

第1章 Android 程序分析环境搭建

在实际的 Android 软件开发过程中，可能很多开发人员有过这样的经历：

- 我有一个不错的 idea，正在开发一款类似想法的软件，可是涉及到的一些功能上的具体代码细节却难以下手，我看到别人的程序中有这个功能，它们是如何实现的呢？
- 我不小心安装了一个流氓软件，软件运行时会自动下载木马程序、恶意扣费、篡改手机系统，它是如何做到这些的呢？
- 我按照网上介绍的方法来分析 Android 程序，可是根本就无法正确地反编译程序，或是反编译出的代码语法混乱，根本无法阅读。

这些场景都提出了一个疑问，那就是如何分析一个 Android 应用程序？如何掌握这些软件的架构思想？分析别人的程序在很多人看来是不能够接受的行为，在他们眼中这种行为都应被视为盗窃。其实任何技术的起源本身就是从学习开始的，用正确的态度对待程序分析技术是可以的。

如果说，开发 Android 程序是一种学问，那么分析 Android 程序更像是一门艺术。在浩瀚如海的反汇编代码中分析出程序的执行流程与架构思想是一件很了不起的事情，这需要分析人员有着扎实的编程基础与深厚的思维分析能力。分析软件的过程犹如一次艰难的旅程，这条旅程会有多长？该怎么走？会有多少崎岖险路？没有人知道，但是先行者已经为我们铺下了台阶，我们只需沿着它慢慢前行。

1.1 Windows 分析环境搭建

搭建 Windows 分析平台的系统版本要求不高，Windows XP 或以上即可。本书的 Windows 平台的分析环境采用 Windows XP 32 位系统，如果读者使用 Windows 7 或其它版本，操作上是大同小异的。

1.1.1 安装 JDK

JDK 是 Android 开发必须的运行环境，在安装 JDK 之前，首先到 Oracle 公司官网上下载它。下载地址为：<http://www.oracle.com/technetwork/java/javase/downloads/index.html>，打开下载页面，目前最新版本为 Java SE 6 Update 33，如图 1-1 所示。

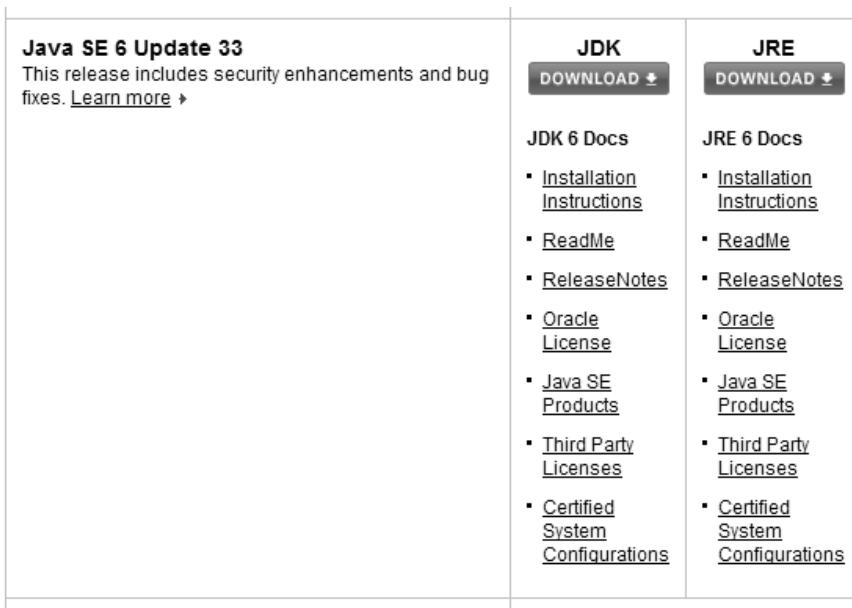


图1-1 下载JDK

点击 JDK 下面的 DOWNLOAD 按钮进入下载页面，勾选“Accept License Agreement”单选框，然后点击 jdk-6u33-windows-i586.exe 进行下载。下载完成后双击安装文件，启动 JDK 安装界面，如图 1-2 所示。



图1-2 JDK安装界面

与安装其它 Windows 软件一样，JDK 的安装过程也很简单，只需要不停点击下一步就可以顺利安装完成。安装完成后手动添加 JAVA_HOME 环境变量，值为“C:\Program Files\Java\jdk1.6.0_33”，并将“C:\Program Files\Java\jdk1.6.0_33\bin”添加到 PATH 变量中。如图 1-3 所示。



图1-3 设置Java环境变量

完成所有步骤后检查一下 Java 是否安装成功。单击“开始”按钮，选择“运行”，在出现的对话框中输入 CMD 命令打开 CMD 窗口，在 CMD 窗口中输入 java -version，如果屏幕上出现如图 1-4 所示的提示，说明安装成功。

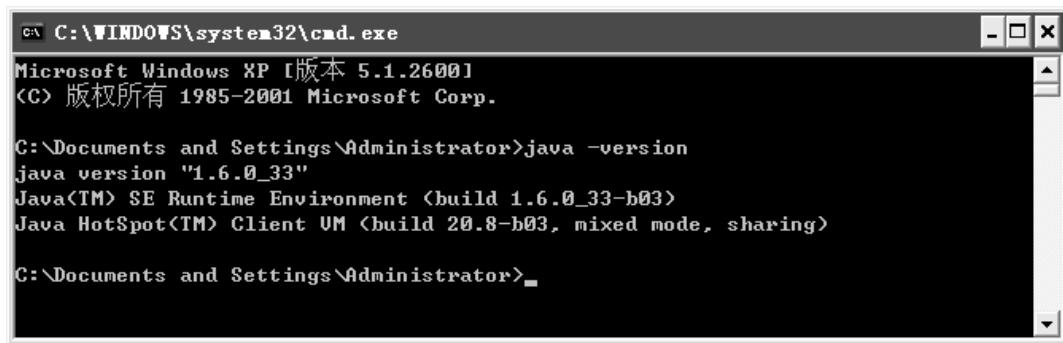


图1-4 查看Java是否正确安装

1.1.2 安装 Android SDK

Android SDK 是以 zip 压缩包的形式提供给开发人员的。首先到 Android 官网下载最新版本的 SDK，下载地址为：<http://developer.android.com/sdk/index.html>。SDK 提供了压缩包与安装

文件两种方式供开发者下载，为了方便部署，本书采用下载安装文件的方式直接安装，目前 Android SDK 的最新版本为 r20，完整下载地址为：http://dl.google.com/androidinstaller_r20-windows.exe。

双击下载后的安装文件，将 Android SDK 安装到任意位置，本书安装环境为 D:\android-sdk 目录，然后将“D:\android-sdk\tools”与“D:\android-sdk\platform-tools”目录添加到系统的 PATH 环境变量中。添加完成后打开一个 CMD 窗口，输入“emulator -version”与“adb version”命令查看是否能成功运行。执行结果如图 1-5 所示。

```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>emulator -version

C:\Documents and Settings\Administrator>Android emulator version 20.0 <build_id
OPENMASTER-391819>
Copyright (C) 2006-2011 The Android Open Source Project and many others.
This program is a derivative of the QEMU CPU emulator (<www.qemu.org>).

This software is licensed under the terms of the GNU General Public
License version 2, as published by the Free Software Foundation, and
may be copied, distributed, and modified under those terms.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

C:\Documents and Settings\Administrator>adb version
Android Debug Bridge version 1.0.29

C:\Documents and Settings\Administrator>
```

图1-5 检查Android SDK是否正确安装

Android SDK 安装成功后，需要通过 SDK 管理器下载具体版本的 SDK，双击“D:\android-sdk\SDK Manager.exe”文件，打开 Android SDK Manager，运行后如图 1-6 所示。

读者可以根据自己的需要选择相应的一个或多个版本进行下载，本书选择了 2.2、2.3.3、4.0、4.0.3、4.1 等几个版本，点击 Install package 按钮打开“Choose Package to Install”对话框，选择“Accept All”单选框，最后点击“Install”按钮开始下载，下载所需的时间根据网络环境差异可能会有所不同。

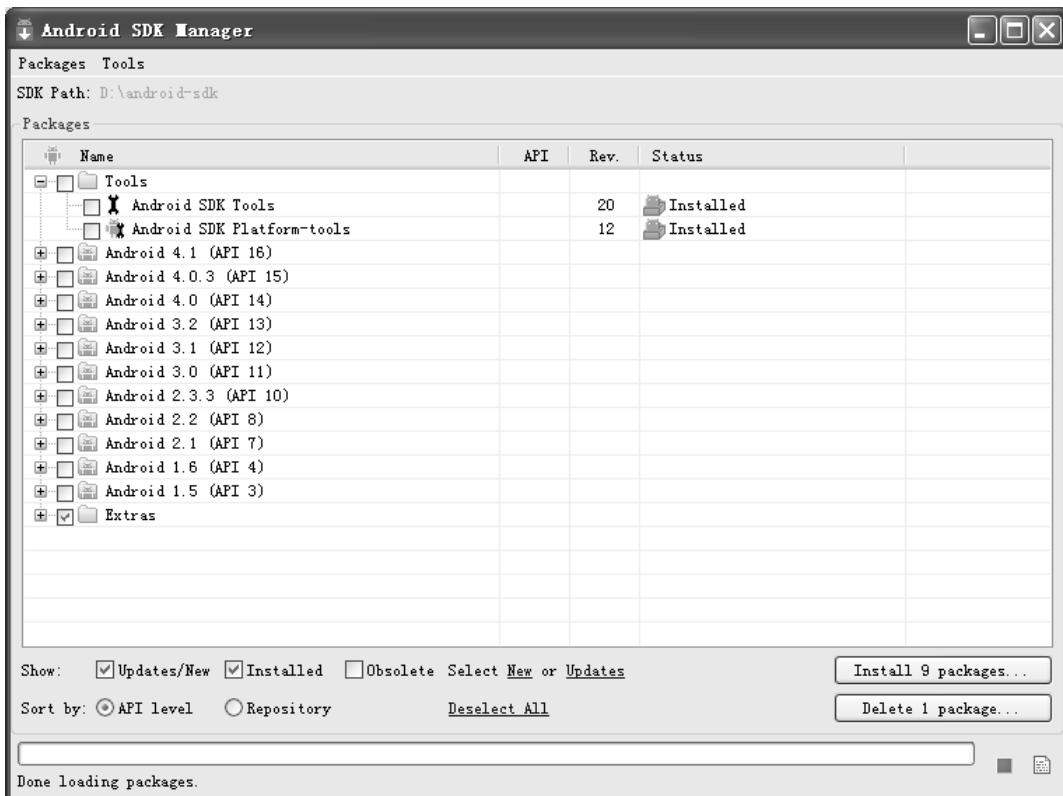


图1-6 Android SDK Manager运行界面

1.1.3 安装Android NDK

Android NDK是Google提供的开发Android原生程序的工具包。如今越来越多的软件与病毒采用了基于Android NDK动态库的调用技术，隐藏了程序在实现上的很多细节，掌握Android NDK程序的分析技术也成为了分析人员必备的技能，本书将会在第7章对Android NDK程序的特点以及分析技术进行详细的讲解。

Android NDK的下载地址为：<http://developer.android.com/sdk/ndk/index.html>，目前最新版本为R8，Windows平台下的完整下载链接为：<http://dl.google.com/android/ndk/android-ndk-r8-windows.zip>。将下载后的压缩包解压到硬盘任意位置，本书为D盘根目录。新建环境变量ANDROID_NDK，值为D:\android-ndk-r8，然后将ANDROID_NDK添加到PATH环境变量中，做好这一步，Android NDK就算安装完成了。

接下来测试配置是否正确，打开一个CMD窗口，进入目录“D:\android-ndk-r8\samples\hello-jni”，输入“ndk-build”命令编译Android NDK中自带的hello-jni工程，如果输出如图1-7所示的结果，就说明Android NDK安装成功了。

```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:

D:>cd d:\android-ndk-r8\samples\hello-jni

D:\android-ndk-r8\samples\hello-jni>ndk-build
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
"Compile thumb": hello-jni <= hello-jni.c
SharedLibrary   : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so

D:\android-ndk-r8\samples\hello-jni>

```

图1-7 使用Android NDK编译hello-jni工程

1.1.4 Eclipse 集成开发环境

Eclipse 是 Android 开发推荐使用的 IDE。它的下载地址为：<http://www.eclipse.org/downloads>，选择下载“Eclipse IDE for Java Developers”或“Eclipse for Mobile Developers”版本即可。强烈建议下载使用后者，后者自带了 CDT（C/C++Development Tools）插件，并针对手机开发做了优化。

Eclipse 是一款绿色软件，下载完成后解压到硬盘任意目录，本书为 D 盘根目录。进入“D:\eclipse”目录，运行 eclipse.exe，Eclipse 会根据前面设置的环境变量自动进行初始化，如果启动时没有提示错误说明安装成功。

1.1.5 安装 CDT、ADT 插件

如果读者使用的 Eclipse 是 For Mobile Developers 版本或自带 CDT 插件，可以跳过 CDT 插件的安装；否则需要手动安装 CDT 插件。安装 Eclipse 的插件比较简单，有在线安装与离线安装两种方式，步骤分别为：

- 启动 Eclipse，点击菜单“Help→Install New Software”打开 Install 对话框，在“Work With”旁边的编辑框中输入 <http://download.eclipse.org/tools/cdt/releases/juno> 并回车，稍等片刻后下面列表框就会解析出 CDT 插件。
- 到 Eclipse 官网上手动下载最新版的 CDT 插件，目前最新 8.1.0。下载地址为：<http://www.eclipse.org/cdt/downloads.php>。启动 Eclipse，点击菜单“Help→Install New Software”打开 Install 对话框，点击界面上的 Add 按钮，打开 Add Repository 对话框，接着点击 Archive 按钮，选择下载的 CDT 压缩包，点击 OK 按钮返回。

无论采用上面哪一种方式进行安装，最终都会在 Name 下面的列表中列出可供安装的 CDT 插件，如图 1-8 所示，全部勾选后点击 Next 按钮即可安装，在线安装耗费的时间根据网络环境差异可能有所不同。

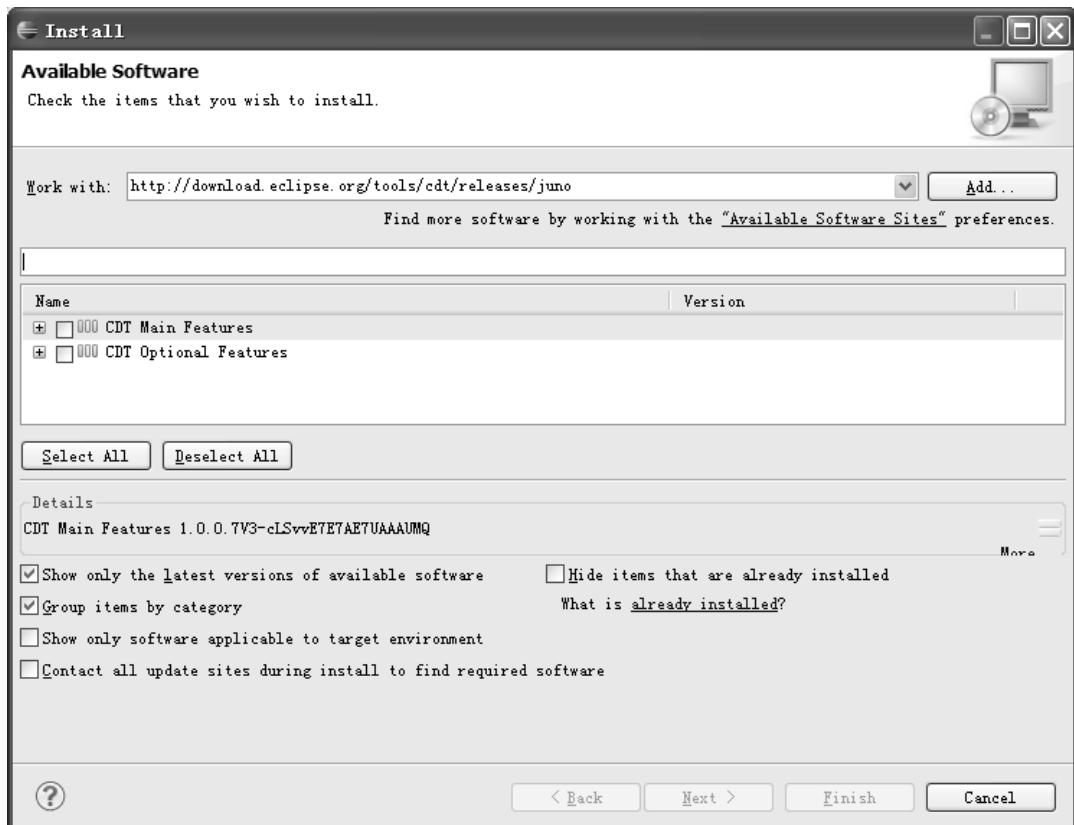


图1-8 安装CDT插件

ADT 插件是 Google 为 Android 开发提供的 Eclipse 插件，方便在 Eclipse 开发环境中创建、编辑、调试 Android 程序。安装过程与 CDT 插件类似。目前最新版本为 20.0.0，官方下载地址为：<http://dl.google.com/android/ADT-20.0.0.zip>，在线安装的 repository 地址为：<https://dl-ssl.google.com/android/eclipse/>，读者可以按照上面的步骤自行完成安装。

ADT 插件安装完成后需要进行相应的配置。点击 Eclipse 菜单项“Window-Preferences”，选择 Android 列表项，在右侧 SDK Location 处选择 Android SDK 的安装位置，如 D:\android-sdk，展开 Android 列表项，选择 NDK，在右侧 NDK Location 处选择 Android NDK 的安装位置，如 D:\android-ndk-r8。设置完后点击 OK 按钮关闭对话框。到此，CDT 与 ADT 插件就安装完成了。

1.1.6 创建 Android Virtual Device

Android SDK 中提供了“Android Virtual Device Manager”工具，方便在没有真实 Android 设备环境的情况下调试运行 Android 程序。

双击运行“D:\android-sdk-windows\AVD Manager.exe”，点击“New”按钮，打开 AVD 创建对话框，在“Name”栏输入 AVD 的名称，如输入“Android2.3.3”，在 Target 一栏选择要模拟的 Android 版本，这里选择“Android 2.3.3 – API Level 10”，SD Card 大小指定为 256MB，其它选项保持不变，结果如图 1-9 所示。

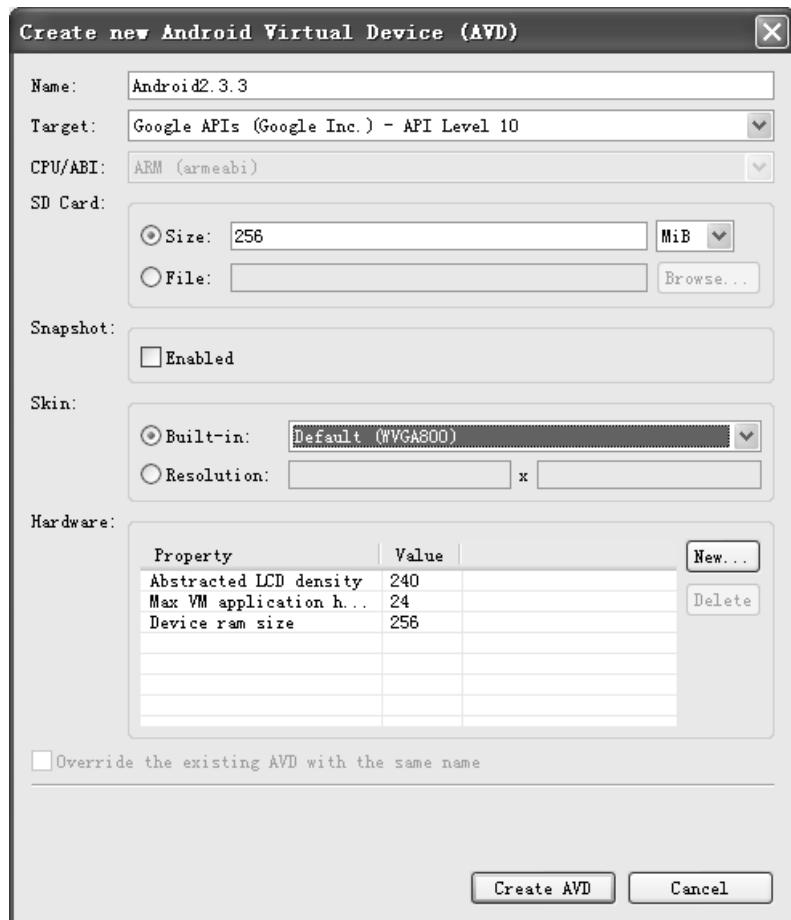


图1-9 创建AVD

点击“Create AVD”按钮完成 AVD 的创建。选中创建的 AVD，点击右侧的“Start”按钮，如果没有错误会成功启动这个 Android 虚拟设备。

如果使用真实Android设备来调试程序，需要先在设备的“设置→程序→开发”选项中开启“USB调试”，然后安装相应设备厂商提供的USB驱动程序。一份常见的USB驱动程序下载地址列表可以在<http://developer.android.com/tools/extras/oem-usb.html>中找到。安装完驱动后在命令提示符下输入“adb devices”就可以列出连接的Android设备了。

1.1.7 使用到的工具

分析Android程序需要用到很多工具，包括：反编译工具、静态分析工具、动态调试工具、集成分析环境等。所有的工具中，大多数是开源或免费的软件，笔者在此不详细列出使用到的工具，在每一章对知识点进行介绍时，会同时介绍相关工具的使用方法与下载地址。

1.2 Linux分析环境搭建

本书中介绍的Windows环境下的大多数操作，在Linux环境下同样适用。本节将介绍如何在Linux环境下搭建Android程序的分析环境。

1.2.1 本书的Linux环境

本书在写作时使用的Linux环境为Ubuntu 10.04 32位系统，在最后一章进行Android病毒分析时使用的是Ubuntu 12.04 32位系统。

1.2.2 安装JDK

首先在当前用户主目录下新建tools目录来存放Android分析常用的工具。

到Oracle官方网站下载JDK安装包。本书Ubuntu平台使用的版本为jdk-6u33-linux-i586，下载地址为<http://www.oracle.com/technetwork/java/javase/downloads/index.html>，将下载的jdk-6u33-linux-i586.bin文件放到/home/feicong/tools目录（feicong为本机用户名），打开一个终端环境输入以下命令：

```
cd tools  
chmod +x ./jdk-6u33-linux-i586.bin  
./jdk-6u33-linux-i586.bin
```

此时安装程序会自动将JDK安装到当前目录的jdk1.6.0_33目录下。接着设置环境变量，执行：

```
sudo gedit /etc/profile
```

在配置文件中加入如下部分：

```

export JAVA_HOME=/home/feicong/jdk1.6.0_33
export JRE_HOME=/home/feicong/jdk1.6.0_33/jre
export PATH=/home/feicong/jdk1.6.0_33/bin:$PATH
export CLASSPATH=.:./home/feicong/jdk1.6.0_33/lib:./home/feicong/jdk1.6.0_33/jre/lib

```

保存后退出，在终端提示符中输入如下命令使环境变量生效：

```
source /etc/profile
```

最后输入命令“java -version”验证 JDK 是否安装成功，如图 1-10 所示。

The screenshot shows a terminal window with the following text:

```

文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
opening the register.html file (located in the JDK installation directory) in a browser.
For more information on what data Registration collects and how it is managed and used, see:
http://java.sun.com/javase/registration/JDKRegistrationPrivacy.html
Press Enter to continue.....
export JAVA_HOME=/home/feicong/jdk1.6.0_33
Done.
feicong@feicong-ubuntu:~/tools$ sudo gedit /etc/profile
[sudo] password for feicong: 6.0_33lib/home/feicong/jdk1.6.0_33/jre/lib
feicong@feicong-ubuntu:~/tools$ source /etc/profile
feicong@feicong-ubuntu:~/tools$ java -version
java version "1.6.0_33"
Java(TM) SE Runtime Environment (build 1.6.0_33-b03)
Java HotSpot(TM) Server VM (build 20.8-b03, mixed mode)
feicong@feicong-ubuntu:~/tools$ 

```

图1-10 验证JDK是否安装成功

1.2.3 在 Ubuntu 上安装 Android SDK

Ubuntu 上安装 Android SDK 与 Windows 安装步骤类似，首先到官方网站 <http://developer.android.com/sdk/index.html> 下载 Android SDK，目前最新版本为 r20，下载后将 android-sdk_r20-linux.tgz 压缩包文件放到 tools 目录，执行以下命令进行解包：

```
tar zvxf android-sdk_r20-linux.tgz
```

解包完毕后就会在当前目录下出现 android-sdk-linux 目录了。这个目录的内容与 Windows 平台提供的工具类似。接着设置环境变量，执行：

```
sudo gedit /etc/profile
```

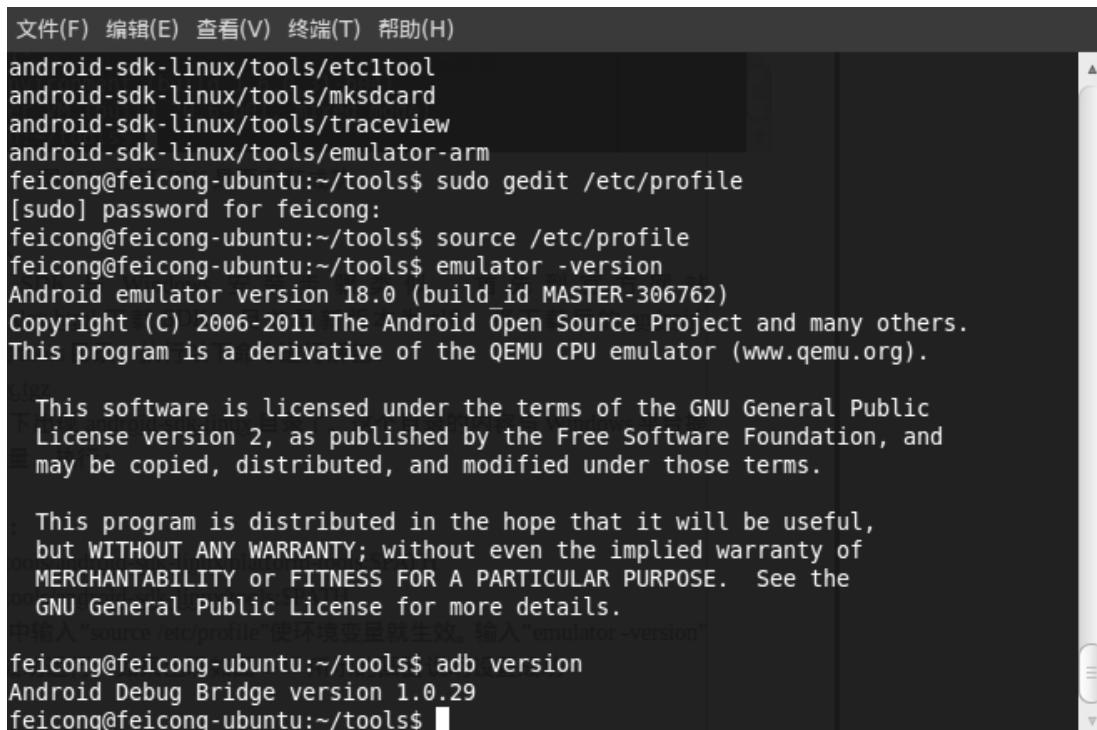
在配置文件中加入如下部分：

```

export PATH=/home/feicong/tools/android-sdk-linux/platform-tools:$PATH
export PATH=/home/feicong/tools/android-sdk-linux/tools:$PATH

```

保存后退出，在终端提示符中输入“source /etc/profile”使环境变量生效。输入“emulator -version”与“adb version”命令查看是否能成功运行。如果出现如图 1-11 所示的画面说明设置成功。



```

文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
android-sdk-linux/tools/etc1tool
android-sdk-linux/tools/mksdcard
android-sdk-linux/tools/traceview
android-sdk-linux/tools/emulator-arm
feicong@feicong-ubuntu:~/tools$ sudo gedit /etc/profile
[sudo] password for feicong:
feicong@feicong-ubuntu:~/tools$ source /etc/profile
feicong@feicong-ubuntu:~/tools$ emulator -version
Android emulator version 18.0 (build id MASTER-306762)
Copyright (C) 2006-2011 The Android Open Source Project and many others.
This program is a derivative of the QEMU CPU emulator (www.qemu.org).

This software is licensed under the terms of the GNU General Public
License version 2, as published by the Free Software Foundation, and
may be copied, distributed, and modified under those terms.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

feicong@feicong-ubuntu:~/tools$ adb version
Android Debug Bridge version 1.0.29
feicong@feicong-ubuntu:~/tools$ 

```

图1-11 测试Android SDK

配置好环境后就需要下载具体版本的 SDK 了，在终端提示符中输入 android 命令启动 Android SDK Manager，接下来的下载步骤与 Windows 平台是一样的，具体操作这里就不再赘述了。

1.2.4 在 Ubuntu 上安装 Android NDK

首先到 Android 官方网站 <http://developer.android.com/sdk/ndk/index.html> 下载 Android NDK，目前 Linux 平台的最新版本为 r18，将下载的 android-ndk-r8-linux-x86.tar.bz2 文件放到 tools 目录，在终端提示符下输入以下命令解包：

```
tar jxvf ./android-ndk-r8-linux-x86.tar.bz2
```

解包完毕后就会在当前目录下出现 android-ndk-r8 目录了。接着设置环境变量，执行：

```
sudo gedit /etc/profile
```

在配置文件中加入如下部分：

```
export ANDROID_NDK=/home/feicong/tools/android-ndk-r8
export PATH=/home/feicong/tools/android-ndk-r8:$PATH
```

保存文件后退出，在终端提示符中输入“source /etc/profile”使环境变量生效。接下来在终端提示符下进入 android-ndk-r8/samples/hello-jni 目录，然后输入 ndk-build 命令编译 hello-jni 工程，如果配置正确，执行结果如图 1-12 所示。

```
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
android-ndk-r8/build/tools/toolchain-patches/gdb/0004-ndk-Fix-MIPS-builds-on-Darwin.patch
android-ndk-r8/build/tools/toolchain-symbols/
android-ndk-r8/build/tools/toolchain-symbols/arm/
android-ndk-r8/build/tools/toolchain-symbols/arm/libgcc.a.functions.txt
android-ndk-r8/build/tools/toolchain-symbols/mips/
android-ndk-r8/build/tools/toolchain-symbols/mips/libgcc.a.functions.txt
android-ndk-r8/build/tools/toolchain-symbols/x86/
android-ndk-r8/build/tools/toolchain-symbols/x86/libgcc.a.functions.txt
android-ndk-r8/RELEASE.TXT
android-ndk-r8/README.TXT
android-ndk-r8/GNUmakefile
feicong@feicong-ubuntu:~/tools$ sudo gedit /etc/profile
[sudo] password for feicong:
feicong@feicong-ubuntu:~/tools$ cd android-ndk-r8/
feicong@feicong-ubuntu:~/tools/android-ndk-r8$ cd samples/
feicong@feicong-ubuntu:~/tools/android-ndk-r8/samples$ cd hello-jni/
feicong@feicong-ubuntu:~/tools/android-ndk-r8/samples/hello-jni$ ndk-build
Gdbserver : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup   : libs/armeabi/gdb.setup
Compile thumb : hello-jni <= hello-jni.c
SharedLibrary : libhello-jni.so
Install      : libhello-jni.so => libs/armeabi/libhello-jni.so
feicong@feicong-ubuntu:~/tools/android-ndk-r8/samples/hello-jni$
```

图1-12 使用Android NDK编译工程

1.2.5 在 Ubuntu 上安装 Eclipse 集成开发环境

首先到 Eclipse 官方网站 <http://www.eclipse.org/downloads/> 下载 Eclipse IDE for Java Developers 版本，将下载到的 eclipse-jee-indigo-SR2-linux-gtk.tar.gz 文件放到 tools 目录，输入以下命令解包：

```
tar zxvf ./eclipse-jee-indigo-SR2-linux-gtk.tar.gz
```

解包完毕后就会在当前目录下出现 eclipse 目录。目录中的 eclipse 文件就是主程序，为方便以后使用可以在桌面上建立快捷方式。

1.2.6 在 Ubuntu 上安装 CDT、ADT 插件

Ubuntu 上安装 CDT、ADT、Pydev 插件与 Windows 平台上的安装步骤是一样的，读者可以参考 1.1.5 小节内容进行安装，这里不再赘述。

1.2.7 创建 Android Virtual Device

Linux 版的 Android SDK 没有提供可视化的 AVD Manager 管理工具，创建 AVD 可以使用 android 命令。在终端提示符下输入“android list targets”列出本机已经下载好的 SDK，本机输出结果如下：

```
feicong@feicong-ubuntu:~/tools$ android list targets
Available Android targets:
.....
-----
id: 2 or "android-8"
    Name: Android 2.2
    Type: Platform
    API level: 8
    Revision: 3
    Skins: WVGA800 (default), HVGA, QVGA, WVGA854, WQVGA432, WQVGA400
    ABIs : armeabi
-----
id: 3 or "android-10"
    Name: Android 2.3.3
    Type: Platform
    API level: 10
    Revision: 2
    Skins: WVGA800 (default), HVGA, QVGA, WVGA854, WQVGA432, WQVGA400
    ABIs : armeabi
.....
-----
id: 6 or "android-15"
    Name: Android 4.0.3
    Type: Platform
    API level: 15
    Revision: 3
    Skins: WSVA, WVGA800 (default), HVGA, QVGA, WVGA854, WXGA720, WXGA800,
    WQVGA432, WQVGA400
    ABIs : armeabi-v7a
```

每一个 id 对应一个版本的 SDK。这个 id 在创建 AVD 时会使用到。创建 AVD 的命令格式为“`android create avd --name <your_avd_name> --target <targetID>`”，比如想要创建 Android 系统版本为 2.3.3 且名称为 android2.3.3 的 AVD 只需在终端提示符下输入如下命令：

```
android create avd --name android2.3.3 --target android-10
```

创建 AVD 完成后可以使用 `emulator` 来启动它，在终端提示符下输入命令：

```
emulator -avd android2.3.3
```

最终运行效果如图 1-13 所示。

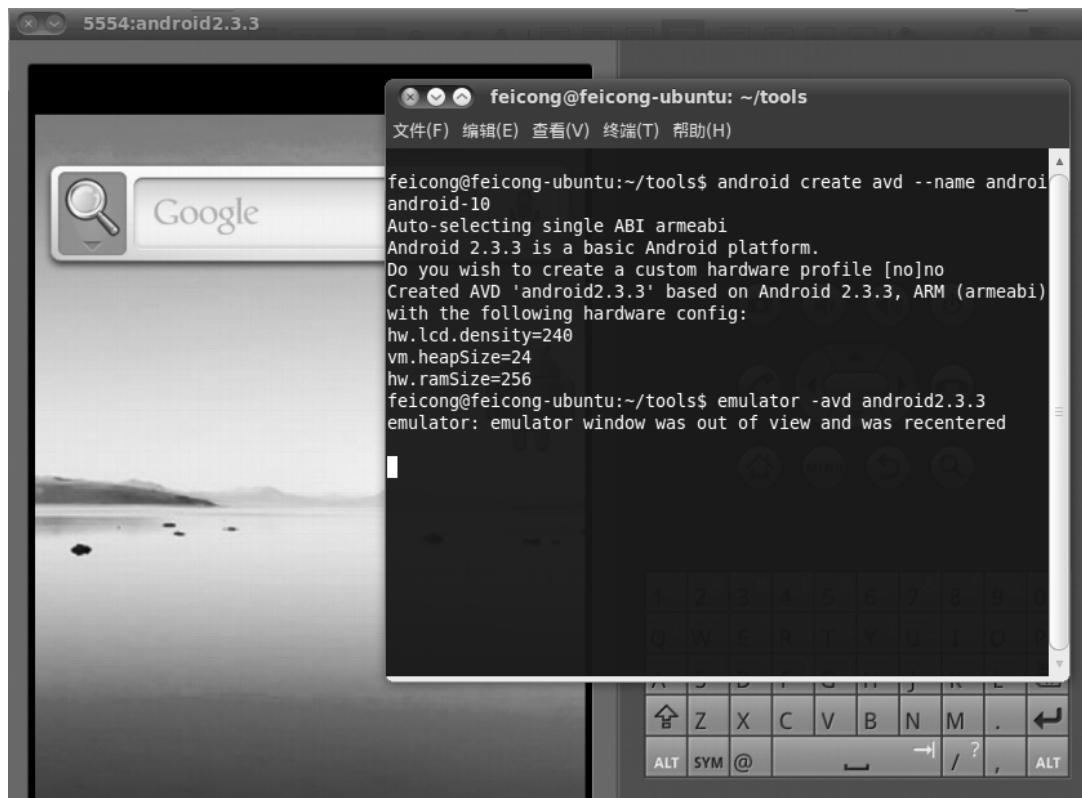


图1-13 Android模拟器运行界面

如果使用真实 Android 设备来调试程序，还需要做一些工作。首先需要在设备的“设置→程序→开发”选项中开启“USB 调试”，接着将设备连接电脑，在终端提示符中输入 `lsusb` 命令查看连接的 USB 设备。我的测试机型为 Moto XT615，命令执行后会得到如下输入。

```
feicong@feicong-ubuntu:~$ lsusb
Bus 003 Device 002: ID 15d9:0a4c Dexon
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 005: ID 0bda:0158 Realtek Semiconductor Corp. Mass Storage Device
Bus 001 Device 004: ID 22b8:2de6 Motorola PCS
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

其中 22b8 为 Vendor id 值，2de6 为 Product id。不同的设备厂商 Vendor id 值不同。可以在<http://developer.android.com/tools/device.html#VendorIds>找到一份常见设备厂商的 Vendor id 列表。Product id 则是具体产品的 id 值。同一厂商的不同设备 Vendor id 相同而 Product id 不同。记录下 Vendor id 与 Product id 值，然后编辑 udev 规则文件/etc/udev/rules.d/70-android.rules，没有则创建，内容如下。

```
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", MODE="2de6", GROUP="plugdev"
```

其中的 22b8 与 2de6 根据自己的 Vendor id 与 Product id 值进行相应的更改，修改保存后退出，在终端提示符中输入命令“adb devices”就能列出配置好的 Android 设备了。

1.2.8 使用到的工具

本书讲解 Android 程序分析时使用到的工具大多数是跨平台的，这些工具同时拥有 Windows 版本与 Linux 版本，笔者在介绍它们时会给出相应工具的下载地址，并且给出其安装方法。另外，本书中也有个别的工具是与平台相关的，笔者在书中都有详细的说明。

1.3 本章小结

本章主要介绍了分析 Android 程序时常用到的一些工具，熟练掌握这些工具的使用是分析 Android 程序的基础，接着介绍了 Windows 平台与 Ubuntu 平台下 Android 开发环境的搭建，为后面的分析环境做准备。在学习完本章后，读者可以自行下载本章所提及到的工具，动手练习如何使用它们，这些工具的具体使用会在本书后面章节逐一进行介绍。

第2章 如何分析Android程序

分析Android程序是开发Android程序的一个逆过程。然而作为分析人员，掌握分析技术还得从开发学起，这个学习的路线应该是呈线性的、循序渐进的。要想分析一个Android程序，首先应该了解Android程序开发的流程、程序结构、语句分支、解密原理等。本章将以走马观花的形式，从开发Android程序开始，到最终分析并破解这个程序，将这一完整路线展现出来。

2.1 编写第一个Android程序

本节将采用Android官方推荐的Eclipse集成开发环境来编写一个Android应用程序。

2.1.1 使用Eclipse创建Android工程

启动Eclipse，新建一个Android工程。“Application Name”为Crackme0201，“Project Name”为crackme02，“Package Name”为com.droider.crackme0201，其它保持默认，设置好后如图2-1所示。

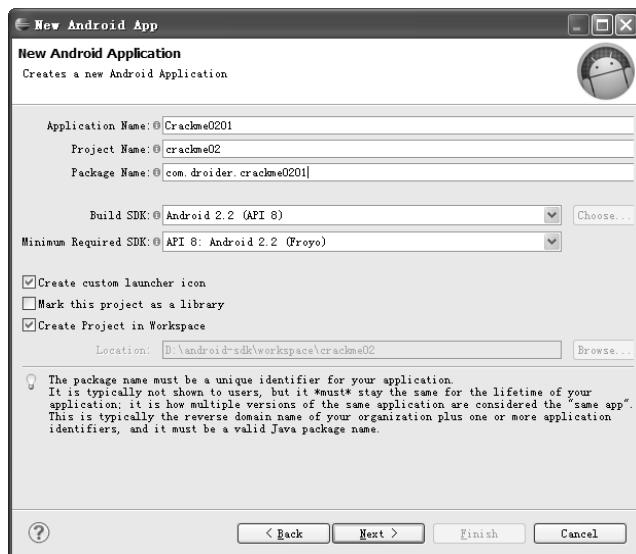


图2-1 使用Eclipse创建Android工程

一路单击 Next 按钮，最后点击 Finish 按钮完成工程的创建。打开工程的 activity_main.xml 布局文件，添加用户名与注册码编辑框，修改完成后，最终界面效果如图 2-2 所示。

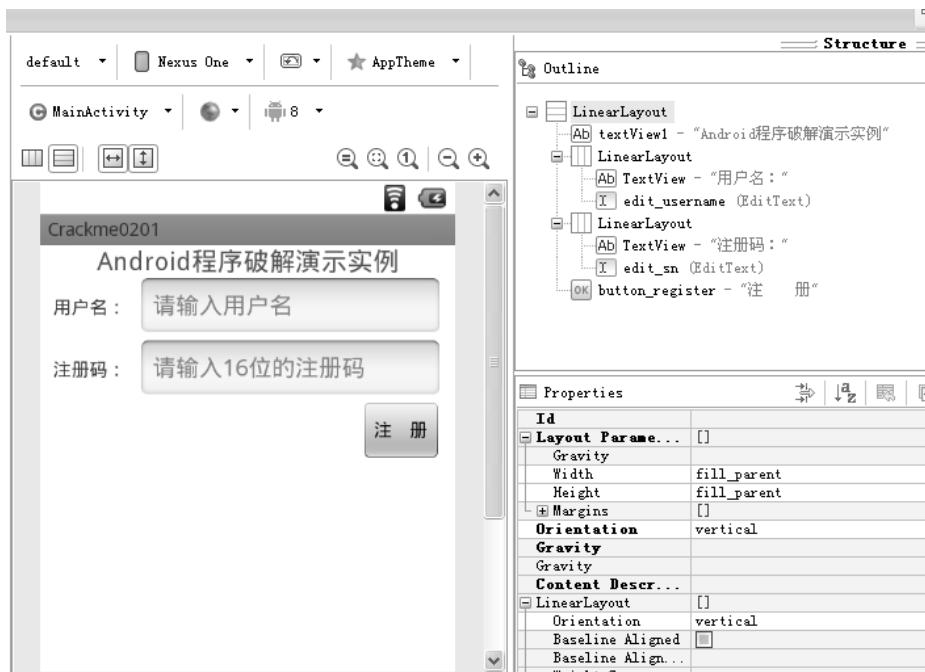


图2-2 crackme程序主界面

接着编写 MainActivity 类的代码，添加一个 checkSN()方法，代码如下：

```
private boolean checkSN(String userName, String sn) {
    try {
        if ((userName == null) || (userName.length() == 0))
            return false;
        if ((sn == null) || (sn.length() != 16))
            return false;
        MessageDigest digest = MessageDigest.getInstance("MD5");
        digest.reset();
        digest.update(userName.getBytes());
        byte[] bytes = digest.digest(); //采用MD5对用户名进行Hash
        String hexstr = toHexString(bytes, ""); //将计算结果转化成字符串
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < hexstr.length(); i += 2) {
            sb.append(hexstr.charAt(i));
        }
    }
```

```

        String userSN = sb.toString(); //计算出的SN
        if (!userSN.equalsIgnoreCase(sn)) //比较注册码是否正确
            return false;
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}

```

这个方法的主要功能是计算用户名与注册码是否匹配。计算的步骤为：使用 MD5 算法计算用户名字符串的 Hash，将计算所得的结果转换成长度为 32 位的十六进制字符串，然后取字符串的所有奇数位重新组合生成新的字符串，这个字符串就是最终的注册码，最后将它与传入的注册码进行比较，如果相同表示注册码是正确的，反之注册码是错误的。

接着在 MainActivity 的 OnCreate() 方法中加入注册按钮点击事件的监听器，如果用户名与注册码匹配就弹出注册成功的提示，不匹配则提示无效的用户名或注册码，代码如下：

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    setTitle(R.string.unregister); //模拟程序未注册
    edit_userName = (EditText) findViewById(R.id.edit_username);
    edit_sn = (EditText) findViewById(R.id.edit_sn);
    btn_register = (Button) findViewById(R.id.button_register);
    btn_register.setOnClickListener(new OnClickListener() {

        public void onClick(View v) {
            if (!checkSN(edit_userName.getText().toString().trim(),
                edit_sn.getText().toString().trim())) {
                Toast.makeText(MainActivity.this, //弹出无效用户名或注册码提示
                    R.string.unsuccessed, Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(MainActivity.this, //弹出注册成功提示
                    R.string.successed, Toast.LENGTH_SHORT).show();
                btn_register.setEnabled(false);
                setTitle(R.string.registered); //模拟程序已注册
            }
        }
    });
}

```

2.1.2 编译生成 APK 文件

首先启动 Android 程序运行环境，然后在 crackme02 工程上点击右键，在弹出的菜单中选择“Run As→Android Application”，如果工程没有任何错误，Eclipse 会根据默认配置编译并启动 crackme02 程序。可以通过菜单项“Run As→Run Configuration”更改默认的启动配置选项，如程序启动的第一个 Activity、Android 平台的版本等。

注意 Android 程序运行环境可以使用真实 Android 设备或 Android 模拟器，如果读者手上没有 Android 设备。可以按照本书 1.2.7 小节讲解步骤创建模拟器，然后在命令行下输入以下命令启动它：

emulator -avd <模拟器名称>

以后在讲解过程中启动 Android 程序运行环境的步骤不再赘述。

程序启动后，输入任意长度的用户名与 16 位长度的注册码，点击注册按钮，程序会模拟注册软件的执行效果，如图 2-3 所示。



图2-3 crackme02程序的运行效果

2.2 破解第一个程序

本节将以上一节编写的 crackme02 程序为例，讲解破解它的完整流程。

2.2.1 如何动手？

破解 Android 程序通常的方法是将 apk 文件利用 ApkTool 反编译，生成 Smali 格式的反汇编代码，然后阅读 Smali 文件的代码来理解程序的运行机制，找到程序的突破口进行修改，最后使用 ApkTool 重新编译生成 apk 文件并签名，最后运行测试，如此循环，直至程序被成功破解。

在实际的分析过程中，还可以使用 IDA Pro 直接分析 apk 文件，或者 dex2jar 与 jd-gui 配合来进行 Java 源码级的分析等，这些分析方法会在本书的第 5 章进行详细的介绍。

2.2.2 反编译 APK 文件

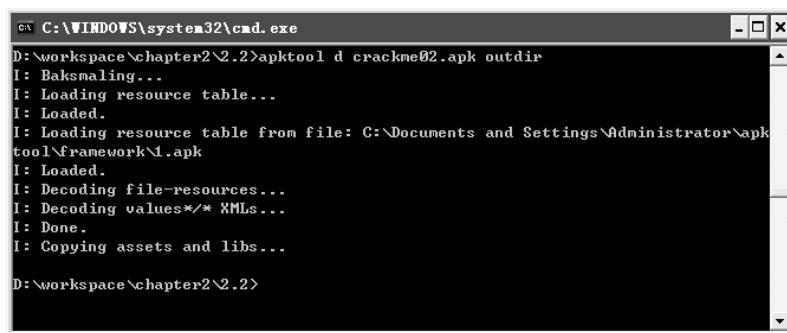
ApkTool 是跨平台的工具，可以在 Windows 平台与 Ubuntu 平台下直接使用。使用前到 <http://code.google.com/p/android-apktool/> 下载 ApkTool，目前最新版本为 1.4.3，Windows 平台需要下载 apktool1.4.3.tar.bz2 与 apktool-install-windows-r04-brut1.tar.bz2 两个压缩包，将下载后的文件解压到同一目录下，本书的 Windows 平台将其解压到“D:\tools\Android\apktool”目录，解压完成后将该目录添加到系统的 PATH 环境变量中，以便在命令行下直接使用，如果是 Ubuntu 系统则需要下载 apktool1.4.3.tar.bz2 与 apktool-install-linux-r04-brut1.tar.bz2，本书的 Ubuntu 平台将其解压到“/home/feicong/tools/apktool”目录下。

在 Windows 平台下，打开一个 CMD 窗口，在命令行下直接输入 apktool 会列出程序的用法。

反编译 apk 文件的命令为： *apktool d[ecode] [OPTS] <file.apk> [<dir>]*

编译 apk 文件的命令为： *apktool b[uild] [OPTS] [<app_path>] [<out_file>]*

在命令行下进入要反编译的 apk 文件目录，输入命令：“apktool d crackme02.apk outdir”，稍等片刻，程序就会反编译完成，如图 2-4 所示。



```
C:\WINDOWS\system32\cmd.exe
D:\workspace\chapter2\2.2>apktool d crackme02.apk outdir
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Documents and Settings\Administrator\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
D:\workspace\chapter2\2.2>
```

图2-4 使用Apktool反编译apk文件

在Ubuntu平台下，使用Apktool与在Windows平台下基本相同，具体步骤读者可自行实践。

2.2.3 分析APK文件

反编译apk文件成功后，会在当前的outdir目录下生成一系列目录与文件。其中smali目录下存放了程序所有的反汇编代码，res目录则是程序中所有的资源文件，这些目录的子目录和文件与开发时的源码目录组织结构是一致的。

如何寻找突破口是分析一个程序的关键。对于一般的Android来说，错误提示信息通常是指引关键代码的风向标，在错误提示附近一般是程序的核心验证代码，分析人员需要阅读这些代码来理解软件的注册流程。

错误提示是Android程序中的字符串资源，开发Android程序时，这些字符串可能硬编码到源码中，也可能引用自“res\values”目录下的strings.xml文件，apk文件在打包时，strings.xml中的字符串被加密存储为resources.arsc文件保存到apk程序包中，apk被成功反编译后这个文件也被解密出来了。

还记得2.1.2小节运行程序时的错误提示吗？在软件注册失败时会Toast弹出“无效用户名或注册码”，我们以此为线索来寻找关键代码。打开“res\values\string.xml”文件，内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Crackme0201</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>
    <string name="title_activity_main">crackme02</string>
    <string name="info">Android程序破解演示实例</string>
    <string name="username">用户名: </string>
    <string name="sn">注册码: </string>
    <string name="register">注 册</string>
    <string name="hint_username">请输入用户名</string>
    <string name="hint_sn">请输入16位的注册码</string>
    <string name="unregister">程序未注册</string>
    <string name="registered">程序已注册</string>
    <string name="unsuccessed">无效用户名或注册码</string>
    <string name="successeed">恭喜您！注册成功</string>
</resources>
```

开发Android程序时，String.xml文件中的所有字符串资源都在“gen/<packagename>/R.java”文件的String类中被标识，每个字符串都有唯一的int类型索引值，使用Apktool反编译apk文件后，所有的索引值保存在string.xml文件同目录下的public.xml文件中。

从上面列表中找到“无效用户名或注册码”的字符串名称为 unsuccessed。打开 public.xml 文件，它的内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <public type="drawable" name="ic_launcher" id="0x7f020001" />
    <public type="drawable" name="ic_action_search" id="0x7f020000" />
    <public type="layout" name="activity_main" id="0x7f030000" />
    <public type="dimen" name="padding_small" id="0x7f040000" />
    <public type="dimen" name="padding_medium" id="0x7f040001" />
    <public type="dimen" name="padding_large" id="0x7f040002" />
    <public type="string" name="app_name" id="0x7f050000" />
    <public type="string" name="hello_world" id="0x7f050001" />
    <public type="string" name="menu_settings" id="0x7f050002" />
    <public type="string" name="title_activity_main" id="0x7f050003" />
    <public type="string" name="info" id="0x7f050004" />
    <public type="string" name="username" id="0x7f050005" />
    <public type="string" name="sn" id="0x7f050006" />
    <public type="string" name="register" id="0x7f050007" />
    <public type="string" name="hint_username" id="0x7f050008" />
    <public type="string" name="hint_sn" id="0x7f050009" />
    <public type="string" name="unregister" id="0x7f05000a" />
    <public type="string" name="registered" id="0x7f05000b" />
    <b><public type="string" name="unsuccessed" id="0x7f05000c" /></b>
    <public type="string" name="successted" id="0x7f05000d" />
    <public type="style" name="AppTheme" id="0x7f060000" />
    <public type="menu" name="activity_main" id="0x7f070000" />
    <public type="id" name="textView1" id="0x7f080000" />
    <public type="id" name="edit_username" id="0x7f080001" />
    <public type="id" name="edit_sn" id="0x7f080002" />
    <public type="id" name="button_register" id="0x7f080003" />
    <public type="id" name="menu_settings" id="0x7f080004" />
</resources>
```

unsuccessed 的 id 值为 0x7f05000c，在 smali 目录中搜索含有内容为 0x7f05000c 的文件，最后发现只有 MainActivity\$1.smali 文件一处调用，代码如下：

```
# virtual methods
.method public onClick(Landroid/view/View;)V
    .locals 4
    .parameter "v"
    .prologue
```

```
const/4 v3, 0x0
.....
.line 32
#calls:
Lcom/droider/crackme0201/MainActivity;->checkSN(Ljava/lang/String;
Ljava/lang/String;)Z
invoke-static {v0, v1, v2}, Lcom/droider/crackme0201/MainActivity;-> # 检查注册码是否合法
access$2(Lcom/droider/crackme0201/MainActivity;Ljava/lang/String;Ljava/
lang/String;)Z
move-result v0
if-nez v0, :cond_0 #如果结果不为0, 就跳转到cond_0标号处
.line 34
iget-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
    this$0:Lcom/droider/crackme0201/MainActivity;
.line 35
const v1, 0x7f05000c# unsuccessted字符串
.line 34
invoke-static {v0, v1, v3}, Landroid/widget/Toast;->
    makeText(Landroid/content/Context;II)Landroid/widget/Toast;
move-result-object v0
.line 35
invoke-virtual {v0}, Landroid/widget/Toast;->show()V
.line 42
:goto_0
return-void
.line 37
:cond_0
iget-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
    this$0:Lcom/droider/crackme0201/MainActivity;
.line 38
const v1, 0x7f05000d# successed字符串
.line 37
invoke-static {v0, v1, v3}, Landroid/widget/Toast;->
    makeText(Landroid/content/Context;II)Landroid/widget/Toast;
move-result-object v0
.line 38
invoke-virtual {v0}, Landroid/widget/Toast;->show()V
.line 39
iget-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
    this$0:Lcom/droider/crackme0201/MainActivity;
```

```

#getter for: Lcom/droider/crackme0201/MainActivity;->btn_register:Landroid/
    widget/Button;
invoke-static {v0}, Lcom/droider/crackme0201/MainActivity;->
    access$3(Lcom/droider/crackme0201/MainActivity;)Landroid/widget
    /Button;
move-result-object v0
invoke-virtual {v0, v3}, Landroid/widget/Button;->setEnabled(z)V
#设置注册按钮不可用
.line 40
iget-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
    this$0:Lcom/droider/crackme0201/MainActivity;
const v1, 0x7f05000b# registered字符串, 模拟注册成功
invoke-virtual {v0, v1}, Lcom/droider/crackme0201/MainActivity;->
    setTitle(I)V
goto :goto_0
.end method

```

Smali 代码中添加的注释使用井号“#”开头，“.line 32”行调用了 checkSN() 函数进行注册码的合法检查，接着下面有如下两行代码：

```

move-result v0
if-nez v0, :cond_0

```

checkSN() 函数返回 Boolean 类型的值。这里的第一行代码将返回的结果保存到 v0 寄存器中，第二行代码对 v0 进行判断，如果 v0 的值不为零，即条件为真的情况下，跳转到 cond_0 标号处，反之，程序顺利向下执行。

如果代码不跳转，会执行如下几行代码：

```

.line 34
iget-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
    this$0:Lcom/droider/crackme0201/MainActivity;
.line 35
const v1, 0x7f05000c    # unsuccessed字符串
.line 34
invoke-static {v0, v1, v3}, Landroid/widget/Toast;->
    makeText(Landroid/content/Context;II)Landroid/widget/Toast;
move-result-object v0
.line 35
invoke-virtual {v0}, Landroid/widget/Toast;->show()V
.line 42
:goto_0
return-void

```

“.line 34”行使用 ige-object 指令获取 MainActivity 实例的引用。代码中的->this\$0 是内部类 MainActivity\$1 中的一个 synthetic 字段，存储的是父类 MainActivity 的引用，这是 Java 语言的一个特性，类似的还有->access\$0，这一类代码会在本书的第 5 章进行详细介绍。

“.line 35”行将 v1 寄存器传入 unsuccessed 字符串的 id 值，接着调用 Toast;->makeText() 创建字符串，然后调用 Toast;->show()V 方法弹出提示，最后.line 40 行调用 return-void 函数返回。

如果代码跳转，会执行如下代码：

```
:cond_0
    ige-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
        this$0:Lcom/droider/crackme0201/MainActivity;
    .line 38
    const v1, 0x7f05000d      # successed字符串
    .line 37
    invoke-static {v0, v1, v3}, Landroid/widget/Toast;->
        1makeText(Landroid/content/Context;II)Landroid/widget/Toast;
    move-result-object v0
    .line 38
    invoke-virtual {v0}, Landroid/widget/Toast;->show()V
    .line 39
    ige-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
        this$0:Lcom/droider/crackme0201/MainActivity;
    #getter for: Lcom/droider/crackme0201/MainActivity;->btn_register:Landroid/
        widget/Button;
    invoke-static {v0}, Lcom/droider/crackme0201/MainActivity;->
        access$3(Lcom/droider/crackme0201/MainActivity;)Landroid/widget/Button;
    move-result-object v0
    invoke-virtual {v0, v3}, Landroid/widget/Button;->setEnabled(Z)V  #设置注
    册按钮不可用
    .line 40
    ige-object v0, p0, Lcom/droider/crackme0201/MainActivity$1;->
        this$0:Lcom/droider/crackme0201/MainActivity;
    const v1, 0x7f05000b      # registered字符串，模拟注册成功
    invoke-virtual {v0, v1}, Lcom/droider/crackme0201/MainActivity;->setTitle(I)V
    goto :goto_0
```

这段代码的功能是弹出注册成功提示，也就是说，上面的跳转如果成功意味着程序会成功注册。

注意 Smali 代码的语法与格式会在本书第 3 章进行详细介绍。

2.2.4 修改 Smali 文件代码

经过上一小节的分析可以发现，“.line 32”行的代码“if-nez v0, :cond_0”是程序的破解点。if-nez 是 Dalvik 指令集中的一个条件跳转指令，类似的还有 if-eqz、if-gez、if-lez 等。这些指令会在本书第 3 章进行介绍，读者在这里只需要知道，与 if-nez 指令功能相反的指令为 if-eqz，表示比较结果为 0 或相等时进行跳转。

用任意一款文本编辑器打开 MainActivity\$1.smali 文件，将“.line 32”行的代码“if-nez v0, :cond_0”修改为“if-eqz v0, :cond_0”，保存后退出，代码就算修改完成了。

2.2.5 重新编译 APK 文件并签名

修改完 Smali 文件代码后，需要将修改后的文件重新进行编译打包成 apk 文件。2.2.2 小节中我们已经知道编译 apk 文件的命令格式为“*apktool b[uild] [OPTS] [<app_path> [<out_file>]*”，打开 CMD 命令提示符窗口，进入到 outdir 同目录，执行以下命令。

```
apktool b outdir
```

不出意外的话，程序就会编译成功。命令输出结果如图 2-5 所示，编译成功后会在 outdir 目录下生成 dist 目录，里面存放着编译成功的 apk 文件。

```
C:\WINDOWS\system32\cmd.exe
D:\workspace\chapter2\2.2>apktool b outdir
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building apk file...

D:\workspace\chapter2\2.2>
```

图2-5 使用ApkTool重新打包Android程序

编译生成的 crackme02.apk 没有签名，还不能安装测试，接下来需要使用 signapk.jar 工具对 apk 文件进行签名。signapk.jar 是 Android 源码包中的一个签名工具。代码位于 Android 源码目录下的/build/tools/signapk/SignApk.java 文件中，源码编译后可以在/out/host/linux-x86/framework 目录中找到它。使用 signapk.jar 签名时需要提供签名文件，我们在此可以使用 Android 源码中提供的签名文件 testkey.pk8 与 testkey.x509.pem，它们位于 Android 源码的 build/target/product/security 目录。新建 signapk.bat 文件，内容为：

```
java -jar "%~dp0signapk.jar" "%~dp0testkey.x509.pem" "%~dp0testkey.pk8" %1
signed.apk
```

将 signapk.jar、signapk.bat、testkey.x509.pem、testkey.pk8 等 4 个文件放到同一目录并添

加到系统 PATH 环境变量中，然后在命令提示符下输入如下命令对 APK 文件进行签名。

```
signapk crackme02.apk
```

签名成功后会在同目录下生成 signed.apk 文件。

2.2.6 安装测试

现在是时候测试修改后的成果了。

启动一个 Android AVD，或者使用数据线连接手机与电脑，然后在命令提示符下执行以下命令安装破解后的程序。

```
adb install signed.apk
```

不出意外会得到以下输出提示：

```
adb install signed.apk
822 KB/s (39472 bytes in 0.046s)
pkg: /data/local/tmp/signed.apk
Success
```

安装完成后启动 crackme02，在用户名与注册码输入框中输入任意字符，点击注册按钮，程序会弹出注册成功提示，并且标题栏字符会变成已注册字样。如图 2-6 所示。



图2-6 测试破解后的程序

2.3 本章小结

本章通过一个实例介绍了 Android 程序的一般分析与破解流程。在实际的分析过程中，接触到的代码远比这些要复杂得多，有些代码甚至经过混淆处理过，很难阅读，这样就需要使用其它手段如动态调试结合一些其它的技巧来辅助分析，这些更深入的内容会在本书的后续章节中进行介绍。

第 5 章 静态分析 Android 程序

如果说前面的章节是在“扎马步”，那么从本章起，就是真正的武功招式了。静态分析是探索 Android 程序内幕的一种最常见的方法，它与动态调试双剑合璧，帮助分析人员解决分析时遇到的各类“疑难”问题。然而，静态分析技术本身需要分析人员具备较强的代码理解能力，这些都需要在平时的开发过程中不断地积累经验，很难想象一个连 Android 应用程序源码都看不懂的人去逆向分析 Android 程序。因此，在开始本章内容之前，假定读者已经具备了基本的 Android 程序开发知识与代码阅读能力。

5.1 什么是静态分析

静态分析（Static Analysis）是指在不运行代码的情况下，采用词法分析、语法分析等各种技术手段对程序文件进行扫描从而生成程序的反汇编代码，然后阅读反汇编代码来掌握程序功能的一种技术。在实际的分析过程中，完全不运行程序是不太可能的，分析人员时常需要先运行目标程序来寻找程序的突破口。静态分析强调的是静态，在整个分析的过程中，阅读反汇编代码是主要的分析工作。生成反汇编代码的工具称为反汇编工具或反编译工具，选择一个功能强大的反汇编工具不仅能获得更好的反汇编效果，而且也能为分析人员节省不少时间。

静态分析 Android 程序有两种方法：一种方法是阅读反汇编生成的 Dalvik 字节码，可以使用 IDA Pro 分析 dex 文件，或者使用文本编辑器阅读 bksmali 反编译生成的 smali 文件；另一种方法是阅读反汇编生成的 Java 源码，可以使用 dex2jar 生成 jar 文件，然后再使用 jd-gui 阅读 jar 文件的代码。

5.2 快速定位 Android 程序的关键代码

在逆向一个 Android 软件时，如果盲目的分析，可能需要阅读成千上万行的反汇编代码才能找到程序的关键点，这无疑是浪费时间的表现，本小节将介绍如何快速的定位程序的关键代码。

5.2.1 反编译 apk 程序

每个 apk 文件中都包含有一个 AndroidManifest.xml 文件，它记录着软件的一些基本信息。包括软件的包名、运行的系统版本、用到的组件等。并且这个文件被加密存储进了 apk 文件

中，在开始分析前，有必要先反编译 apk 文件对其进行解密。反编译 apk 的工具使用前面章节介绍过的 Apktool。Apktool 提供了反编译与打包 apk 文件的功能。本小节使用到的实例程序为 crackme0502.apk，按照前面使用 Apktool 的步骤，在命令提示符下输入“apktool d crackme0502.apk”即可反编译成功。

5.2.2 程序的主 Activity

我们知道，一个 Android 程序由一个或多个 Activity 以及其它组件组成，每个 Activity 都是相同级别的，不同的 Activity 实现不同的功能。每个 Activity 都是 Android 程序的一个显示“页面”，主要负责数据的处理及展示工作，在 Android 程序的开发过程中，程序员很多时候是在编写用户与 Activity 之间的交互代码。

每个 Android 程序有且只有一个主 Activity（隐藏程序除外，它没有主 Activity），它是程序启动的第一个 Activity。打开 crackme0502 文件夹下的 AndroidManifest.xml 文件，其中有如下片断的代码。

```
<activity android:label="@string/title_activity_main" android:name=".  
MainActivity">  
    <intent-filter>  
        <action android:name="android.intent.action.MAIN" />  
        <category android:name="android.intent.category.LAUNCHER" />  
    </intent-filter>  
</activity>
```

在程序中使用到的 Activity 都需要在 AndroidManifest.xml 文件中手动声明，声明 Activity 使用 activity 标签，其中 android:label 指定 Activity 的标题，android:name 指定具体的 Activity 类，“.MainActivity”前面省略了程序的包名，完整类名应该为 com.droider.crackme0502.MainActivity，intent-filter 指定了 Activity 的启动意图，android.intent.action.MAIN 表示这个 Activity 是程序的主 Activity。android.intent.category.LAUNCHER 表示这个 Activity 可以通过 LAUNCHER 来启动。如果 AndroidManifest.xml 中，所有的 Activity 都没有添加 android.intent.category.LAUNCHER，那么该程序安装到 Android 设备上后，在程序列表中是不可见的，同样的，如果没有指定 android.intent.action.MAIN，Android 系统的 LAUNCHER 就无法匹配程序的主 Activity，因此该程序也不会有图标出现。

在反编译出的 AndroidManifest.xml 中找到主 Activity 后，可以直接去查看其所在类的 OnCreate()方法的反汇编代码，对于大多数软件来说，这里就是程序的代码入口处，所有的功能都从这里开始得到执行，我们可以沿着这里一直向下查看，追踪软件的执行流程。

5.2.3 需重点关注的 Application 类

如果需要在程序的组件之间传递全局变量，或者在 Activity 启动之前做一些初始化工作，

就可以考虑使用 Application 类了。使用 Application 时需要在程序中添加一个类继承自 android.app.Application，然后重写它的 OnCreate()方法，在该方法中初始化的全局变量可以在 Android 其它组件中访问，当然前提条件是这些变量具有 public 属性。最后还需要在 AndroidManifest.xml 文件的 Application 标签中添加“android:name”属性，取值为继承自 android.app.Application 的类名。

鉴于 Application 类比程序中其它的类启动得都要早，一些商业软件将授权验证的代码都转移到了该类中。例如，在 OnCreate()方法中检测软件的购买状态，如果状态异常则拒绝程序继续运行。因此，在分析 Android 程序过程中，我们需要先查看该程序是否具有 Application 类，如果有，就要看看它的 OnCreate()方法中是否做了一些影响到逆向分析的初始化工作。

5.2.4 如何定位关键代码——六种方法

一个完整的 Android 程序反编译后的代码量可能非常庞大，要想在这浩如烟海的代码中找到程序的关键代码，是需要很多经验与技巧的。笔者经过长时间的探索，总结了以下几种定位代码的方法。

- 信息反馈法

所谓信息反馈法，是指先运行目标程序，然后根据程序运行时给出的反馈信息作为突破口寻找关键代码。在第 2 章中，我们运行目标程序并输入错误的注册码时，会弹出提示“无效用户名或注册码”，这就是程序反馈给我们的信息。通常情况下，程序中用到的字符串会存储在 String.xml 文件或者硬编码到程序代码中，如果是前者的话，字符串在程序中会以 id 的形式访问，只需在反汇编代码中搜索字符串的 id 值即可找到调用代码处；如果是后者的话，在反汇编代码中直接搜索字符串即可。

- 特征函数法

这种定位代码的方法与信息反馈法类似。在信息反馈法中，无论程序给出什么样的反馈信息，终究是需要调用 Android SDK 中提供的相关 API 函数来完成的。比如弹出注册码错误的提示信息就需要调用 Toast.MakeText().Show()方法，在反汇编代码中直接搜索 Toast 应该很快就能定位到调用代码，如果 Toast 在程序中有多处的话，可能需要分析人员逐个甄别。

- 顺序查看法

顺序查看法是指从软件的启动代码开始，逐行的向下分析，掌握软件的执行流程，这种分析方法在病毒分析时经常用到。

- 代码注入法

代码注入法属于动态调试方法，它的原理是手动修改 apk 文件的反汇编代码，加入 Log 输出，配合 LogCat 查看程序执行到特定点时的状态数据。这种方法在解密程序数据时经常使用，详细的内容会在本书的第 8 章介绍。

- 栈跟踪法

栈跟踪法属于动态调试方法，它的原理是输出运行时的栈跟踪信息，然后查看栈上的函数调用序列来理解方法的执行流程，这种方法的详细内容会在本书的第 8 章介绍。

- Method Profiling

Method Profiling（方法剖析）属于动态调试方法，它主要用于热点分析和性能优化。该功能除了可以记录每个函数占用的 CPU 时间外，还能够跟踪所有的函数调用关系，并提供比栈跟踪法更详细的函数调用序列报告，这种方法在实践中可帮助分析人员节省很多时间，也被广泛使用，详细的内容会在本书的第 8 章介绍。

5.3 smali 文件格式

使用 Apktool 反编译 apk 文件后，会在反编译工程目录下生成一个 smali 文件夹，里面存放着所有反编译出的 smali 文件，这些文件会根据程序包的层次结构生成相应的目录，程序中所有的类都会在相应的目录下生成独立的 smali 文件。如上一节中程序的主 Activity 名为 com.droider.crackme0502.MainActivity，就会在 smali 目录下依次生成 com\droider\crackme0502 目录结构，然后在这个目录下生成 MainActivity.smali 文件。

smali 文件的代码通常情况下比较长，而且指令繁多，在阅读时很难用肉眼捕捉到重点，如果有阅读工具能够将特殊指令（例如条件跳转指令）高亮显示，势必会让分析工作事半功倍，为此笔者专门为文本编辑器 Notepad++ 编写了 smali 语法文件来支持高亮显示与代码折叠，并以此作为 smali 代码的阅读工具。

无论是普通类、抽象类、接口类或者内部类，在反编译出的代码中，它们都以单独的 smali 文件来存放。每个 smali 文件都由若干条语句组成，所有的语句都遵循着一套语法规规范。在 smali 文件的头 3 行描述了当前类的一些信息，格式如下。

```
.class <访问权限> [修饰关键字] <类名>
.super <父类名>
.source <源文件名>
```

打开 MainActivity.smali 文件，头 3 行代码如下。

```
.class public Lcom/droider/crackme0502/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"
```

第 1 行 “.class” 指令指定了当前类的类名。在本例中，类的访问权限为 public，类名为“Lcom/droider/crackme0502/MainActivity;”，类名开头的 L 是遵循 Dalvik 字节码的相关约定，表示后面跟随的字符串为一个类。

第 2 行的 “.super” 指令指定了当前类的父类。本例中的 “Lcom/droider/crackme0502/MainActivity;” 的父类为 “Landroid/app/Activity;”。

第 3 行的 “.source” 指令指定了当前类的源文件名。

回想一下，在上一章中讲解 dex 文件格式时介绍的 DexClassDef 结构，这个结构描述了一个类的详细信息，该结构的第 1 个字段 classIdx 就是类的类型索引，第 3 个字段 superClassIdx 就是指向类的父类类型索引，第 5 个字段 sourceFileIdx 就是指向类的源文件名的字符串索引。baksmbi 在解析 dex 文件时，也是通过这 3 个字段来获取相应的类的值。

注意 经过混淆的 dex 文件，反编译出来的 smali 代码可能没有源文件信息，因此，“.source”行的代码可能为空。

前 3 行代码过后就是类的主体部分了，一个类可以由多个字段或方法组成。smali 文件中字段的声明使用 “.field” 指令。字段有静态字段与实例字段两种。静态字段的声明格式如下。

```
# static fields
.field <访问权限> static [修饰关键字] <字段名>:<字段类型>
```

baksmbi 在生成 smali 文件时，会在静态字段声明的起始处添加“static fields”注释，smali 文件中的注释与 Dalvik 语法一样，也是以井号 “#” 开头。“.field” 指令后面跟着的是访问权限，可以是 public、private、protected 之一。修饰关键字描述了字段的其它属性，如 synthetic。指令的最后是字段名与字段类型，使用冒号 “:” 分隔，语法上与 Dalvik 也是一样的。

实例字段的声明与静态字段类似，只是少了 static 关键字，它的格式如下。

```
# instance fields
.field <访问权限> [修饰关键字] <字段名>:<字段类型>
```

比如以下的实例字段声明。

```
# instance fields
.field private btnAnno:Landroid/widget/Button;
```

第 1 行的 “instance fields” 是 baksmbi 生成的注释，第 2 行表示一个私有字段 btnAnno，它的类型为 “Landroid/widget/Button;”。

如果一个类中含有方法，那么类中必然会有相关方法的反汇编代码，smali 文件中方法的声明使用 “.method” 指令。方法有直接方法与虚方法两种。直接方法的声明格式如下。

```
# direct methods
.method <访问权限> [修饰关键字] <方法原型>
<.locals>
[.parameter]
```

```
[.prologue]
[.line]
<代码体>
.end method
```

“direct methods”是 bksmali 添加的注释，访问权限和修饰关键字与字段的描述相同，方法原型描述了方法的名称、参数与返回值。“.locals”指定了使用的局部变量的个数。“.parameter”指定了方法的参数，与 Dalvik 语法中使用“.parameters”指定参数个数不同，每个“.parameter”指令表明使用一个参数，比如方法中有使用到 3 个参数，那么就会出现 3 条“.parameter”指令。“.prologue”指定了代码的开始处，混淆过的代码可能去掉了该指令。“.line”指定了该处指令在源代码中的行号，同样的，混淆过的代码可能去除了行号信息。

虚方法的声明与直接方法相同，只是起始处的注释为“virtual methods”。

如果一个类实现了接口，会在 smali 文件中使用“.implements”指令指出。相应的格式声明如下。

```
# interfaces
.implements <接口名>
```

“# interfaces”是 bksmali 添加的接口注释，“.implements”是接口关键字，后面的接口名是 DexClassDef 结构中 interfacesOff 字段指定的内容。

如果一个类使用了注解，会在 smali 文件中使用“.annotation”指令指出。注解的格式声明如下。

```
# annotations
.annotation [注解属性] <注解类名>
[注解字段 = 值]
.end annotation
```

注解的作用范围可以是类、方法或字段。如果注解的作用范围是类，“.annotation”指令会直接定义在 smali 文件中，如果是方法或字段，“.annotation”指令则会包含在方法或字段定义中。例如下面的代码。

```
# instance fields
.field public sayWhat:Ljava/lang/String;
.annotation runtime Lcom/droider/anno/MyAnnoField;
    info = "Hello my friend"
.end annotation
.end field
```

实例字段 sayWhat 为 String 类型，它使用了 com.droider.anno.MyAnnoField 注解，注解字段 info 值为“Hello my friend”。将其转换为 Java 代码为：

```
@ com.droider.anno MyAnnoField(info = "Hello my friend")
public String sayWhat;
```

5.4 Android 程序中的类

介绍完了 smali 文件格式，下面我们来看看 Android 程序反汇编后生成了哪些 smali 文件，这些 smali 文件的代码又有些什么特点。

5.4.1 内部类

Java 语言允许在一个类的内部定义另一个类，这种在类中定义的类被称为内部类（Inner Class）。内部类可分为成员内部类、静态嵌套类、方法内部类、匿名内部类。前面我们曾经说过，baksmali 在反编译 dex 文件的时候，会为每个类单独生成了一个 smali 文件，内部类作为一个独立的类，它也拥有自己独立的 smali 文件，只是内部类的文件名形式为 “[外部类]\$/[内部类].smali”，例如下面的类。

```
class Outer {
    class Inner{}}
}
```

baksmali 反编译上述代码后会生成两个文件：Outer.smali 与 Outer\$Inner.smali。查看 5.2 节生成的 smali 文件，发现在 smali\com\droider\crackme0502 目录下有一个 MainActivity\$SNChecker.smali 文件，这个 SNChecker 就是 MainActivity 的一个内部类。打开这个文件，代码结构如下。

```
.class public Lcom/droider/crackme0502/MainActivity$SNChecker;
.super Ljava/lang/Object;
.source "MainActivity.java"

# annotations
.annotation system Ldalvik/annotation/EnclosingClass;
    value = Lcom/droider/crackme0502/MainActivity;
.end annotation
.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x1
    name = "SNChecker"
.end annotation

# instance fields
.field private sn:Ljava/lang/String;
```

```

.field final synthetic this$0:Lcom/droider/crackme0502/MainActivity;

# direct methods
.method public constructor
<init>(Lcom/droider/crackme0502/MainActivity;Ljava/lang/String;)V
.....
.end method

# virtual methods
.method public isRegistered()Z
.....
.end method

```

发现它有两个注解定义块“Ldalvik/annotation/EnclosingClass;”与“Ldalvik/annotation/InnerClass;”、两个实例字段 sn 与 this\$0、一个直接方法 init()、一个虚方法 isRegistered()。注解定义块我们稍后进行讲解。先看它的实例字段，sn 是字符串类型，this\$0 是 MainActivity 类型，synthetic 关键字表明它是“合成”的，那 this\$0 到底是个什么东西呢？

其实 this\$0 是内部类自动保留的一个指向所在外部类的引用。左边的 this 表示为父类的引用，右边的数值 0 表示引用的层数。我们看下面的类。

```

public class Outer { //this$0
    public class FirstInner { //this$1
        public class SecondInner { //this$2
            public class ThirdInner {
            }
        }
    }
}

```

每往里一层右边的数值就加一，如 ThirdInner 类访问 FirstInner 类的引用为 this\$1。在生成的反汇编代码中，this\$X 型字段都被指定了 synthetic 属性，表明它们是被编译器合成的、虚构的，代码的作者并没有声明该字段。

我们再看看 MainActivity\$SNChecker 的构造函数，看它是如何初始化的。代码如下。

```

# direct methods
.method public constructor
<init>(Lcom/droider/crackme0502/MainActivity;Ljava/lang/String;)V
    .locals 0
    .parameter      #第一个参数MainActivity引用
    .parameter "sn" #第二个参数字符串sn

```

```

.prologue
.line 83
input-object p1, p0, Lcom/droider/crackme0502/MainActivity$SNChecker;
->this$0:Lcom/droider/crackme0502/MainActivity;      #将MainActivity引用赋值
给this$0
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V  #调用默认的构造函数
    .line 84
    input-object p2, p0, Lcom/droider/crackme0502/MainActivity$SNChecker; ->
    sn:Ljava/lang/String;
        #将sn字符串的值赋给sn字段
    .line 85
    return-void
.end method

```

细心的读者会发现，这段代码声明时使用“.parameter”指令指定了两个参数，但实际上却使用了 p0~p2 共 3 个寄存器，为什么会出现这种情况呢？在第 3 章介绍 Dalvik 虚拟机时曾经讲过，对于一个非静态的方法而言，会隐含的使用 p0 寄存器当作类的 this 引用。因此，这里的确是使用了 3 个寄存器：p0 表示 MainActivity\$SNChecker 自身的引用，p1 表示 MainActivity 的引用，p2 表示 sn 字符串。另外，从 MainActivity\$SNChecker 的构造函数可以看出，内部类的初始化共有以下 3 个步骤：首先是保存外部类的引用到本类的一个 synthetic 字段中，以便内部类的其它方法使用，然后是调用内部类的父类的构造函数来初始化父类，最后是对内部类自身进行初始化。

5.4.2 监听器

Android 程序开发中大量使用到了监听器，如 Button 的点击事件响应 OnClickListener、Button 的长按事件响应 OnLongClickListener、ListView 列表项的点击事件响应 OnItemSelectedListener 等。由于监听器只是临时使用一次，没有什么复用价值，因此，在实际编写代码的过程中，多采用匿名内部类的形式来实现。如下面的按钮点击事件响应代码。

```

btn.setOnClickListener(new android.view.View.OnClickListener() {

    @Override
    public void onClick(View v) {
        .....
    }
});

```

监听器的实质就是接口，在 Android 系统源码的 frameworks\base\core\java\android\view\View.java 文件中可以发现 OnClickListener 监听器的代码如下。

```

public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}

```

设置按钮点击事件的监听器只需要实现 View.OnClickListener 的 onClick()方法即可。打开5.2节的 MainActivity.smali 文件，在 OnCreate()方法中找到设置按钮点击事件监听器的代码如下。

```

.method public onCreate(Landroid/os/Bundle;)V
    .locals 2
    .parameter "savedInstanceState"
    .....
    .line 32
    ige-object v0, p0, Lcom/droider/crackme0502/MainActivity;->btnAnno:
    Landroid/widget/Button;
    new-instance v1, Lcom/droider/crackme0502/MainActivity$1; #新建一个
    MainActivity$1实例
    invoke-direct {v1, p0}, Lcom/droider/crackme0502/MainActivity$1;
-><init>(Lcom/droider/crackme0502/MainActivity;)V #初始化MainActivity$1
实例
    invoke-virtual {v0, v1}, Landroid/widget/Button;
->setOnClickListener(Landroid/view/View$OnClickListener;)V #设置按钮点击事件
监听器
    .line 40
    ige-object v0, p0, Lcom/droider/crackme0502/MainActivity;
->btnCheckSN:Landroid/widget/Button;
    new-instance v1, Lcom/droider/crackme0502/MainActivity$2; #新建一个
MainActivity$2实例
    invoke-direct {v1, p0}, Lcom/droider/crackme0502/MainActivity$2
-><init>(Lcom/droider/crackme0502/MainActivity;)V; #初始化MainActivity$2实例
    invoke-virtual {v0, v1}, Landroid/widget/Button;
->setOnClickListener(Landroid/view/View$OnClickListener;)V#设置按钮点击事件
监听器
    .line 50
    return-void

```

```
.end method
```

OnCreate() 方法分别调用了按钮对象的 setOnClickListener() 方法来设置点击事件的监听器。第一个按钮传入了一个 MainActivity\$1 对象的引用，第二个按钮传入了一个 MainActivity\$2 对象的引用，我们到 MainActivity\$1.smali 文件中看一下前者的实现，它的代码大致如下。

```
.class Lcom/droider/crackme0502/MainActivity$1;
.super Ljava/lang/Object;
.source "MainActivity.java"

# interfaces
.implements Landroid/view/View$OnClickListener;

# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
    value =
Lcom/droider/crackme0502/MainActivity;->onCreate(Landroid/os/Bundle;)V
.end annotation
.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x0
    name = null
.end annotation

# instance fields
.field final synthetic this$0:Lcom/droider/crackme0502/MainActivity;

# direct methods
.method constructor <init>(Lcom/droider/crackme0502/MainActivity;)V
.....
.end method

# virtual methods
.method public onClick(Landroid/view/View;)V
.....
.end method
```

在 MainActivity\$1.smali 文件的开头使用了 “.implements” 指令指定该类实现了按钮点击事件的监听器接口，因此，这个类实现了它的 OnClickListener() 方法，这也是我们在分析程序时关心的地方。另外，程序中的注解与监听器的构造函数都是编译器为我们自己生成的，实际分析过程中不必关心。

5.4.3 注解类

注解是Java的语言特性，在Android的开发过程中也得到了广泛的使用。Android系统中涉及到注解的包共有两个：一个是dalvik.annotation，该程序包下的注解不对外开放，仅供核心库与代码测试使用，所有的注解声明位于Android系统源码的libcore\dalvik\src\main\java\dalvik\annotation目录下；另一个是android.annotation，相应注解声明位于Android系统源码的frameworks\base\core\java\android\annotation目录下。

在前面介绍的smali文件中，可以发现很多代码都使用到了注解类，首先是MainActivity.smali文件，其中有一段代码如下。

```
# annotations
.annotation system Ldalvik/annotation/MemberClasses;
    value = {
        Lcom/droider/crackme0502/MainActivity$SNChecker;
    }
.end annotation
```

MemberClasses注解是编译时自动加上的，查看MemberClasses注解的源码，代码如下。

```
/**
 * A "system annotation" used to provide the MemberClasses list.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@interface MemberClasses {}
```

从注释可以看出，MemberClasses注解是一个“系统注解”，作用是为父类提供一个MemberClasses列表。MemberClasses即子类成员集合，通俗的讲就是一个内部类列表。

接着是MainActivity\$1.smali文件，其中有一段代码如下。

```
# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
    value = Lcom/droider/crackme0502/MainActivity;->onCreate(Landroid/os/Bundle;)V
.end annotation
```

EnclosingMethod注解用以说明整个MainActivity\$1类的作用范围，其中的Method表明它作用于一个方法，而注解的value表明它位于MainActivity的onCreate()方法中。与EnclosingMethod对应的还有EnclosingClass注解，在MainActivity\$SNChecker.smali文件中有如下一段代码。

```
# annotations
.annotation system Ldalvik/annotation/EnclosingClass;
    value = Lcom/droider/crackme0502/MainActivity;
.end annotation

.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x1
    name = "SNCHECKER"
.end annotation
```

EnclosingClass 注解表明 MainActivity\$SNCHECKER 作用于一个类, 注解的 value 表明这个类是 MainActivity。在 EnclosingClass 注解的下面是 InnerClass, 它表明自身是一个内部类, 其中的 accessFlags 访问标志是一个枚举值, 声明如下。

```
enum {
    kDexVisibilityBuild      = 0x00,      /* annotation visibility */
    kDexVisibilityRuntime     = 0x01,
    kDexVisibilitySystem      = 0x02,
};
```

为 1 表明它的属性是 Runtime。name 为内部类的名称, 本例为 SNCHECKER。

如果注解类在声明时提供了默认值, 那么程序中会使用到 AnnotationDefault 注解。打开 5.2 小节 smali\com\droider\anno 目录下的 MyAnnoClass.smali 文件, 有如下一段代码。

```
# annotations
.annotation system Ldalvik/annotation/AnnotationDefault;
    value = .subannotation Lcom/droider/anno/MyAnnoClass;
        value = "MyAnnoClass"
    .end subannotation
.end annotation
```

可以看到, 此处的 MyAnnoClass 类有一个默认值为 “MyAnnoClass”。

除了以上介绍的注解外, 还有 Signature 与 Throws 注解。Signature 注解用于验证方法的签名, 如下面的代码中, onItemClick()方法的原型与 Signature 注解的 value 值是一致的。

```
.method public
onItemClick(Landroid/widget/AdapterView;Landroid/view/View;IJ)V
    .locals 6
    .parameter
        .parameter "v"
        .parameter "position"
```

```

.parameter "id"
.annotation system Ldalvik/annotation/Signature;
    value = {
        "(",
        "Landroid/widget/AdapterView",
        "<*>;",
        "Landroid/view/View;",
        "IJ)V"
    }
.end annotation
.....
.end method

```

如果方法的声明中使用 throws 关键字抛出异常，则会生成相应的 Throws 注解。示例代码如下。

```

.method public final get()Ljava/lang/Object;
    .locals 1
.annotation system Ldalvik/annotation/Throws;
    value = {
        Ljava/lang/InterruptedException|,
        Ljava/util/concurrent/ExecutionException|
    }
.end annotation
.....
.end method

```

示例的 get() 方法抛出了 InterruptedException 与 ExecutionException 两个异常，将其转换为 Java 代码如下。

```

public final Object get() throws InterruptedException, ExecutionException {
    .....
}

```

以上介绍的注解都是自动生成的，用户不可以在代码中添加使用。在 Android SDK r17 版本中，`android.annotation` 增加了一个 `SuppressLint` 注解，它的作用是辅助开发人员去除代码检查器（Lint API check）添加的警告信息。比如在代码中声明了一个常量但在代码中没有使用它，代码检查器检测到后会在变量所在的代码行添加警告信息（通常的表现为在代码行的最左边添加一个黄色惊叹号的小图标以及在变量名底部会加上一条黄色波浪线），将鼠标指向变量并停留片刻，代码检查器会给出提示建议，如图 5-1 所示。

```

private Button btn1;

@Override
public void onC
super.onCre
setContentV
btnAnno = (Button) findViewById(R.id.btn_annotation);

```

The value of the field MainActivity.btn1 is not used
3 quick fixes available:
Remove 'btn1', keep assignments with side effects
Create getter and setter for 'btn1'...
Add @SuppressWarnings 'unused' to 'btn1'
Press 'F2' for focus

图5-1 代码检查器提示

根据最后一条提示建议添加`@SuppressWarnings("unused")`注解后，警告信息消失。

另外，如果在程序代码中使用到的 API 等级比 `AndroidManifest.xml` 文件中定义的 `minSdkVersion` 要高，代码检查器会在 API 所在代码行添加错误信息。比如在代码中使用了 `File` 类的 `getUsableSpace()` 方法，该 API 要求的最低 SDK 版本为 9，如果 `minSdkVersion` 指定的值为 8，那么代码检查器就会报错误提示，解决方法是在方法或方法所在类的前面添加 “`@TargetApi(9)`”。

除了 `SuppressLint` 与 `TargetApi` 注解，`android.annotation` 包还提供了 `SdkConstant` 与 `Widget` 两个注解，这两个注解在注释中被标记为 “`@hide`”，即在 SDK 中是不可见的。`SdkConstant` 注解指定了 SDK 中可以被导出的常量字段值，`Widget` 注解指定了哪些类是 UI 类，这两个注解在分析 Android 程序时基本上碰不到，此处就不去探究了。

5.4.4 自动生成的类

使用 Android SDK 默认生成的工程会自动添加一些类。这些类在程序发布后会仍然保留在 apk 文件中，目前最新版本的 Android SDK 为 r20 版，经过笔者研究，发现会自动生成如下的类。

首先是 `R` 类，这个类开发 Android 程序的读者应该会很熟悉，工程 `res` 目录下的每个资源都会有一个 `id` 值，这些资源的类型可以是字符串、图片、样式、颜色等。例如我们常见的 `R.java` 代码如下。

```

package com.droider.crackme0502;
public final class R {
    public static final class attr {           //属性
    }
    public static final class dimen {          //尺寸
        public static final int padding_large=0x7f040002;
        public static final int padding_medium=0x7f040001;
        public static final int padding_small=0x7f040000;
    }
}

```

```

public static final class drawable {           //图片
    public static final int ic_action_search=0x7f020000;
    public static final int ic_launcher=0x7f020001;
}
public static final class id {                //id标识
    public static final int btn_annotation=0x7f080000;
    public static final int btn_checksn=0x7f080002;
    public static final int edt_sn=0x7f080001;
    public static final int menu_settings=0x7f080003;
}
public static final class layout {            //布局
    public static final int activity_main=0x7f030000;
}
public static final class menu {              //菜单
    public static final int activity_main=0x7f070000;
}
public static final class string {            //字符串
    public static final int app_name=0x7f050000;
    public static final int hello_world=0x7f050001;
    public static final int menu_settings=0x7f050002;
    public static final int title_activity_main=0x7f050003;
}
public static final class style {             //样式
    public static final int AppTheme=0x7f060000;
}
}

```

由于这些资源类都是 R 类的内部类，因此它们都会独立生成一个类文件，在反编译出的代码中，可以发现有 R.smali、R\$attr.smali、R\$dimen.smali、R\$drawable.smali、R\$id.smali、R\$layout.smali、R\$menu.smali、R\$string.smali、R\$style.smali 等几个文件。

接下来是 BuildConfig 类，该类是在 Android SDK r17 版本中添加的，以后版本的 Android 程序中都有它的身影。这个类中只有一个 boolean 类型的名为 DEBUG 的字段，用来标识程序发布的版本类型。它的值默认是 true，即程序以调试版本发布。由于这个类是自动生成的，如果想将它改为 false，需要先在 Eclipse 开发环境中点击菜单“Project→Build Automatically”关闭自动构建，然后点击菜单“Project→Clean”，现在使用右键菜单“Android Tools→Export Signed Application Package”导出程序，会发现此时 BuildConfig.DEBUG 的值为 false 了。

然后是注解类，如果在代码中使用了 SuppressLint 或 TargetApi 注解，程序中将会包含相应的注解类，在反编译后会在 smali\android\annotation 目录下生成相应的 smali 文件。

Android SDK r20 更新后，会在默认生成的工程中添加 android-support-v4.jar 文件。这个 jar 包是 Android SDK 中提供的兼容包，里面提供了高版本才有的如 Fragment、ViewPager 等控件供低版本的程序调用。关于该包的详细信息请参看 Android 官方文档：<http://developer.android.com/tools/extras/support-library.html>。

5.5 阅读反编译的 smali 代码

在介绍完了 smali 文件的格式与自动生成的类后，我们再来看看，Java 语言编写的不同结构的代码在 smali 文件中都有些什么特点。

5.5.1 循环语句

循环语句是程序开发中最常用的语句结构，在 Android 开发过程中，常见的循环结构有迭代器循环、for 循环、while 循环、do while 循环。我们在编写迭代器循环代码时，一般是如下形式的代码。

```
Iterator<对象> <对象名> = <方法返回一个对象列表>;
for (<对象> <对象名> : <对象列表>) {
    [处理单个对象的代码体]
}
```

或者：

```
Iterator<对象> <迭代器> = <方法返回一个迭代器>;
while (<迭代器>.hasNext()) {
    <对象> <对象名> = <迭代器>.next();
    [处理单个对象的代码体]
}
```

第一种方式的迭代是 for 关键字中将对象名与对象列表用冒号“：“隔开，然后在循环体中直接访问单个对象，这种方式的代码简练、可读性好，在实际的编程过程中使用颇多。第二种方式是手动获取一个迭代器，然后在一个循环中调用迭代器中的 hasNext() 方法检测是否为空，最后在代码循环体中调用其 next() 方法来遍历迭代器。

将本节提供的示例程序 Circulate.apk 反编译，然后打开反编译工程 smali\com\droider\circulate 目录下的 MainActivity.smali 文件，找到 iterator() 方法的代码如下。

```
.method private iterator()V
.locals 7
.prologue
```

```

.line 34
const-string v4, "activity"
invoke-virtual {p0, v4}, Lcom/droider/circulate/MainActivity;->
getSystemService
(Ljava/lang/String;)Ljava/lang/Object; #获取ActivityManager
move-result-object v0
check-cast v0, Landroid/app/ActivityManager;
.line 35
.local v0, activityManager:Landroid/app/ActivityManager;
invoke-virtual {v0}, Landroid/app/ActivityManager;->getRunningAppProcesses()
Ljava/util/List;
move-result-object v2          #正在运行的进程列表
.line 36
.local v2, psInfos:Ljava/util/List|,
"Ljava/util/List<Landroid/app/ActivityManager$RunningAppProcessInfo;>;"
new-instance v3, Ljava/lang/StringBuilder; #新建一个StringBuilder对象
invoke-direct {v3}, Ljava/lang/StringBuilder;-><init>()V    #调用
StringBuilder构造函数
.line 37
.local v3, sb:Ljava/lang/StringBuilder;
invoke-interface {v2}, Ljava/util/List;->iterator()Ljava/util/Iterator;
#获取进程列表的迭代器
move-result-object v4
:goto_0 #迭代循环开始
invoke-interface {v4}, Ljava/util/Iterator;->hasNext()Z#开始迭代
move-result v5
if-nez v5, :cond_0  #如果迭代器不为空就跳走
.line 40
invoke-virtual {v3}, Ljava/lang/StringBuilder;->toString()Ljava/lang/
String;
move-result-object v4      # StringBuilder转为字符串
const/4 v5, 0x0
invoke-static {p0, v4, v5}, Landroid/widget/Toast;->makeText
(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/
widget/Toast;
move-result-object v4
invoke-virtual {v4}, Landroid/widget/Toast;->show()V# 弹出StringBuilder
的内容
.line 41
return-void #方法返回

```

```

.line 37
:cond_0
invoke-interface {v4}, Ljava/util/Iterator; ->next()Ljava/lang/Object;
#循环获取每一项
move-result-object v1
check-cast v1, Landroid/app/ActivityManager$RunningAppProcessInfo;
.line 38
.local v1, info:Landroid/app/ActivityManager$RunningAppProcessInfo;
new-instance v5, Ljava/lang/StringBuilder; #新建一个临时的StringBuilder
get-object v6, v1, Landroid/app/ActivityManager$RunningAppProcessInfo;
    ->processName:Ljava/lang/String; #获取进程的进程名
invoke-static {v6}, Ljava/lang/String; ->valueOf(Ljava/lang/Object;)
Ljava/lang/String;
move-result-object v6
invoke-direct {v5, v6}, Ljava/lang/StringBuilder; -><init>(Ljava/lang/String;)V
const/16 v6, 0xa#换行符
invoke-virtual {v5, v6}, Ljava/lang/StringBuilder; ->append(C)Ljava/lang/StringBuilder;
move-result-object v5 #组合进程名与换行符
invoke-virtual {v5}, Ljava/lang/StringBuilder; ->toString()Ljava/lang/String;
move-result-object v5
invoke-virtual {v3, v5}, Ljava/lang/StringBuilder; #将组合后的字符串添加到
StringBuilder末尾
    ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
goto :goto_0 #跳转到循环开始处
.end method

```

这段代码的功能是获取正在运行的进程列表，然后使用 Toast 弹出所有的进程名。获取正在运行的进程列表使用 ActivityManager 类的 getRunningAppProcesses()方法，后者会返回一个 List<RunningAppProcessInfo>对象，在上面的代码中，调用了 List 的 iterator()来获取进程列表的迭代器，然后从标号 goto_0 开始进入迭代循环。在循环中首先调用迭代器的 hasNext()方法检测迭代器是否为空，如果迭代器为空就调用 Toast 弹出所有进程信息，如果不为空，说明迭代器中的内容还没有取完，调用迭代器的 next()方法获取单个 RunningAppProcessInfo 对象，接着新建一个临时的 StringBuilder，将进程名与换行符组合后添加到循环开始前创建的 StringBuilder 中，最后使用 goto 语句跳转到循环体的开始处。

看完这一段代码，读者肯定会发现它与上面列出的 while 循环声明非常相似了！没错，第一种迭代器循环展开后就是第二种循环的实现，虽然彼此的 Java 代码不同，但生成的反

汇编代码极具相似。总结一下，迭代器循环有如下特点：

- 迭代器循环会调用迭代器的 `hasNext()`方法检测循环条件是否满足。
- 迭代器循环中调用迭代器的 `next()`方法获取单个对象。
- 循环中使用 `goto` 指令来控制代码的流程。
- `for` 形式的迭代器循环展开后即为 `while` 形式的迭代器循环。

下面看看传统的 `for` 循环，找到 `MainActivity.smali` 文件中的 `forCirculate()` 方法，代码如下。

```
.method private forCirculate()V
    .locals 8
    .prologue
    .line 47
    invoke-virtual {p0}, Lcom/droider/circulate/MainActivity;-
        >getApplicationContext()Landroid/content/Context;
    move-result-object v6
    invoke-virtual {v6}, Landroid/content/Context;      # 获取 PackageManager
        ->getPackageManager()Landroid/content/pm/PackageManager;
    move-result-object v3
    .line 49
    .local v3, pm:Landroid/content/pm/PackageManager;
    const/16 v6, 0x2000
    .line 48
    invoke-virtual {v3, v6}, Landroid/content/pm/PackageManager;
        ->getInstalledApplications(I)Ljava/util/List;    # 获取已安装的程序列表
    move-result-object v0
    .line 50
    .local v0, appInfos:Ljava/util/List;,"Ljava/util/List<Landroid/content/pm/
    /ApplicationInfo;>;"
    invoke-interface {v0}, Ljava/util/List;->size()I    # 获取列表中ApplicationInfo
    对象的个数
    move-result v5
    .line 51
    .local v5, size:I
    new-instance v4, Ljava/lang/StringBuilder;          # 新建一个
    StringBuilder对象
    invoke-direct {v4}, Ljava/lang/StringBuilder;-><init>()V    # 调用
    StringBuilder的构造函数
    .line 52
    .local v4, sb:Ljava/lang/StringBuilder;
    const/4 v1, 0x0
```

```

.local v1, i:I #初始化v1为0
:goto_0 #循环开始
if-lt v1, v5, :cond_0      #如果v1小于v5，则跳转到cond_0 标号处
.line 56
invoke-virtual {v4}, Ljava/lang/StringBuilder;->toString()Ljava/
lang/String;
move-result-object v6
const/4 v7, 0x0
invoke-static {p0, v6, v7}, Landroid/widget/Toast; #构造Toast
->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
Landroid/widget/Toast;
move-result-object v6
invoke-virtual {v6}, Landroid/widget/Toast;->show()V#显示已安装的程序列表
.line 57
return-void #方法返回
.line 53
:cond_0
invoke-interface {v0, v1}, Ljava/util/List;->get(I)Ljava/lang/Object;
#单个ApplicationInfo
move-result-object v2
check-cast v2, Landroid/content/pm/ApplicationInfo;
.line 54
.local v2, info:Landroid/content/pm/ApplicationInfo;
new-instance v6, Ljava/lang/StringBuilder;   #新建一个临时StringBuilder对象
iget-object v7, v2, Landroid/content/pm/ApplicationInfo;->packageName:
Ljava/lang/String;
invoke-static {v7}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)
Ljava/lang/String;
move-result-object v7  #包名
invoke-direct {v6, v7}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/
String;)V
const/16 v7, 0xa#换行符
invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;->append(C)Ljava/
lang/StringBuilder;
move-result-object v6  #组合包名与换行符
invoke-virtual {v6}, Ljava/lang/StringBuilder;->toString()Ljava/lang/
String; #转换为字符串
move-result-object v6
invoke-virtual {v4, v6}, Ljava/lang/StringBuilder;-
>append(Ljava/lang/String;)Ljava/lang/StringBuilder;    #添加到循环外

```

```

    的StringBuilder中
.line 52
add-int/lit8 v1, v1, 0x1      #下一个索引
goto :goto_0 #跳转到循环起始处
.end method

```

这段代码的功能是获取所有安装的程序，然后使用 Toast 弹出所有的软件包名。获取所有安装的程序使用 PackageManager 类的 getInstalledApplications()方法，代码首先创建了一个 StringBuilder 对象用来存放所有的字符串信息，接着初始化 v1 寄存器为 0 作为获取列表项的索引，for 循环的起始处是 goto_0 标号，循环条件的代码为 “if-lt v1, v5, :cond_0”，v1 为索引值，v5 为列表中 ApplicationInfo 的个数，cond_0 标号处的代码为循环体，如果没有索引到最后一项，代码都会跳到 cond_0 标号处去执行，相反，如果索引完了，代码会顺序执行 Toast 显示所有的字符串信息。cond_0 标号处的第一行代码调用 List 的 get()方法获取列表中的单个 ApplicationInfo 对象，然后组合包名与换行符后添加到先前声明的 StringBuilder 中，最后将 v1 索引值加一后调用 “goto :goto_0” 语句跳转到循环起始处。

看完了 for 循环的代码，可以发现它有如下特点：

- 在进入循环前，需要先初始化循环计数器变量，且它的值需要在循环体中更改。
- 循环条件判断可以是条件跳转指令组成的合法语句。
- 循环中使用 goto 指令来控制代码的流程。

接下来是 while 循环与 do while 循环，两者结构差异不大，只是循环条件判断的位置有所不同。并且它们的代码与前面介绍的迭代器循环代码十分相似。笔者在此就不列出了，有兴趣的读者可自行阅读 MainActivity.smali 文件中的 whileCirculate()与 dowhileCirculate()方法的代码。

5.5.2 switch 分支语句

switch 分支也是比较常见的语句结构，经常出现在判断分支比较多的代码中。使用 Apktool 反编译本书配套源代码中 5.5.2 小节提供的 SwitchCase.apk 文件，打开反编译后工程目录中的 smali\com\droider\

switchcase\MainActivity.smali 文件，找到 packedSwitch()方法的代码如下。

```

.method private packedSwitch(I)Ljava/lang/String;
.locals 1
.parameter "i"
.prologue
.line 21
const/4 v0, 0x0
.line 22

```

```

.local v0, str:Ljava/lang/String;      #v0为字符串, 0表示null
packed-switch p1, :pswitch_data_0    #packed-switch分支, pswitch_data_0指定case区域
.line 36
const-string v0, "she is a person"   #default分支
.line 39
:goto_0          #所有case的出口
return-object v0#返回字符串v0
.line 24
:pswitch_0       #case 0
const-string v0, "she is a baby"
.line 25
goto :goto_0     #跳转到goto_0标号处
.line 27
:pswitch_1       #case 1
const-string v0, "she is a girl"
.line 28
goto :goto_0     #跳转到goto_0标号处
.line 30
:pswitch_2       #case 2
const-string v0, "she is a woman"
.line 31
goto :goto_0     #跳转到goto_0标号处
.line 33
:pswitch_3       #case 3
const-string v0, "she is an obasan"
.line 34
goto :goto_0     #跳转到goto_0标号处
.line 22
nop
:pswitch_data_0
.packed-switch 0x0      #case 区域, 从0开始, 依次递增
    :pswitch_0   #case 0
    :pswitch_1   #case 1
    :pswitch_2   #case 2
    :pswitch_3   #case 3
.end packed-switch
.end method

```

代码中的 switch 分支使用的是 packed-switch 指令。p1 为传递进来的 int 类型的数值，

`pswitch_data_0` 为 case 区域，在 case 区域中，第一条指令“`.packed-switch`”指定了比较的初始值为 0，`pswitch_0~pswitch_3` 分别是比较结果为“case 0”到“case 3”时要跳转到的地址。可以发现，标号的命名采用 `pswitch_` 开关，后面的数值为 case 分支需要判断的值，并且它的值依次递增。再来看看这些标号处的代码，每个标号处都使用 `v0` 寄存器初始化一个字符串，然后跳转到了 `goto_0` 标号处，可见 `goto_0` 是所有的 case 分支的出口。另外，“`.packed-switch`”区域指定的 case 分支共有 4 条，对于没有被判断的 default 分支，会在代码的 `packed-switch` 指令下面给出。

`packed-switch` 指令在 Dalvik 中的格式如下：

`packed-switch vAA, +BBBBBBBB`

指令后面的“`+BBBBBBBB`”被指明为一个 `packed-switch-payload` 格式的偏移。它的格式如下。

```
struct packed-switch-payload {
    ushort ident; /* 值固定为 0x0100 */
    ushort size; /* case 数目 */
    int first_key; /* 初始 case 的值 */
    int[] targets; /* 每个 case 相对 switch 指令处的偏移 */
};
```

打开 IDA Pro 找到“`packed-switch p1, :pswitch_data_0`”指令位于 `0x2cb1a` 处，相应的机器码为“`2B 02 13 00 00 00`”，手动分析机器码如下：

`2B` 为 `packed-switch` 的 OpCode。

`02` 为寄存器 `p1`。

`00000013` 为偏移量 `0x13`。

Dalvik 中计算偏移是以两个字节为单位，因为实际该指令指向的 `packed-switch-payload` 结构体的偏移量为 `0x2cb1a + 2 * 0x13 = 0x2cb40`。使用 C32asm 查看该处的数据如图 5-2 所示。



图5-2 `packed-switch-payload`结构体数据

第 1 个 `ident` 字段为 `0x100`，标识 `packed-switch` 有效的 case 区域。第 2 个字段 `size` 为 4，表明有 4 个 case。第 3 个字段 `first_key` 为 0，表明初始 case 值为 0。第 4 个字段为偏移量，

分别为 0x6、0x9、0xc、0xf，加上 packed-switch 指令的偏移值 0x2cb1a，计算可得：

```
case 0 位置 = 0x2cb1a + 2 * 0x6 = 0x2cb26
case 1 位置 = 0x2cb1a + 2 * 0x9 = 0x2cb2c
case 2 位置 = 0x2cb1a + 2 * 0xc = 0x2cb32
case 3 位置 = 0x2cb1a + 2 * 0xf = 0x2cb38
```

至此，有规律递增的 switch 分支就算是搞明白了。最后，将这段 smali 代码整理为 Java 代码如下。

```
private String packedSwitch(int i) {
    String str = null;
    switch (i) {
        case 0:
            str = "she is a baby";
            break;
        case 1:
            str = "she is a girl";
            break;
        case 2:
            str = "she is a woman";
            break;
        case 3:
            str = "she is an obasan";
            break;
        default:
            str = "she is a person";
            break;
    }
    return str;
}
```

现在我们来看看无规律的 case 分支语句代码会有什么不同，找到 MainActivity.smali 文件的 sparseSwitch() 方法代码如下。

```
.method private sparseSwitch(I)Ljava/lang/String;
.locals 1
.parameter "age"
.prologue
.line 43
const/4 v0, 0x0
.line 44
```

```

.local v0, str:Ljava/lang/String;
sparse-switch p1, :sswitch_data_0 # sparse-switch分支, sswitch_data_0
指定case区域
.line 58
const-string v0, "he is a person" #case default
.line 61
:goto_0 #case 出口
return-object v0 #返回字符串
.line 46
:sswitch_0 #case 5
const-string v0, "he is a baby"
.line 47
goto :goto_0 #跳转到goto_0标号处
.line 49
:sswitch_1 #case 15
const-string v0, "he is a student"
.line 50
goto :goto_0 #跳转到goto_0标号处
.line 52
:sswitch_2 #case 35
const-string v0, "he is a father"
.line 53
goto :goto_0 #跳转到goto_0标号处
.line 55
:sswitch_3 #case 65
const-string v0, "he is a grandpa"
.line 56
goto :goto_0 #跳转到goto_0标号处
.line 44
nop
:sswitch_data_0
.sparse-switch #case区域
    0x5 -> :sswitch_0 #case 5(0x5)
    0xf -> :sswitch_1 #case 15(0xf)
    0x23 -> :sswitch_2 #case 35(0x23)
    0x41 -> :sswitch_3 #case 65(0x41)
.end sparse-switch
.end method

```

代码中的 switch 分支使用的是 sparse-switch 指令。按照分析 packed-switch 的方法，我们直接

查看 `sswitch_data_0` 标号处的内容。可以看到 “`.sparse-switch`” 指令没有给出初始 `case` 的值，所有的 `case` 值都使用 “`case 值 -> case 标号`” 的形式给出。此处共有 4 个 `case`，它们的内容都是构造一个字符串，然后跳转到 `goto_0` 标号处，代码架构上与 `packed-switch` 方式的 `switch` 分支一样。

sparse-switch 指令在 Dalvik 中的格式如下：

sparse-switch vAA, +BBBBBBBBBB

指令后面的“+BBBBBBBB”被指明为一个 sparse-switch-payload 格式的偏移。它的格式如下。

```
struct sparse-switch-payload {
    ushort ident; /* 值固定为 0x0200 */
    ushort size; /* case 数目 */
    int[] keys; /* 每个 case 的值，顺序从低到高 */
    int[] targets; /* 每个 case 相对 switch 指令处的偏移 */
};
```

同样地，打开 IDA Pro 找到“sparse-switch p1, :sswitch_data_0”指令位于 0x2cb6a 处，相应的机器码为“2C 02 13 00 00 00”，手动分析机器码如下：

2C 为 sparse -switch 的 OpCode。

02 为寄存器 p1。

00000013 为偏移量 0x13。

因为实际该指令指向的 sparse-switch-payload 结构体的偏移量为 $0x2cb6a + 2 * 0x13 = 0x2cb90$ 。该处的数据如图 5-3 所示。



图5-3 sparse-switch-payload结构体数据

第 1 个 ident 字段为 0x200，标识 sparse-switch 有效的 case 区域。第 2 个字段 size 为 4，表明有 4 个 case。第 3 个字段 keys 为 4 个 case 的值，分别为 0x5、0xf、0x23、0x41。第 4 个字段分别为偏移量，分别为 0x6、0x9、0xc、0xf，加上 sparse -switch 指令的偏移值 0x2cb6a，计算可得：

```
case 0 位置 = 0x2cb6a + 2 * 0x6 = 0x2cb76
case 1 位置 = 0x2cb6a + 2 * 0x9 = 0x2cb7c
case 2 位置 = 0x2cb6a + 2 * 0xc = 0x2cb82
case 3 位置 = 0x2cb6a + 2 * 0xf = 0x2cb88
```

最后，将这段 smali 代码整理为 Java 代码如下。

```
private String sparseSwitch(int age) {
    String str = null;
    switch (age) {
        case 5:
            str = "he is a baby";
            break;
        case 15:
            str = "he is a student";
            break;
        case 35:
            str = "he is a father";
            break;
        case 65:
            str = "he is a grandpa";
            break;
        default:
            str = "he is a person";
            break;
    }
    return str;
}
```

5.5.3 try/catch 语句

在实际编写代码过程中，各种预想不到的结果都有可能出现，为了尽可能的捕捉到异常信息，有必要在代码中使用 Try/Catch 语句将可能发生问题的代码“包裹”起来。使用 Apktool 反编译随书 5.5.3 小节提供的 TryCatch.apk 文件，打开反编译后工程目录中的 smali\com\droider\trycatch\MainActivity.smali 文件，找到 tryCatch()方法代码如下。

```
.method private tryCatch(ILjava/lang/String;)V
    .locals 10
    .parameter "drumsticks"
    .parameter "peple"
    .prologue
```

```

const/4 v9, 0x0
.line 19
:try_start_0      # 第1个try开始
invoke-static {p2}, Ljava/lang/Integer;->parseInt(Ljava/lang/String;)I
#将第2个参数转换为int型
:try_end_0        # 第1个try结束
.catch Ljava/lang/NumberFormatException; {:try_start_0 .. :try_end_0} :
catch_1 # catch_1
move-result v1  #如果出现异常这里不会执行，会跳转到catch_1标号处
.line 21
.local v1, i:I      #.local声明的变量作用域在.local声明与.end local之间
:try_start_1      #第2个try开始
div-int v2, p1, v1  #第1个参数除以第2个参数
.line 22
.local v2, m:I      #m为商
mul-int v5, v2, v1  #m * i
sub-int v3, p1, v5  #v3为余数
.line 23
.local v3, n:I
const-string v5, "\u5171\u6709%d\u53ea\u9e21\u817f\uff0c%d
    \u4e2a\u4eba\u5e73\u5206\uff0c\u6bcf\u4eba\u53ef\u5206\u5f97%d
    \u53ea\uff0c\u8fd8\u5269\u4e0b%d\u53ea"      #格式化字符串
const/4 v6, 0x4
new-array v6, v6, [Ljava/lang/Object;
const/4 v7, 0x0
.line 24
invoke-static {p1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v8
aput-object v8, v6, v7
const/4 v7, 0x1
invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v8
aput-object v8, v6, v7
const/4 v7, 0x2
invoke-static {v2}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v8
aput-object v8, v6, v7
const/4 v7, 0x3
invoke-static {v3}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
move-result-object v8

```

```

        aput-object v8, v6, v7
    .line 23
    invoke-static {v5, v6}, Ljava/lang/String;
        ->format(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
move-result-object v4
    .line 25
    .local v4, str:Ljava/lang/String;
const/4 v5, 0x0
    invoke-static {p0, v4, v5}, Landroid/widget/Toast;
        ->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
            Landroid/widget/Toast;
move-result-object v5
    invoke-virtual {v5}, Landroid/widget/Toast;->show()V      #使用Toast显示格式化后的结果
:try_end_1  #第2个try结束
    .catch Ljava/lang/ArithmeticException; {:try_start_1 .. :try_end_1} :
catch_0      # catch_0
    .catch Ljava/lang/NumberFormatException; {:try_start_1 .. :try_end_1} :
catch_1      # catch_1
    .line 33
    .end local v1          #i:I
    .end local v2          #m:I
    .end local v3          #n:I
    .end local v4          #str:Ljava/lang/String;
:goto_0
return-void #方法返回
    .line 26
    .restart local v1      #i:I
:catch_0
move-exception v0
    .line 27
    .local v0, e:Ljava/lang/ArithmeticException;
:try_start_2      #第3个try开始
const-string v5, "\u4eba\u6570\u4e0d\u80fd\u4e3a0" # “人数不能为0”
const/4 v6, 0x0
    invoke-static {p0, v5, v6}, Landroid/widget/Toast;
        ->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
            Landroid/widget/Toast;
move-result-object v5
    invoke-virtual {v5}, Landroid/widget/Toast;->show()V      #使用Toast显示异常

```

常原因

```
:try_end_2      #第3个try结束
.catch Ljava/lang/NumberFormatException; { :try_start_2 .. :try_end_2 } :
    catch_1
    goto :goto_0 #返回
    .line 29
    .end local v0          #e:Ljava/lang/ArithmeticException;
    .end local v1          #i:I
:catch_1
move-exception v0
.line 30
.local v0, e:Ljava/lang/NumberFormatException;
const-string v5, "\u65e0\u6548\u7684\u6570\u503c\u5b57\u7b26\u4e32"
# “无效的数值字符串”
invoke-static {p0, v5, v9}, Landroid/widget/Toast;
->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
    Landroid/widget/Toast;
move-result-object v5
invoke-virtual {v5}, Landroid/widget/Toast;->show()V  #使用Toast显示异常
常原因
goto :goto_0 #返回
.end method
```

整段代码的功能比较简单，输入鸡腿数与人数，然后使用 Toast 弹出鸡腿的分配方案。传入人数时为了演示 Try/Catch 效果，使用了 String 类型。代码中有两种情况下会发生异常：第一种是将 String 类型转换成 int 类型时可能会发生 NumberFormatException 异常；第二种是计算分配方法时除数为零的 ArithmeticException 异常。

代码中的 try 语句块使用 try_start_ 开头的标号注明，以 try_end_ 开头的标号结束。第一个 try 语句的开头标号为 try_start_0，结束标号为 try_end_0。使用多个 try 语句块时标号名称后面的数值依次递增，本实例代码中最多使用到了 try_end_2。

在 try_end_0 标号下面使用“.catch”指令指定处理到的异常类型与 catch 的标号，格式如下。

```
.catch <异常类型> {<try 起始标号> .. <try 结束标号>} <catch 标号>
```

查看 catch_1 标号处的代码发现，当转换 String 到 int 时发生异常会弹出“无效的数值字符串”的提示。对于代码中的汉字，bksmali 在反编译时将其使用 Unicode 进行编码，因此，在阅读前需要使用相关的编码转换工具进行转换。

仔细阅读代码会发现在 try_end_1 标号下面使用“.catch”指令定义了 catch_0 与 catch_1 两个 catch。catch_0 标号的代码开头又有一个标号为 try_start_2 的 try 语句块，其实这个 try 语句块是虚构的，假如下面的代码。

```

private void a() {
    try {
        .....
        try {
            .....
            } catch (XXX) {
                .....
            }
        } catch (YYY) {
            .....
        }
    }
}

```

当执行内部的 try 语句时发生了异常，如果异常类型为 XXX，则内部 catch 就会捕捉到并执行相应的处理代码，如果异常类型不是 XXX，那么就会到外层的 catch 中去查找异常处理代码，这也就是为什么实例的 try_end_1 标号下面会有两个 catch 的原因，另外，如果在执行 XXX 异常的处理代码时又发生了异常，这个时候该怎么办？此时这个异常就会扩散到外层的 catch 中去，由于 XXX 异常的外层只有一个 YYY 的异常处理，这时会判断发生的异常是否为 YYY 类型，如果是就会进行处理，不是则抛给应用程序。回到本实例中来，如果在执行内部的 ArithmeticException 异常处理时再次发生别的异常，就会调用外层的 catch 进行异常捕捉，因此在 try_end_2 标号下面有一个 catch_1 就很好理解了。

在 Dalvik 指令集中，并没有与 Try/Catch 相关的指令，在处理 Try/Catch 语句时，是通过相关的数据结构来保存异常信息的。回忆一下上一章讲解 dex 文件格式时，曾经介绍过的 DexCode 数据结构，它的声明如下。

```

struct DexCode {
    u2 registersSize; /* 使用的寄存器个数 */
    u2 insSize; /* 参数个数 */
    u2 outsSize; /* 调用其它方法时使用的寄存器个数 */
    u2 triesSize; /* Try/Catch个数 */
    u4 debugInfoOff; /* 指向调试信息的偏移 */
    u4 insnsSize; /* 指令集个数，以2字节为单位 */
    u2 insns[1]; /* 指令集 */
    /* 2字节空间用于结构对齐 */
    /* try_item[triesSize] DexTry 结构*/
    /* Try/Catch中handler的个数 */
    /* catch_handler_item[handlersSize] , DexCatchHandler结构*/
};

```

该结构下面的 try_item 就保存了 try 语句的信息，它的结构 DexTry 声明如下。

```

struct DexTry {
    u4 startAddr;          /* 起始地址 */
    u2 insnCount;          /* 指令数量 */
    u2 handlerOff;         /* handler的偏移 */
};


```

每个 DexTry 保存了 try 语句的起始地址和指令的数量，这样就可以计算出 try 语句块包含的地址范围。在 try_item 字段的下面就是 handler 的个数。下面我们来看看在 dex 文件中存储的 Try/Catch 信息，该实例的类个数较多，手动查找比较慢，在这里使用 Android SDK 中的 dexdump 工具，首先使用解压缩软件取出 TryCatch.apk 中的 classes.dex 文件，然后在命令提示符下输入以下命令：

```
dexdump classes.dex > dump.txt
```

打开生成的 dump.txt 文件，搜索 tryCatch 可找到如下内容。

```

.....
#1           : (in Lcom/droider/trycatch/MainActivity;
name        : 'tryCatch'
type        : '(ILjava/lang/String;)V'
access      : 0x0002 (PRIVATE)
code        -
registers   : 13
ins         : 3
outs        : 3
insn_size  : 80 16-bit code units
catches     : 3
0x0001 - 0x0004
    Ljava/lang/NumberFormatException; -> 0x0045
0x0005 - 0x0038
    Ljava/lang/ArithmeticException; -> 0x0039
    Ljava/lang/NumberFormatException; -> 0x0045
0x003a - 0x0044
    Ljava/lang/NumberFormatException; -> 0x0045
.....

```

从上面的输出信息中，可以发现 tryCatch()方法是私有方法，使用了 13 个寄存器，共 80 条指令，有 3 个 try 语句块，共有 2 个异常处理 Handler。其中，0x0001 - 0x0004 为第一个 try 语句块的代码范围，tryCatch()方法的代码位于 0x2cb08，因此计算可得到第 1 个 try 语句块的代码范围为：

$$(0x2cb08 + 1 * 2) \sim (0x2cb08 + 4 * 2) = 0x2cb0a \sim 0x2cb10$$

同样可计算得到第2与第3个try语句块的代码范围是“0x2cb12~0x2cb78”与“0x2cb7c~0x2cb90”。最后，将这段smali代码整理为Java代码如下。

```
private void tryCatch(int drumsticks, String peple) {
    try {
        int i = Integer.parseInt(peple);
        try {
            int m = drumsticks / i;
            int n = drumsticks - m * i;
            String str = String.format(
                "共有%d只鸡腿, %d个人平分, 每人可分得%d只, 还剩下%d只",
                drumsticks, i, m, n);
            Toast.makeText(MainActivity.this, str,
                    Toast.LENGTH_SHORT).show();
        } catch (ArithmetricException e) {
            Toast.makeText(MainActivity.this, "人数不能为0",
                    Toast.LENGTH_SHORT).show();
        }
    } catch (NumberFormatException e) {
        Toast.makeText(MainActivity.this, "无效的数值字符串",
                    Toast.LENGTH_SHORT).show();
    }
}
```

5.6 使用IDA Pro静态分析Android程序

IDA Pro是目前全世界最强大的静态反汇编分析工具。它具备可交互、可编程、可扩展、多处理器支持等众多特点。使用IDA Pro来静态分析程序是一门大学问，关于它的完整功能与使用方法，绝不是本书一到两节的内容就可以阐述清楚的，如果读者对IDA Pro不了解或者没有使用过该软件，请读者先查看相关的技术书籍来掌握基本的使用方法，推荐阅读《IDA Pro权威指南（第2版）》。

5.6.1 IDA Pro对Android的支持

IDA Pro从6.1版本开始，提供了对Android的静态分析与动态调试支持。包括Dalvik指令集的反汇编、原生库（ARM/Thumb代码）的反汇编、原生库（ARM/Thumb代码）的动态调试等。具体可查看IDA Pro官方的更新日志，链接如下：<http://www.hex-rays.com/products/ida/6.1/index.shtml>。

注意 IDA Pro 是商业收费软件，而且价格不菲。本节在对反汇编代码进行演示与讲解时，假定读者已经通过正规渠道获得了 IDA Pro 6.1 或更高版本的使用授权，并且已经安装配置好 IDA Pro 软件。笔者在编写本章时 IDA Pro 最新版本为 6.3，本书讲解时使用了 IDA Pro 6.1。

5.6.2 如何操作

以 5.2 节的 crackme0502.apk 为例，首先解压出 classes.dex 文件，然后打开 IDA Pro，将 classes.dex 拖放到 IDA Pro 的主窗口，会弹出加载新文件对话框，如图 5-4 所示，IDA Pro 解析出了该文件属于“Android DEX File”，保持默认的选项，点击 OK 按钮，稍等片刻 IDA Pro 就会分析完 dex 文件。

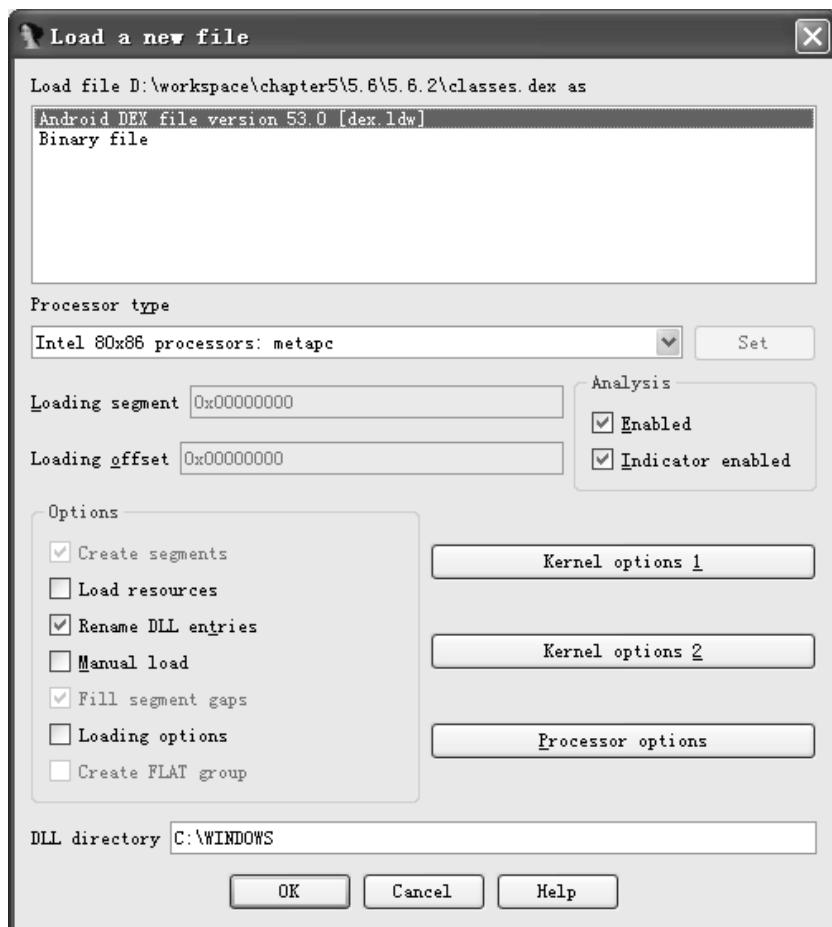


图5-4 使用IDA Pro加载classes.dex文件

IDA Pro 支持结构化形式显示数据结构，因此，我们有必要先整理一下反汇编后的数据。dex 文件的数据结构大部分在 Android 系统源码中 dalvik\libdex\DumpFile.h 文件中，笔者将其中的结构整理为 dex.idc 脚本，在分析 dex 文件时直接导入即可使用。导入的方法为点击 IDA Pro 的菜单项“File→Script file”，然后选择 dex.idc 即可。点击 IDA Pro 主界面的 Structures 选项卡，如图 5-5 所示。

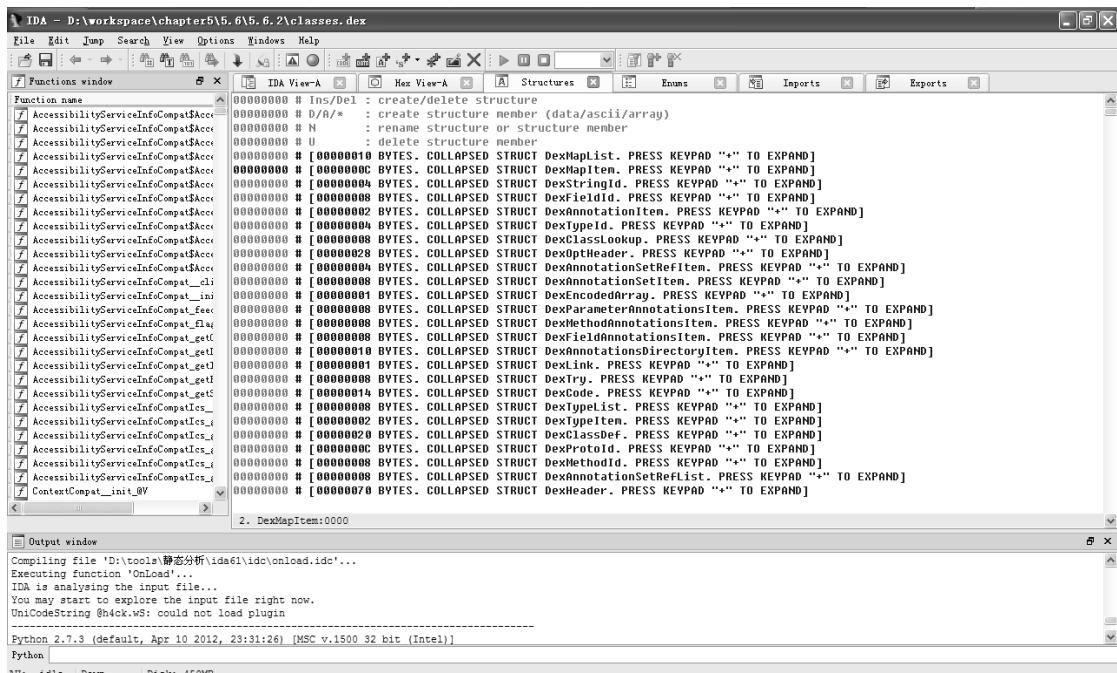


图5-5 导入dex.idc

点击 IDA View-A 选项卡，回到反汇编代码界面，然后点击菜单项“Jump→Jump to address”，或者按下快捷键 G，弹出地址跳转对话框，输入 0 让 IDA Pro 跳转到 dex 文件开头。将鼠标定位到注释“# Segment type: Pure data”所在的行，然后点击菜单项“Edit→Structs →Struct var”，或者按下快捷键 ALT+Q，弹出选择结构类型对话框，如图 5-6 所示，选择 DexHeader 后点击 OK 按钮返回。

此时，dex 文件开头的 0x70 个字节就会格式化显示，效果如图 5-7 所示。同样，读者可以手动对 dex 文件中其它的结构进行整理，如 DexHeader 下面的 DexStringId 结构。

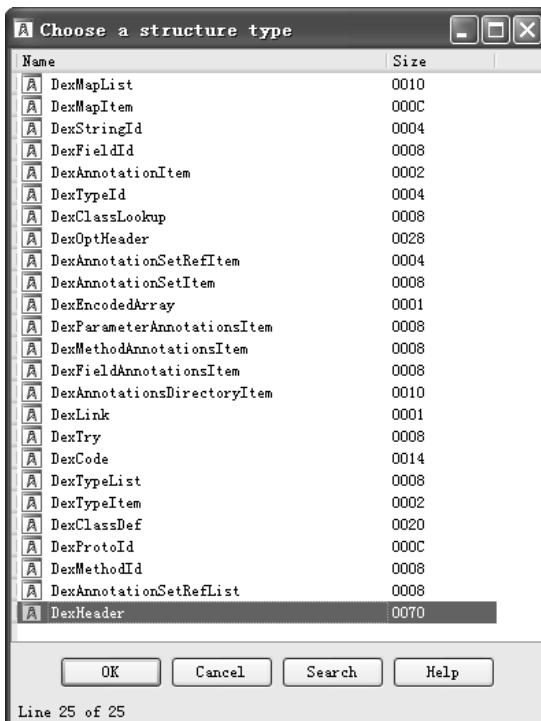


图5-6 选择结构类型

```

HEADER:00000000 # Segment type: Pure data
HEADER:00000000 stru_0:           .byte 0x64, 0x65, 0x78, 0xA, 0x30, 0x33, 0x35, 0# magic
HEADER:00000000                           # DATA XREF: AccessibilityServiceInfoCompat$AccessibilityService
HEADER:00000000                           # AccessibilityServiceInfoCompat$AccessibilityService
HEADER:00000000 .int 0x3518E66A          # checksum
HEADER:00000000 .byte 0x35, 0xE7, 5, 0x20, 0x75, 0x16, 0x5E, 0xA8, 0x2E# signature
HEADER:00000000 .byte 0xE5, 0x27, 0xB8, 0xB9, 0xFC, 0xD9, 0x2F, 0x17, 5# signature
HEADER:00000000 .byte 0xAF, 0xBA          # signature
HEADER:00000000 .int 0x4F42C             # fileSize
HEADER:00000000 .int 0x70                # headerSize
HEADER:00000000 .int 0x12345678         # endianTag
HEADER:00000000 .int 0                  # linkSize
HEADER:00000000 .int 0                  # linkOff
HEADER:00000000 .int 0x4F35C             # mapOff
HEADER:00000000 .int 0xE56               # stringIdsSize
HEADER:00000000 .int 0x70                # stringIdsOff
HEADER:00000000 .int 0x210               # typeIdsSize
HEADER:00000000 .int 0x39C8              # typeIdsOff
HEADER:00000000 .int 0x2B3               # protoIdsSize
HEADER:00000000 .int 0x4208              # protoIdsOff
HEADER:00000000 .int 0x2E9               # fieldIdsSize
HEADER:00000000 .int 0x626C              # fieldIdsOff
HEADER:00000000 .int 0xC54                # methodIdsSize
HEADER:00000000 .int 0x79B4              # methodIdsOff
HEADER:00000000 .int 0x127               # classDefsSize
HEADER:00000000 .int 0xDC54              # classDefsOff
HEADER:00000000 .int 0x3F2F8              # dataSize
HEADER:00000000 .int 0x10134              # dataOff

```

图5-7 格式化后的DexHeader结构

点击菜单项“Jump→Jump to segment”，或者按下快捷键CTRL+S，弹出段选择对话框，如图5-8所示，IDA Pro将dex文件一共分成了9个段，其中前7个段由DexHeader结构给出，最后2个段可以通过计算得出。仔细查看段名，可以发现IDA Pro对其命名不是很好，有3个HEADER段与2个CODE段，笔者觉得第3个段改名为PROTOS更合适一些，还有第6个段改名为CLASSDEFS更好，IDA Pro为什么这样命名我们不得而知，不过，我们需要知道每个段具体所代表的含义。

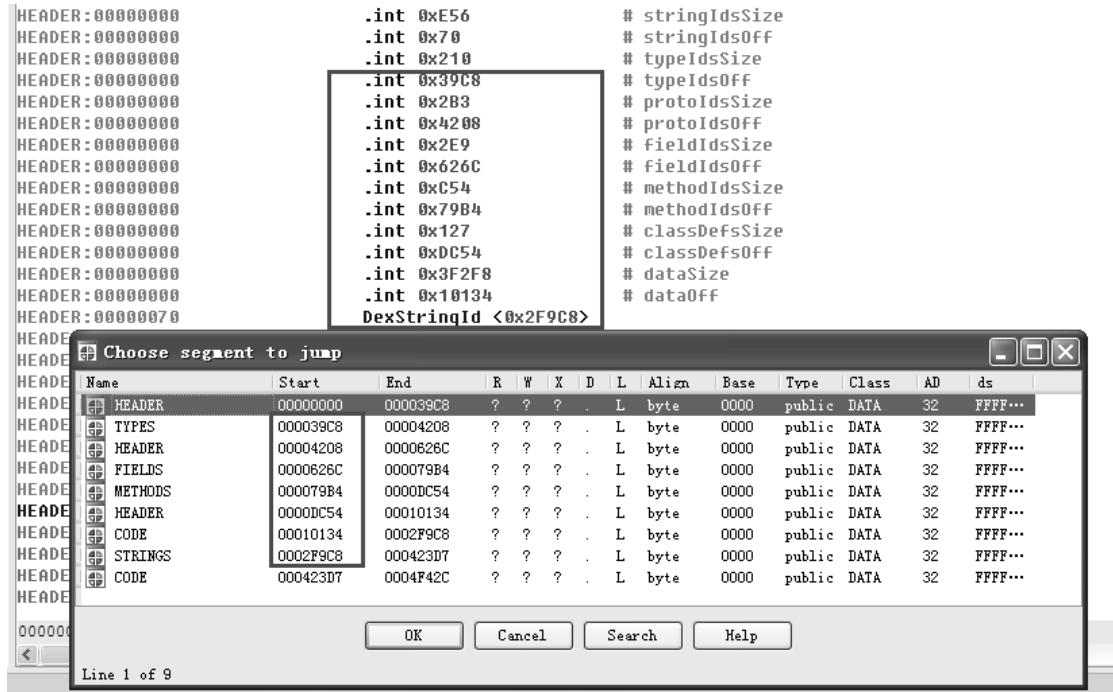


图5-8 dex文件的9个段

dex文件中所有方法可以点击Exports选项卡查看。方法的命名规则为“类名.方法名@方法声明”。在Exports选项卡中随便选择一项，如SimpleCursorAdapter.swapCursor@LL，然后双击跳转到相应的反汇编代码处，该处的代码如下。

```

CODE:0002CFCC Method 2589 (0xa1d):
CODE:0002CFCC     public android.database.Cursor
CODE:0002CFCC         android.support.v4.widget.SimpleCursorAdapter.
                     swapCursor(
CODE:0002CFCC             android.database.Cursor p0)      #方法声明
CODE:0002CFCC     this = v2    #this引用
CODE:0002CFCC     p0 = v3    #第一个参数

```

```

CODE:0002CFCC      invoke-super    {this, p0}, <ref ResourceCursorAdapter.
                     swapCursor(ref) imp. @ _def_ResourceCursorAdapter_
                     swapCursor@LL>
CODE:0002CFD2      move-result-object          v0
CODE:0002CFD4      igeget-object   v1, this, SimpleCursorAdapter_mOriginalFrom
CODE:0002CFD8      invoke-direct   {this, v1}, <void SimpleCursorAdapter.
                     findColumns(ref) SimpleCursorAdapter_findColumns@VL>
CODE:0002CFDE
CODE:0002CFDE      locret:
CODE:0002CFDE      return-object           v0
CODE:0002CFDE      Method End

```

IDA Pro 的反汇编代码使用 ref 关键字来表示非 Java 标准类型的引用，如方法第 1 行的 invoke-super 指令的前半部分如下。

```
invoke-super    {this, p0}, <ref ResourceCursorAdapter.swapCursor(ref)
```

前面的 ref 是 swapCursor() 方法的返回类型，后面括号中的 ref 是参数类型。

后半部分的代码是 IDA Pro 智能识别的。IDA Pro 能智能识别 Android SDK 的 API 函数并使用 imp 关键字标识出来，如第 1 行的 invoke-super 指令的后半部分如下。

```
imp. @ _def_ResourceCursorAdapter_swapCursor@LL
```

imp 表明该方法为 Android SDK 中的 API，@ 后面的部分为 API 的声明，类名与方法名之间使用下划线分隔。

IDA Pro 能识别隐式传递过来的 this 引用，在 smali 语法中，使用 p0 寄存器传递 this 指针，此处由于 this 取代了 p0，所以后面的寄存器命名都依次减了 1。

IDA Pro 能识别代码中的循环、switch 分支与 Try/Catch 结构，并能将它们以类似高级语言的结构形式显示出来，这在分析大型程序时对了解代码结构有很大的帮助。具体的代码反汇编效果读者可以打开 5.2 节使用到的 SwitchCase.apk 与 TryCatch.apk 的 classes.dex 文件自行查看。

5.6.3 定位关键代码——使用 IDA Pro 进行破解的实例

使用 IDA Pro 定位关键代码的方法整体上与定位 smali 关键代码差不多。

第一种方法是搜索特征字符串。首先按下快捷键 CTRL+S 打开段选择对话框，双击 STRINGS 段跳转到字符串段，然后点击菜单项“Search→text”，或者按下快捷键 ALT+T，打开文本搜索对话框，在 String 旁边的文本框中输入要搜索的字符串后点击 OK 按钮，稍等片刻就会定位到搜索结果。不过目前 IDA Pro 对中文字符串的显示与搜索都不支持，如果字符串中的中文字符显示为乱码，需要编写相关的字符串处理插件来解决，这个工作就交给读

者去完成了。

第二种方法是搜索关键 API。首先按下快捷键 CTRL+S 打开段选择对话框，双击第一个 CODE 段跳转到数据起始段，然后点击菜单项“Search→text”，或者按下快捷键 ALT+T，打开文本搜索对话框，在 String 旁边的文本框中输入要搜索的 API 名称后点击 OK 按钮，稍等片刻就会定位到搜索结果。如果 API 被调用多次，可以按下快捷键 CTRL+T 来搜索下一项。

第三种方法是通过方法名来判断方法的功能。这种办法比较笨拙，对于混淆过的代码，定位关键代码比较困难。比如，我们知道 crackme0502.apk 程序的主 Activity 类为 MainActivity，于是在 Exports 选项卡页面上输入 Main，代码会自动定位到以 Main 开头的所在行，如图 5-9 所示，可粗略判断出每个方法的作用。

BuildConfig. <init>@V	0002D028	2974
MainActivity\$1. <init>@VL	0002D040	2975
MainActivity\$1. onClick@VL	0002D05C	2976
MainActivity\$2. <init>@VL	0002D078	2977
MainActivity\$2. onClick@VL	0002D094	2978
MainActivity\$SMChecker. <init>@VLL	0002D0FC	2979
MainActivity\$SMChecker. isRegistered@Z	0002D11C	2980
MainActivity. <init>@V	0002D200	2981
MainActivity. access\$0@VL	0002D218	2982
MainActivity. access\$1@LL	0002D230	2983
MainActivity. getAnnotations@V	0002D248	2985
MainActivity. onCreate@VL	0002D384	2987
MainActivity. onCreateOptionsMenu@ZL	0002D410	2988

图5-9 定位MainActivity

下面我们来尝试破解一下 crackme0502.apk。首先安装运行 apk 程序，程序运行后有两个按钮，点击“获取注解”按钮会 Toast 弹出 3 条信息。在文本框中输入任何字符串后，点击“检测注册码”按钮，程序弹出注册码错误的提示信息。这里我们以按钮事件响应为突破口来查找关键代码，通过图 5-9 我们可以发现有两个名为 OnClick() 的方法，那具体是哪一个呢？我们分别进去看看。前者调用了 MainActivity.access\$0() 方法，在 IDA Pro 的反汇编界面双击 MainActivity_access 可以看到它其实调用了 MainActivity 的 getAnnotations() 方法，看到这里应该可以明白，MainActivity\$1.onClick() 方法是前面按钮的事件响应代码。接下来查看 MainActivity\$2.onClick() 方法，双击代码行，来到相应的反汇编代码处，按下空格键切换到 IDA Pro 的流程视图，如图 5-10 所示，代码的“分水岭”就是“if-eqz v2, loc_2D0DC”。图中左边红色箭头表示条件不满足时执行的路线，右边的绿色箭头是条件满足时执行的路线。

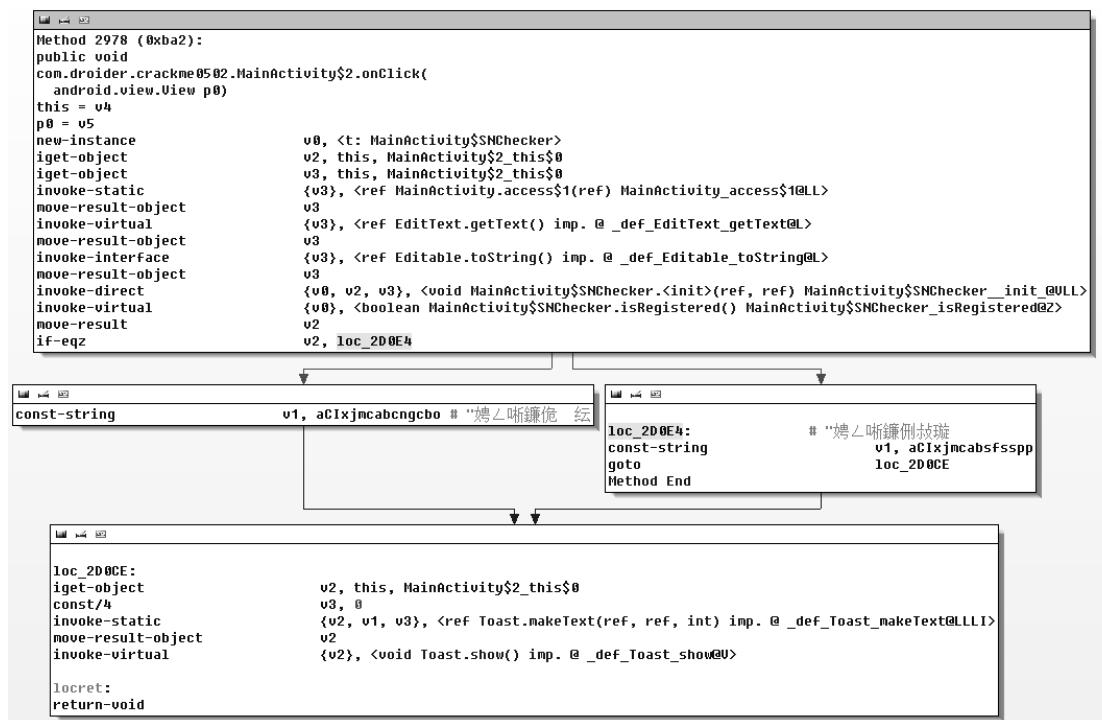


图5-10 IDA Pro的流程视图

虽然不知道这堆乱码字符串分别是什么，但通过最后调用的 Toast 来看，直接修改 if-eqz 即可将程序破解。将鼠标定位到指令“if-eqz v2, loc_2D0DCD”所在行，然后点击 IDA Pro 主界面的“Hex View-A”选项卡，可看到这条指令所在的文件偏移为 0x2D0BE，相应的字节码为“38 02 0f 00”，通过前面的学习，我们知道只需将 if-eqz 的 OpCode 值 38 改成 if-nez 的 OpCode 值 39 即可。说干说干，使用 C32asm 打开 classes.dex 文件，将 0x2D0BE 的 38 改为 39，然后保存退出。接着按照本书 4.6 小节的介绍，将 dex 文件进行 Hash 修复后导入 apk 文件，对 apk 重新签名后安装测试发现程序已经破解成功了。

为了让读者看到一种常见的 Android 程序的保护手段，这里更换一下破解思路。通过图 5-10 可发现，MainActivity\$SNChecker.isRegistered()方法实际上返回一个 Boolean 值，通过判断它的返回值来确定注册码是否正确。现在的问题是，如果该程序是一个大型的 Android 软件，而且调用注册码判断的地方可能不止一处，这种情况时，通常有两种解决方法：第一种是使用 IDA Pro 的交叉引用功能查找到所有方法被调用的地方，然后修改所有的判断结果；第二种方法是直接给 isRegistered()方法“动手术”，让它的结果永远返回为真。很显然，第二种方法解决问题更利落，而且一劳永逸。

下面尝试使用这种方法进行破解，首先按下空格键切换到反汇编视图，发现直接修改方

法的第二条指令为“return v9”即可完成破解，对应机器码为“0F 09”，将其修改完成后重新修复与签名，安装测试发现程序启动后就立即退出了。这时最先怀疑的是程序是否修改正确，使用 IDA Pro 重新导入修改过的 classes.dex 文件，发现修改的地方没错，看来是程序采取了某种保护措施！回想一下前面提到的两种程序退出方法：Context 的 finish()方法与 android.os.Process 的 killProcess()方法，按下快捷键 CTRL+S 并双击 CODE 回到代码段，接着按下快捷键 ALT+T 搜索 finish 与 killProcess，最后在 MyApp 类的 onCreate()方法中找到了相应的调用，查看相应的反汇编代码，发现这段代码使用 Java 的反射机制，手工调用 isRegistered()方法检查字符串“11111”是否为合法注册码，如果是或者调用 isRegistered()失败都说明程序被修改过，从而调用 killProcess()来杀死进程。明白了保护手段，解决方法就简单多了，直接将两处 killProcess()的调用直接 nop 掉（修改相应地方的指令为 0）就可以了。

5.7 恶意软件分析工具包——Androguard

对于 Android 恶意软件分析人员来说，提起 Androguard 应该不会感到陌生，Androguard 提供了一组工具包来辅助分析人员快速鉴别与分析 APK 文件。

5.7.1 Androguard 的安装与配置

Androguard 的安装过程比较复杂，而且容易出错，作者自己制作了一个安装有 Androguard 的 Ubuntu 系统镜像 ARE (Android Reverse Engineering) 供用户下载使用，不过就目前来说，ARE 默认安装的 Androguard 版本过低，作者又没有更新，已经失去了使用的价值。

目前 Androguard 最新版本为 1.6，笔者以 Ubuntu 10.04 演示，其安装方法如下。

首先下载版本控制软件，用来下载工具源码。执行以下命令：

```
sudo apt-get install subversion mercurial git-core
```

安装下载工具 wget 与解压工具 unzip：

```
sudo apt-get install wget unzip
```

安装 setuptools：

```
cd ~/Downloads  
wget http://pypi.python.org/packages/2.6/s/setuptools/setuptools-0.6c11-  
py2.6.egg  
chmod a+x setuptools-0.6c11-py2.6.egg  
sudo ./setuptools-0.6c11-py2.6.egg
```

安装依赖库：

```
sudo apt-get install python2.6-dev python-bzutils libbz2-dev libmuparser-dev  
libsparsehash-dev
```

```
python-ptrace python-pgments graphviz liblzma-dev libsnavy-dev
```

安装 pydot:

```
cd ~/Downloads  
svn checkout http://pydot.googlecode.com/svn/trunk/ pydot  
cd pydot  
sudo python setup.py instal
```

安装 psyco:

```
cd ~/Downloads  
svn co http://codespeak.net/svn/psyco/dist/ psyco  
cd psyco  
sudo python setup.py install
```

安装 networkx:

```
cd ~/Downloads  
git clone https://github.com/networkx/networkx.git  
cd networkx  
sudo python setup.py install
```

安装 IPython (注意: 最好使用 easy_install 安装, 避免 IPython 版本冲突):

```
sudo easy_install ipython
```

安装 Chilkat:

```
cd ~/Downloads  
wget http://www.chilkatsoft.com/download/chilkat-9.3.2-python-2.6-i686  
-linux.tar.gz  
sudo tar zxvf chilkat-9.3.2-python-2.6-i686-linux.tar.gz -C /
```

安装 python-magic:

```
cd ~/Downloads  
git clone git://github.com/ahupp/python-magic.git  
cd python-magic  
sudo python setup.py install
```

安装 pyfuzzy:

```
cd ~/Downloads  
wget http://nchc.dl.sourceforge.net/project/pyfuzzy/pyfuzzy/pyfuzzy-0.1.0  
/pyfuzzy-0.1.0.tar.gz
```

```
tar zxvf pyfuzzy-0.1.0.tar.gz
cd pyfuzzy-0.1.0
sudo python setup.py install
```

安装 Androguard:

```
hg clone https://androguard.googlecode.com/hg/ androguard
```

下载完 Androguard 源码后，打开 androguard 目录下的 elsim/elsign/formula/Makefile 文件，在 CFLAGS 声明处添加如下代码：

```
CFLAGS += -I/usr/include/muParser
```

打开 androguard 目录下的 elsim/elsign/libelsign/Makefile 文件，在 CFLAGS 声明处添加如下代码：

```
CFLAGS += -I/usr/include/muParser -I/usr/include/python2.6
```

修改完成后在终端提示符下进入 androguard 目录并执行 make，Androguard 就安装配置完成了。

最后说下安装 Mercury，Mercury 需要 Python 2.7 运行环境，笔者 Ubuntu 10.04 的 Python 2.6 无法运行它，这里的安装只做演示，读者在安装有 Python 2.7 的环境下使用下面的命令安装即可。

```
cd <Androguard目录>
mkdir mercury
wget http://labs.mwrinfosecurity.com/assets/299/mercury-v1.1.zip
unzip mercury-v1.1.zip
```

在安装 Androguard 的时候，各个依赖库的版本差异与网络环境都有可能导致安装失败，随着 Androguard 版本的不断更新，其依赖库也有可能发生变化，笔者无法保证上述的安装方法能够适应您的操作系统，读者可以严格按照 Androguard 项目的 wiki 安装页的说明进行安装配置，网址是：<http://code.google.com/p/androguard/wiki/Installation>。

5.7.2 Androguard 的使用方法

Androguard 中提供的每个工具都是一个独立的 py 文件，我以上一节中的 crackme0502.apk 与破解后的 crackme0502_cracked.apk 为例，来讲解它们的使用方法。

- androapkinfo.py

androapkinfo.py 用来查看 apk 文件的信息。该工具会输入 apk 文件的包、资源、权限、组件、方法等信息，输出的内容比较详细，建议使用时将输出信息重定向到文件后再进行查看。使用方法：

```
./androapkinfo.py -i ./crackme0502.apk
```

命令执行后输出信息如下。

```

./androapkinfo.py -i ./crackme0502.apk
crackme0502.apk :
FILES:
res/layout/activity_main.xml Android's binary XML -51d837ba
res/menu/activity_main.xml Android's binary XML 2471e50a
AndroidManifest.xml Android's binary XML b5b7132
resources.arsc data 2a26ed7f
res/drawable-hdpi/ic_action_search.png PNG image, 48 x 48, 8-bit colormap,
non-interlaced 64275be8
.....
MAIN ACTIVITY: com.droider.crackme0502.MainActivity
ACTIVITIES: ['com.droider.crackme0502.MainActivity']
SERVICES: []
RECEIVERS: []
PROVIDERS: []
Native code: False
Dynamic code: False
Reflection code: True
.....
Lcom/droider/crackme0502/MainActivity; <init> ['ANDROID', 'APP']
Lcom/droider/crackme0502/MainActivity; getAnnotations ['ANDROID', 'WIDGET']
Lcom/droider/crackme0502/MainActivity; onCreate ['ANDROID', 'WIDGET', 'APP']
Lcom/droider/crackme0502/MainActivity; onCreateOptionsMenu ['ANDROID',
'VIEW']
Lcom/droider/crackme0502/MyApp; <init> ['ANDROID', 'APP']
Lcom/droider/crackme0502/MyApp; onCreate ['ANDROID', 'OS', 'APP']

```

● androaxml.py

androaxml.py 用来解密 apk 包中的 AndroidManifest.xml 文件。使用方法：

```
./androaxml.py -i ./crackme0502.apk
```

输出结果如下。

```

./androaxml.py -i ./crackme0502.apk
<?xml version="1.0" ?>
<manifest android:versionCode="1" android:versionName="1.0"
package="com.droider.crackme0502"
xmlns:android="http://schemas.android.com/apk/res/android">
<uses-sdk android:minSdkVersion="8" android:targetSdkVersion="15">
</uses-sdk>
<application android:debuggable="true" android:icon="@7F020001"
```

```

    android:label="@+id/app_name" android:name=".MyApp" android:theme=
    "@+id/app_theme">
<activity android:label="@+id/main_activity" android:name=".MainActivity">

    <intent-filter>
        <action android:name="android.intent.action.MAIN">

        </action>
        <category android:name="android.intent.category.LAUNCHER">

        </category>
    </intent-filter>
</activity>
</application>
</manifest>

```

● androcsign.py

androcsign.py 用于添加 apk 文件的签名信息到一个数据库文件中。Androguard 工具目录下的 signatures/dbandroguard 文件为收集的恶意软件信息数据库。在开始使用 androcsign.py 前需要为 apk 文件编写一个 sign 文件，这个文件采用 json 格式保存。下面是笔者编写的 crackme0502.apk 的 sign 文件 crackme0502.sign 的内容：

```

[
{
    "SAMPLE": "apks/crackme0502.apk"
},
{
    "BASE": "AndroidOS",
    "NAME": "DroidDream",
    "SIGNATURE": [
        {
            "TYPE": "METHSIM",
            "CN": "Lcom/droider/crackme0502/MainActivity$SNChecker;",
            "MN": "isRegistered",
            "D": "()Z"
        }
    ],
    "BF": "0"
}
]

```

SAMPLE 指定需要添加信息的 apk 文件。BASE 指定文件运行的系统，目前固定为 AndroidOS。NAME 是该签名的名字。SIGNATURE 为具体的签名规则，其中 TYPE 用来指定签名的类型，METHSIM 表示的是方法的签名，此外还有 CLASSSIM 表示为类签名；CN 用来指定方法所在的类；MN 指定了方法名；D 指定了方法的签名信息。BF 用来指定签名的检测规则，可以同时满足 1 条或多条，例如，使用 SIGNATURE 定义了 3 条签名规则，当软件的代码同时满足规则 1 或规则 2 且满足规则 3 时说明样本符合检测条件，那么 BF 可定义为 “"BF" : "(0 or 1) and 2"”。

在 Androguard 目录下新建一个 apks 目录，将 crackme0502.apk 复制进去，然后将 crackme-0502.sign 文件复制到 Androguard 的 signatures 目录下，在终端提示符下执行下面的命令：

```
./androcsign.py -i signatures/crackme0502.sign -o signatures/dbandroguard
```

命令执行后 crackme0502.apk 的信息就存入 dbandroguard 数据库了，输出信息如图 5-11 所示。

```
File Edit View Terminal Help
android@ubuntu10:~/tools/androguard$ ./androcsign.py -i signatures/crackme0502.s
ign -o signatures/dbandroguard
B[F0I]B[F0P1{Ljava/lang/String;length()I}I]B[.]B[R]B[F0P1{Ljava/lang/String;length()I}I]B[F0P1{Ljava/lang/String;charAt(I)C}G]B[.]B[I]B[F0P1{Ljava/lang/String;charAt(I)C}G]B[.]B[G]B[G]B[I]B[.]B[I]B[.]B[G]B[F0P1{Ljava/lang/String;charAt(I)C}G]B[G]B[.]
[{'DroidDream': [[[0, 'QltGMElldQltGMFAxe0xqYXZhL2xhbmvcvU3RyaW5n02xlbdm0aCgpSX1J
XUJbXUJbUl1CW0YwUDF7TGphdmEvbGFuZy9TdHJpbmc7bGVuZ3RoKClJfUldQltGMFAxe0xqYXZhL2xh
bmvcvU3RyaW5n02NoYXJBdChJKUN9R11CW11CW0ldQltGMFAxe0xqYXZhL2xhbmvcvU3RyaW5n02NoYXJB
dChJKUN9R11CW11CW0ddQltHXUJbR11CW0ldQltDQltJXUJbSV1CW11CW0ddQltGMFAxe0xqYXZhL2xh
bmvcvU3RyaW5n02NoYXJBdChJKUN9R11CW0ddQlt', 1.0, 4.455759067518712, 4.53904268152
44052, 0.0]], 'a'}]}
android@ubuntu10:~/tools/androguard$
```

图5-11 androcsign.py执行效果

- `androdd.py`

androdd.py 用来生成 apk 文件中每个类的方法的调用流程图。使用方法:

```
./androdd.py -i ./crackme0502.apk -o ./out -d -f PNG
```

这里需要使用`-o` 选项强制指定一个输入目录, `-d` 指定生成 dot 图形文件, `-f` 用来指定输出的图片格式, 可以是 PNG 或 JPG。不过在笔者的 Ubuntu 10.04 上, 该工具并没有很好的工作。

- androdiff.py

`androdiff.py` 用来比较两个 apk 文件的差异。使用方法如下：

```
./androdiff.py -i ./crackme0502.apk -o crackme0502_cracked.apk
```

命令执行后输出结果如下。

```
./androdiff.py -i ./crackme0502.apk ./crackme0502_cracked.apk
.....
[ ('Lcom/droider/crackme0502/MainActivity$2;', 'onClick',
  '(Landroid/view/View;)V' ) ]
<-> [ ('Lcom/droider/crackme0502/MainActivity$2;', 'onClick',
  '(Landroid/view/View;)V' ) ]
onClick-BB@0x0 onClick-BB@0x0
Added Elements(1)
0x32 12 if-nez v2, +15
Deleted Elements(1)
0x32 12 if-eqz v2, +15
Elements:
IDENTICAL: 3
SIMILAR: 1
NEW: 0
DELETED: 0
SKIPPED: 0
NEW METHODS
DELETED METHODS
```

通过结果可以发现，androdiff分析出了两个apk之间的差异，MainActivity\$2类的onClick()方法中有一行代码不同。

- **androdump.py**

androdump.py 用来 dump 一个 Linux 进程的信息。使用方法如下。

```
./androdump.py -i pid
```

pid 为一个 Linux 进程的 ID，该工具使用的时候较少，这里就不做介绍了。

- **androgexf.py**

androgexf.py 用来生成 APK 的 GEXF 格式的图形文件。该文件可以使用 Gephi 查看。使用方法：

```
./androgexf.py -i ./crackme0502.apk -o ./crackme0502.gexf
```

命令执行完后会在 crackme0502.apk 目录下生成 crackme0502.gexf，gexf 是图形数据文件，可以使用 Gephi 打开，关于 Gephi 的详细使用方法将在下一小节介绍。crackme0502.gexf 打开后效果如图 5-12 所示（Windows 版本的 Gephi）。

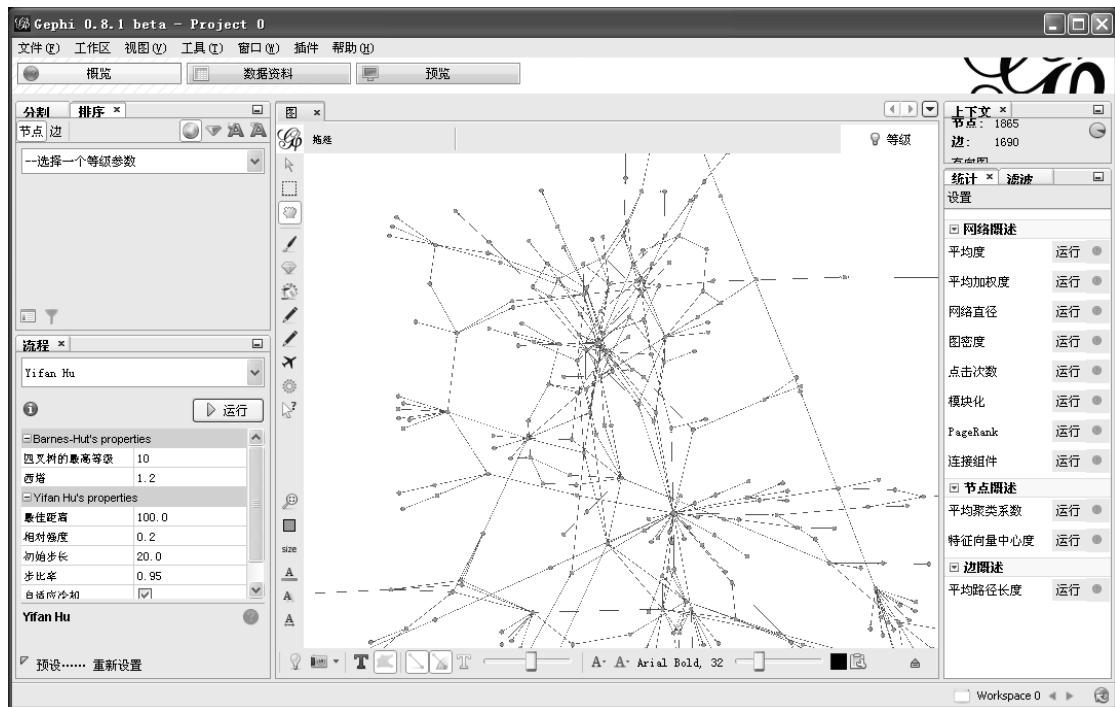


图5-12 使用Gephi查看GEXF文件

- **androlyze.py**

androlyze.py 提供了一个交互环境方便分析人员静态分析 Android 程序，该工具的功能非常强大，而且涉及的内容较多，详细的用法将在 5.7.4 小节介绍。

- **andromercury.py**

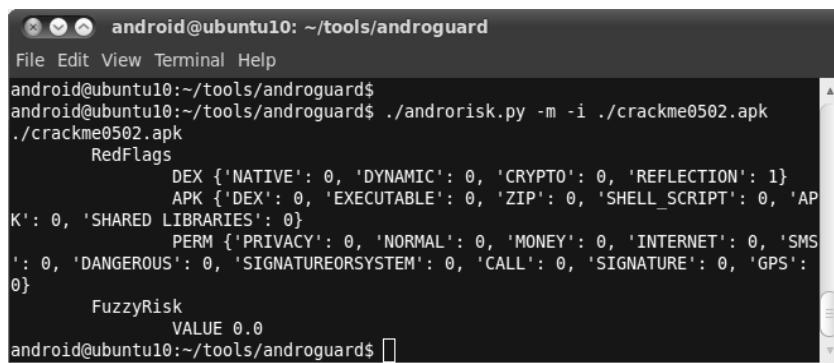
andromercury.py 是 Mercury 工具的框架。功能上是对 Mercury 的包装，Mercury 需要的 Python 版本为 2.7，此处不展示该工具，Mercury 工具的详细使用方法会在本书的第 11 章进行介绍。

- **andrорisk.py**

andrорisk.py 用于评估 apk 文件中潜在的风险。使用方法如下。

```
./andrорisk.py -m -i ./crackme0502.apk
```

-m 选项表明需要分析 apk 中的每一个方法，命令执行后效果如图 5-13 所示。



```
android@ubuntu10:~/tools/androguard
File Edit View Terminal Help
android@ubuntu10:~/tools/androguard$ ./androrisk.py -m -i ./crackme0502.apk
./crackme0502.apk
RedFlags
    DEX {'NATIVE': 0, 'DYNAMIC': 0, 'CRYPTO': 0, 'REFLECTION': 1}
    APK {'DEX': 0, 'EXECUTABLE': 0, 'ZIP': 0, 'SHELL_SCRIPT': 0, 'APK': 0, 'SHARED_LIBRARIES': 0}
        PERM {'PRIVACY': 0, 'NORMAL': 0, 'MONEY': 0, 'INTERNET': 0, 'SMS': 0, 'DANGEROUS': 0, 'SIGNATUREORSYSTEM': 0, 'CALL': 0, 'SIGNATURE': 0, 'GPS': 0}
    FuzzyRisk
        VALUE 0.0
android@ubuntu10:~/tools/androguard$
```

图5-13 androrisk.py执行效果

从输出结果来看，crackme0502.apk 中没有发现风险，唯独有一项 REFLECTION 的值为 1，表示程序中有使用到 Java 反射技术。

- androsign.py

androsign.py 用于检测 apk 的信息是否存在于特定的数据库中，它的作用与 androcsign.py 恰好相反。使用方法：

```
./androsign.py -i apks/crackme0502.apk -b signatures/dbandroguard -c signatures/dbconfig
```

- androsim.py

androsim.py 用于计算两个 apk 文件的相似度，它是唯一一个有 Windows 移植版的工具，Windows 平台上该工具为 androsim.exe，使用方法为：

```
./androsim.py -i ./crackme0502.apk ./crackme0502_cracked.apk
```

命令执行后输出结果如下。

```
Elements:
```

```
IDENTICAL:      717
SIMILAR:       1
NEW:            0
DELETED:       0
SKIPPED:       0
--> methods: 99.983498% of similarities
```

可以看到两个程序的相似度为 99.983498%。

- androxgml.py

androxgml.py 用来生成 apk/jar/class/dex 文件的控制流程及功能调用图，输出格式为 xgmml。使用方法如下。

```
./androxgml.py -i ./crackme0502.apk -o crackme0502.xgmml
```

不过很可惜的是，目前不管是在 Ubuntu 10.04，还是 Ubuntu 12.04 上使用该功能，都会

运行错误，并输出以下错误提示：

```
AttributeError: DVMBasicBlock instance has no attribute 'get_ins'
```

后者笔者发现这是 Androguard 的一个 bug，截止到笔者编写完本章，该 bug 都还没出解决方案。

- apkviewer.py

apkviewer.py 用来为 apk 文件中每一个类生成一个独立的 graphml 图形文件，使用方法如下。

```
./apkviewer.py -i ./crackme0502.apk -o output
```

命令执行完后，可以使用 Gephi 打开生成后的 graphml 文件，不过图形中的每个节点是指令级别的，在查看时效果没有方法级别的 gexf 文件直观。

5.7.3 使用 Androguard 配合 Gephi 进行静态分析

Androguard 可以生成 Java 方法级与 Dalvik 指令级的图形文件，配合 Gephi 工具查看图形文件，可以快速地了解程序的执行流程，在静态分析 Android 程序时，这个功能非常方便。

下面我们以 crackme0502.apk 为例，介绍如何使用 Gephi 来静态分析它。首先下载 Gephi 程序，Gephi 是开源的，支持 Mac OSX/Windows/Linux 三种平台，目前最新版本为 0.8.1 beta，笔者演示时下载的是 Windows 版本的安装程序，顺利安装完成后启动界面如图 5-14 所示。

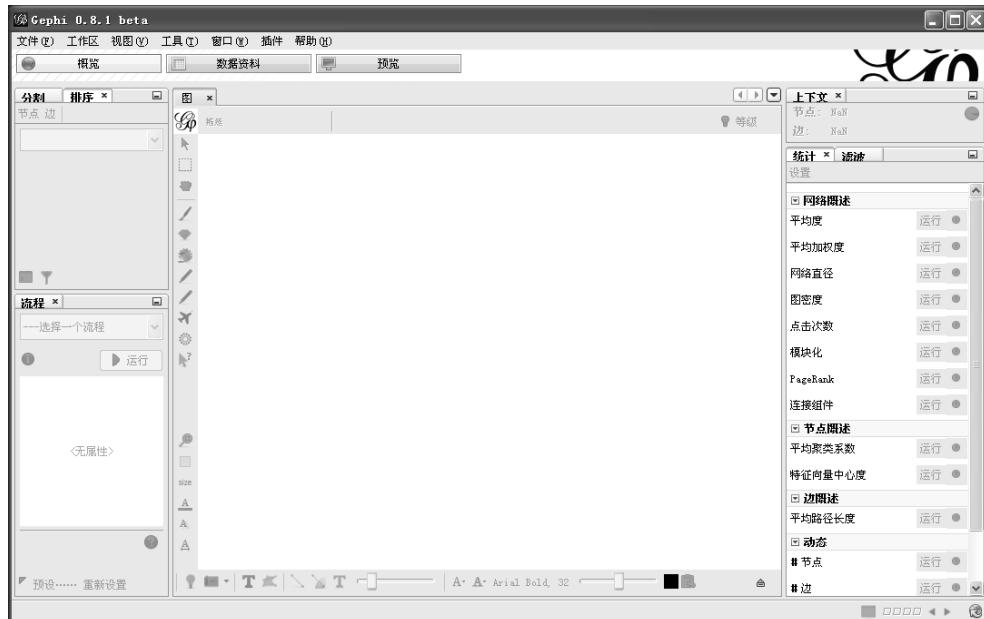


图5-14 Gephi启动界面

点击菜单项“文件→打开”，选择上一小节使用 Androguard 生成的 crackme0502.gexf，Gephi 会分析出 gexf 文件的版本为 1.2，然后点击确定按钮进入图形显示界面。在流程中选择“Yifan Hu”，然后点击运行按钮来生成分析图，如图 5-15 所示。

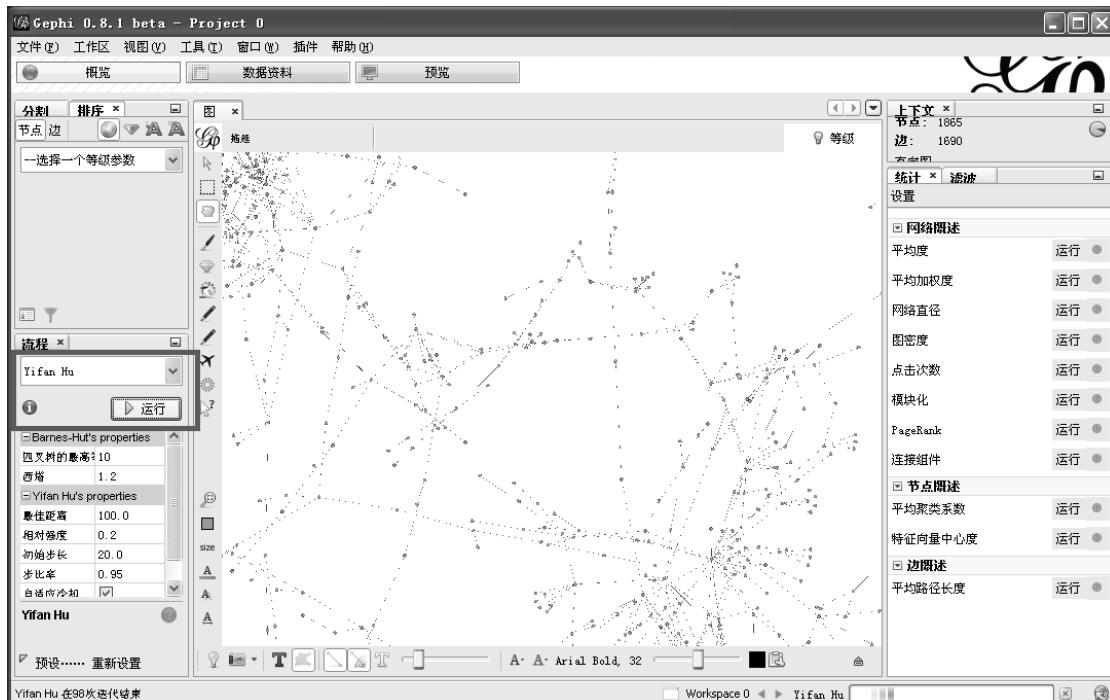


图5-15 生成分析图

分析图生成完毕后，点击图形下方的“T”按钮显示标签（label）的内容，然后拖动旁边的两个滑块来调整连接线的粗细与文本的字体大小，如图 5-16 所示，左边的滑块用来调整节点与节点之间连接线的粗细，右边的滑块用来调整文本的字体大小。

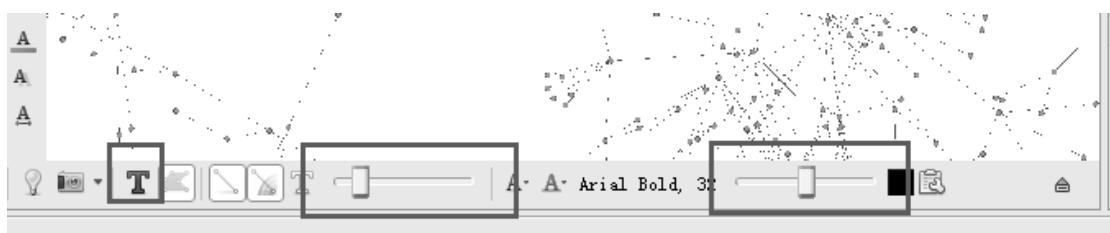


图5-16 调整连接线与文本大小

接下来点击 Gephi 菜单栏下方的“数据资料”按钮，切换到“数据资料”选项卡，

在过滤标签旁边的文本框中输入“OnCreate”查找所有 OnCreate()方法，结果如图 5-17 所示。

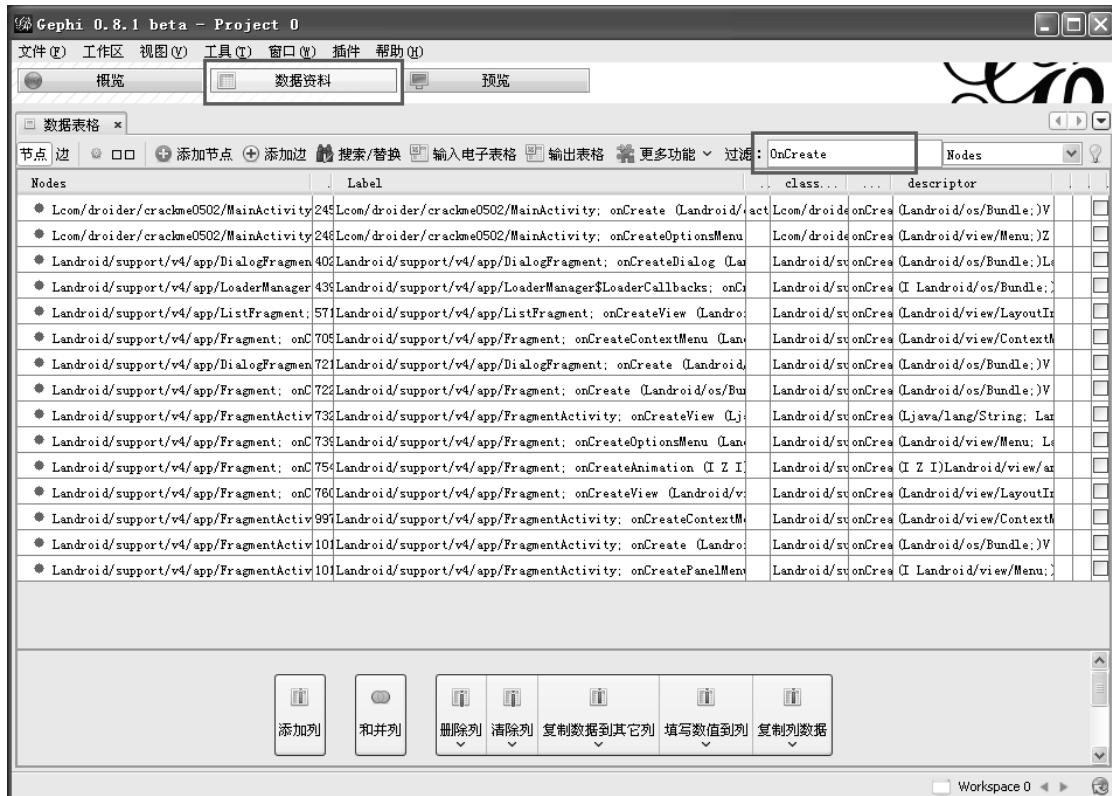


图5-17 查找OnCreate()方法

结果中的第一条记录就是 MainActivity 的 OnCreate()，在第一条记录上点击右键，选择菜单项“在概述选择”(这个 Gephi 的中文翻译有些别扭，其含义应该是“在概览图中选中”)，然后点击菜单下方的概览按钮，切换到图形显示页面，发现 ACTIVITY 节点与 OnCreate 节点，以及它们之间的连接线都是绿色的，将鼠标放在 OnCreate 节点上，可以看到它向下关联了 MainActivity\$1.<init>、findViewById、setContentView、MainActivity\$2.<init>共 4 个节点，拖动所有的节点调整至合适位置，完成后效果如图 5-18 所示，MainActivity 在 OnCreate() 方法中执行了哪些内容一目了然。

按照上面的步骤，我们来查看 OnClick()方法的节点，找到 MainActivity\$2 的 OnClick() 方法，然后在记录上点击右键选择“编辑节点”，在颜色一栏将其修改为 “[255,0,0]”，设置节点为红色，然后如法炮制的设置 OnClick 节点的连接线连接的几个节点，最后在设置 isRegistered 节点时，将其尺寸调整为 15.0，最后效果如图 5-19 所示。

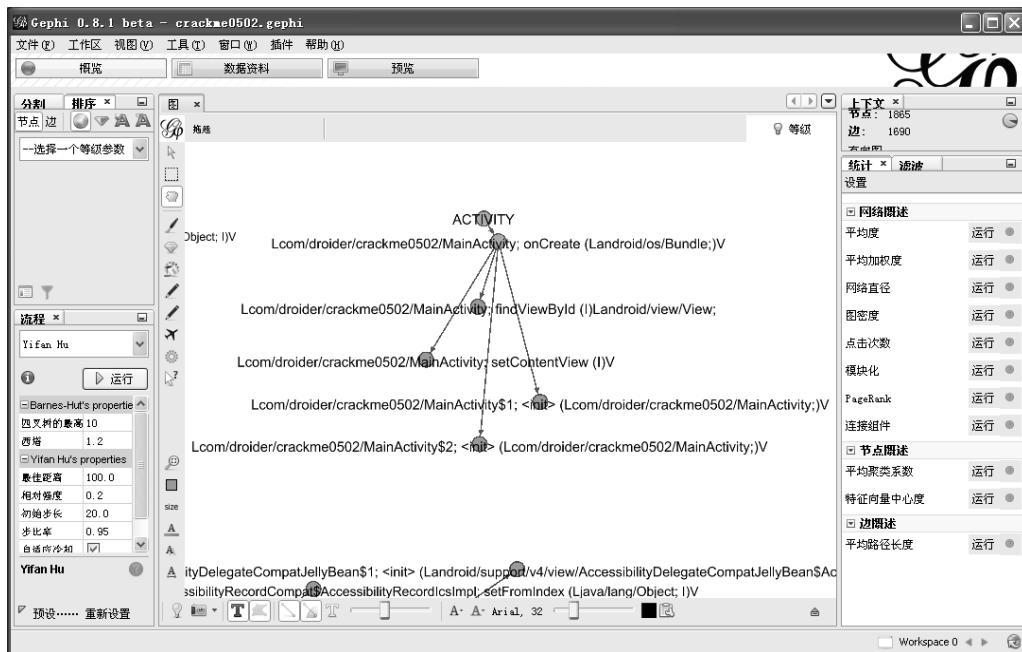


图5-18 OnCreate()方法的执行流程

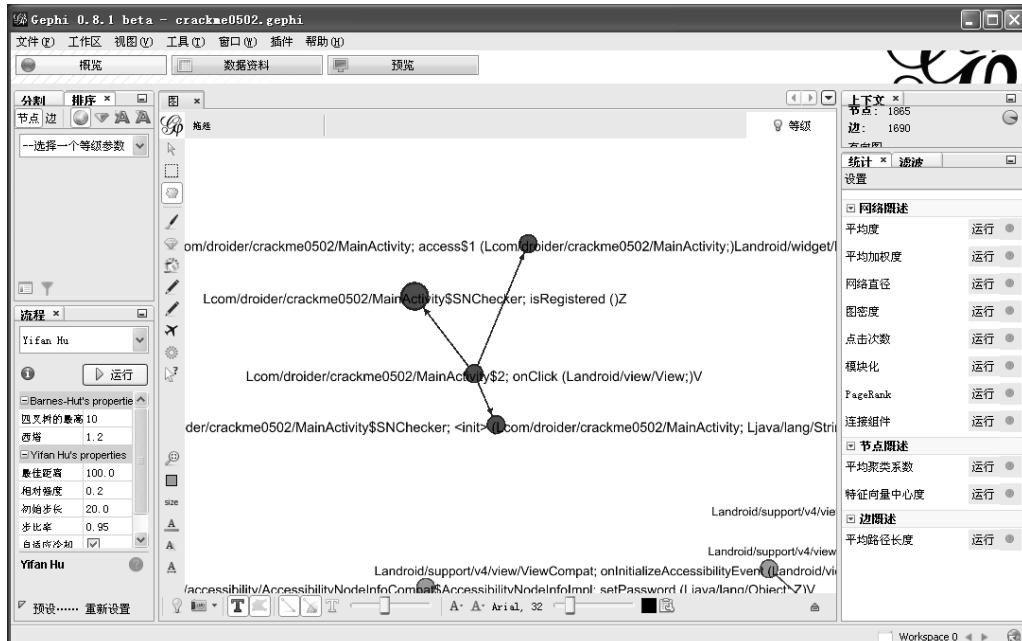


图5-19 MainActivity\$2的onClick()方法

点击 Gephi 的菜单项“文件→保存”，将修改后的 gexf 文件存为 crackme0502.gephi，方便以后查看。可以发现，使用 Gephi 分析 apk 文件比 IDA Pro 分析还要直观。除了简单的显示方法调用外，Gephi 还有很多强大的功能，这些就留给读者自己慢慢去挖掘了。

5.7.4 使用 androlyze.py 进行静态分析

Androguard 工具包中的 androlyze.py 与其它的 py 文件不同，它不是单一功能的脚本，而是一个强大的静态分析工具，它提供的一个独立的 Shell 环境来辅助分析人员执行分析工作。

在终端提示符下执行“./androlyze.py -s”会进入 androlyze 的 Shell 交互环境，分析人员可以在其中执行不同的命令，来满足不同情况下的分析需求。androlyze.py 通过访问对象的字段与方法的方式来提供反馈结果，分析过程中可能会用到 3 个对象：apk 文件对象、dex 文件对象、分析结果对象。这 3 个对象是通过 androlyze.py 的 Shell 环境（以下简称 Shell 环境）来获取的。首先是 apk 文件对象，以 5.2 小节的 crackme0502.apk 为例，在 Shell 环境下执行以下命令：

```
a = APK("./crackme0502.apk")
```

APK()方法返回一个 apk 文件对象，并赋值给 a。androlyze.py 的使用有一个技巧，就是在输入对象名后加上一个点“.”，然后按钮 Tab 键，终端提示符下显示该类所有的方法与字段，输入部分方法或字段名按 Tab 键，终端提示符会补全提示。如图 5-20 所示。

```
android@ubuntu10: ~/tools/androguard
File Edit View Terminal Help
android@ubuntu10:~/tools/androguard$ ./androlyze.py -s
Androlyze version 1.6
In [1]: a = APK("./crackme0502.apk")

In [2]: a.
a.androidversion      a.get_libraries
a.filename            a.get_main_activity
a.files               a.get_min_sdk_version
a.files_crc32        a.get_package
a.format_value        a.get_permissions
a.get_AndroidManifest a.get_providers
a.get_activities      a.get_raw
a.get_androidversion_code a.get_receivers
a.get_androidversion_name a.get_services
a.get_certificate     a.get_target_sdk_version
a.get_details_permissions a.is_valid_APK
a.get_dex              a.magic_file
a.get_element          a.new_zip
a.get_elements         a.package
a.get_file              a.permissions
a.get_filename          a.show
a.get_files             a.validAPK
a.get_files_crc32       a.xml
a.get_files_information a.zip
a.get_files_types       a.zipmodule

In [2]: a.
```

图5-20 apk文件对象可用的方法

接着是 dex 文件对象的获取，执行以下命令。

```
d = DalvikVMFormat(a.get_dex())
```

DalvikVMFormat()执行后会返回一个 dex 文件对象，它的可用方法如图 5-21 所示。

```
File Edit View Terminal Help
default_colors      disable_print_colors
del                dist/
In [3]: d.
d.CM              d.get_cm_method        d.get_methods_id_item
d.add_idx          d.get_cm_string       d.get_regex_strings
d.classes          d.get_cm_type         d.get_string_data_item
d.classes_names    d.get_codes_item     d.get_strings
d.codes            d.get_debug_info_item d.header
d.create_dref     d.get_determineException d.length_buff
d.create_python_export d.get_determineNext d.map_list
d.create_xref      d.get_field           d.methods
d.debug            d.get_field_descriptor d.pretty_show
d.dotbuff          d.get_fields          d.read
d.fields           d.get_fields_class   d.read_b
d.fix_checksums   d.get_fields_id_item d.readat
d.get_BRANCH_DVM_OPCODES d.get_format_type  d.register
d.get_DVM_TOSTRING d.get_header_item   d.save
d.get_all_fields  d.get_idx            d.set_buff
d.get_buff         d.get_len_methods  d.set_decompiler
d.get_class_manager d.get_method        d.set_gvmanalysis
d.get_classes      d.get_method_by_idx d.set_idx
d.get_classes_def_item d.get_method_descriptor d.set_vmanalysis
d.get_classes_names d.get_methods      d.show
d.get_cm_field    d.get_methods_class d.strings
In [3]: d.[]
```

图5-21 dex文件对象可用的方法

接下来是分析结果对象的获取，执行以下命令。

```
dx = VMAnalysis(d)
```

VMAnalysis()执行返回后将分析结果对象赋给 dx，dx 可用的方法较少，如图 5-22 所示。

使用 3 条命令获取 3 个对象的方法比较麻烦，Shell 环境下可以执行以下命令一次获取这 3 个对象。

```
a, d, dx = AnalyzeAPK("./crackme0502.apk", decompiler="dad")
```

AnalyzeAPK()一次性完成上面介绍的 3 个方法调用，其中 decompiler 指定反编译器的名称，Androguard 自带并且默认使用 dad 作为 dex 文件的反编译器。

```

android@ubuntu10: ~/tools/androguard
File Edit View Terminal Help
In [3]: dx = VMAnalysis(d)

In [4]:
In [4]: dx.
dx.get_method          dx.get_tainted_variables
dx.get_method_signature dx.get_vm
dx.get_methods         dx.hmethods
dx.get_permissions     dx.methods
dx.get_permissions_method dx.signature
dx.get_tainted_field   dx.tainted
dx.get_tainted_fields  dx.tainted_packages
dx.get_tainted_packages dx.tainted_variables

In [4]: dx.□

```

图5-22 结果对象可用的方法

在获得这 3 个对象后，我们看看如何使用它们来分析 Android 程序。首先我们查看 apk 文件的信息，可以执行 `a.show()`，该命令执行后会输出 apk 压缩包中所有的文件信息。我们也可以执行 `a.files` 命令获得相近的输出结果。还可以执行：

- `a.get_permissions()`: 输出 apk 用到的全部权限。
- `a.get_providers()`: 输出程序中所有的 Content Provider。
- `a.get_receivers()`: 输出程序中所有的 Broadcast Receiver。
- `a.get_services()`: 输出程序中所有的 Service。
-

其它的命令读者可以自己手动尝试运行一遍。接着是 dex 文件对象。该对象保存了 dex 文件中所有的类、方法、字段的信息，这些信息都是以对象的方式进行提供的，而且都以 `d.CLASS_` 开头。例如 “`d.CLASS_Laaa_bbb_ccc`”，表示 dex 文件中的 `aaa.bbb.ccc` 类。方法的名称是在类名称后添加以 `METHOD_` 开头的方法字符串，例如 “`d.CLASS_Laaa_bbb_ccc.METHOD_ddd_Ljava_lang_StringV`”，表示 `aaa.bbb.ccc` 类的 “`void ddd(String)`” 方法。字段的名称是在类名称后添加以 `FIELD_` 开头的字段声明字符串，例如 “`d.CLASS_Laaa_bbb_ccc.FIELD_this_0`”，表示 `aaa.bbb.ccc` 类的 `this$0` 字段。

按照上面的命名规则，我们查看 `MainActivity$2` 的 `onClick()` 方法，执行以下命令。

`d.CLASS_Lcom_droider_crackme0502_MainActivity_2.METHOD_onClick.pretty_show()`
`pretty_show()` 用来显示 `onClick()` 方法的代码，如图 5-23 所示。



```

x y z android@ubuntu10: ~/tools/androguard
File Edit View Terminal Help

In [13]: d.CLASS_Lcom_droider_crackme0502_MainActivity_2.METHOD_onClick.pretty_sh
ow()
#####
Method Information
Lcom/droider/crackme0502/MainActivity$2;->onClick(Landroid/view/View;)V [access_
flags=public]
#####
Params
- local registers: v0...v4
- v5:android.view.View
- return:void
#####
*****
onClick-BB@0x0 :
    0 (00000000) new-instance          v0, Lcom/droider/crackme0502/MainActi
vity$SNChecker;
    1 (00000004) iget-object          v2, v4, Lcom/droider/crackme0502/Main
Activity$2;->this$0 Lcom/droider/crackme0502/MainActivity;
    2 (00000008) iget-object          v3, v4, Lcom/droider/crackme0502/Main
Activity$2;->this$0 Lcom/droider/crackme0502/MainActivity;
    3 (0000000c) invoke-static         v3, Lcom/droider/crackme0502/MainActi
vity;->access$1(Lcom/droider/crackme0502/MainActivity;)Landroid/widget/EditText;
    4 (00000012) move-result-object   v3
    5 (00000014) invoke-virtual       v3, Landroid/widget/EditText;->getTex
t()Landroid/text/Editable;
    6 (0000001a) move-result-object   v3
    7 (0000001c) invoke-interface    v3, Landroid/text/Editable;->toString
()Ljava/lang/String;
    8 (00000022) move-result-object   v3
    9 (00000024) invoke-direct        v0, v2, v3, Lcom/droider/crackme0502/
MainActivity$SNChecker;-><init>(Lcom/droider/crackme0502/MainActivity; Ljava/lan
g/String;)V
    10 (0000002a) invoke-virtual      v0, Lcom/droider/crackme0502/MainActi
vity$SNChecker;->isRegistered()Z
    11 (00000030) move-result         v2
    12 (00000032) if-eqz             v2, +15 [ onClick-BB@0x36 onClick-BB@0
0x50 ]

onClick-BB@0x36 :
    13 (00000036) const-string        v1, '\xe6\xb3\xa8\xe5\x86\x8c\xe7\x
a0\x81\xe6\xad\x
a3\xe7\x
a1\xae' [ onClick-BB@0x3a ]

```

图5-23 MainActivity\$2的onClick()方法

在代码的最下面，有如下一段内容。

```

#####
XREF
T: Lcom/droider/crackme0502/MainActivity; access$1
(Lcom/droider/crackme0502/MainActivity;)Landroid/widget/EditText; c

```

```

T: Lcom/droider/crackme0502/MainActivity$SNChecker; <init>
    (Lcom/droider/crackme0502/MainActivity; Ljava/lang/String;)V 24
T: Lcom/droider/crackme0502/MainActivity$SNChecker; isRegistered ()Z 2a
#####

```

上面的代码为方法的交叉引用区，开头的字母 T 表示后面指定的方法被本方法引用，除此之外，它还可以是字母 F，表示本方法被其它的方法所引用。

除了使用 pretty_show() 显示反汇编代码外，还可以使用 source() 直接显示 Java 源码，不过笔者本机并未测试成功。

最后是 dx 对象，它可以实现字符串、字段、方法、包名的搜索，使用方法如下。

```

dx.tainted_variables.get_string(<要搜索的字符串>)
show_Path(d, dx.tainted_packages.search_packages(<包名>))

```

这两个方法是 Androguard 作者在 wiki 页上公布的，笔者本机并未测试成功。

androlyze.py 的其它的功能笔者就不介绍了，读者可以自己动手多试。同时，从本节对 Androguard 的使用介绍中可以看出，目前 Androguard 在兼容性与稳定性方面有待加强。

5.8 其它静态分析工具

在静态分析 Android 程序时，除了使用 IDA Pro 与 Androguard 外，还有很多其它的静态工具。它们大多是以 ApkTool、BakSmali、JAD 与 Androguard 为基础，进行扩展实现的，其中就包括一款名为 ApkInspector 的静态分析工具。ApkInspector 由国人郑聰开发，拥有类似 IDA Pro 的流程图显示功能，支持反汇编代码语法高亮、字符串搜索、函数和变量重命名。它是 honeynet 2011 中的一个项目，在 2011 年的 GSOC (Google Summer Of Code) 中该软件表现突出。今年的 GSOC2012 上该工具有提出更新与改进，不过目前仍未有它的更新版本可供下载，该软件的安装配置过程比较繁琐，笔者在此不介绍它的使用，感兴趣的读者可以访问它的项目主页 <https://bitbucket.org/ryanwsmith/apkinspector> 获取到它。

5.9 阅读反编译的 Java 代码

在分析大型软件时，为了弄清程序的结构框架，需要花费掉大量的时间与精力来阅读 smali 代码，这无疑是分析成本的一大开销。然而，Android 程序大多数情况下是采用 Java 语言开发的，传统意义上的 Java 反汇编工具依然能够派上用场。

5.9.1 使用 dex2jar 生成 jar 文件

在第 4 章中介绍 Android 程序生成步骤时曾经讲到过，生成 apk 文件的其中一个环节就是将 Java 语言的字节码转换成 Dalvik 虚拟机的字节码。那么，这个转换的过程可逆吗？答

案是：可以的。使用开源的 dex2jar 工具即可。

dex2jar 的官网是 <http://code.google.com/p/dex2jar/>，目前最新版本为 0.0.9.9，将下载下来的 dex2jar 压缩包解压，然后将解压后的文件夹添加到系统的 PATH 环境变量中，在命令提示符下输入以下命令：

```
d2j-dex2jar xxx.apk
```

稍等片刻就会在同目录下生成一个 jar 文件。dex2jar 是一个工具包，除了提供 dex 文件转换成 jar 文件外，还提供了一些其它的功能，每个功能使用一个 bat 批处理或 sh 脚本来包装，只需在 Windows 系统中调用 bat 文件、在 Linux 系统中调用 sh 脚本即可。

d2j-apk-sign 用来为 apk 文件签名。命令格式：d2j-apk-sign xxx.apk。

d2j-asm-verify 用来验证 jar 文件。命令格式：d2j-asm-verify -d xxx.jar。

d2j-dex2jar 用来将 dex 文件转换成 jar 文件。命令格式：d2j-dex2jar xxx.apk

d2j-dex-asmifier 用来验证 dex 文件。命令格式：d2j-dex-asmifier xxx.dex。

d2j-dex-dump 用来转存 dex 文件的信息。命令格式：d2j-dex-dump xxx.apk out.jar。

d2j-init-deobf 用来生成反混淆 jar 文件时的初始化配置文件。

d2j-jar2dex 用来将 jar 文件转换成 dex 文件。命令格式：d2j-jar2dex xxx.apk。

d2j-jar2jasmin 用来将 jar 文件转换成 jasmin 格式的文件。命令格式：d2j-jar2jasmin xxx.jar

d2j-jar-access 用来修改 jar 文件中的类、方法以及字段的访问权限。

d2j-jar-remap 用来重命名 jar 文件中的包、类、方法以及字段的名称。

d2j-jasmin2jar 用来将 jasmin 格式的文件转换成 jar 文件。命令格式：d2j-jasmin2jar dir
dex2jar 为 d2j-dex2jar 的副本。
dex-dump 为 d2j-dex-dump 的副本。

5.9.2 使用 jd-gui 查看 jar 文件的源码

为了达到源码级的反编译效果，可以使用 Java 反编译工具 JAD 将 jar 文件转换成 Java 源文件，目前 JAD 官网已经无法访问，可以通过 <http://www.varaneckas.com/jad/> 下载到 JAD 的可执行文件。

在这里，笔者推荐使用 jd-gui。jd-gui 是一款用 C++ 开发的 Java 反编译工具，支持 Windows、Linux 和苹果 Mac OS 三个平台。jd-gui 是免费的，而且反编译效果不错，该工具省掉了将 jar 文件转换成 Java 源文件的步骤，直接以源码的形式显示 jar 文件中的内容，可以从官方免费获取。官方主页为：<http://java.decompiler.free.fr/>，jd-gui 运行后效果如图 5-24 所示。

除了反编译功能外，jd-gui 还带有强大的搜索功能，在主界面按下快捷键 CTRL+F，会在程序的状态栏显示一个搜索工具条，输入要搜索的内容，当前打开的反编译窗口会高亮显示搜索结果。除此之外，点击菜单项“Search→Search”会弹出搜索对话框，如图 5-25 所示，搜索框列举出了 isRegistered() 方法在哪些文件中被引用过。

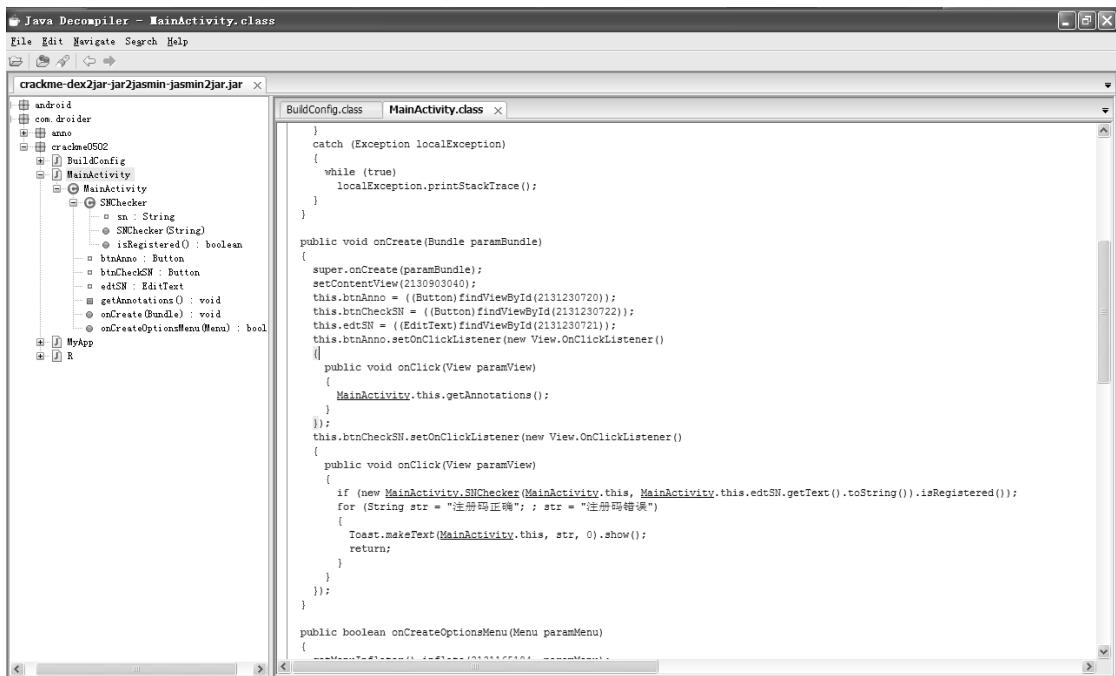


图5-24 使用jd-gui查看jar文件的源码

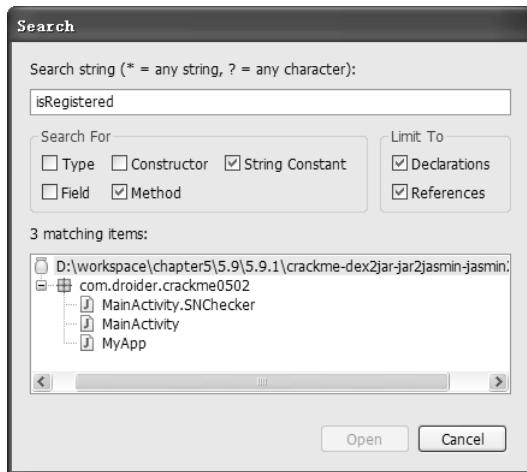


图5-25 jd-gui的搜索功能

5.10 集成分析环境——santoku

本章的前面部分介绍的 Android 静态分析工具包括 ApkTool、BakSmali、Androguard、

dex2jar、jd-gui，这些工具中除了 Androguard 不能在 Windows 平台上运行外，其它的都能支持跨平台，可以在 Windows 平台上良好的运行。

如果读者觉得单独下载配置这些工具麻烦（其实本书配套源代码中有提供），不妨使用另一款集成分析环境 santoku。santoku 实质是一款定制的 Ubuntu 12.04 系统镜像，与其它 Ubuntu 系统相比，它具有如下特点：

1. 集成了大量主流的 Android 程序分析工具，为分析人员节省分析环境配置所需的时间。
2. 集成移动设备取证工具。支持 Android、iPhone 等移动设备的取证工作。
3. 集成渗透测试工具。
4. 集成网络数据分析工具。在分析 Android 病毒、木马等程序时，这些工具特别有用。
5. 采用 LXDE 作为系统的桌面环境，界面与 Windows XP 非常相似，符合中国人使用习惯。
6. 正处于 beta 阶段，但整个项目显得很有活力，相信将来的更新和维护也会不错。

santoku 的初衷是为了提供一套完整的移动设备司法取证环境。但很显然它集成的 Android 程序分析工具，会给我们的分析工作带来很多便捷。santoku-linux 的官方网站为 <https://santoku-linux.com>，目前最新版本 alpha 0.3。如图 5-26 所示，santoku-linux 启动后的界面非常清新。

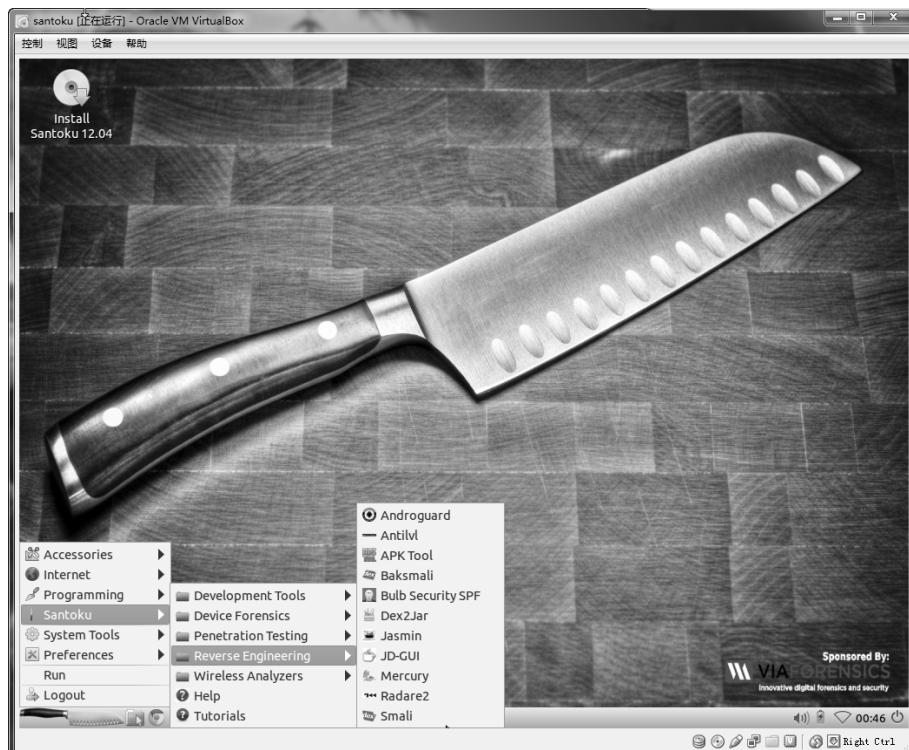


图5-26 santoku-linux启动界面

如果读者还在为安装配置 Androguard 烦恼的话，不妨下载安装 santoku 试试，目前它集成的 Androguard 版本为 1.5，虽然不是最新版本，但通过它可更新到 1.6 版，只需下载 Androguard 的源码后，修改两个库文件的 makefile（详细位置请参考 Androguard 安装方法），然后执行一次 make 命令即可。

santoku 详细的安装与使用方法笔者就不介绍了，它里面集成的大多数静态分析工具在前面的章节中都已经介绍过了，相信读者也都已经掌握了。

5.11 本章小结

静态分析是软件分析过程中最基础也是最重要的一种手段，本章主要从 Android 程序的特点、smali 文件的代码结构、静态分析工具的使用等几个方面来介绍如何分析一个完整的 Android 程序。对于刚接触 Android 程序分析的读者来说，建议多阅读反编译后的 smali 文件，而不是直接使用 jd-gui 等工具来阅读 Java 源码，这样有助于提高反编译代码的阅读能力，以后分析混淆过的 APK 文件或者 jd-gui 派不上用场时就不至于手足无措。