第 1 章

开发环境搭建

万事开头难，学习一门新技术最怕的就是刚开始的拦路虎：编译环境、Demo 工程、Server 部署等。这 3 个问题（尤其是最后一个）想必会浇灭很多人的热情之火。本章将会手把手地带着大家搭建好 WebRTC 的开发环境。

1.1 WebRTC 简介

WebRTC 是一项致力于把实时通信功能引入到浏览器中的技术，由万维网联盟（World Wide Web Consortium，W3C）和互联网工程任务组（Internet Engineering Task Force，IETF）联合制定标准，并在 2018 年 9 月份宣布进入 CR（Candidate Recommendation）阶段，这意味着 WebRTC 1.0 的功能和 API 定义已经基本完成。

在浏览器支持方面，Edge 和 Safari 也在 2017 年正式宣布开始支持 WebRTC，加上之前就已经支持 WebRTC 的 Chrome、Firefox 和 Opera，目前主流的浏览器都已经支持 WebRTC 了。当然，不同的浏览器支持程度可能有细微的区别，不过我们可以利用 adapter.js（https://github.com/webrtcHacks/adapter）这一层包装来处理不同浏览器的差异。

Google Chrome 对 WebRTC 的实现（简称 Google WebRTC）是完全开源的，其核心代码用 C++ 编写，实现了 WebRTC 标准里定义的 API，用于 Chromium 浏览器内核。此外，WebRTC 的代码库里也包含了 Android 和 iOS/macOS 平台的 SDK 封装，分别提供 Java 和 Objective-C 接口，便于这些平台的开发者调用，Windows 和 Linux 平台则可以直接调用 WebRTC 的 C++接口进行开发。

WebRTC 应用——无论是 Web 端还是 Native 端——通常都会有 3 种运行模式：P2P（Peer to Peer）模式，SFU（Selective Forwarding Unit）模式和 MCU（Multi-point Control Unit）模式。其中，P2P 模式会尽量在客户端之间直接建立媒体数据连接，避免使用服务器转发媒体数据，SFU 和 MCU 模式则一定会由服务器转发媒体数据。P2P 模式的优点是可以减少租用服务器和网络带宽的成本，

但是客户端之间的网络连接质量难以得到保证，音视频通话的效果往往会差一些，而 SFU 和 MCU 模式下，服务端的网络状况是很稳定的，能有效提高通话质量，但需要付出更高的成本。

WebRTC Native App 使用最多的 SDK 就是 Google WebRTC，但现在也有一些其他的开源实现，比如 https://github.com/awslabs/amazon-kinesis-video-streams-webrtc-sdk-c 和 https://github.com/pion/webrtc。从功能完整度来看，Google WebRTC 是最完整的，但后两者更轻量，而且 pion/webrtc 还是纯 Go 语言实现，有其独特的应用场景。

WebRTC 服务端——3 种模式下的信令逻辑，以及 SFU 和 MCU 模式下的媒体处理逻辑——有非常多的开源或商用实现，比如 Google WebRTC 团队开源的非常简略的 P2P 信令服务端 AppRTC（https://github.com/webrtc/apprtc），以及 Janus（https://github.com/meetecho/janus-gateway）、Jitsi（https://github.com/jitsi/jitsi-meet）、Kurento（https://github.com/Kurento/kurento-media-server）、Licode（https://github.com/lynckia/licode）、mediasoup（https://github.com/versatica/mediasoup）、Medooze（https://github.com/medooze/media-server）和 OWT（https://github.com/open-webrtc-toolkit/owt-server）等。

尽管在 iOS/macOS、Android/Linux、Windows 这 3 类平台中 WebRTC 自身的代码是同一个仓库，但编译依赖不同，所以不可能放到一起。另外，Android/Linux 的编译环境要求文件系统区分大小写；对于 macOS 系统，如果没有重新格式化系统盘重装系统，就不用尝试下载到系统盘了。笔者的移动硬盘里曾经有 webrtc_ios、webrtc_android 和 webrtc_windows 三个目录，一共 60 GB。

下面的步骤对网络环境有较高的要求，大家可以上网查找相关教程，这里就不赘述了。

1.2 iOS/macOS 编译环境

WebRTC iOS/macOS 的开发需要在 macOS 下进行，本书使用的开发环境是 macOS 10.15.3（19D76）和 Xcode 11.3.1（11C504），存放代码库的磁盘建议至少有 30GB 的空间。

首先我们需要下载 WebRTC iOS/macOS 的代码库，以及相关的编译依赖，而这需要我们先下载安装 depot_tools。depot_tools 是 Chromium 代码库管理工具，提供了代码管理、依赖管理、工作流程管理等功能，运行 depot_tools 需要 Python 2.x 环境，而且需要是官方 build（--version 选项不能输出额外信息，macOS 系统自带的 Python 环境就是 2.x，没有问题，但如果使用 virtualenv 或 conda 等工具管理系统的 Python 环境，就需要把当前环境的 Python 可执行文件链接到系统的 Python 程序上去）。

下载 depot_tools 和 WebRTC iOS/macOS 代码的命令如下：

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
export PATH=$PATH:<depot_tools clone 到本地的路径>
# cd 到希望放置 WebRTC iOS/macOS 代码库的目录中
fetch --nohooks webrtc_ios
# 上条命令执行完毕后，可以编辑当前目录下的 .gclient 文件
# 在 target_os 里加上 mac，这套代码就可以用于 macOS 开发了
gclient sync
```

上面执行的 fetch 和 gclient 命令，就是 depot_tools 提供的。

1.3　Android/Linux 编译环境

WebRTC Android/Linux 的开发需要在 Linux 下进行，安装依赖的过程比较烦琐，所以笔者制作好了一个 Docker 镜像（Docker 的安装和基本教程，大家可以上网查阅相关资料，这里不做展开），通过 Docker 一键启动即可，非常简单：

```
docker run --rm \
  -v <希望放置 WebRTC Android/Linux 代码库的绝对路径>:/webrtc \
  -it piasy/webrtc-build
```

首次执行会下载 Docker 镜像，需要等一会。启动成功后，命令行会变成 Docker 镜像实例的命令行，这个命令行里已经配好了 WebRTC Android/Linux 开发所需的环境，在这个命令行里下载 WebRTC Android/Linux 代码的命令如下：

```
cd /depot_tools && git pull
cd /webrtc
fetch --nohooks webrtc_android
# 上条命令执行完毕后，可以编辑当前目录下的 .gclient 文件
# 在 target_os 里加上 linux，这套代码就可以用于 Linux 开发了
gclient sync
```

1.4　Windows 编译环境

WebRTC Windows 的开发需要在 Windows 下进行，本书使用的开发环境是 Windows 10 专业版（Parallel Desktops 虚拟机）和 Visual Studio 2017（简称 VS 2017）社区版。

下载代码之前，我们同样需要先下载 depot_tools，Windows 平台不是通过 git clone 下载，而是从官网下载一个最新的 zip 包（下载链接为 https://storage.googleapis.com/chrome-infra/depot_tools.zip），下载后解压到某个目录，注意不要在资源浏览器里拖曳或复制，一定是解压到某个目录，否则.git 目录可能会丢失。解压之后，把 depot_tools 路径添加到 Path 环境变量中，在环境变量中新建 DEPOT_TOOLS_WIN_TOOLCHAIN 并设置为 0。设置完毕后，打开命令行窗口，执行一次 gclient，不用提供任何参数，这次执行是为了让 gclient 安装相关依赖工具，注意一定要使用 cmd.exe，否则依赖工具可能无法正确安装。

下载 WebRTC Windows 代码的命令如下：

```
# cd 到希望放置 WebRTC Windows 代码库的目录中
fetch --nohooks webrtc
gclient sync
```

在开始编译 WebRTC Windows 代码之前，我们需要先配置好 Visual Studio 开发环境。

下载 VS 2017 installer 之后，可以通过命令行启动它，启动时添加如下参数：

```
    --add Microsoft.VisualStudio.Workload.NativeDesktop --add
Microsoft.VisualStudio.Component.VC.ATLMFC --includeRecommended
```

启动后选择安装 10.0.17134 版本的 Windows 10 SDK（必须要选择这个版本），然后安装即可。编译 WebRTC Windows 还需要我们安装 Debugging Tools for Windows，但它不是通过 VS 2017 installer 安装的，而是在安装完 Windows 10 SDK 之后，在控制面板中右击 Windows Software Development Kit，然后选择“修改”选项后勾选并安装的。

1.5 更新 WebRTC 编译环境

WebRTC 的开发非常活跃，有时一天能有几十个提交，建议大家定期更新（笔者通常是在 WebRTC 的 Android 库在 bintray 上发布更新后应用一次更新）。

更新已有的 WebRTC 代码库主要分为两步，且各个平台的操作都一样：更新 WebRTC 主代码仓库；更新编译依赖。更新 WebRTC 主代码仓库可以通过 git 操作完成，如果没有对代码做过修改，在命令行中进入 src 目录后执行 git pull 即可，否则可能需要切换分支或者通过 git rebase 等命令调整分支（git 操作这里不做展开，大家可以自行查阅资料）。更新编译依赖需要在更新主代码仓库后进行，在命令行中进入 src 父目录后执行 gclient sync 即可。

1.6 macOS 上的 Android 编译环境

虽然 WebRTC 官方建议我们使用 Linux 环境进行 Android 开发，但无论是虚拟机还是 Docker 镜像都无法完全释放宿主机的性能。本节将介绍如何在 macOS 下编译 WebRTC Android 的全套代码，充分释放 MacBook 的强劲性能。下载代码虽然还是得借助 Docker 镜像，不过下载代码的过程和性能关系不大。

具体的步骤如下：

- 在 Docker 镜像里同步（gclient sync 命令）好 Android/Linux 的代码库。
- 查看 src/third_party/android_ndk/source.properties 里 Linux NDK 的版本，然后下载对应版本的 macOS NDK，并替换原有 src/third_party/android_ndk 目录里的内容，但需要把原有 NDK 目录里的 BUILD.gn 复制过来。当然这里建议大家把原有的 src/third_party/android_ndk 目录备份，同步前还原为 Linux NDK，编译前替换为 macOS NDK，这样就不用每次都下载 NDK 了。
- 修改 src/build/toolchain/gcc_solink_wrapper.py 末尾部分为：

```
# Finally, strip the linked shared object file (if desired).
if args.strip:
  result = subprocess.call(wrapper_utils.CommandToRun(
      [args.readelf[:-7] + "strip", '-o', args.output, args.sofile]))
```

- 下载最新的 gn（https://chrome-infra-packages.appspot.com/dl/gn/gn/mac-amd64/+/latest）和 clang-format

（ https://storage.googleapis.com/chromium-clang-format/0679b295e2ce2fce7919d1e8d003e497475f24a3，需要替换最后的 hash 值为 src/buildtools/mac/clang-format.sha1 的内容）并放到 src/buildtools/mac 目录中。

- 本地编译 llvm（因为下载的 macOS 版本都没有 llvm-ar 这个程序），编译命令为“./src/tools/clang/scripts/build.py --without-fuchsia”。若更新代码之后遇见编译错误，则需要重新执行此命令编译 llvm。
- 把$JAVA_HOME/bin 加到 PATH /usr/bin 的前面，这样找到的就会是正确的 jdk 路径，也就能找到 rt.jar 了，否则会报错“No such file or directory: '/System/Library/Frameworks/JavaVM.framework/Versions/A/jre/lib/rt.jar'”。
- 确保在 Python 2.x 的 shell 里执行 gn 和 ninja 即可编译。
- 可能运行 gn 命令会报错 string_replace 函数找不到，这是因为本地的 gn 可执行程序太旧，下载最新版应该就可以了。

```
    ERROR at //build/config/BUILDCONFIG.gn:598:24: Unknown function.
          _output_name =
string_replace("${_magic_prefix}${_output_name}",
                       ^-------------
    See //testing/test.gni:123:7: whence it was called.
         shared_library(_library_target) {
         ^--------------------------------
    See //webrtc.gni:379:3: whence it was called.
      test(target_name) {
      ^------------------
    See //BUILD.gn:487:3: whence it was called.
      rtc_test("rtc_unittests") {
      ^--------------------------
```

- 修改 src/third_party/ijar/BUILD.gn 的“if (is_linux) {”为“if (is_linux || is_mac) {”。
- 下载 macOS 版本的 JDK 11.0.4(https://www.oracle.com/technetwork/java/javase/downloads/java-archive-javase11-5116896.html)，下载 jdk-11.0.4_osx-x64_bin.tar.gz 并解压后用 jdk-11.0.4.jdk/Contents/Home/目录替换原有的 src/third_party/jdk/current 目录。当然这里还是建议先备份原有的目录。

第 2 章

运行官方 Demo

搭建好开发环境后，下一步就是运行官方 Demo 了。通过官方 Demo，我们既可以了解 WebRTC API 的基本使用，也可以基于它们分析 WebRTC 内部的代码流程，探究其内部原理。

2.1 官方 Demo 简介

WebRTC 本身不包括信令，需要有一个信令服务器才能让客户端之间进行通话，信令服务器负责房间管理、长连接等功能。

WebRTC 的官方 Demo 分为两套，使用不同的信令服务器：Android/iOS/macOS 是一套，使用的信令服务器是 AppRTC Server，客户端叫作 AppRTCMobile；Windows/Linux 是另一套，使用的信令服务器是 PeerConnection Server，客户端叫作 PeerConnection Client。各个平台的 AppRTCMobile 之间可以实现自推自收（Loopback）和两终端互相推收（PeerConnection Client）。

2.2 部署 AppRTC Server

AppRTC Server 的部署也是一件比较复杂的工作，为了方便读者朋友快速上手，笔者也制作好了一个 Docker 镜像，通过 Docker 一键启动即可，非常简单。由于需要开放一段 UDP 端口，macOS 和 Linux 的开放方式有所区别，因此两个平台的启动方式略有不同。

macOS 系统启动命令为：

```
docker run --rm \
  -p 8080:8080 -p 8089:8089 -p 3033:3033 \
  -p 3478:3478 -p 3478:3478/udp \
```

```
  -p 59000-65000:59000-65000/udp \
  -e PUBLIC_IP=<服务器访问的 IP 地址> \
  -it piasy/apprtc-server
```

提　示

请将“<服务器访问的 IP 地址>”替换为实际部署的 AppRTC Server 的 IP 地址，下文中涉及 AppRTC Server 地址的内容也需要做同样的替换。

Linux 系统启动命令为：

```
sudo docker run --rm --net=host \
  -e PUBLIC_IP=<服务器访问的 IP 地址> \
  -it piasy/apprtc-server
```

服务器防火墙需要配置开放的端口如下：

- TCP 8080 用于 Signal Server 的 HTTP 服务。
- TCP 8089 用于 Signal Server 的 WebSocket 服务。
- TCP 3033 用于 ICE Server。
- TCP 3478 和 UDP 3478 用于 TURN Server 供客户端获取公网 IP 地址。
- UDP 59000-65000 用于 TURN Server 实现媒体数据中转。

提　示

如果对上述概念不熟悉，不要担心，下一章将会详细解答。

2.3　运行 iOS AppRTCMobile

下载好 WebRTC iOS/macOS 的代码之后，为了利用我们刚才部署的 AppRTC Server，同时也便于 Xcode 打包真机调试，我们需要对 WebRTC iOS/macOS 的代码进行少许修改。

提　示

以下操作均以 WebRTC iOS/macOS 代码库的 src 目录为基础目录。

- 编辑 examples/objc/AppRTCMobile/ios/Info.plist 中的 Bundle ID，搜索 com.google.AppRTCMobile，将其修改为合适的 Bundle ID，例如 com.piasy.AppRTCMobile。
- 编辑 examples/objc/AppRTCMobile/ARDAppClient.m 中的 kARDIceServerRequestUrl 常量值，将其修改为@"http://<服务器访问的 IP 地址>:3033/iceconfig"。
- 编辑 examples/objc/AppRTCMobile/ARDAppEngineClient.m，将启动所有的 https://appr.tc 替换为 http://<服务器访问的 IP 地址>:8080。
- 编辑 examples/objc/AppRTCMobile/ARDTURNClient.m，完成如下替换：

```
// 将原有的如下 4 行代码
```

```
    NSDictionary *responseDict = [NSDictionary dictionaryWithJSONData:data];
    NSString *iceServerUrl = responseDict[@"ice_server_url"];
    [self makeTurnServerRequestToURL:[NSURL URLWithString:iceServerUrl]
              WithCompletionHandler:completionHandler];

    // 替换为如下新代码
    NSDictionary *turnResponseDict = [NSDictionary dictionaryWithJSONData:data];
    NSMutableArray *turnServers = [NSMutableArray array];
    [turnResponseDict[@"iceServers"] enumerateObjectsUsingBlock:
                    ^(NSDictionary *obj, NSUInteger idx, BOOL *stop){
       [turnServers addObject:[RTCIceServer serverFromJSONDictionary:obj]];
    }];
    if (!turnServers) {
    NSError *responseError =
       [[NSError alloc] initWithDomain:kARDTURNClientErrorDomain
                                  code:kARDTURNClientErrorBadResponse
                              userInfo:@{
       NSLocalizedDescriptionKey: @"Bad TURN response.",
       }];
    completionHandler(nil, responseError);
    return;
    }
    completionHandler(turnServers, nil);
```

修改完成后，我们可以通过 gn 命令生成 Xcode 工程，然后通过 Xcode 编译、运行、调试。生成 iOS arm64 架构 Xcode 工程的命令如下：

```
    # cd 到 WebRTC iOS/macOS 代码库的 src 目录中
    # 确保当前命令行使用的是 Python 2.x 环境
    gn gen out/xcode_ios_arm64 --args='target_os="ios" target_cpu="arm64"'
--ide=xcode
```

如果上述命令报类似如下错误：

```
    Automatic code signing identity selection was enabled but could
    not find exactly one code signing identity matching
    Apple Developer. Check that your keychain
    is accessible and that there is a valid code signing identity
    listed by `xcrun security find-identity -v -p codesigning`
    TIP: Simulator builds don't require code signing...
    ERROR at //build/config/ios/ios_sdk.gni:155:7: Assertion failed.
         assert(false)
         ^-----
    See //build/config/sysroot.gni:67:3: whence it was imported.
      import("//build/config/ios/ios_sdk.gni")
      ^-------------------------------------
    See //build/config/linux/pkg_config.gni:5:1: whence it was imported.
    import("//build/config/sysroot.gni")
    ^----------------------------------
    See //BUILD.gn:15:1: whence it was imported.
    import("//build/config/linux/pkg_config.gni")
```

```
^----------------------------------------
```

可以尝试编辑 build/config/ios/ios_sdk.gni，把 ios_code_signing_identity_description = "iPhone Developer"修改为 ios_code_signing_identity_description = "Apple Development"。

成功执行上述命令后，就可以在 out/xcode_ios_arm64 目录中找到一个名为 all.xcworkspace 的文件，双击即可通过 Xcode 打开，如图 2-1 和图 2-2 所示。

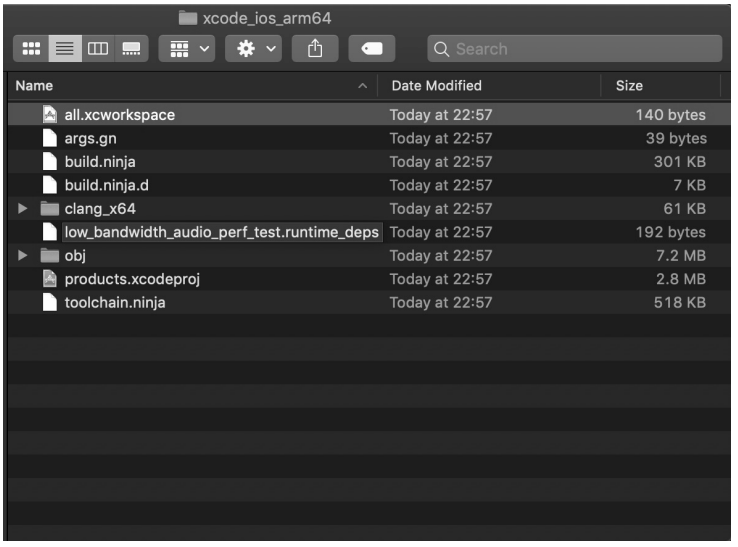


图 2-1　WebRTC iOS Demo Xcode 工程生成结果

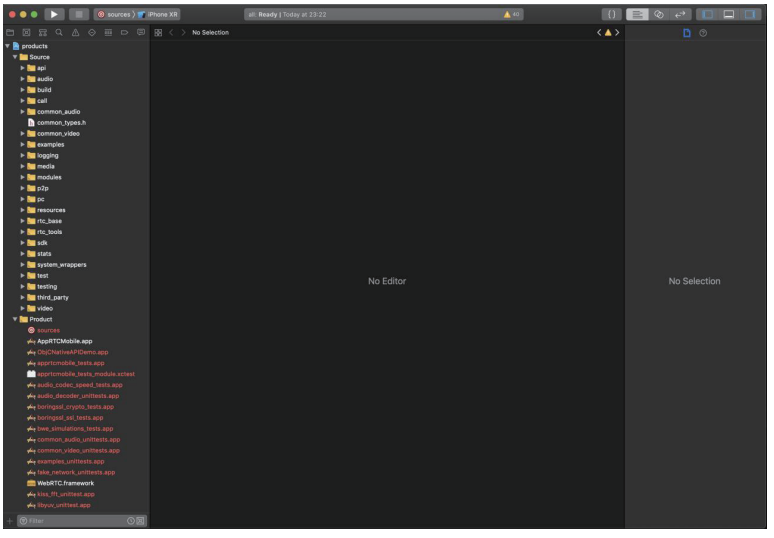


图 2-2　WebRTC iOS Demo Xcode 工程

在 Xcode 左侧导航栏中选中“products”，run target 选择“AppRTCMobile”，工程文件的设置 target 也选择“AppRTCMobile”，如图 2-3 所示。

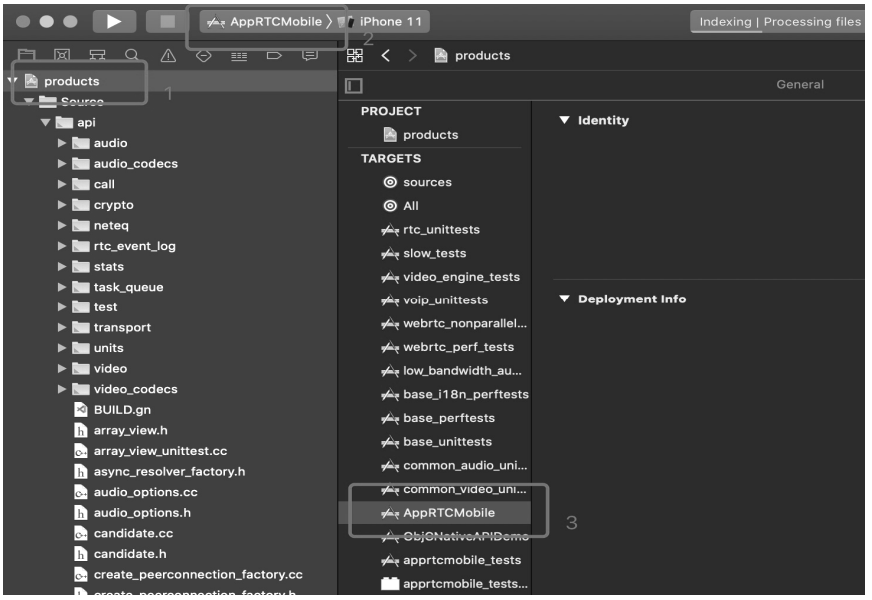


图 2-3　选择 target

接下来，单击 Xcode 工程设置中“General->Identity”部分的“Choose an Info.plist file”，选择 examples/objc/AppRTCMobile/ios/Info.plist，如图 2-4 所示。

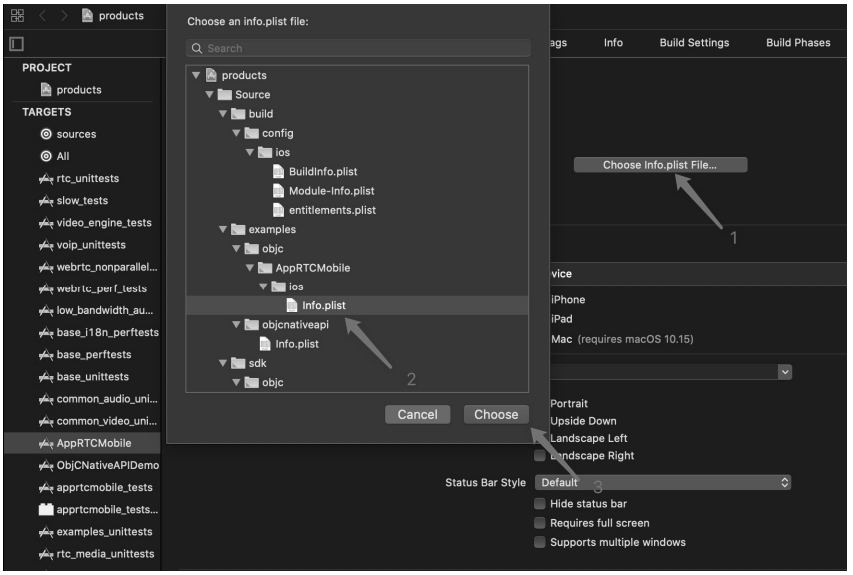


图 2-4　选择 Info.plist

提　示

选择之后，Xcode 显示的内容可能不会发生任何变化，仍然显示“Choose an Info.plist file”。不要担心，这应该是 Xcode 的一个 bug。我们可以直接点击“Signing & Capabilities”标签页选择签名配置，这时我们会发现 Info.plist 的选择已经生效了，因为已经能看到正确的 Bundle Identifier 等信息了。

接下来，在 Xcode 工程设置的“Signing”中勾选“Automatically manage signing”复选框，并选择合适的 Developer Team，如图 2-5 所示。

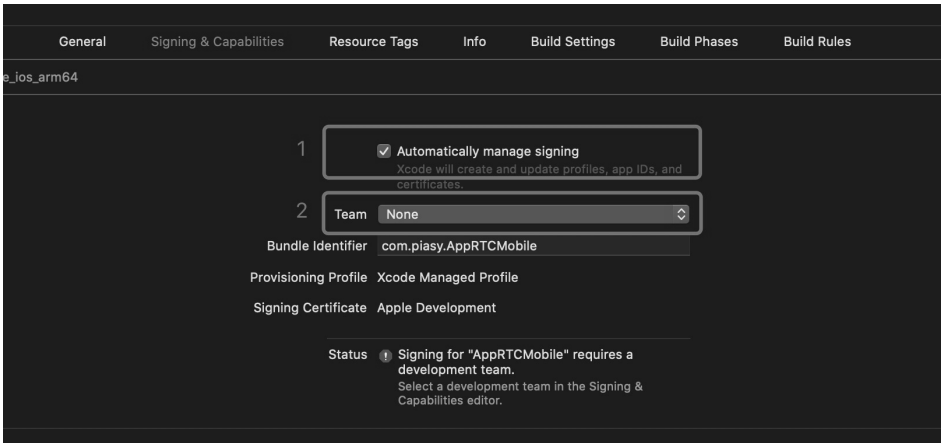


图 2-5　设置签名

由于 WebRTC 核心代码编译生成的 WebRTC.framework 是一个动态库，因此我们需要在 Xcode 工程设置的“Frameworks，Libraries，and Embedded Content”部分添加 WebRTC.framework，如图 2-6 所示。

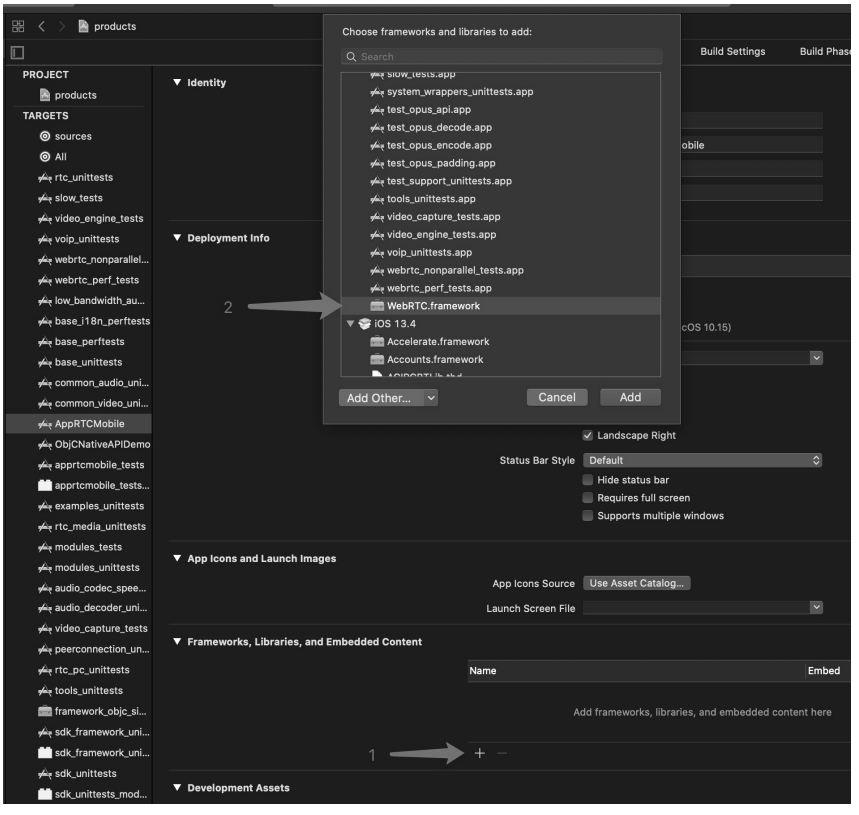


图 2-6　在 Frameworks，Libraries，and Embedded Content 里添加 WebRTC.framework

现在就可以把 iPhone 连接到电脑，在 Xcode 里选择设备，并单击编译运行了。WebRTC iOS 的编译实际是通过 ninja 完成的。编译过程如图 2-7 所示。

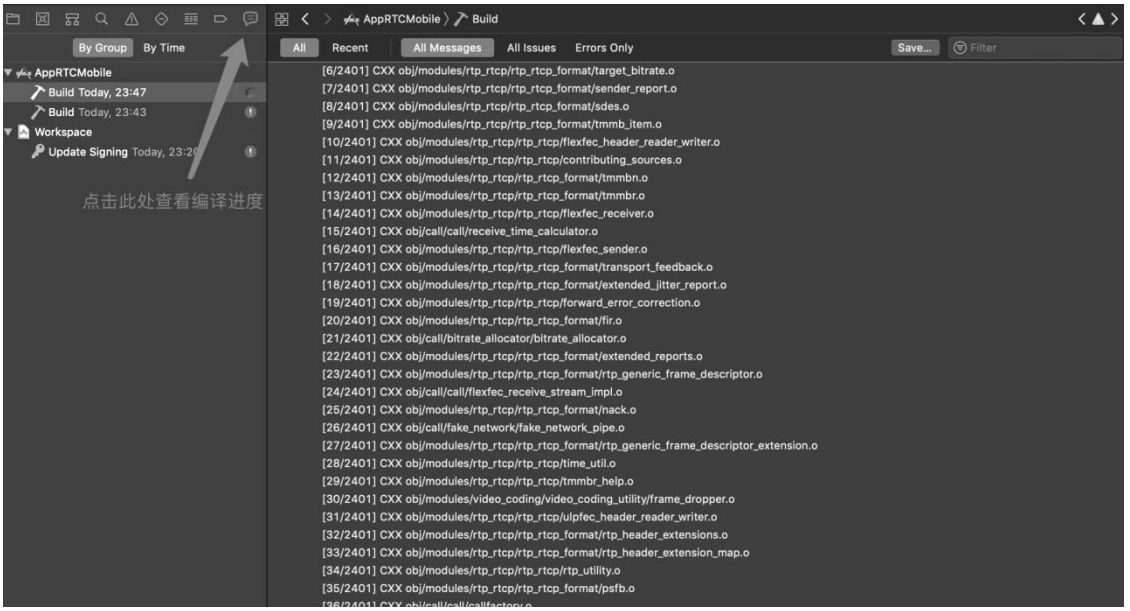


图 2-7　编译过程

提　示

单击运行后，Xcode 可能很快会报错，并且 Identity 的 Bundle Identifier 等信息会还原为“Choose Info.plist File…”，这是因为 ninja 又重新生成了一遍 Xcode 工程。重复一次图 2-4~图 2-6 所示的操作，再次运行即可。

成功运行后，启动界面如图 2-8 左图所示。

单击中间的“Loopback call”即可开始自推自收测试，效果如图 2-8 中图所示。

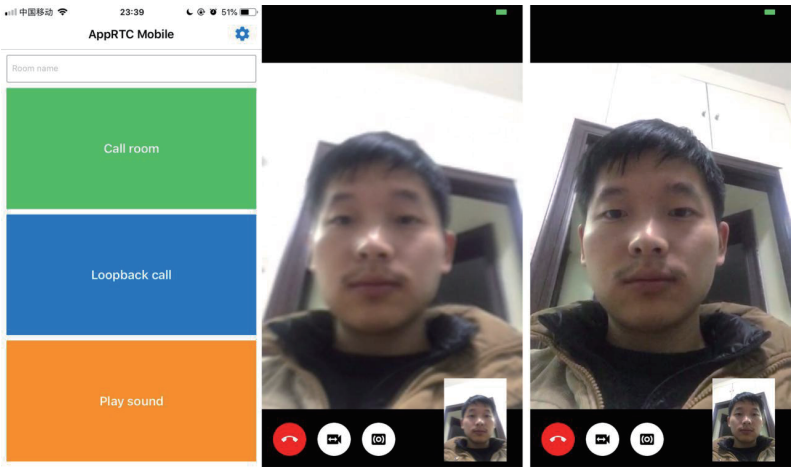


图 2-8　AppRTCMobile iOS

图 2-8 中图比右图模糊很多，这是因为 iOS AppRTCMobile 默认使用的发送视频分辨率是 192×144。我们可以单击右上角的设置按钮进入设置页面，选择发送视频分辨率为 640×480（图 2-8 右图所示的效果）。

2.4　运行 macOS AppRTCMobile

WebRTC macOS 的 Demo 也叫 AppRTCMobile，当然这只是 Xcode 工程的 target 名字，实际上它是一个 macOS 的 App。由于 macOS AppRTCMobile 和 iOS AppRTCMobile 共用了大部分代码，因此，如果我们在运行 iOS AppRTCMobile 时已经对代码做了修改，这里就不需要额外修改了，否则需要参考 2.3 节开头的代码修改说明进行修改。

在 macOS 下，我们也是使用 gn 命令生成 Xcode 工程，然后通过 Xcode 编译、运行、调试。生成 macOS x64 架构 Xcode 工程的命令如下：

```
    # cd 到 WebRTC iOS/macOS 代码库的 src 目录中
    # 确保当前命令行使用的是 Python 2.x 环境
    gn gen out/xcode_mac_x64 --args='target_os="mac" target_cpu="x64"'
--ide=xcode
```

如果命令没有报错，就可以在 out/xcode_mac_x64 目录中找到一个名为 all.xcworkspace 的文件，如图 2-9 所示。

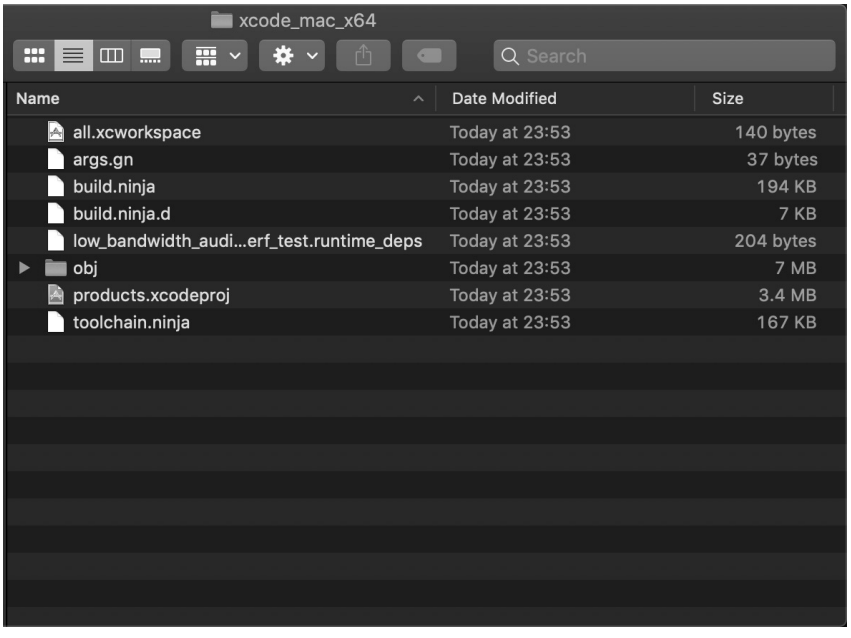


图 2-9　WebRTC macOS Demo Xcode 工程生成结果

双击该文件即可通过 Xcode 打开，首次打开 Xcode 会提示是否自动创建 scheme，单击“Automatically Create Schemes”按钮即可，如图 2-10 所示。

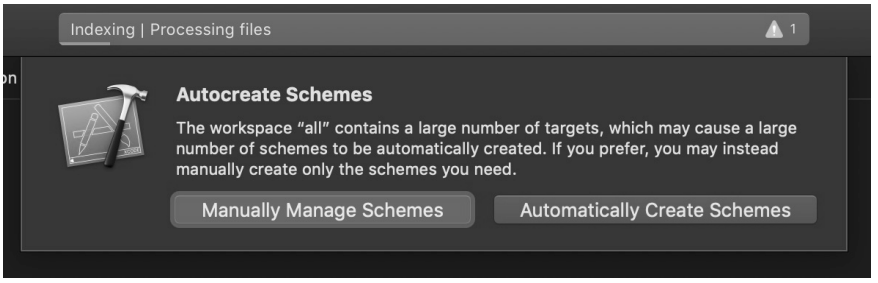


图 2-10　Automatically Create Schemes

选择后 Xcode 界面如图 2-11 所示。

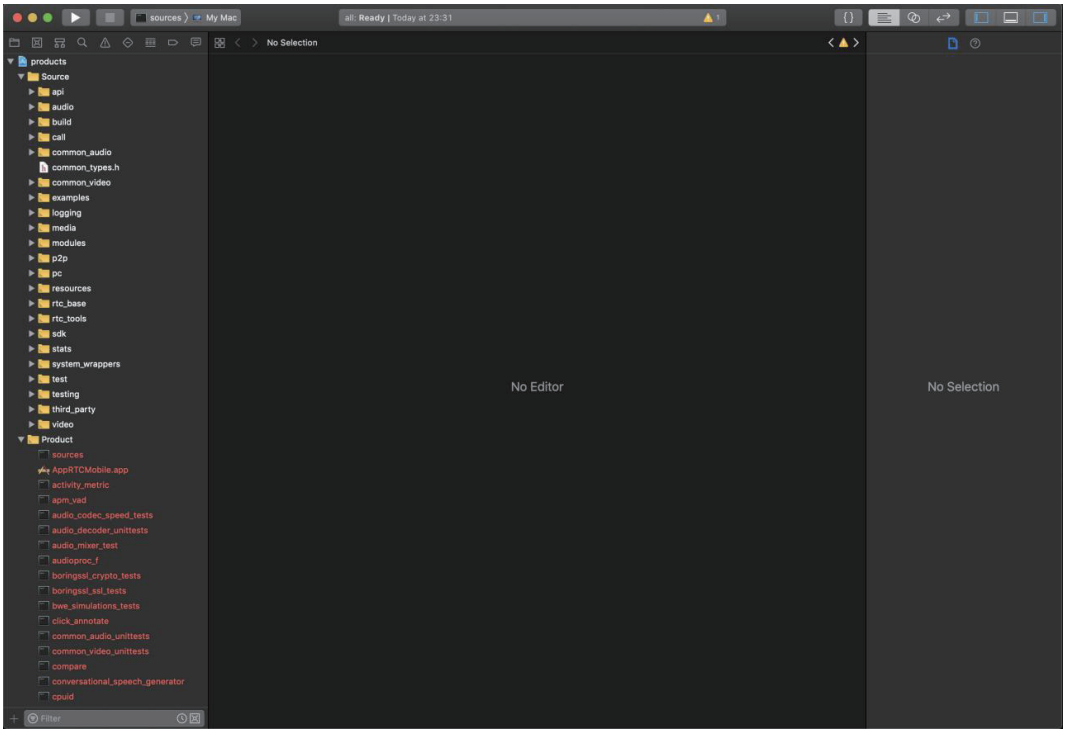


图 2-11　WebRTC macOS Demo Xcode 工程

和 iOS 工程一样，我们也需要在 Xcode 左侧导航栏中选中“products”，run target 选择“AppRTCMobile”，工程文件的设置 target 也选择“AppRTCMobile”，如图 2-12 所示。

接下来，单击 Xcode 工程设置的“General->Identity”部分的“Choose Info.plist File…”，选择 examples/objc/AppRTCMobile/mac/Info.plist，如图 2-13 所示。

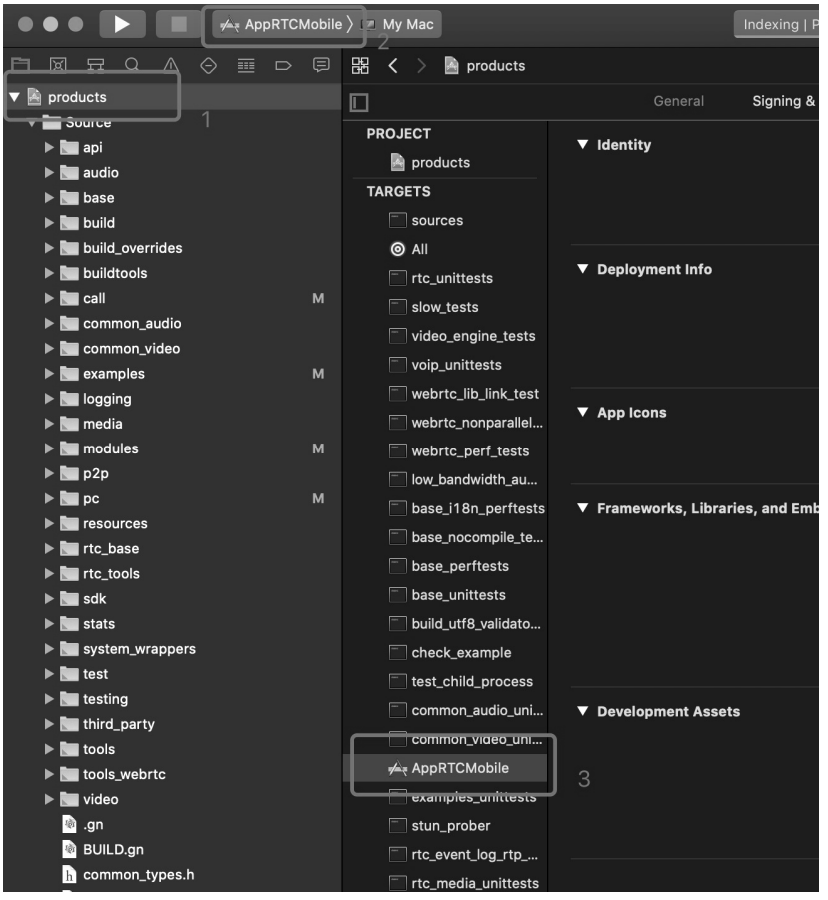


图 2-12　选择 target

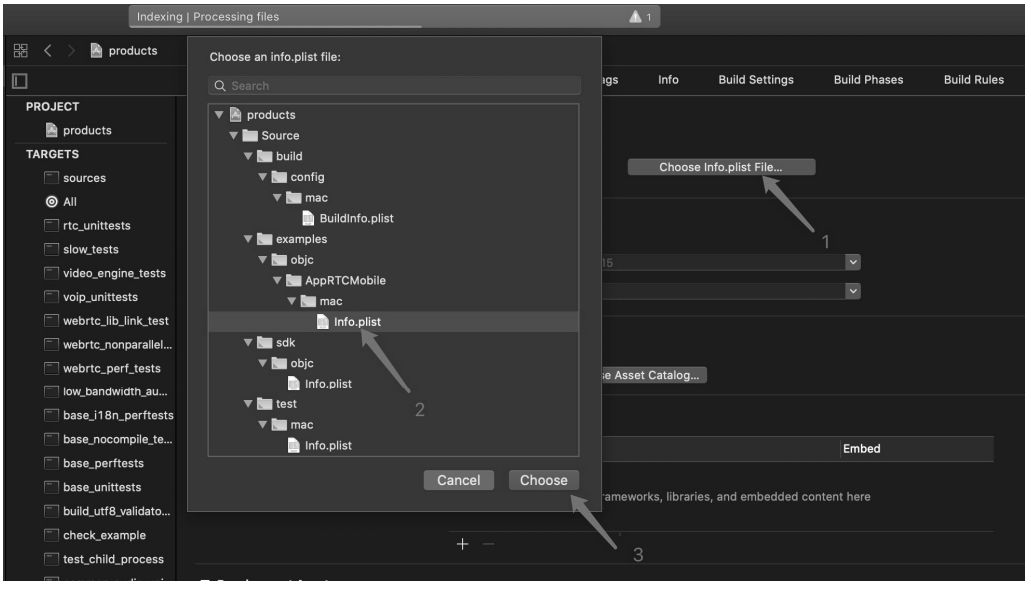


图 2-13　选择 Info.plist

提　示

和 iOS 工程一样，很可能选择之后 Xcode 显示的内容也不会发生任何变化，仍然显示“Choose Info.plist File…”处理方式与 2.3 节相同，不再赘述。

运行 macOS Demo 无须设置签名，所以我们现在就可以单击运行按钮进行编译了。WebRTC macOS 的编译实际也是通过 ninja 完成的，编译过程如图 2-14 所示。

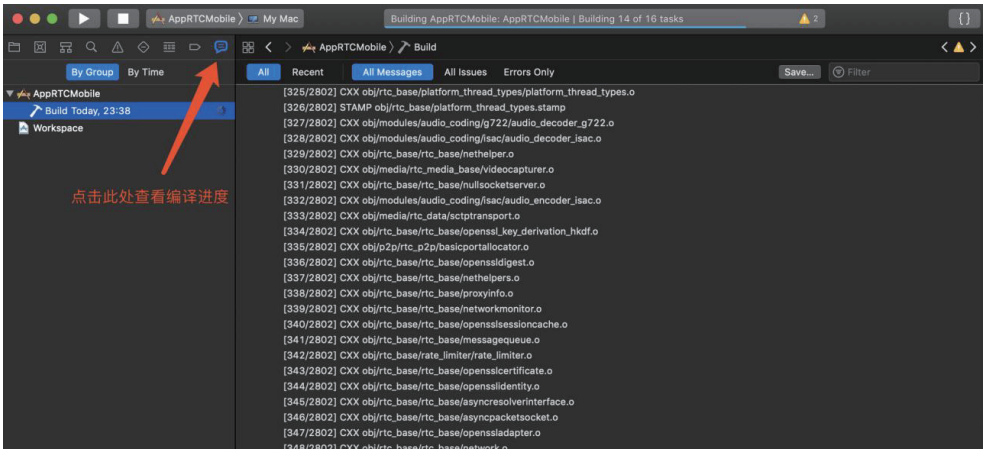


图 2-14　编译过程

提　示

报错后的处理方式同 2.3 节类似。

成功运行后，启动界面如图 2-15 所示。

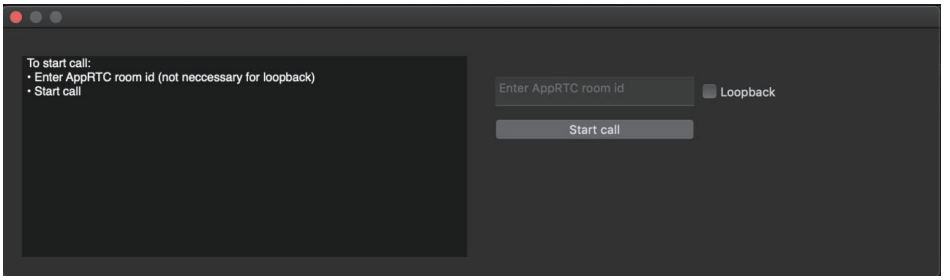


图 2-15　AppRTCMobile macOS

勾选“Loopback”复选框后单击“Start call”按钮即可开始自推自收测试，效果如图 2-16 所示。

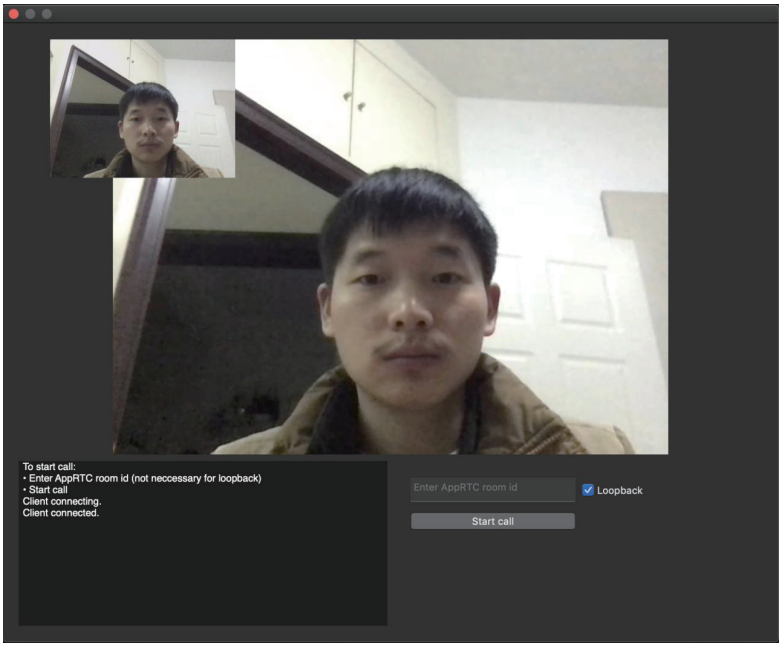


图 2-16　AppRTCMobile macOS 自推自收

2.5　运行 Android AppRTCMobile

WebRTC 主代码库里并不包含 Android Studio 工程，因此很多开发人员都不知道如何运行 Android AppRTCMobile 项目。为了方便读者快速上手，笔者编写了一些 Gradle 脚本，并把 WebRTC 官方发布在 Bintray 的 C++代码动态链接库（.so）放了进来，这样如果只需要开发调试 Java 代码，下载主代码库后用 Android Studio 打开即可，每次编译运行也只涉及 Java 代码。

这些修改都放在了笔者的 GitHub 仓库中，通过如下命令即可下载：

```
git clone https://github.com/Piasy/webrtc.git
```

克隆完成后，在 Android Studio 启动界面单击“Open an existing Android Studio project”按钮，并在弹出的路径选择框中选择 webrtc/sdk/android_gradle 目录 open。

打开工程后，Android Studio 界面如图 2-17 所示。

这里打开的实际上是 WebRTC iOS/macOS 的代码仓库，因为这个 Android Studio 工程每次编译运行只涉及 Java 代码，所以不需要任何 WebRTC Android 的编译依赖。

Gradle Sync 结束后，运行 AppRTCMobile 即可安装运行 Android AppRTCMobile。

成功运行后，启动界面如图 2-18 左图所示。

启动后，点击右上角的“设置”按钮，进入图 2-18 中图所示的设置界面。滑动到最底部，点击“Room server URL”，将 https://appr.tc 修改为 2.2 节部署的服务器地址（http://<服务器访问的 IP 地址>:8080），如图 2-18 右图所示。

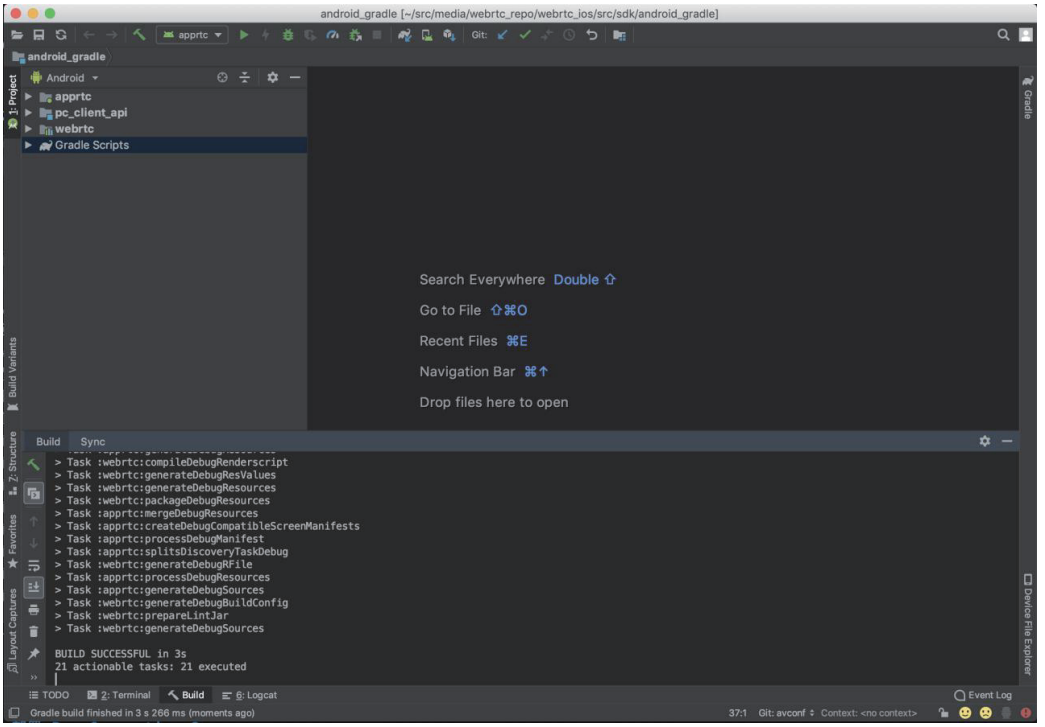


图 2-17　WebRTC Android Demo Android Studio 工程

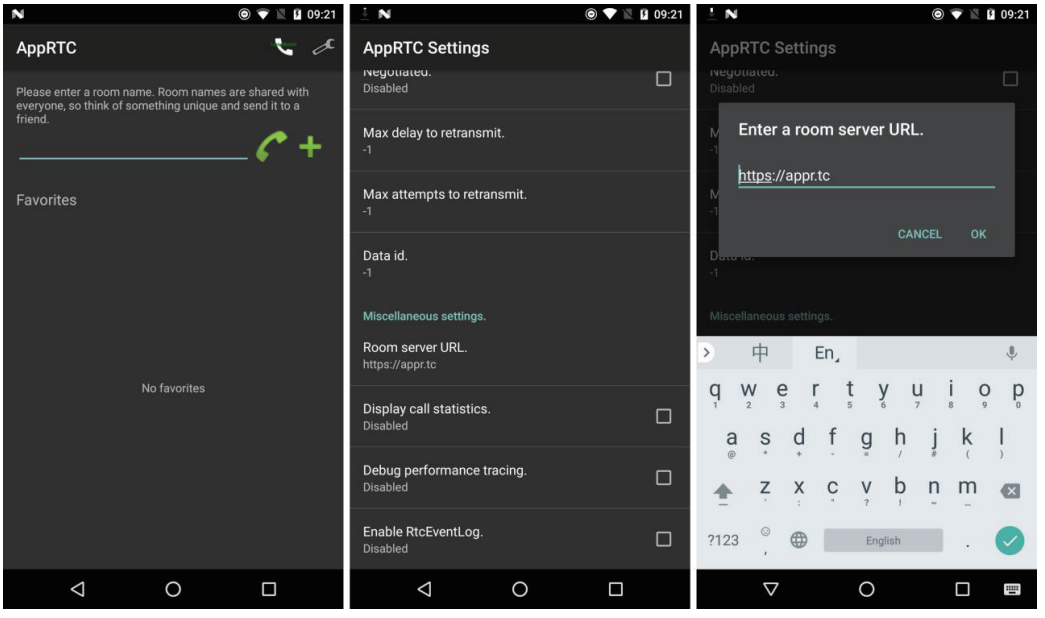


图 2-18　AppRTCMobile Android

返回启动界面，点击右上角的“Loopback”按钮，即可开始自推自收测试，效果如图 2-19 所示。

图 2-19　AppRTCMobile Android 自推自收

2.6　iOS/macOS/Android AppRTCMobile 互通测试

前面已经提到，各个平台的 AppRTCMobile 之间可以互通，前提是使用同一个 AppRTC Server。下面就简单介绍一下操作过程。

首先，Android AppRTCMobile 默认使用的是 VP8 视频编码，我们需要在设置页面中将其修改为 H264 Baseline（H264 High 也可以，不过只有 7.0 以上的手机才能生效），否则其他端接收视频时可能会很卡。设置界面如图 2-20 所示。

之后在首页输入框中输入房间号，在 Android AppRTCMobile 中点击输入框右侧的打电话图标即可加入该房间，在 iOS AppRTCMobile 中是点击顶部的“Call room”，在 macOS AppRTCMobile 中则是点击“Start call”，分别如图 2-21 上左、上右、下图所示。

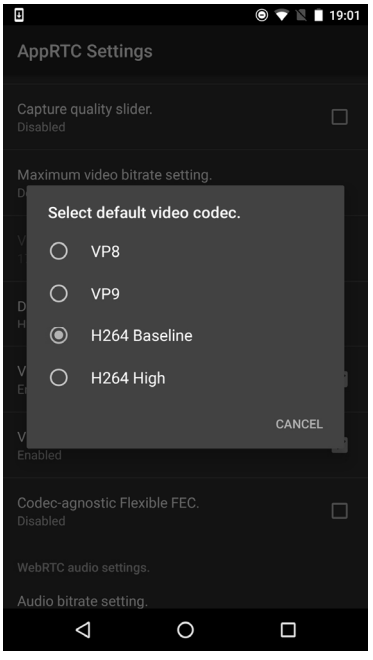


图 2-20　设置视频编码为 H264 Baseline

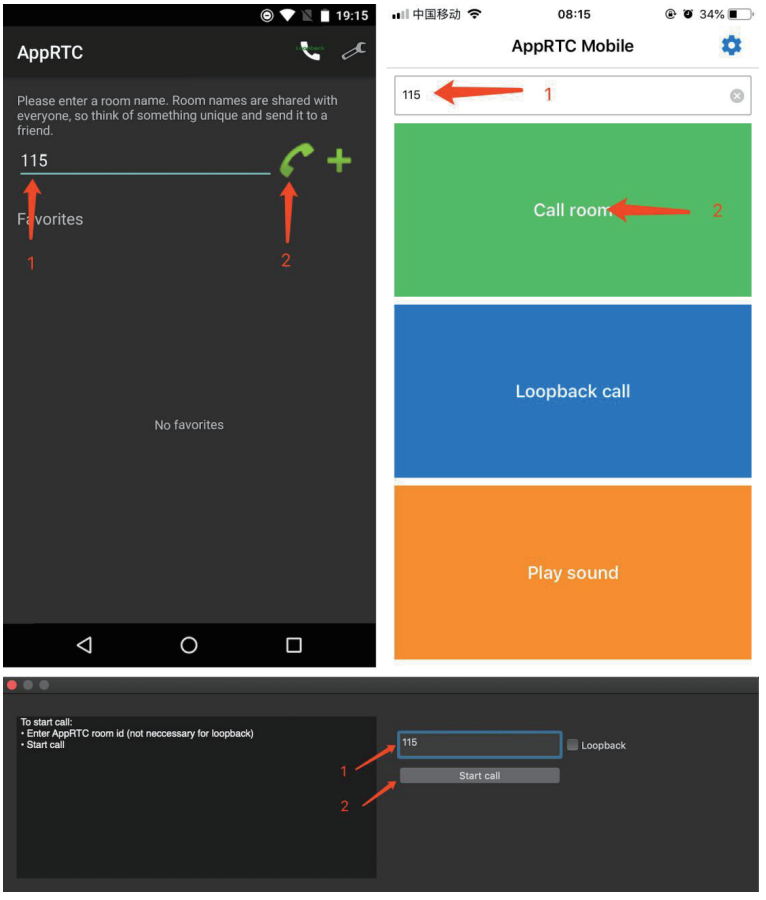


图 2-21　Android/iOS/macOS 加入房间

互通成功的截图这里就不展示了。有一点需要注意：AppRTC Server 对离开房间的处理不是很完善，尤其是异常离开，因此建议每次测试都使用不同的房间号，否则可能会报错房间已满。

2.7　Android C++代码单步调试

iOS/macOS AppRTCMobile 的 Xcode 工程可以给所有的代码设置断点进行单步调试，分析代码流程非常方便，但是 Android 项目却没有 C++代码调试的官方支持，分析代码流程时会有诸多不便。虽然我们可以为 Android 代码库生成 Xcode 项目，但是代码跳转不是很好用，所以如果能有 Android Studio 的 Native 开发支持，就非常方便了。

笔者经过了很长时间的努力，最终解决了这个问题，并在本节中分享给大家。

细心的读者可能在克隆 WebRTC Android Studio 项目（https://github.com/Piasy/webrtc.git）时就发现了一些端倪：其中包含了一些 CMakeLists。是的，它们就是为了实现 WebRTC Android C++ 代码调试的。不过这些 CMakeLists 只是告诉 Android Studio 怎么编译 WebRTC Android C++的代码，并不包含 WebRTC Android C++的代码，因此我们还是需要有一份准备好的 WebRTC Android 编译

环境。

接下来在 2.5 节里打开的 Android Studio 工程中编辑 gradle.properties 文件：

步骤 01　点击 Android Studio 左侧的“Project”工具栏，展开 Project 视图。

步骤 02　调整 Project 视图模式，选择“Project”模式，该模式和文件系统展现形式保持一致，便于查找文件。

步骤 03　找到 gradle.properties 文件，双击进行编辑，如图 2-22 所示。

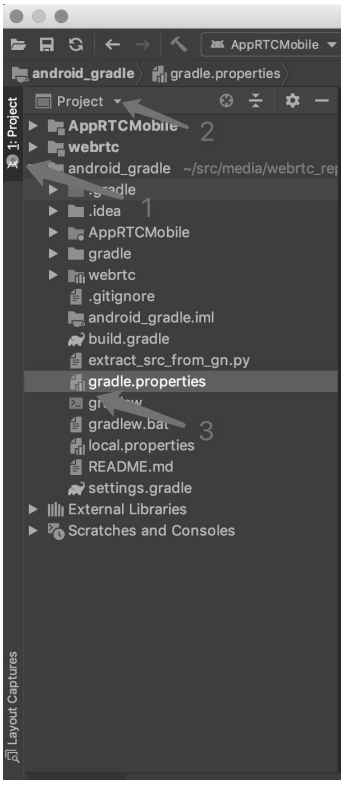


图 2-22　编辑 gradle.properties 文件

该文件底部和调试 Android C++代码相关的配置如下：

```
    compile_native_code=true

    webrtc_repo=/Users/piasy/src/media/webrtc_repo/webrtc_android/src
    webrtc_build_dir=out/android_studio

    android_jar=third_party/android_tools_mac/sdk/platforms/android-28/android
.jar

    py2=/Users/piasy/anaconda3/envs/py2/bin/python
    protoc=/Users/piasy/src/media/webrtc_repo/webrtc_android/src/out/protoc
```

- compile_native_code 用于控制 Android Studio 工程是否启用 C++代码的编译和调试，此时需要将其修改为 true;

- webrtc_repo 用于指定 WebRTC Android/Linux 代码库所在的绝对路径，前面已经提到，我们仍需要一份准备好的 WebRTC Android 编译环境，就是在这里进行设置的。
- webrtc_build_dir 用于指定编译生成的代码保存路径，是一个相对路径，相对于 webrtc_repo。
- android_jar 用于指定 Android SDK 的 android.jar 相对路径，也是相对于 webrtc_repo。
- py2 用于指定 Python 2.x 可执行程序的绝对路径。
- protoc 用于指定 protobuf 编译器程序的绝对路径。这个程序需要先在命令行里通过 ninja 编译一次特定的目标来生成，编译命令如下：

```
# cd 到放置 WebRTC Android/Linux 代码库的目录的 src 子目录中
gn gen out/android_debug_arm --args='target_os="android" target_cpu="arm"'
ninja -C out/android_debug_arm network_tester_config_proto_gen
```

上述命令执行完毕后，我们需要的 prococ 就生成了，路径为 out/android_debug_arm/clang_x64/protoc。

设置完毕后，点击 Android Studio 菜单栏的 File->Sync Project with Gradle Files，sync 完毕后运行 AppRTCMobile 就会开始编译 C++代码。运行成功后，就可以 attach debugger 了，如图 2-23 所示。

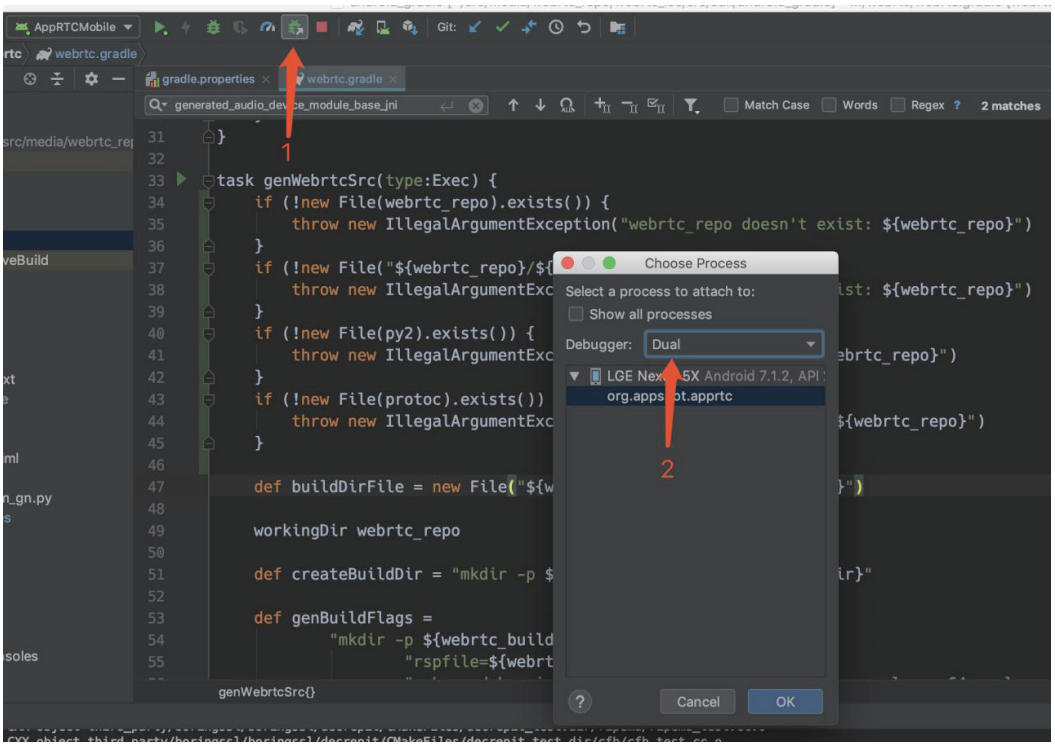


图 2-23　attach debugger

之后就可以在 C++代码里设置断点，或者通过 step in 从 Java 代码单步进入到 JNI 的 C++代码了。这里有几点注意事项：

- WebRTC Android/Linux 代码库的提交，需要和 https://github.com/Piasy/webrtc.git 基于的提交保持一致，可以通过 git reset 或 branch 命令选择 git 提交。

- 更新 WebRTC Android 和 https://github.com/Piasy/webrtc.git 后，需要删除 webrtc_build_dir 目录。
- 如果中途由于设置错误等原因导致编译出错后续重试之前，需要删除 webrtc_build_dir 目录和.externalNativeBuild 目录，以确保一个干净的编译环境。

2.8　部署 PeerConnection Server

PeerConnection Server 是供 Windows/Linux PeerConnection Client 使用的信令服务器，部署非常简单，通过 ninja 命令编译出可执行程序后直接运行即可。

PeerConnection Server 在 Windows 和 Linux 上都可以运行，本文以 Linux 版本为例：

```
# cd 到放置 WebRTC Android/Linux 代码库目录的 src 子目录中
gn gen out/linux_debug_x64 --args='target_os="linux" target_cpu="x64"'
ninja -C out/linux_debug_x64 peerconnection_server
./out/linux_debug_x64/peerconnection_server
```

最后的命令是运行 PeerConnection Server 程序。为了让客户端能访问到这个服务器，我们需要确保运行程序的服务器防火墙开放了 TCP 8888 端口。

2.9　运行 Windows PeerConnection Client

Windows 端 PeerConnection Client 程序的编译和运行都非常简单，编译命令如下：

```
# cd 到放置 WebRTC Windows 代码库的目录的 src 子目录中
gn gen out\x86\debug_clang_vs2017 --args='target_cpu="x86"' --ide=vs2017
ninja -C out\x86\debug_clang_vs2017 peerconnection_client
```

执行成功后，找到 out\x86\debug_clang_vs2017 目录中的 peerconnection_client.exe，双击即可运行，启动界面如图 2-24 所示。

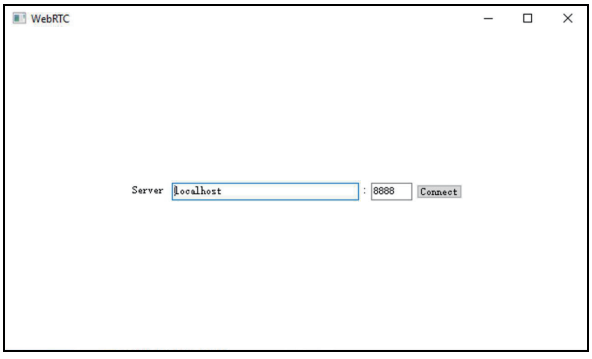


图 2-24　PeerConnection Client 启动界面

输入正确的 PeerConnection Server IP 和端口后，点击“Connect”即可连接到 PeerConnection Server，进入 peer 列表界面。如果之前或之后有其他 Client 连接到 Server，就会在列表中显示，如图 2-25 所示。

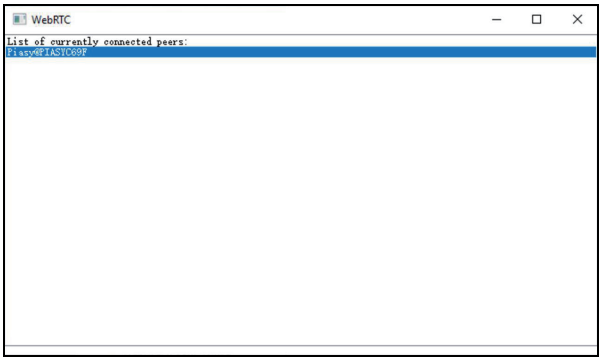


图 2-25　peer 列表界面

双击列表中的一项，即可和该 peer 发起 P2P 音视频通话。

Windows 和 Linux PeerConnection Client 互通的屏幕截图如图 2-26 和图 2-27 所示。

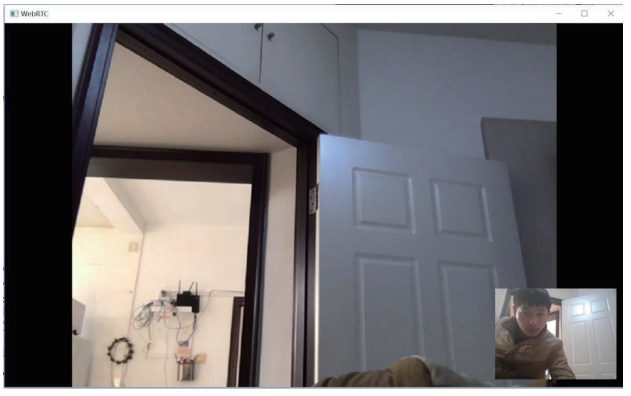


图 2-26　Windows 端 PeerConnection Client

图 2-27　Linux 端 PeerConnection Client

Linux 端的本地预览（小窗）和 Windows 端的远程视频（大窗）都存在画面偏蓝的问题，应该是 Linux 端对 YUV 数据的处理出了一点问题导致的。

2.10　运行 Linux PeerConnection Client

Linux 端 PeerConnection Client 程序的编译和运行与 Windows 端一样简单：

```
# cd 到放置 WebRTC Android/Linux 代码库目录的 src 子目录中
gn gen out/linux_debug_x64 --args='target_os="linux" target_cpu="x64"'
ninja -C out/linux_debug_x64 peerconnection_client
```

Linux 端 PeerConnection Client 的界面和操作与 Windows 端完全一致，参考 2.9 节即可。

第 3 章

基本流程分析

在上一章里我们成功运行了 WebRTC 各个平台的 Demo 程序，本章将分析一下两端音视频通话的基本调用流程，同时也熟悉一下 WebRTC 的核心 API。

3.1 拓扑结构

AppRTCMobile 的拓扑结构如图 3-1 所示。

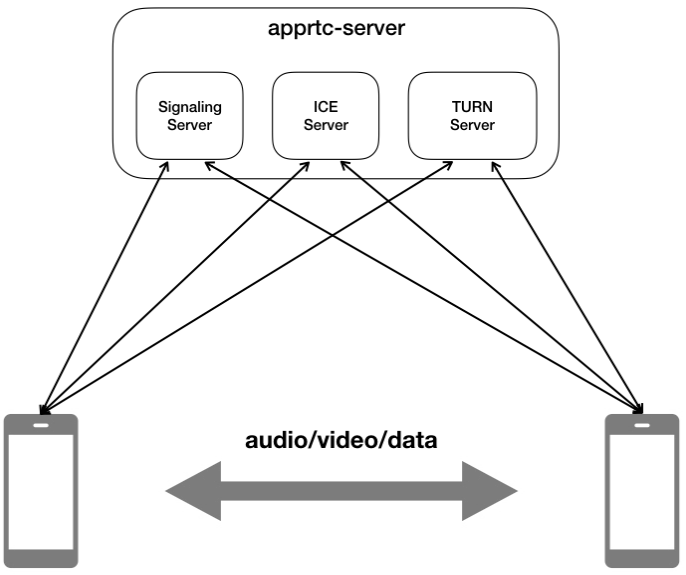


图 3-1 AppRTCMobile 拓扑结构

apprtc-server 这个 Docker 镜像里打包了 3 种服务器程序：

- Signaling Server：用于实现房间、用户的管理，以及 WebSocket 长连接消息；前者基于 Google App Engine 的 Python SDK 实现，后者基于 Golang 实现。
- ICE Server：用于下发 STUN/TURN Server 的 url、用户名、密码信息，只是一个简单的 NodeJS 程序。
- TURN Server：ICE 协议里的 TURN Server，供客户端获取公网 IP，也能提供媒体数据中转服务，这里使用的是 coturn（https://github.com/coturn/coturn）开源项目。

客户端会依次访问这 3 种服务器程序，完成加入房间、获取 TURN Server 配置、执行 ICE 协议过程的逻辑。

3.2　通话过程

了解了拓扑结构后，让我们看一下音视频通话的过程，宏观上两个客户端实现音视频通话的步骤如图 3-2 所示。

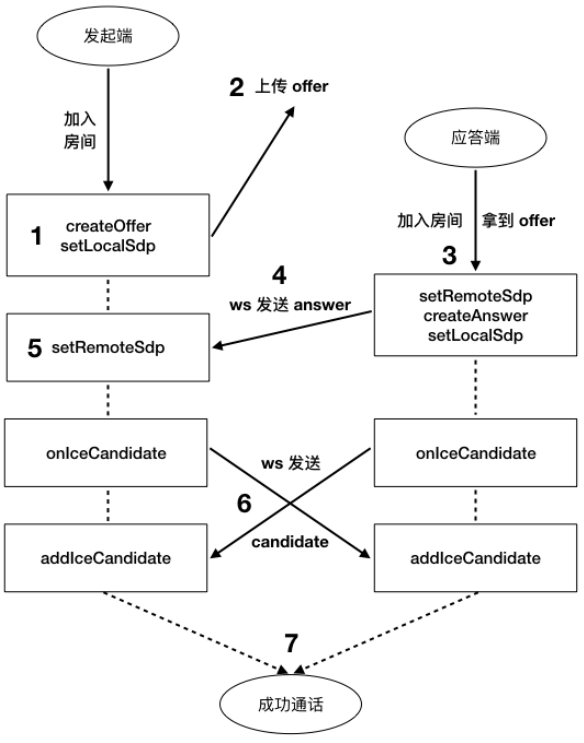


图 3-2　音视频通话的步骤

步骤 01　发起端创建本地 PeerConnection（简称 PC）对象，并创建 Offer。

步骤 02　发起端通过 Signaling Server（HTTP 服务）把 Offer 送到应答端。

步骤03 应答端创建本地 PC 对象，把发起端的 Offer 设置给 PC，然后获得 Answer。

步骤04 应答端通过 Signaling Server（长连接）把 Answer 发给发起端。

步骤05 发起端把应答端的 Answer 设置给 PC。

步骤06 两端都收集本地 PC 的 ICE Candidate(包括访问 TURN Server)，通过 Signaling Server（长连接）发送给对端，对端把 ICE Candidate 设置给本地的 PC。

步骤07 两端开始建立 P2P 的 Socket，并收发音视频数据。

接下来我们做一个简单的名词解释。

3.2.1 PeerConnection

WebRTC 的初心可以说是为浏览器带来无插件化的 P2P 媒体通信解决方案。这个 P2P 的解决方案核心类是 PeerConnection，通常简称 PC。

3.2.2 Offer、Answer 和 SDP

Offer 和 Answer 都属于 SDP（Session Description Protocol）。顾名思义，SDP 是一种描述会话（Session）的协议。一次电话会议、一次网络电话、一次视频流传输等都是一次会话。那么会话需要哪些描述呢？最基础的有多媒体数据格式和网络传输地址，当然还包括很多其他的配置信息。

为什么需要描述会话？因为参与会话的各个成员能力不对等。有人可能会想到使用所有人都支持的媒体格式，我们暂且不考虑这样的格式是否存在，而是思考另一个问题：如果参与本次会话的成员都可以支持更高质量的通话，那么使用通用的、普通质量的格式是不是很亏？既然无法使用固定的配置，那么对会话的描述就很有必要了。

另外，一次会话用什么配置也不是由某一个人说了算，必须所有人的意见达成一致，这样才能保证所有人都能参与会话。这就涉及一个协商的过程了，会话发起者先提出一些建议（Offer），其他参与者再根据 Offer 给出自己的选择（Answer），最终意见达成一致后才能开始会话。意见不一致怎么办？WebRTC 的处理方式就是报错了。

3.2.3 ICE

ICE 是用于 UDP 媒体传输的 NAT 穿透协议（适当扩展也能支持 TCP 协议），是对上述 Offer/Answer 模型的扩展，会利用 STUN、TURN 协议完成工作。ICE 会在 SDP 中增加传输地址记录值（IP+port+协议），然后对其进行连通性测试，测试通过之后就可以用于传输媒体数据了。

3.2.4 STUN

STUN 只是 NAT 穿透的一套工具，并非完整解决方案，它提供了获取一个内网连接（IP+port）对应的公网连接映射关系（NAT Binding）的机制，也提供了 NAT Binding 保活机制。WebRTC 里就用到了这两种机制。

3.2.5　TURN

TURN 协议是 STUN 协议的一个扩展，允许一个 peer 只使用一个 relay address 就可以和多个 peer 实现通信。怎么实现的呢？就是为每个 peer 分配一个中继地址，其他 peer 向 A 的中继地址发数据，TURN Server 就会把数据转发给 A。

这其实不是 P2P，因为数据都经过了一次 TURN Server 的中转，但在有些情况下，必须借助中转才能实现通信，具体在后面会展开介绍。

3.2.6　ICE Candidate

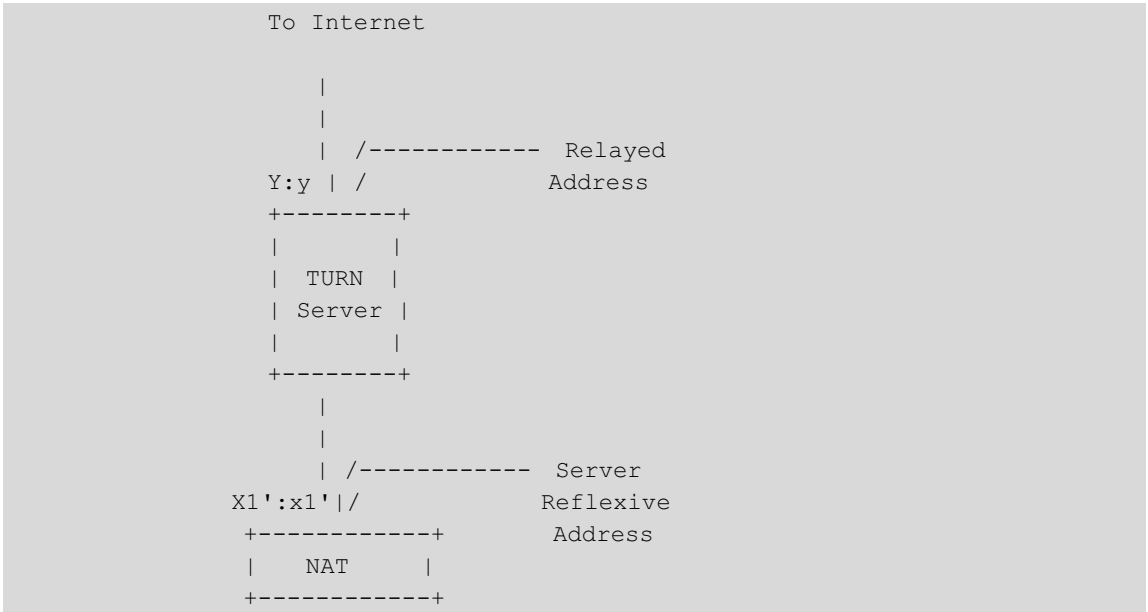
每个传输地址记录值都是一个 ICE Candidate，可能有以下 4 种：

- 客户端从本机网络接口上获取的地址（host）。
- STUN Server 看到的该客户端的地址（server reflexive，缩写为 srflx）。
- TURN Server 为该客户端分配的中继地址（relayed）。
- 连通性测试过程中，在来自对方的数据报文里看到的地址（peer reflexive，缩写为 prflx）。

两个客户端上述 Candidate 的任意组合也许都能连通，但实际上很多组合都不可用，例如 L、R 两个客户端处于两个不同的 NAT 网络后面时，网络接口地址都是内网地址，显然无法连通。ICE 的任务就是找出哪些组合可以连通。怎么找？逐个尝试！只不过要有条理地按照某种顺序去尝试。

网络接口地址对应的端口号是客户端自己分配的，如果有多个网络接口地址，就要带着（不是瞎猜哪个地址可用）。TURN Server 可以同时取得 srflx 和 relayed Candidate，而 STUN Server 只能取得 srflx Candidate（所以 coturn 是一个 TURN Server）。

前 3 种 Candidate 地址的关系如下：（引用自 RFC 5245）



```
                |
                | /------------ Local
              X:x |/             Address
              +--------+
              |        |
              | Agent  |
              |        |
              +--------+
```

3.3 核心 API 和 Demo 代码位置

在本节我们会介绍 WebRTC 核心的 API。为了方便查阅源码，我们也会给出各端 Demo 调用这些核心 API 的代码位置。

3.3.1 全局初始化

任何平台的客户端在使用 WebRTC 的 API 之前都需要进行一些初始化操作，主要是设置实验性功能开关、初始化 SSL，当然也可以启用 trace、设置日志输出等。

1. iOS 端初始化

iOS 端的初始化代码在 examples/objc/AppRTCMobile/ios/ARDAppDelegate.m 的 application:didFinishLaunchingWithOptions:函数（iOS App 进程启动完成的回调）里。

```
NSDictionary *fieldTrials = @{};
RTCInitFieldTrialDictionary(fieldTrials);
RTCInitializeSSL();
RTCSetupInternalTracer();
```

实验性功能（fieldTrials）的设置我们在后面的章节做介绍，这里暂且跳过。

2. macOS 端初始化

macOS 端的初始化代码在 examples/objc/AppRTCMobile/mac/APPRTCAppDelegate.m 的 applicationDidFinishLaunching:函数（macOS App 进程启动完成的回调）里。

```
RTCInitializeSSL();
```

macOS 端只初始化了 SSL，没有开启任何实验性功能。

3. Android 端初始化

Android 端的初始化代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的构造函数里，会在 activity 的 onCreate 函数中被构造。

```
final String fieldTrials = ...;
PeerConnectionFactory.initialize(
    PeerConnectionFactory.InitializationOptions.builder(appContext)
```

```
            .setFieldTrials(fieldTrials)
            .setEnableInternalTracer(true)
            .createInitializationOptions());
```

Android 端的 SDK 对初始化过程进行了封装，我们可以通过 PeerConnectionFactory.InitializationOptions 对象的各个字段控制初始化的过程。其内部实现包括加载动态库（libjingle_peerconnection_so.so）、设置实验性功能开关、初始化 SSL 等。

4. Windows 端初始化

Windows 端的初始化代码在 examples/peerconnection/client/main.cc 的 wWinMain 函数（程序入口函数）里。

```
    rtc::WinsockInitializer winsock_init;
    rtc::Win32SocketServer w32_ss;
    rtc::Win32Thread w32_thread(&w32_ss);
    rtc::ThreadManager::Instance()->SetCurrentThread(&w32_thread);

    webrtc::field_trial::InitFieldTrialsFromString(FLAG_force_fieldtrials);

    rtc::InitializeSSL();
```

Windows 端和 Linux 端的初始化里还需要先准备 SocketServer 线程，然后才能使用其他接口。

5. Linux 端初始化

Linux 端的初始化代码在 examples/peerconnection/client/linux/main.cc 的 main 函数（程序入口函数）里。

```
    webrtc::field_trial::InitFieldTrialsFromString(FLAG_force_fieldtrials);

    CustomSocketServer socket_server(&wnd);
    rtc::AutoSocketServerThread thread(&socket_server);

    rtc::InitializeSSL();
```

3.3.2　PeerConnectionFactory

在初始化之后、使用 PC 之前，我们需要先创建和初始化 PeerConnectionFactory 对象，因为 PC 的创建使用了工厂模式。

1. iOS/macOS 端创建 PC Factory

iOS、macOS 对 WebRTC 核心 API 的调用都是同一套代码。创建 PC Factory 的代码在 examples/objc/AppRTCMobile/ARDAppClient.m 的 connectToRoomWithId:settings:isLoopback:函数里，这个函数会在 ViewController 显示之前被调用。

```
    RTCDefaultVideoDecoderFactory *decoderFactory =
[[RTCDefaultVideoDecoderFactory alloc] init];
    RTCDefaultVideoEncoderFactory *encoderFactory =
[[RTCDefaultVideoEncoderFactory alloc] init];
```

```
    _factory = [[RTCPeerConnectionFactory alloc]
initWithEncoderFactory:encoderFactory decoderFactory:decoderFactory];
```

iOS/macOS 创建 PC Factory 时指定了视频编解码 factory，没有特殊需求时，使用 WebRTC 内置的默认实现即可。

2. Android 端创建 PC Factory

Android 端创建 PC Factory 的代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 createPeerConnectionFactoryInternal 函数里，这个函数会在 activity 的 onCreate 函数中被调用。

```
    final AudioDeviceModule adm = createJavaAudioDevice();

    final boolean enableH264HighProfile =
       VIDEO_CODEC_H264_HIGH.equals(peerConnectionParameters.videoCodec);
    final VideoEncoderFactory encoderFactory;
    final VideoDecoderFactory decoderFactory;

    if (peerConnectionParameters.videoCodecHwAcceleration) {
      encoderFactory = new DefaultVideoEncoderFactory(
         rootEglBase.getEglBaseContext(), true /* enableIntelVp8Encoder */,
enableH264HighProfile);
      decoderFactory = new
DefaultVideoDecoderFactory(rootEglBase.getEglBaseContext());
    } else {
      encoderFactory = new SoftwareVideoEncoderFactory();
      decoderFactory = new SoftwareVideoDecoderFactory();
    }

    factory = PeerConnectionFactory.builder()
                .setOptions(options)
                .setAudioDeviceModule(adm)
                .setVideoEncoderFactory(encoderFactory)
                .setVideoDecoderFactory(decoderFactory)
                .createPeerConnectionFactory();
```

逻辑和 iOS/macOS 端是基本一样的，但 Android 端除了视频编解码 factory 之外，还有更多组件可以设置。这些组件在后续涉及时再展开介绍，这里先跳过。

3. Windows/Linux 端创建 PC Factory

Windows、Linux 对 WebRTC 核心 API 的调用都是同一套代码。创建 PC Factory 的代码在 examples/peerconnection/client/conductor.cc 的 Conductor::InitializePeerConnection 函数里，这个函数会在用户呼叫对方（或收到对方呼叫信息）时调用。

```
    peer_connection_factory_ = webrtc::CreatePeerConnectionFactory(
       nullptr /* network_thread */, nullptr /* worker_thread */,
       nullptr /* signaling_thread */, nullptr /* default_adm */,
       webrtc::CreateBuiltinAudioEncoderFactory(),
       webrtc::CreateBuiltinAudioDecoderFactory(),
```

```
    webrtc::CreateBuiltinVideoEncoderFactory(),
    webrtc::CreateBuiltinVideoDecoderFactory(), nullptr /* audio_mixer */,
    nullptr /* audio_processing */);
```

其实 WebRTC 核心的 API 都是 C++实现，Java/Objective-C 都只是一层包装，由于 Google 官方对 Java/Objective-C 的实际使用少一些，因此这层包装肯定会比 C++ API 要落后一些。

3.3.3　创建 PeerConnection

PeerConnection 类是 WebRTC 里名副其实的主角，所以我们要分好几小节来介绍，首先我们看一下 PC 对象的创建。

1. iOS/macOS 端创建 PC

iOS/macOS 端创建 PC 的代码在 examples/objc/AppRTCMobile/ARDAppClient.m 的 startSignalingIfReady 函数里，这个函数会在客户端访问 Signaling Server 和 ICE Server 完成后被调用，即拿到了加入房间、建立通话所需的信息后调用。

```
RTCMediaConstraints *constraints = [self defaultPeerConnectionConstraints];
RTCConfiguration *config = [[RTCConfiguration alloc] init];
RTCCertificate *pcert = [RTCCertificate generateCertificateWithParams:@{
   @"expires" : @100000,
   @"name" : @"RSASSA-PKCS1-v1_5"
}];
config.iceServers = _iceServers;
config.sdpSemantics = RTCSdpSemanticsUnifiedPlan;
config.certificate = pcert;

_peerConnection = [_factory peerConnectionWithConfiguration:config
                                        constraints:constraints
                                          delegate:self];
```

创建 PC 时，我们提供了 3 个参数：

- RTCConfiguration：创建 PC 对象时的主要配置参数，包括 IceServer 信息、SDP 语义版本等，内容较多，后面的章节涉及时再展开介绍。
- RTCMediaConstraints：创建 PC 对象时的媒体相关配置，后面的章节涉及时再展开介绍。
- RTCPeerConnectionDelegate：PC 相关事件的回调，比如连接状态变化回调、拿到本地 ICE Candidate 后的回调等，内容也不少，后面的章节涉及时再展开介绍。

2. Android 端创建 PC

Android 端创建 PC 的代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 createPeerConnectionInternal 函数里，这个函数会在客户端访问 Signaling Server 和 ICE Server 完成后被调用，即拿到了加入房间、建立通话所需的信息后调用。

```
PeerConnection.RTCConfiguration rtcConfig =
   new PeerConnection.RTCConfiguration(signalingParameters.iceServers);
// TCP candidates are only useful when connecting to a server that supports
```

```
    // ICE-TCP.
    rtcConfig.tcpCandidatePolicy = PeerConnection.TcpCandidatePolicy.DISABLED;
    rtcConfig.bundlePolicy = PeerConnection.BundlePolicy.MAXBUNDLE;
    rtcConfig.rtcpMuxPolicy = PeerConnection.RtcpMuxPolicy.REQUIRE;
    rtcConfig.continualGatheringPolicy =
PeerConnection.ContinualGatheringPolicy.GATHER_CONTINUALLY;
    // Use ECDSA encryption.
    rtcConfig.keyType = PeerConnection.KeyType.ECDSA;
    // Enable DTLS for normal calls and disable for loopback calls.
    rtcConfig.enableDtlsSrtp = !peerConnectionParameters.loopback;
    rtcConfig.sdpSemantics = PeerConnection.SdpSemantics.UNIFIED_PLAN;

    peerConnection = factory.createPeerConnection(rtcConfig, pcObserver);
```

逻辑和 iOS/macOS 端是基本一样的。PeerConnection.RTCConfiguration 和 PeerConnection.Observer 也留在后面的章节展开介绍。

3. Windows/Linux 端创建 PC

Android 端创建 PC 的代码在 examples/peerconnection/client/conductor.cc 的 Conductor::CreatePeerConnection 函数里，这个函数会在 Demo 首页 peer 列表元素被双击或收到其他 peer 的呼叫信息后被调用。

```
    webrtc::PeerConnectionInterface::RTCConfiguration config;
    config.sdp_semantics = webrtc::SdpSemantics::kUnifiedPlan;
    config.enable_dtls_srtp = dtls;
    webrtc::PeerConnectionInterface::IceServer server;
    server.uri = GetPeerConnectionString();
    config.servers.push_back(server);

    peer_connection_ = peer_connection_factory_->CreatePeerConnection(
        config, nullptr, nullptr, this);
```

逻辑和 iOS/macOS 端是基本一样的。RTCConfiguration 和 PeerConnectionObserver 也留在后面的章节展开介绍。

3.3.4 创建 Source 和 Track

创建 PC 的目的是为了收发音视频数据，收发的载体就是 Track，而 Track 的数据则来自于 Source。我们接下来就看看创建 Source 和 Track 的代码。

1. iOS/macOS 端创建 Source 和 Track

iOS/macOS 创建 Source 和 Track 的代码在 examples/objc/AppRTCMobile/ARDAppClient.m 的 createMediaSenders 函数里，创建完 PC 后就会被立即调用。

首先我们看一下音频 Source 和 Track 的创建：

```
    RTCMediaConstraints *constraints = [self defaultMediaAudioConstraints];
    RTCAudioSource *source = [_factory audioSourceWithConstraints:constraints];
    RTCAudioTrack *track = [_factory audioTrackWithSource:source
```

```
                                                trackId:kARDAudioTrackId];
    [_peerConnection addTrack:track streamIds:@[ kARDMediaStreamId ]];
```

音频比较简单，只需要调用 PC Factory 的接口创建 Source 和 Track，然后把 Track 添加到 PC 中即可。

接下来我们看一下视频 Source 和 Track 的创建：

```
    RTCVideoSource *source = [_factory videoSource];
    RTCVideoTrack *track = [_factory videoTrackWithSource:source
trackId:kARDVideoTrackId];
    [_peerConnection addTrack:track streamIds:@[ kARDMediaStreamId ]];
```

创建对象和音频完全一样，但视频还有两个特殊之处：视频的采集是一个独立的模块，叫作 VideoCapturer，此外视频还需要本地预览。

创建 VideoCapturer、启动预览的代码分布在 3 处，这里将其合并到一起，以便于理解：

```
    _localVideoView = [[RTCCameraPreviewView alloc] initWithFrame:CGRectZero];

    RTCCameraVideoCapturer *capturer = [[RTCCameraVideoCapturer alloc]
initWithDelegate:source];

    _videoCallView.localVideoView.captureSession =
localCapturer.captureSession;
    ARDSettingsModel *settingsModel = [[ARDSettingsModel alloc] init];
    _captureController =
      [[ARDCaptureController alloc] initWithCapturer:localCapturer
settings:settingsModel];
    [_captureController startCapture];
```

创建 Capturer 时，我们需要传入一个 Source 对象，因为 Capturer 会把采集到的数据交给 Source（用于发送）。此外 Capturer 采集到的数据还需要进行预览，iOS 的预览是把 AVCaptureSession 设置给 RTCCameraPreviewView，在其中通过系统 API 自动把视频数据渲染到预览视图上。

上面 3 处代码的位置分别是：

- 创建预览视图：examples/objc/AppRTCMobile/ios/ARDVideoCallView.m 的 initWithFrame:函数。
- 创建 Capturer：examples/objc/AppRTCMobile/ARDAppClient.m 的 createLocalVideoTrack 函数。
- 启动 Capturer，关联 Capturer 和预览：examples/objc/AppRTCMobile/ios/ARDVideoCallViewController.m 的 appClient:didCreateLocalCapturer:函数。

2. Android 端创建 Source 和 Track

Android 创建 Source 和 Track 的代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 createPeerConnectionInternal 函数里，创建完 PC 后就会被立即执行。

首先我们看一下音频 Source 和 Track 的创建：

```
    List<String> mediaStreamLabels = Collections.singletonList("ARDAMS");
    peerConnection.addTrack(createAudioTrack(), mediaStreamLabels);

    private AudioTrack createAudioTrack() {
      audioSource = factory.createAudioSource(audioConstraints);
```

```
      localAudioTrack = factory.createAudioTrack(AUDIO_TRACK_ID, audioSource);
      localAudioTrack.setEnabled(enableAudio);
      return localAudioTrack;
    }
```

和 iOS/macOS 一样，只需要调用 PC Factory 的接口创建 Source 和 Track，然后把 Track 添加到 PC 中即可。

创建视频 Source 和 Track 的代码，核心逻辑一致，但涉及更多的类：

```
    peerConnection.addTrack(createVideoTrack(videoCapturer),
mediaStreamLabels);

    private VideoTrack createVideoTrack(VideoCapturer capturer) {
      surfaceTextureHelper =
          SurfaceTextureHelper.create("CaptureThread",
rootEglBase.getEglBaseContext());
      videoSource = factory.createVideoSource(capturer.isScreencast());
      capturer.initialize(surfaceTextureHelper, appContext,
videoSource.getCapturerObserver());
      capturer.startCapture(videoWidth, videoHeight, videoFps);

      localVideoTrack = factory.createVideoTrack(VIDEO_TRACK_ID, videoSource);
      localVideoTrack.setEnabled(renderVideo);
      localVideoTrack.addSink(localRender);
      return localVideoTrack;
    }
```

Android 的 Capturer 比 iOS/macOS 稍微复杂一些，因为 Android 的 Camera API 有两套：

```
    if (useCamera2) {
      videoCapturer = createCameraCapturer(new Camera2Enumerator(this));
    } else {
      videoCapturer = createCameraCapturer(new
Camera1Enumerator(captureToTexture()));
    }

    private VideoCapturer createCameraCapturer(CameraEnumerator enumerator) {
      final String[] deviceNames = enumerator.getDeviceNames();
      return enumerator.createCapturer(deviceNames[0], null);
    }
```

上述代码是 Android Demo 的简化版本，完整代码在 examples/androidapp/src/org/appspot/apprtc/CallActivity.java 的 createVideoCapturer 函数中，它会在创建 PC 前执行。不过 Capturer 和 PC 的创建顺序不重要，可以随意调换。

SurfaceTextureHelper 是用于采集的辅助类，这里暂且跳过。

localRender 是用于本地预览的对象。和 iOS/macOS 不一样的是，Android 的本地预览需要添加给 Track，而非利用系统 API“一键预览”。视频数据传递的路径是 Capturer->Source->Track->Sink(Preview)。

Android 的预览也做了一点特殊处理：在实现本地预览和收远端流的位置切换时，Android

Demo 不是交换视图位置，而是把视图的数据源做交换，具体逻辑在 CallActivity.java 中，由于不是核心流程，因此这里就不做展开了。

3. Windows/Linux 端创建 Source 和 Track

Windows/Linux 创建 Source 和 Track 的代码在 examples/peerconnection/client/conductor.cc 的 Conductor::AddTracks 函数里，创建完 PC 后就会被立即调用。

首先我们看一下音频 Source 和 Track 的创建：

```
    rtc::scoped_refptr<webrtc::AudioTrackInterface> audio_track(
       peer_connection_factory_->CreateAudioTrack(
          kAudioLabel, peer_connection_factory_->CreateAudioSource(
                          cricket::AudioOptions())));
    auto result_or_error = peer_connection_->AddTrack(audio_track, {kStreamId});
```

同样，我们只需要调用 PC Factory 的接口创建 Source 和 Track，然后把 Track 添加到 PC 中即可。

接下来我们看一下视频 Source 和 Track 的创建：

```
    rtc::scoped_refptr<CapturerTrackSource> video_device =
       CapturerTrackSource::Create();
    if (video_device) {
      rtc::scoped_refptr<webrtc::VideoTrackInterface> video_track_(
         peer_connection_factory_->CreateVideoTrack(kVideoLabel,
video_device));
      main_wnd_->StartLocalRenderer(video_track_);

      result_or_error = peer_connection_->AddTrack(video_track_, {kStreamId});
    }
```

CapturerTrackSource 是 Windows/Linux Demo 对 WebRTC 视频采集类接口做的封装，实现了 Source 接口，所以可以用于创建 Track。它里面会创建一个 Capturer 对象，视频数据的传递路径也是 Capturer->Source->Track。具体细节代码这里不做展开，感兴趣的朋友可以阅读其源码。

3.3.5　创建 Offer

添加完 Track 之后，我们明确了需要发送哪些数据，这时就可以开始 SDP 协商的过程了，首先是发起端创建 Offer，并将其作为本地 SDP 设置给 PC 对象，然后把 Offer 通过 Signaling Server 交给应答端。

1. iOS/macOS 端创建 Offer

iOS/macOS 创建 Offer 的代码在 examples/objc/AppRTCMobile/ARDAppClient.m 的 startSignalingIfReady 函数里，添加完 Track 后就会被立即调用。

```
    __weak ARDAppClient *weakSelf = self;
    [_peerConnection offerForConstraints:[self defaultOfferConstraints]
                   completionHandler:^(RTCSessionDescription *sdp,
                                       NSError *error) {
```

```
    ARDAppClient *strongSelf = weakSelf;
    [strongSelf peerConnection:strongSelf.peerConnection
       didCreateSessionDescription:sdp
                             error:error];
  }];

  - (RTCMediaConstraints *)defaultOfferConstraints {
    NSDictionary *mandatoryConstraints = @{
      @"OfferToReceiveAudio" : @"true",
      @"OfferToReceiveVideo" : @"true"
    };
    RTCMediaConstraints* constraints =
       [[RTCMediaConstraints alloc]
          initWithMandatoryConstraints:mandatoryConstraints
                   optionalConstraints:nil];
    return constraints;
  }
```

值得指出的是，添加 Track 只是表明我们需要发送数据，那怎么表明我们是否需要接收数据呢？就靠 constraints 里的 OfferToReceiveAudio/Video 选项了。

创建完 Offer 后，我们需要将其设置给 PC 对象，并通过 Signaling Server 发送给应答端，这段代码在 peerConnection:didCreateSessionDescription:error:函数中：

```
  __weak ARDAppClient *weakSelf = self;
  [self.peerConnection setLocalDescription:sdp
                      completionHandler:^(NSError *error) {
                       // ...
                      }];
  ARDSessionDescriptionMessage *message =
     [[ARDSessionDescriptionMessage alloc] initWithDescription:sdp];
  [self sendSignalingMessage:message];
```

2. Android 端创建 Offer

Android 创建 Offer 的代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 createOffer 函数里，也是添加完 Track 后就会被立即调用。

```
  // Create SDP constraints.
  sdpMediaConstraints = new MediaConstraints();
  sdpMediaConstraints.mandatory.add(
     new MediaConstraints.KeyValuePair("OfferToReceiveAudio", "true"));
  sdpMediaConstraints.mandatory.add(new MediaConstraints.KeyValuePair(
     "OfferToReceiveVideo", Boolean.toString(isVideoCallEnabled())));

  peerConnection.createOffer(sdpObserver, sdpMediaConstraints);
```

同样，我们也是通过 constraints 里的 OfferToReceiveAudio/Video 选项来说明是否接收数据的。

创建完 Offer 后设置给 PC 对象的代码在 SDPObserver 类的 onCreateSuccess 回调中：

```
  peerConnection.setLocalDescription(sdpObserver, sdp);
```

Android Demo 里在设置 SDP 之前还对 SDP 做了一个 preferCodec 的处理，基本逻辑就是调整

SDP 里音视频编码的排列顺序。更多关于 SDP 的内容，请阅读 3.4 节。

Android Demo 发送 Offer 的代码在 examples/androidapp/src/org/appspot/apprtc/CallActivity.java 的 onLocalDescription 函数中，由 SDPObserver 类的 onSetSuccess 回调触发。

3. Windows/Linux 端创建 Offer

Windows/Linux 创建 Offer 的代码在 examples/peerconnection/client/conductor.cc 的 Conductor::ConnectToPeer 函数里，也是添加完 Track 后就会被立即调用。

```
peer_connection_->CreateOffer(
    this, webrtc::PeerConnectionInterface::RTCOfferAnswerOptions());
```

注意，这里我们没有设置任何选项表明是否接收数据，因为 AddTrack 默认情况下也会认为需要接收数据，所以 iOS、macOS、Android 其实都不需要显式设置接收数据。关于是否接收数据的内容，请阅读 3.4 节。

创建成功的回调为 Conductor::OnSuccess 函数，其中包含了把 SDP 设置给 PC 和把 Offer 发送给应答端的逻辑：

```
peer_connection_->SetLocalDescription(
    DummySetSessionDescriptionObserver::Create(), desc);

Json::StyledWriter writer;
Json::Value jmessage;
jmessage[kSessionDescriptionTypeName] =
    webrtc::SdpTypeToString(desc->GetType());
jmessage[kSessionDescriptionSdpName] = sdp;
SendMessage(writer.write(jmessage));
```

3.3.6　创建 Answer

应答端拿到发起端的 Offer 后，先将其设置给 PC 对象，然后创建 Answer 并设置给 PC 对象，最后将 Answer 通过 Signaling Server 发送到发起端，发起端拿到 Answer 之后，也需要把 Answer 设置给 PC 对象。

1. iOS/macO 端创建 Answer

首先我们看一下设置 Offer 的代码。iOS/macOS 的代码在 examples/objc/AppRTCMobile/ARDAppClient.m 的 processSignalingMessage: 函数里。这个函数用于处理来自 Signaling Server 的消息。

```
__weak ARDAppClient *weakSelf = self;
[_peerConnection setRemoteDescription:description
                    completionHandler:^(NSError *error) {
                      ARDAppClient *strongSelf = weakSelf;
                      [strongSelf peerConnection:strongSelf.peerConnection
                          didSetSessionDescriptionWithError:error];
                    }];
```

设置成功后，会在 peerConnection:didSetSessionDescriptionWithError: 函数里创建 Answer：

```
    RTCMediaConstraints *constraints = [self defaultAnswerConstraints];
    __weak ARDAppClient *weakSelf = self;
    [self.peerConnection answerForConstraints:constraints
                        completionHandler:^(RTCSessionDescription *sdp, NSError
*error) {
                          ARDAppClient *strongSelf = weakSelf;
                          [strongSelf peerConnection:strongSelf.peerConnection
                             didCreateSessionDescription:sdp
                                                   error:error];
                        }];
```

创建 Answer 成功后，设置给 PC、发送给发起端的代码和创建 Offer 成功后的处理代码是同一个函数，这里就不展开了。

同样的，发起端收到 Answer 后，也是调用 setRemoteDescription 设置给 PC 对象。代码和应答端设置 Offer 是一样的，这里也不做展开。

2. Android 端创建 Answer

Android 设置 Offer 的代码在 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 setRemoteDescription 函数里，这个函数会在收到来自 Signaling Server 的消息后被调用。

```
    peerConnection.setRemoteDescription(sdpObserver, sdpRemote);
```

Android Demo 创建 Answer 没有等待设置 Offer 成功，这样做并不会产生问题，因为 PC 的接口调用都是切换到工作线程异步串行执行的(工作线程是单一线程)，所以 PC 实际执行创建 Answer 时，Offer 一定已经设置完成了。创建 Answer 的调用在 PeerConnectionClient.java 的 createAnswer 函数里：

```
    peerConnection.createAnswer(sdpObserver, sdpMediaConstraints);
```

创建 Answer 成功后，设置给 PC、发送给发起端的代码和创建 Offer 成功后的处理代码是同一个函数，这里就不展开了。

同样的，发起端收到 Answer 后，也是调用 setRemoteDescription 设置给 PC 对象，代码和应答端设置 Offer 是一样的，这里也不做展开。

3. Windows/Linux 端创建 Answer

Windows/Linux 设置 Offer 的代码在 examples/peerconnection/client/conductor.cc 的 Conductor::OnMessageFromPeer 函数里。这个函数用于处理来自 Signaling Server 的消息。

```
    peer_connection_->SetRemoteDescription(
        DummySetSessionDescriptionObserver::Create(),
        session_description.release());
```

和 Android Demo 一样，Windows/Linux Demo 创建 Answer 也没有等待 Offer 设置成功回调。创建 Answer 的代码如下：

```
    peer_connection_->CreateAnswer(
        this, webrtc::PeerConnectionInterface::RTCOfferAnswerOptions());
```

创建 Answer 成功后，设置给 PC、发送给发起端的代码和创建 Offer 成功后的处理代码是同一

个函数，这里就不展开了。

同样的，发起端收到 Answer 后，也是调用 setRemoteDescription 设置给 PC 对象。代码和应答端设置 Offer 是一样的，这里也不做展开。

3.3.7　ICE Candidate 回调和设置

通话双方交换并设置了 SDP 之后，下一步就是交换并设置 ICE Candidate、建立 P2P 连接了。

1. iOS/macOS ICE Candidate 回调和设置

iOS/macOS 收到本地 ICE Candidate 的回调函数为 examples/objc/AppRTCMobile/ARDAppClient.m 的 peerConnection:didGenerateIceCandidate: 函数，它是 RTCPeerConnectionDelegate 的回调函数之一，设置本地 SDP（setLocalDescription）会启动 ICE Candidate 收集过程，收集到之后会回调该函数。

```
ARDICECandidateMessage *message =
    [[ARDICECandidateMessage alloc] initWithCandidate:candidate];
[self sendSignalingMessage:message];
```

可以看到，收集到 ICE Candidate 之后我们会通过 Signaling Server 发送给对端。对端收到 ICE Candidate 后的处理在 ARDAppClient.m 的 processSignalingMessage: 函数里：

```
ARDICECandidateMessage *candidateMessage =
  (ARDICECandidateMessage *)message;
[_peerConnection addIceCandidate:candidateMessage.candidate];
```

我们通过 addIceCandidate 函数把对端的 ICE Candidate 设置给本地的 PC 对象。

值得指出的是，收到 Signaling Server 的消息和本地创建 PC 对象是无法保证先后顺序的，所以 iOS/macOS Demo 层引入了一个消息队列，防止消息丢失。

2. Android ICE Candidate 回调和设置

Android 收到本地 ICE Candidate 的回调函数为 examples/androidapp/src/org/appspot/apprtc/PeerConnectionClient.java 的 PCObserver 类的 onIceCandidate 函数，它是 PeerConnection.Observer 的回调函数之一。设置本地 SDP（setLocalDescription）会启动 ICE Candidate 收集过程，之后会回调该函数。它会进一步回调 CallActivity.java 的 onIceCandidate 函数，并会把 ICE Candidate 发送给对端：

```
appRtcClient.sendLocalIceCandidate(candidate);
```

对端收到 ICE Candidate 后，会在 PeerConnectionClient.java 的 addRemoteIceCandidate 函数里添加给 PC 对象：

```
peerConnection.addIceCandidate(candidate);
```

同样的，由于收到 Signaling Server 的消息和本地创建 PC 对象是无法保证先后顺序的，因此 Android Demo 层也引入了一个消息队列，防止消息丢失。

3. Windows/Linux ICE Candidate 回调和设置

Windows/Linux 收到本地 ICE Candidate 的回调函数为 examples/peerconnection/client/conductor.cc 的 Conductor::OnIceCandidate 函数，它是 PeerConnectionObserver 的回调函数之一。设置本地 SDP（SetLocalDescription）会启动 ICE Candidate 收集过程，收集到之后会回调该函数。

```
Json::StyledWriter writer;
Json::Value jmessage;

jmessage[kCandidateSdpMidName] = candidate->sdp_mid();
jmessage[kCandidateSdpMlineIndexName] = candidate->sdp_mline_index();
std::string sdp;
if (!candidate->ToString(&sdp)) {
  RTC_LOG(LS_ERROR) << "Failed to serialize candidate";
  return;
}
jmessage[kCandidateSdpName] = sdp;
SendMessage(writer.write(jmessage));
```

其中的处理也是把 ICE Candidate 发送给对端。对端收到 ICE Candidate 后，会在 Conductor::OnMessageFromPeer 函数里添加给 PC 对象：

```
std::unique_ptr<webrtc::IceCandidateInterface> candidate(
    webrtc::CreateIceCandidate(sdp_mid, sdp_mlineindex, sdp, &error));
if (!peer_connection_->AddIceCandidate(candidate.get())) {
  RTC_LOG(WARNING) << "Failed to apply the received candidate";
  return;
}
```

Windows/Linux Demo 没有使用一个队列保存创建 PC 前收到的消息。如果收到消息时 PC 尚未创建，它会立即创建 PC，所以也不会有问题。

3.3.8 ICE 连接状态回调

交换了 SDP 后，通话两端就开始建立 P2P 连接了。对于这个过程的状态变化，我们可以监听 PC 的 ICE 连接状态回调。

iOS/macOS 的回调定义为：

```
// RTCPeerConnectionDelegate
- (void)peerConnection:(RTCPeerConnection *)peerConnection
    didChangeIceConnectionState:(RTCIceConnectionState)newState;
```

Android 的回调定义为：

```
// PeerConnection.Observer
void onIceConnectionChange(IceConnectionState newState);
```

Windows/Linux 的回调定义为

```
// PeerConnectionObserver
```

```
virtual void OnIceConnectionChange(
    PeerConnectionInterface::IceConnectionState new_state) = 0;
```

其中状态码的定义为：

```
enum IceConnectionState {
  kIceConnectionNew,
  kIceConnectionChecking,
  kIceConnectionConnected,
  kIceConnectionCompleted,
  kIceConnectionFailed,
  kIceConnectionDisconnected,
  kIceConnectionClosed,
  kIceConnectionMax,
};
```

几个常用的状态为：

- kIceConnectionConnected: ICE 连接建立成功的状态，这种状态下可以收发音视频数据。
- kIceConnectionDisconnected: ICE 连接断开后的状态。
- kIceConnectionFailed: ICE 连接建立失败的状态。

3.3.9　核心 API 回顾

我们已经初步了解了 WebRTC 的核心 API，现在做一个简单的回顾。

首先回顾一下几个重要的概念：

- Capturer: 负责数据采集，只有视频才有这一层抽象，它有多种实现，包括相机采集（Android 还有 Camera1/Camera2 两套）、录屏采集、视频文件采集等。
- Source: 数据源，数据来自于 Capturer。它把数据交给 Track。
- Track: 媒体数据交换的载体，发送端把本地的 Track 发送给远程的接收端。
- Sink: Track 数据的消费者，只有视频才有这一层封装。发送端视频的本地预览、接收端收到远程视频后的渲染都是 Sink。
- Transceiver: 负责收发媒体数据（以 Track 为载体。前文没有介绍 Transceiver 这个概念，我们会在第 5 章展开介绍）。

以视频为例，数据由发送端的 Capturer 采集，交给 Source，再交给本地的 Track，然后兵分两路：一路由本地 Sink 进行预览，一路由 Transceiver 发送给接收端。接收端 Track 把数据交给 Sink 渲染。

Capturer 的创建和销毁完全由 App 层负责，只需要把它和 Source 关联起来即可；创建 Source 需要调用 PC Factory 接口，创建 Track 也是，并且需要提供 Source 参数；Sink 的创建和销毁也由 App 层负责，只需要把它们添加到 Track 里即可；创建 Transceiver 需要调用 PC 接口（前文我们调用的 AddTrack 接口，内部会创建 Transceiver）。

接下来我们回顾一下进行音视频通话的关键 4 步：

- 创建 PC Factory，创建 PC。

- 创建 Capturer、Source、Track、Transceiver。
- 创建 Offer、Answer，并互相交换。
- 互相交换 ICE Candidate。

3.4 SDP 初探

前文做名词解释的时候我们已经简单了解了 SDP：用来协商会话能力的协议，通过 Offer/Answer 机制进行协商。在本节里，我们将掀开 WebRTC SDP 的神秘面纱。

首先我们看一个 WebRTC SDP 的实际例子：

```
    o=- 0 0 IN IP4 127.0.0.1
    s=IntelWebRTCMCU
    t=0 0
    a=group:BUNDLE 0 1
    a=msid-semantic: WMS yDsbzayy4c
    m=audio 1 UDP/TLS/RTP/SAVPF 111
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=candidate:1 1 udp 2013266431 192.168.50.50 20000 typ host
    a=ice-ufrag:v5mR
    a=ice-pwd:mxWxOT1thmNxVyg0ijAYTo
    a=fingerprint:sha-256
4D:58:22:D8:6A:59:AC:50:4D:B2:A3:F3:0B:BC:24:58:1B:DF:39:EB:D0:B7:D3:D3:13:37:
28:74:95:15:B9:BC
    a=setup:active
    a=recvonly
    a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
    a=mid:0
    a=rtcp-mux
    a=rtpmap:111 opus/48000/2
    a=fmtp:111 minptime=10;useinbandfec=1
    m=video 1 UDP/TLS/RTP/SAVPF 98 125
    c=IN IP4 0.0.0.0
    a=rtcp:1 IN IP4 0.0.0.0
    a=candidate:1 1 udp 2013266431 192.168.50.50 20000 typ host
    a=ice-ufrag:v5mR
    a=ice-pwd:mxWxOT1thmNxVyg0ijAYTo
    a=extmap:2 urn:ietf:params:rtp-hdrext:toffset
    a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
    a=extmap:5
http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
    a=fingerprint:sha-256
4D:58:22:D8:6A:59:AC:50:4D:B2:A3:F3:0B:BC:24:58:1B:DF:39:EB:D0:B7:D3:D3:13:37:
28:74:95:15:B9:BC
    a=setup:active
    a=recvonly
```

```
    a=mid:1
    a=rtcp-mux
    a=rtpmap:98 H264/90000
    a=rtcp-fb:98 ccm fir
    a=rtcp-fb:98 transport-cc
    a=rtcp-fb:98 nack
    a=rtcp-fb:98 goog-remb
    a=fmtp:98
level-asymmetry-allowed=1;packetization-mode=1;profile-level-id=42e01f
    a=rtpmap:125 red/90000
```

WebRTC SDP 总体上可以分为以下几个部分：

- session metadata: 描述整个会话的基本信息，行开头为 v=、o=、s=、t= 等。
- network description: 描述网络传输的信息，行开头为 c=、a=candidate 等。
- stream description: 描述媒体数据流的信息，行开头为 m=、a=rtpmap、a=fmtp、a=sendrecv 等。
- security description: 描述数据加密、身份认证相关信息，行开头为 a=crypto、a=ice-frag、a=ice-pwd、a=fingerprint 等。
- QoS, grouping description: 描述服务质量、传输层复用相关信息，行开头为 a=rtcp-fb、a=group、a=rtcpmux 等。

上面的例子就是包含了音视频数据的 SDP，初次见面，我们先挑几个简单的行认识一下：

- m= 所在的行是 media line（媒体行，又叫 m line）。两个 m line 之间（或最后一个 m line 到结尾）的行都是用来描述一种媒体数据的（叫 media section），包括数据格式、网络传输等信息。例如，以 m=audio 开头的描述的是音频数据的信息，以 m=video 开头的描述的是视频数据的信息。
- a=candidate: 所在的行描述了这个 media section 将要使用的 ICE Candidate。
- a=recvonly 所在的行描述了这个 media section 的方向，可选值包括 recvonly（只接收）、sendonly（只发送）、sendrecv（同时收发）。
- a=rtpmap: 所在的行描述了这个 media section 支持的格式，例如上面的 OPUS（音频）、H264（视频）。

其他的行，我们后文再仔细展开。