = 1,
  TWO = 2,
  THREE = 3,
  FIVE = 5,
  TEN = 10,
  LAZY_ME = 15,
}
```

packet/parser/README

4. 模块之间的调用约定

对于模块之间的大多数通信，开发者应该在类似这样的通用模型中假设一种异步 server-client 模型。

4.1 异步服务器-客户端模型（Asynchronous server-client model）

```
// Define callback function type
using CallbackFunction = std::function<void(ParamType)>;

// Asynchronous method definition
bool Foo(Parameter param, CallbackFunction callback);

// A new callback is passed for each asynchronous call
// Always prefer lambda over std::bind
CallbackFunction callback = [this] {
  // do something
};
Parameter param = {
  // something
};
if (Foo(param, callback)) {
   // The callback will be invoked
   // Callback must be invoked in the future
} else {
   // Failed, no need to wait
}
```

许多协议和配置文件都符合这种模型，例如 AclManager 和 L2cap。

4.2 同步数据库模型（Synchronous database model）

某些情况下，异步 server-client 模型不可行。这时，可以考虑同步的数据库模型。该模型中，可以通过互斥锁来使操作同步执行。当方法返回时，必须将更改反映到所有依赖项中。对内部状态的更改必须是原子性的。

```
// Synchronous method definition
void Foo(Parameter param, Output* output);
int Bar(Parameter param);
Parameter param = {
  // something
};
Output output = {};
Foo(param, &output);
// output can be used immediately
int bar_output = Bar(param);
// bar_output can be used immediately
```

许多存储和信息模块都可以应用这个模型，例如 Metrices 和 Storage。

Bluetooth Reading

- Bluetooth - wiki
- Android Bluetooth - 敲代码的小林哥
- 鸿蒙蓝牙
- nRF Connect SDK
- Gabeldorsche Android Bluetooth stack