# 基 础 篇

本篇是全书的基础知识讲解，共包括 3 章。第 1 章主要介绍 Android 移动应用安全调试工具；第 2 章通过对一个 Android 应用进行逆向、破解、篡改、二次打包等操作，帮助读者快速建立起对 Android 应用逆向攻防的基础概念，形成对该技术方向的直观了解和形象记忆，为后续的理论学习打下基础；第 3 章主要介绍 iOS 相关的基础知识，重点学习 iOS 软件包结构以及 iOS 应用的启动和运行流程。

# 第 1 章 构建 Android 安全分析环境

CHAPTER 1

## 1.1 常用 adb 命令一览

adb 全称是 Android Debug Bridge，通过监听 Socket TCP 5554 等端口实现对 Android 模拟器的链接。对于真机可以在开发者选项中启动 USB 调试，通过 USB 连接到主机，并赋予主机对手机的调试权限，adb 工具就可以通过 USB 对手机进行调试与控制。

本节介绍在 Android 开发与逆向分析中常用的 adb 命令，以下都是分析 Android 应用过程中的常用命令。

安装 Apk 软件包：

```
$ adb install apk_name.apk
```

升级安装 Apk 软件包：

```
$ adb install -r apk_name.apk
```

卸载 Apk 软件包：

```
$ adb uninstall apk_name.apk
```

将本地文件推送到设备：

```
$ adb push local_dir device_dir
```

将设备中的文件拉取到本地：

```
$ adb pull device_dir local_dir
```

拉取时可能会遇到访问文件所在文件夹需要 Root 权限，这时可以先将文件转移到 /sdcard 中再进行拉取。

打印 adb 日志：

```
$ adb logcat > log.txt
```

查看指定应用的详细信息：

```
$ adb shell dumpsys package package_name
```

查看当前应用的 activity 信息：

```
$ adb shell dumpsys activity top
```

快速截取手机屏幕：

```
$ adb shell screencap -p device_dir/screen.png
```

手机录屏：

```
$ adb shell screenrecord device_dir/screen.mp4
```

设备端口转发：

```
$ adb forward tcp:23946 tcp:23946
```

adb 工具实现了主机与手机的交互，不仅可以用在手动调试应用的过程中，许多自动化动态调试工具（比如 MobSF）也会用到它。

## 1.2 NDK 命令行编译 Android 动态链接库

NDK 全称 Native Development Kit，是 Android 的一个工具开发包，通常用于开发 Android 应用调用的 C、C++ 动态库，并将动态库与应用一起打包成 Apk 包，Android 中的 Java 层通过 NDK 使用 Jni 接口与 Native 层代码进行交互。

本节介绍使用 NDK 工具中的几个简单的命令，不需要 Android Studio 等 IDE，通过命令行来编译一个 Android 动态链接库。配置好 JDK 环境与 Android SDK 环境，首先使用 Java 编写一个简单的 Android 程序。

MainActivity.java 文件的主要代码：

```
package test.example;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;
import android.widget.LinearLayout;
import android.widget.Button;

public class MainActivity extends Activity{

  static{
    System.loadLibrary("jni_test");
  }

  public native String stringFromJNI();

  public void onCreate(Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    LinearLayout lla = new LinearLayout(this);
    Button b = new Button(this);
    LinearLayout lla = new LinearLa
    Button b = new Button(this);
    b.setText("click");
    lla.addView(b);
    this.setContentView(lla);
```

```
        final Activity _this = this;

        b.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
              Toast.makeText(_this, stringFromJNI(), Toast.LENGTH_LONG).show();
             }
        });
      }
}
```

调用 javac 命令编译编写的 Java 文件，生成 class 文件：

```
javac -bootclasspath {Android_SDK_HOME}/platforms/android-28/android.jar MainActivity.java
```

使用 javah 命令生成.h 头文件，注意 javah 命令的使用以及执行位置。首先进入 class 文件所在包目录的顶层，本例中就是 test 目录的父目录，在该目录下执行下面命令：

```
javah -d ./test/example/jni/ -bootclasspath {Android_SDK_HOME}/platforms/android-28/
android.jar -classpath . test.example.MainActivity
```

在 test/example/jni 目录下会生成头文件：test_example_MainActivity.h。在 jni 目录下新建 jni.c 文件，编写 Jni 代码：

Jni.c 文件的主要代码：

```
#include <string.h>
#include <jni.h>
#include "test_example_MainActivity.h"

JNIEXPORT jstring JNICALL Java_test_example_MainActivity_stringFromJNI(JNIEnv * env, jobject _
this){
  return (*env)->NewStringUTF(env,"return from c");
}
```

在 test/example/jni 目录下新建 Android.mk 文件(注意字母大小写)，这个文件是说明如何编译动态链接库的。

Android.mk 文件的主要代码：

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := jni_test
LOCAL_SRC_FILES := jni.c
```

打开命令行，进入 test/example/jni 目录，输入以下命令：

```
{Android_NDK_HOME}/ndk-build
```

如图 1.1 所示为使用 ndk-build 编译 so 文件的输出。

此时，项目下会产生 libs 目录，该目录中就是生成的动态链接库。因为 Android 支持多

```
Android NDK: APP_PLATFORM not set. Defaulting to minimum supported version android-14.
[arm64-v8a] Compile        : jni_test <= jni.c
[arm64-v8a] SharedLibrary  : libjni_test.so
[arm64-v8a] Install        : libjni_test.so => libs/arm64-v8a/libjni_test.so
[x86_64] Compile        : jni_test <= jni.c
[x86_64] SharedLibrary  : libjni_test.so
[x86_64] Install        : libjni_test.so => libs/x86_64/libjni_test.so
[mips64] Compile        : jni_test <= jni.c
[mips64] SharedLibrary  : libjni_test.so
[mips64] Install        : libjni_test.so => libs/mips64/libjni_test.so
[armeabi-v7a] Compile thumb  : jni_test <= jni.c
[armeabi-v7a] SharedLibrary  : libjni_test.so
[armeabi-v7a] Install        : libjni_test.so => libs/armeabi-v7a/libjni_test.so
[armeabi] Compile thumb  : jni_test <= jni.c
[armeabi] SharedLibrary  : libjni_test.so
[armeabi] Install        : libjni_test.so => libs/armeabi/libjni_test.so
[x86] Compile        : jni_test <= jni.c
[x86] SharedLibrary  : libjni_test.so
[x86] Install        : libjni_test.so => libs/x86/libjni_test.so
[mips] Compile        : jni_test <= jni.c
[mips] SharedLibrary  : libjni_test.so
[mips] Install        : libjni_test.so => libs/mips/libjni_test.so
```

图 1.1　使用 ndk-build 编译 so 文件的输出

种处理器架构，针对不同架构需要将 C++编译成多个版本的动态链接库，所以可以用 Application.mk 文件来配置生成的平台类型。

在 jni 目录下新建 Application.mk 文件。

Application.mk 文件的主要代码：

```
App_ABI := armeabi armeabi-v7a x86
```

再次使用 ndk-build 命令编译，就会在 libs 下分别生成 armeabi、armeabi-v7a、x86 架构的动态链接库。

## 1.3　NDK 工具链常用工具

NDK 提供了一些用于分析与调试链接库文件的命令行工具，当逆向人员尝试分析某个 Android 应用的 Native 层代码时，这些工具可以发挥作用。

**1. Addr2line**

Addr2line 是一个分析 so 动态链接库文件并根据 so 文件的地址偏移找到对应函数位置的工具。当 Native 层的 C++代码发生错误时，往往很难像 Java 层那样直接定位到问题代码，这是因为 C++代码已经被编译成了汇编语言。而使用 Addr2line 可以将 adb 日志中抛出的 so 文件异常地址转换成对应的函数，大大降低了调试的难度。

Addr2line 是 NDK 中的一个组件，可以使用命令行独立调用，但不同架构的 so 文件对应的 Addr2line 是不同的。

1）在 Windows NDK 目录下

对应 AArch64 架构：Sdk\ndk-bundle\toolchains\aarch64-linux-android-4.9\prebuilt\windows-x86_64\bin。

对应 Arm 架构：Sdk\ndk-bundle\toolchains\arm-linux-androideabi-4.9\prebuilt\windows-x86_64\bin。

2）在 Linux NDK 目录下

对应 AArch64 架构：toolchains/aarch64-linux-android-4.9/prebuilt/linux-x86_64/bin。

对应 Arm 架构：toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86_64/bin。

当 so 文件发生异常，系统会抛出错误堆栈信息。以 libh_db.so 为例，使用 adb logcat 命令获取 Android 应用运行时产生的日志并从中找到 signal 抛出的堆栈信息如下。

adb 获取的日志：

```
backtrace:
10-28 17:13:02.151 7349 7349 F DEBUG:      #00 pc 0000000000000ef8
/data/data/cn.andouya/files/libh_db.so (offset 0xbc000)
10-28 17:13:02.151 7349 7349 F DEBUG:      #01 pc 0000000000043a1c
```

使用 Addr2line 调试出问题的 so 文件，尝试将堆栈中的异常地址转换成 so 文件中的函数。

```
$ aarch64-linux-android-addr2line -C -f -e libh_db.so 00000000000bcef8
```

图 1.2 所示为使用 Addr2line 调试得到的结果。

```
_Unwind_IteratePhdrCallback
/usr/local/google/buildbot/src/android/gcc/toolchain/build/../gcc/gcc-4.9/libgcc/unwind-dw2-fde-dip.c:299
```

图 1.2　使用 Addr2line 调试得到的结果

从图 1.2 的输出中可以看到 so 文件抛出 signal 的函数位置，是在 so 文件 unwind-dw2-fde-dip.c 的第 299 行。需要注意的是，例子中的 libh_db.so 是 Arm64 架构的，所以调用的 Addr2line 工具是 AArch64 目录下的，分析 so 文件需要使用相同架构的工具。

### 2. readelf

readelf 工具一般用于查看 ELF 格式的文件信息，即可查看 Android 编译出来的 so 动态链接库文件，与 Addr2line 工具在同一目录下。通常的用法如下：

```
$ readelf <选项> elf-文件
```

常用参数：

| | |
|---|---|
| -h --file-header | 显示 ELF 文件头 |
| -l --program-headers | 显示程序头 |
| -S --section-headers | 显示节头 |
| -g --section-groups | 显示节组 |
| -t --section-details | 显示节的细节 |
| -s --syms | 显示符号表 |
| --dyn-syms | 显示动态符号表 |

下面使用 readelf 工具来处理一个 so 文件，查看它的头部结构：

```
$ readelf -h libshello_world_normal.so
```

如图 1.3 所示为使用 readelf 命令读取 so 文件头部信息的结果。

如表 1.1 所示为 elf 文件头部信息内容的解析。

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           AArch64
  Version:                           0x1
  Entry point address:               0x431b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6233456 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         8
  Size of section headers:           64 (bytes)
  Number of section headers:         38
  Section header string table index: 35
```

图 1.3　使用 readelf 命令读取 so 文件头部信息的结果

**表 1.1　elf 文件头部信息内容的解析**

| elf 文件头部信息 | 说　　明 |
|---|---|
| Magic | 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |
| 类别 | ELF64 |
| 数据 | 2 补码，小端序 |
| 版本 | 1 |
| OS/ABI | UNIX-System V |
| ABI 版本 | 0 |
| 类型 | DYN(共享目标文件) |
| 系统架构 | AArch64 |
| 版本 | 0x1 |
| 入口点地址 | 0x431b0 |
| 程序头起点 | 64 |
| section 段头部的起点 | 6233456 |
| 标志 | 0x0 |
| 头部的大小 | 64B |
| 程序头部的大小 | 56B |
| 程序头部的数量 | 7 |
| section 段头部的大小 | 64B |
| section 段头部的数量 | 21 |
| section 段头部字符表的索引 | 20 |


视频讲解

## 1.4　解除手机 BL 锁

如果需要逆向分析 Android 应用，真机与模拟器相比限制会少一些。许多应用没有针对模拟器进行优化，在分析的过程中可能会出现闪退或者卡顿的情况。一些具有基本安全防护的应用会检测模拟器环境，从而主动结束进程。接下来的 3 节会介绍如何准备调试应用的真机环境。

在进行所有刷机或 Root 操作之前，都必须要解开 BL 锁。目前国内绝大部分厂商的手机已经不提供解除 BL 锁的方式，因此真机调试环境通常都会选择 Google 公司的 Nexus 系列手机，刷 Google 官方系统镜像。

本书所用的调试真机为 Nexus 6P。Nexus 6P 解除 BL 锁的方法如下：

(1) 进入开发者选项，打开 USB 调试，打开“OEM 解锁”。

打开 USB 调试与 OEM 解锁如图 1.4 和图 1.5 所示。


图 1.4　打开 USB 调试截图


图 1.5　OEM 解锁截图

(2) 使设备进入 fastboot 模式的方法有两种：一是手机在关机状态下长按音量与开机键，二是将手机通过 USB 连接到计算机后，在计算机命令行终端上使用 adb 命令“adb reboot fastboot”进入 fastboot 模式。进入 fastboot 模式后，手机显示的页面被称为 bootloader 界面。

如图 1.6 所示为 Nexus 6P 的 bootloader 界面。


图 1.6　Nexus 6P 的 bootloader 界面

输入 adb 命令检查 fastboot 模式是否正常。

```
$ fastboot devices
```

(3) 输入解锁命令。

```
$ fastboot flashing unlock
```

这样，手机的 BL 锁就被解除，可以进行下一步的刷机操作了。

## 1.5 给手机刷入工厂镜像

Google 为 Nexus 手机准备了各 Android 原生版本的驱动，通过 Google 官网可以找到各 Android 版本的工厂镜像，在刷第三方 ROM 时最好先刷对应版本的 Android 工厂镜像。当刷机出现问题导致无法开机时，可以进入 bootloader 刷回工厂镜像。

刷机时手机内的数据会被清除，如果有需要请提前备份数据，Google 的工厂镜像下载地址：https://developers.google.com/android/images。根据 Nexus 6P 的官方代号找到对应的系统镜像。

如图 1.7 所示为 Google 官网上的系统镜像。

"angler" for Nexus 6P

| Version | Download | SHA-256 Checksum |
|---|---|---|
| 6.0.0 (MDA89D) | Link | 9f001626d37785a4845e2d61c53caaff839a31713062e49b29ede6b5fe807e68 |
| 6.0.0 (MDB08K) | Link | 4655088655f64e4f778adebe9e0ac8099447af643cf983262a95a1abf4401053 |
| 6.0.0 (MDB08L) | Link | bb060be5690856a09325862c94c3568775034bc0c78678d6cdc2ea6dd77feb5d |
| 6.0.0 (MDB08M) | Link | 5690aabf9b18d19ffff5949907eb123f15aff6cbfa31f67c1eeef0650a1fa1fe |
| 6.0.0 (MMB29N) | Link | 63f8243f3bc4fed4638ce9bc943d989da34e73775de6efa1677dad37be3fa1b7 |
| 6.0.1 (MMB29M) | Link | 616cf265c50f3960883f4c0e22b5e795defd8853cfdc83c61448054a06bf6a7f |
| 6.0.1 (MMB29P) | Link | ba26af977515fc9279f78aec766b6e1da33d3ecb8101985a46d7f35b5e7cde7a |
| 6.0.1 (MMB29Q) | Link | 24a6e02f2c3134a32e0d164d0163834595787faab48b07e92fc4a4a89d26e255 |
| 6.0.1 (MMB29V) | Link | 17366b60a63f8d3a9844f7562ea04d1a4409a9ce908452316656d3a3c8207153 |
| 6.0.1 (MHC19I) | Link | 545ef1061782108ad699b96a1f05a59c73eeed6662d8d6fe6448c796a1143752 |
| 6.0.1 (MHC19Q) | Link | e49fbdfd9982787166b5b159861f9d41ba817b87c1ec43f8ec9eb02565d78689 |
| 6.0.1 (MTC19T) | Link | 095027d58b891101167b85a0032998e720da588ceb0b4411c6b1c3f942d68e4c |
| 6.0.1 (MTC19V) | Link | b322694cd8b4f2dfba77889dd9cc645b59e535c4c944816e35632261625f8771 |

图 1.7 Google 官网上的系统镜像

将下载的压缩包解压，然后将 Android SDK 中的 platform-tool 复制到解压目录下，确保手机进入 fastboot 模式，在解压目录中找到 flash-all.bat，以管理员身份运行，并等待刷机完毕。

## 1.6 Root Android 系统

Android 原生系统获取 Root 权限需要借助 Magisk 框架，Magisk 是一个类似于 Xposed 的工具，它会挂载一个与系统文件相隔离的文件系统来加载自定义内容。所以 Magisk 的安装需要借助第三方 recovery 软件。

第三方 recovery 选用 Twrp 软件。首先下载 Magisk 框架的 19.3 版本，将其放入手机 SD 卡中。下载 Twrp 软件对应 Nexus 6P 的版本：twrp-3.3.1-0-angler.img。进入手机的

bootloader 模式，在主机命令行执行下面的命令：

```
$ .\fastboot boot twrp-3.3.1-0-angler.img
```

如图 1.8 所示为 Twrp 软件界面。

在 Twrp 中单击 Install 按钮，选择 Magisk-v 19.3.zip 文件进行安装。

如图 1.9 所示为单击选择 Magisk-v 19.3 安装包。

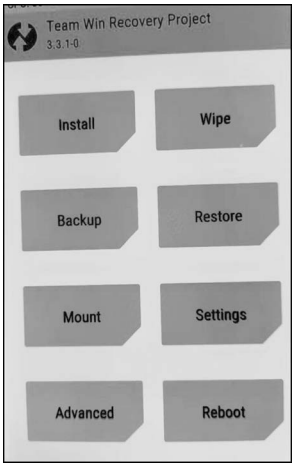


图 1.8　Twrp 软件界面


图 1.9　单击选择 Magisk-v 19.3 安装包

选中 Magisk 安装包后，滑动下方的滑块执行安装，如图 1.10 所示。

此处的 Magisk 安装包是泛指 Magisk 软件的安装包，根据具体情况，软件版本以及安装包的文件名都有可能发生变化。

如图 1.11 所示为安装成功后的输出内容。


图 1.10　安装 Magisk 安装包

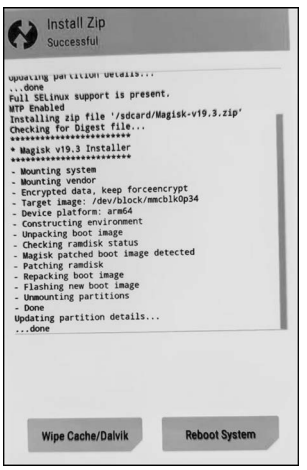


图 1.11　安装成功后的输出内容

安装完毕后直接重启，重启手机后在 Magisk Manager 应用中可以找到启动超级用户的选项，如果有应用需要申请 Root 权限，那么可以在 Magisk Manager 中手动授予。

## 1.7 本章小结

本章介绍了逆向分析过程中所用到的基本环境的配置方法和基本工具。移动应用的逆向分析会用到多种工具，后续章节会具体介绍。读者在后面的学习中选择自己用得顺手的工具即可。

另外，建议读者在学习本书时尽量使用真机作为测试环境，会避免一些不必要的兼容性问题与性能问题。

# 第 2 章

CHAPTER 2

# 破解第一个 Android 应用

## 2.1 反编译 Apk


视频讲解

本章来尝试破解一个简单的 Android 应用，并在这个过程中熟悉反编译、修改、重打包、签名等 Android 应用基本破解流程。

在反编译这一步，需要首先获得 Android 应用的源代码文件，选用的工具是 Apktool。Apktool 是使用 Java 编写的跨平台 Apk 程序反编译工具，从 GitHub 上下载最新 apktool.jar 文件 https://github.com/iBotPeaches/Apktool/releases。

如图 2.1 所示为 GitHub 上 Apktool 的下载页面。

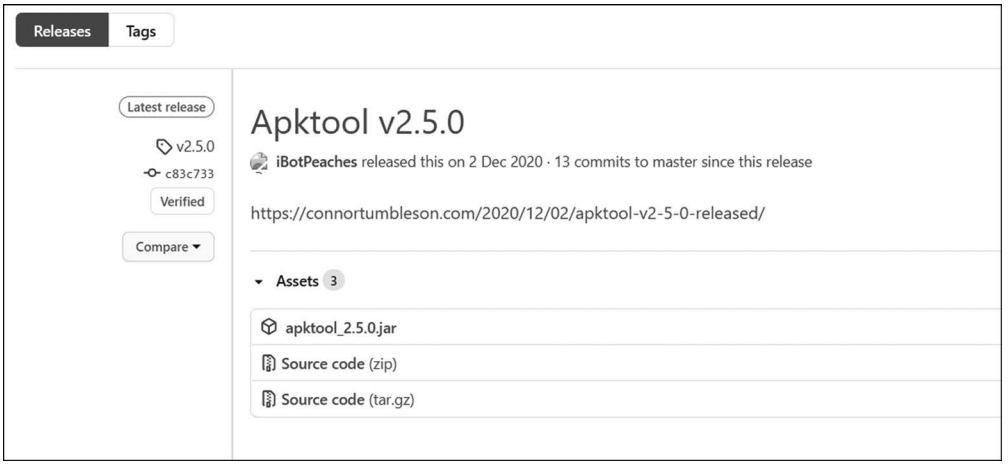


图 2.1 GitHub 上 Apktool 的下载页面

本例使用的 Apk 是 Android Studio 直接创建的 Native Demo 应用，只有一个 MainActivity 显示一段 C++代码返回的字符串。在命令行运行“java -jar”，调用 apktool.jar 对 Apk 包进行反编译，-o 参数指定反编译出的结果所在目录：

```
$ java -jar apktool.jar d hello_world.apk -o output_dir
```

如图 2.2 所示为 Apktool 运行时的输出。

反编译结束后 Apk 包内解码出来的所有文件都保存在 output_dir 目录中。

```
node1@node1:~/test_example/tools$ java -jar apktool.jar d hello_world.apk -o output_dir
I: Using Apktool 2.4.1-dirty on hello_world.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/node1/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: running generatePublicXml
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

图 2.2 Apktool 运行时的输出

## 2.2 分析包内文件

本节将通过逐一分析 2.1 节用 Apktool 工具处理 Apk 文件后得到的内容文件，来了解 Apk 包的结构。

### 1. lib 目录

lib 目录下保存的是 Android Native 层编译链接出来的动态链接库，不同架构的 so 文件分别保存在对应的目录下，常见的有 armeabi(armeabi-v7a)、arm64-v8a、x86、x86_64。应用运行时可以根据运行环境选取对应架构的 so 文件进行加载。

如图 2.3 所示为 lib 目录的截图。

### 2. assets 目录

assets 目录是 Android 应用的另一种资源的打包方式，assets 目录中的所有文件都会随应用打包，通过 Context 获得的 AssetManager 类可以访问到 assets 目录。assets 目录下的文件在打包后会原封不动地保存在 Apk 包内，但是与 res 目录下的不同，assets 目录下的文件不会被映射到 R.java 中，同时 assets 可以保留目录结构。

如图 2.4 所示为 assets 目录的截图。

```
node1@node1:~/output_dir/lib$ ls -l
total 16
drwxrwxr-x 2 node1 node1 4096 Jan 29 11:30 arm64-v8a
drwxrwxr-x 2 node1 node1 4096 Jan 29 11:30 armeabi-v7a
drwxrwxr-x 2 node1 node1 4096 Jan 29 11:30 x86
drwxrwxr-x 2 node1 node1 4096 Jan 29 11:30 x86_64
```

图 2.3 lib 目录的截图

```
node1@node1:~/output_dir/assets$ ls -l
total 4
-rw-rw-r-- 1 node1 node1 13 Jan 29 11:30 test_assets
```

图 2.4 assets 目录的截图

### 3. kotlin 目录

Kotlin 是一个多平台的静态编程语言，可以编译成 Java 字节码，也可以编译成 JavaScript，方便在没有 JVM 的设备上运行，现在已经成为 Android 官方支持的开发语言。如果 Apk 包导入 Kotlin 语言编写的部件时，Kotlin 相关的文件就会保存在 kotlin 目录下。

如图 2.5 所示为 kotlin 目录的截图。

### 4. META-INF 目录

该目录下保存签名相关的信息，编译生成一个 Apk 包时，会对所有要打包的文件进行校验计算，将计算结果保存在 META-INF 目录下。当 Apk 包被安装时，应用管理器会对包内的文件进行校验，如果校验结果不一致，应用就不会被安装。

如图 2.6 所示为 META-INF 目录的截图。

```
node1@node1:~/output1_dir/kotlin$ ls -l
total 336
drwxrwxr-x 2 node1 node1  4096 Jan 29 11:33 annotation
-rw-rw-r-- 1 node1 node1   204 Jan 29 11:33 ArithmeticException.kotlin_metadata
-rw-rw-r-- 1 node1 node1   135 Jan 29 11:33 AssertionError.kotlin_metadata
-rw-rw-r-- 1 node1 node1   443 Jan 29 11:33 BuilderInference.kotlin_metadata
-rw-rw-r-- 1 node1 node1   153 Jan 29 11:33 ClassCastException.kotlin_metadata
drwxrwxr-x 2 node1 node1  4096 Jan 29 11:33 collections
-rw-rw-r-- 1 node1 node1   154 Jan 29 11:33 Comparator.kotlin_metadata
drwxrwxr-x 2 node1 node1  4096 Jan 29 11:33 comparisons
-rw-rw-r-- 1 node1 node1   442 Jan 29 11:33 ConcurrentModificationException.kotlin_metadata
drwxrwxr-x 2 node1 node1  4096 Jan 29 11:33 contracts
drwxrwxr-x 4 node1 node1  4096 Jan 29 11:33 coroutines
-rw-rw-r-- 1 node1 node1   172 Jan 29 11:33 Error.kotlin_metadata
-rw-rw-r-- 1 node1 node1   176 Jan 29 11:33 Exception.kotlin_metadata
drwxrwxr-x 2 node1 node1  4096 Jan 29 11:33 experimental
-rw-rw-r-- 1 node1 node1   467 Jan 29 11:33 Experimental.kotlin_metadata
-rw-rw-r-- 1 node1 node1   584 Jan 29 11:33 ExperimentalMultiplatform.kotlin_metadata
-rw-rw-r-- 1 node1 node1   663 Jan 29 11:33 ExperimentalStdlibApi.kotlin_metadata
-rw-rw-r-- 1 node1 node1   627 Jan 29 11:33 ExperimentalUnsignedTypes.kotlin_metadata
-rw-rw-r-- 1 node1 node1   188 Jan 29 11:33 HashCodeKt.kotlin_metadata
-rw-rw-r-- 1 node1 node1   217 Jan 29 11:33 IllegalArgumentException.kotlin_metadata
-rw-rw-r-- 1 node1 node1   214 Jan 29 11:33 IllegalStateException.kotlin_metadata
-rw-rw-r-- 1 node1 node1   160 Jan 29 11:33 IndexOutOfBoundsException.kotlin_metadata
-rw-rw-r-- 1 node1 node1   262 Jan 29 11:33 InitializedLazyImpl.kotlin_metadata
```

图 2.5　kotlin 目录的截图

```
node1@node1:~/output_dir/original/META-INF$ ls -l
total 220
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.appcompat_appcompat.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.arch.core_core-runtime.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.asynclayoutinflater_asynclayoutinflater.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.coordinatorlayout_coordinatorlayout.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.core_core.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.cursoradapter_cursoradapter.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.customview_customview.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.documentfile_documentfile.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.drawerlayout_drawerlayout.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.fragment_fragment.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.interpolator_interpolator.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.legacy_legacy-support-core-ui.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.legacy_legacy-support-core-utils.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.lifecycle_lifecycle-livedata-core.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.lifecycle_lifecycle-livedata.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.lifecycle_lifecycle-runtime.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.lifecycle_lifecycle-viewmodel.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.loader_loader.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.localbroadcastmanager_localbroadcastmanager.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.print_print.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.slidingpanelayout_slidingpanelayout.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.swiperefreshlayout_swiperefreshlayout.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.vectordrawable_vectordrawable-animated.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.vectordrawable_vectordrawable.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.versionedparcelable_versionedparcelable.version
-rw-rw-r-- 1 node1 node1     6 Jan 29 11:30 androidx.viewpager_viewpager.version
-rw-rw-r-- 1 node1 node1  1328 Jan 29 11:30 CERT.RSA
-rw-rw-r-- 1 node1 node1 54795 Jan 29 11:30 CERT.SF
-rw-rw-r-- 1 node1 node1 54733 Jan 29 11:30 MANIFEST.MF
```

图 2.6　META-INF 目录的截图

### 5. original 目录

经过 Apktool 反编译后产生的目录，用来保存一些文件的备份。

如图 2.7 所示为 original 目录的截图。

```
node1@node1:~/output_dir/original$ ls -l
total 8
-rw-rw-r-- 1 node1 node1 2188 Jan 29 11:30 AndroidManifest.xml
drwxrwxr-x 2 node1 node1 4096 Jan 29 11:30 META-INF
```

图 2.7　original 目录的截图

### 6. res 目录

保存工程的资源文件，打包时 values 文件被编译进 resource.arsc 文件中，使用 Apktool 反编译后会将 resource.arsc 中的内容还原成 values 文件。res 目录下的文件会被映射到 R.java 中，应用中访问资源可以使用资源 ID：R.id.filename。

如图 2.8 所示为 res 目录的截图。

```
total 0
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 anim
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 color
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 color-v21
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 color-v23
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-anydpi-v21
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-hdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldrtl-hdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldrtl-mdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldrtl-xhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldrtl-xxhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-ldrtl-xxxhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-mdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-v21
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-v23
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-v24
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-watch-v20
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-xhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-xxhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 drawable-xxxhdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 layout
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 layout-v21
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 layout-v22
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 layout-v26
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 layout-watch-v20
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 mipmap-anydpi-v26
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 mipmap-hdpi
drwxrwxrwx 1 charles charles 4096 Sep 11 15:54 mipmap-mdpi
```

图 2.8　res 目录的截图

**7. Unknown 目录**

Unknown 目录是 Apktool 反编译后生成的目录，用来保存暂时无法被处理的文件，在重打包时直接复制回包内。

**8. AndroidManifest.xml 文件**

AndroidManifest 的官方解释是应用清单(manifest 意思是货单)，每个应用的根目录中都必须包含一个，并且文件名必须一模一样。这个文件中包含了 App 的配置信息，系统需要根据里面的内容运行 App 的代码，显示界面。后面会详细解析 AndroidManifest.xml 文件。

**9. Apktool.yml 文件**

Apktool.yml 文件是 Apktool 工具的描述文件，记录了 Apk 反编译信息，方便对 Apk 目录进行反编译操作。


视频讲解

## 2.3　修改 Smali 代码文件

Apktool 默认的反编译设置会把 Apk 包内的 dex 文件反编译成 Smali 文件，保存在 Smali 目录下，Smali 文件可以直接用编辑器打开，像编辑 Java 源码一样对 Smali 文件进行修改。本节中将通过直接修改 Smali 文件使程序在启动时跳出一个弹窗。

首先定位插入代码的位置，要在启动时执行插入的代码，则代码需要插在入口 Activity 的 onCreate 函数内。下面给出在正常开发过程控制弹窗的 Java 代码：

```
Toast.makeText(this, stringFromJNI, 1).show();
```

此处计划在弹窗内显示 Native 方法返回的字符串，用 Java 来实现就是简单的一句将它翻译成 Smali 代码。

Smali 代码：

```
const/4 v1, 0x1

invoke-static {p0, v0, v1},
Landroid/widget/Toast; -> makeText (Landroid/content/Context; Ljava/lang/CharSequence; I)
Landroid/widget/Toast;

move-result-object v1

invoke-virtual {v1}, Landroid/widget/Toast; ->show()V
```

如图 2.9 所示为弹窗代码的具体插入点。

```
.line 22
invoke-virtual {p0}, Lcom/example/hello_world/MainActivity;->stringFromJNI()Ljava/lang/String;

move-result-object v0

# insert code
const/4 v1, 0x1
invoke-static {p0, v0, v1}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/(
move-result-object v1
invoke-virtual {v1}, Landroid/widget/Toast;->show()V
# insert end

invoke-virtual {p1, v0}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
```

图 2.9　弹窗代码的具体插入点

原逻辑 stringFromJNI()方法返回的字符串作为参数传入 setText()方法，在 TextView 中显示出来。调用 stringFromJNI()方法的返回值保存在 v0 中，弹窗要显示 v0 的值，因此插入点在 move-result-object v0 之后，根据 Toast.makeText()方法的参数表，v0 作为第二个参数传入，同时需要构造一个整型值 v1 作为第三个参数。Toast.makeText()方法完成调用后 v1 就没有用了，可以用来保存 makeText 的返回值。

此时，还不能进行重编译，插入代码时引入了一个新的局部变量 v1，原来代码中只有一个局部变量 v0，而在 onCreate()方法的开头有一个局部变量数量的声明。

如图 2.10 所示为修改 onCreate()方法 locals 值的效果。

```
# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 1
```

图 2.10　修改 onCreate()方法 locals 值的效果

当引入新的局部变量后，必须要修改这里的值，否则无法完成重编译。这里将.locals 1 改为.locals 2，就可以顺利进入下一步的重编译环节。

## 2.4　重编译并签名

在命令行执行 Apktool 的重编译命令：

```
$ java -jar apktool.jar b output_dir -o hello_world_unsigned.apk
```

重新打包之后将 Apk 拖入 Jadx-gui 中，验证插入的代码是否存在语法问题。

如图 2.11 所示为 Jadx-gui 验证插入的代码逻辑。

```
/* access modifiers changed from: protected */
@Override // androidx.core.app.ComponentActivity, androidx.appcompat.app.AppCompatAc
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.activity_main);
    String stringFromJNI = stringFromJNI();
    Toast.makeText(this, stringFromJNI, 1).show();
    ((TextView) findViewById(R.id.sample_text)).setText(stringFromJNI);
}
```

图 2.11　Jadx-gui 验证插入的代码逻辑

重编译出来的应用还没有签名，无法直接运行，需要对应用进行签名，签名的方法与工具有多种，这里介绍用 Apksigner 签名的方式。

apksigner.jar 可以从 Android SDK 的 build-tools 的 lib 目录下找到。apksigner.jar 签名用到的 jks 文件可以用 Android Studio 生成。在 Android Studio 中选择 Build 选项卡中的 Generate Signed APK 选项，在弹出的窗口中创建新的签名文件。

如图 2.12 所示为在 Android Studio 中创建 jks 文件的界面。

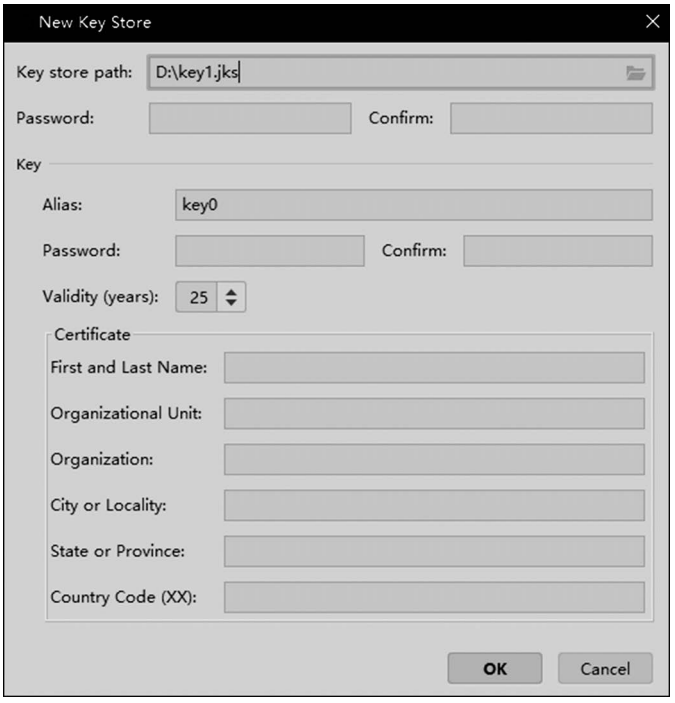


图 2.12　在 Android Studio 中创建 jks 文件的界面

在命令行执行下面的命令：

```
java -jar apksigner.jar sign -verbose --ks key1.jks --v1-signing-enabled true --v2-signing-enabled true --ks-pass pass:password --ks-key-alias key0 --out hello_world_signed.apk hello_world_unsigned.apk
```

以上这条命令较长，其中涉及的参数及其功能如表 2.1 所示。

表 2.1　对应用签名的命令参数

| 参　数 | 功　能 | 参　数 | 功　能 |
|---|---|---|---|
| --ks | 签名使用的 jks 文件 | --ks-pass pass | KeyStore 的密码 |
| --v1-signing-enabled | 使用 jar 包签名方式 | --ks-key-alias | 生成 jks 文件时指定的 alias |
| --v2-signing-enabled | 使用全 Apk 包签名方式 | | |

该命令启用了 v1 和 v2 两个版本的签名，这是 Android 签名的两种机制。v1 签名是基于 jar 文件的签名方案，该方案会遍历 Apk 中所有条目，提取出包中文件的消息摘要并写入 MANIFEST.MF 文件中。再对 MANIFEST.MF 文件做二次摘要生成 CERT.SF 文件并使用私钥对 CERT.SF 文件签名，将 3 个文件一起打包保存到 META-INF 文件夹中。

v1 版本签名存在两个较大的缺陷：一是校验时对所有文件的摘要计算对某些资源比较多的应用或者性能较差的平台是不小的性能负担，会导致应用安装时间过长；二是用来存放签名的 META-INF 目录本身不会被校验，形成了校验环节的漏洞。

基于 v1 基础上推出的 v2 签名则是将验证归档中所有的字节，并在原先 Apk 块中增加一个新的签名块用于存储签名、摘要、签名算法、证书链等属性信息。

相比 v1 签名方案，v2 签名支持将 Apk 分割成小块，分别计算小块的摘要，再计算得到最终的摘要。v1 签名与 v2 签名可以共存，校验的过程中如果检测到使用了 v2 签名块，则必须通过 v2 的校验流程。如果没找到 v2 签名的块，则降级到 v1 签名的校验流程。

下面将签名完毕的 Apk 安装到手机中运行，查看运行效果。

如图 2.13 所示为重编译应用的运行效果。

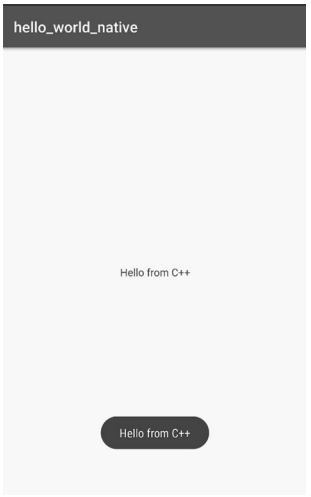


图 2.13　重编译应用的运行效果

## 2.5　本章小结

本章通过一个最简单的 Android 应用介绍了分析与破解 Android 程序的基本流程。而实际逆向人员遇到的应用复杂度要更高，有的甚至采用了一定的保护措施，需要借助一些辅助手段进行分析，后面的章节中将会具体介绍。

# 第 3 章

CHAPTER 3

# iOS 基础知识

作为移动设备市场的重要组成部分，苹果公司开发的 iPhone 设备也是攻防双方对抗的战场之一。本章将介绍有关 iOS 应用的相关知识，包括 iOS 应用包结构以及 iOS 应用的启动流程。

视频讲解

视频讲解

## 3.1 iOS 包结构分析

每一个应用程序都有一个载体，载体内部保存着程序的字节码以及程序运行过程中的各种配置文件，所有的文件会被打包在一起，形成一个压缩包，以便于程序的复制分发以及网络传输。

不同平台的程序包文件扩展名以及结构都不一样，例如，Windows 平台的程序包通常是以. exe 为扩展名的可执行文件；Android 系统的程序包则是以. apk 为扩展名的压缩包文件。对于 iOS 系统，iOS 程序包文件是以. ipa 为扩展名的压缩包。

IPA 包文件与 APK 包文件类似，以 ZIP 压缩包格式作为基础。如果将一个 iOS 应用的 IPA 包文件的扩展名修改为. zip，则可以使用解压缩工具打开。IPA 包解压效果如图 3.1 所示。

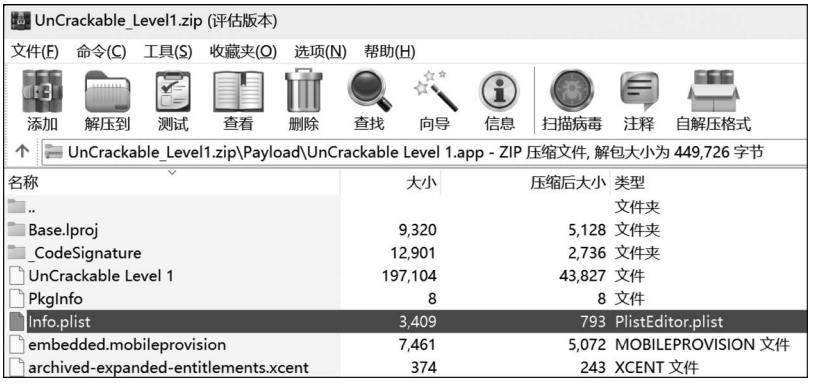


图 3.1 IPA 包解压效果

### 3.1.1 _CodeSignature 文件夹

_CodeSignature 文件夹内有一个 CodeResources 文件，该文件内包含一个字典，字典的内容是 IPA 包内文件的哈希表。字典的键是文件名，字典的值是 Base64 格式的哈希值。