《移动终端开发技术》

电子教材

所属专业（教研室）： 计算机软技术

制定人： 移动终端开发技术项目组

制定时间： 2017 年 10 月

日照职业技术学院

目　录

第 1 章　Android 系统简介

欢迎你来到 Android 世界！Android 系统是目前世界上市场占有率最高的移动操作系统，不管你在哪里，几乎都可以看到人人手中都会有一部 Android 手机。虽然今天的 Android 世界欣欣向荣，可是你知道它的过去是什么样的吗？我们一起来看一看它的发展史吧。

2003 年 10 月，Andy Rubin 等人一起创办了 Android 公司。2005 年 8 月谷歌收购了这家仅仅成立了 22 个月的公司，并让 Andy Rubin 继续负责 Android 项目。在经过了数年的研发之后，谷歌终于在 2008 年推出了 Android 系统的第一个版本。但自那之后，Android 的发展就一直受到重重阻挠。乔布斯自始至终认为 Android 是一个抄袭 iPhone 的产品，里面剽窃了诸多 iPhone 的创意，并声称一定要毁掉 Android。而本身就是基于 Linux 开发的 Android 操作系统，在 2010 年被 Linux 团队从 Linux 内核主线中除名。又由于 Android 中的应用程序都是使用 Java 开发的，甲骨文则针对 Android 侵犯 Java 知识产权一事对谷歌提起了诉讼……

可是，似乎再多的困难也阻挡不了 Android 快速前进的步伐。由于谷歌的开放政策，任何手机厂商和个人都能免费地获取到 Android 操作系统的源码，并且可以自由地使用和定制。三星、HTC、摩托罗拉、索爱等公司都推出了各自系列的 Android 手机，Android 市场上百花齐放。仅仅推出两年后，Android 就超过了已经霸占市场逾十年的诺基亚 Symbian，成为了全球第一大智能手机操作系统，并且每天都还会有数百万台新的 Android 设备被激活。目前 Android 已经占据了全球智能手机操作系统 70%以上的份额。

说了这些，想必你已经体会到 Android 系统炙手可热的程度，并且迫不及待地想要加入到 Android 开发者的行列当中了吧。试想一下，十个人中有七个人的手机都可以运行你编写的应用程序，还有什么能比这个更诱人的呢？那么从今天起，我就作为你 Android 旅途中的导师，一步步地引导你成为一名出色的 Android 开发者。

好了，现在我们就来一起初窥一下 Android 世界吧。

1.1 认识 Android 系统

Android 从面世以来到现在已经发布了近二十个版本了。在这几年的发展过程中，谷歌为 Android 王国建立了一个完整的生态系统。手机厂商、开发者、用户之间相互依存，共同推进着 Android 的蓬勃发展。开发者在其中扮演着不可或缺的角色，因为再优秀的操作系统没有开发者来制作丰富的应用程序也是难以得到大众用户喜爱的，相信没有多少人能够忍受没有 QQ、微信的手机吧？而谷歌推出的 Google Play 更是给开发者带来了大量的机遇，只要你能制作出优秀的产品，在 Google Play 上获得了用户的认可，你就完全可以得到不错的经济回报，从而成为一名独立开发者，甚至是成功创业！

那我们现在就以一个开发者的角度，去了解一下这个操作系统吧。纯理论型的东西也比较无聊，怕你看睡着了，因此我只挑重点介绍，这些东西跟你以后的开发工作都是息息相关的。

1.1.1 Android 系统架构

为了让你能够更好地理解 Android 系统是怎么工作的，我们先来看一下它的系统架构。Android 大致可以分为四层架构，五块区域。

1. Linux 内核层

   Android 系统是基于 Linux 2.6 内核的，这一层为 Android 设备的各种硬件提供了底层的驱动，如显示驱动、音频驱动、照相机驱动、蓝牙驱动、Wi-Fi 驱动、电源管理等。

2. 系统运行库层

   这一层通过一些 C/C++库来为 Android 系统提供了主要的特性支持。如 SQLite 库提供了数据库的支持，OpenGL|ES 库提供了 3D 绘图的支持，Webkit 库提供了浏览器内核的支持等。

   同样在这一层还有 Android 运行时库，它主要提供了一些核心库，能够允许开发者使用 Java 语言来编写 Android 应用。另外 Android 运行时库中还包含了 Dalvik 虚拟机，它使得每一个 Android 应用都能运行在独立的进程当中，并且拥有一个自己的 Dalvik 虚拟机实例。相较于 Java 虚拟机，Dalvik 是专门为移动设备定制的，它针对手机内存、CPU 性能有限等情况做了优化处理。

3. 应用框架层

   这一层主要提供了构建应用程序时可能用到的各种 API，Android 自带的一些核心应用就是使用这些 API 完成的，开发者也可以通过使用这些 API 来构建自己的应用程序。

4. 应用层

   所有安装在手机上的应用程序都是属于这一层的，比如系统自带的联系人、短信等程序，或者是你从 Google Play 上下载的小游戏，当然还包括你自己开发的程序。

结合图 1.1 你将会理解得更加深刻，图片源自维基百科。

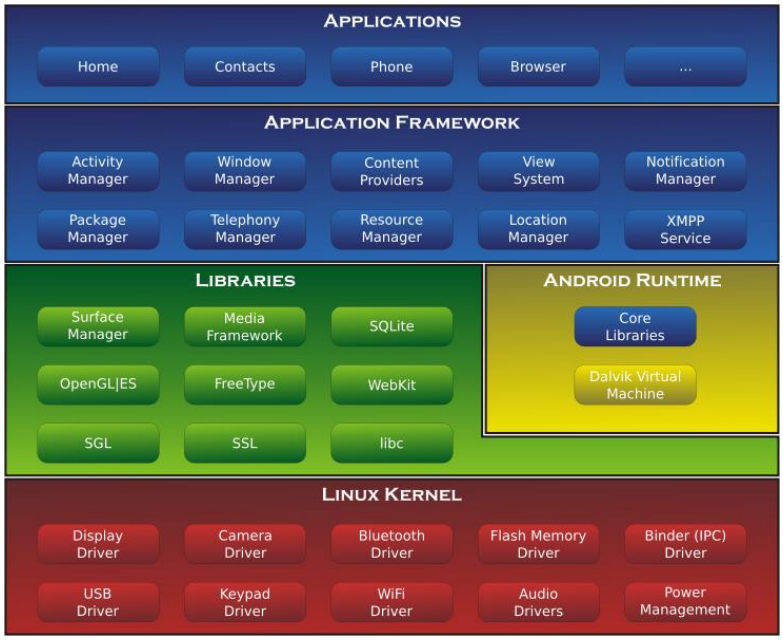

图　1.1

1.1.2　Android 已发布的版本

2008 年 9 月，谷歌正式发布了 Android 1.0 系统，这也是 Android 系统最早的版本。随后的几年，谷歌以惊人的速度不断地更新 Android 系统，2.1、2.2、2.3 系统的推出使 Android 占据了大量的市场。2011 年 2 月，谷歌发布了 Android 3.0 系统，这个系统版本是专门为平板电脑设计的，但也是 Android 为数不多比较失败的版本，推出之后一直不见什么起色，市场份额也少得可怜。不过很快，在同年的 10 月，谷歌又发布了 Android 4.0 系统，这个版本不再对手机和平板进行差异化区分，既可以应用在手机上也可以应用在平板上，除此之外还引入了不少新特性。目前最新的系统版本已经是 4.4 KitKat。

下表中列出了目前市场上主要的一些 Android 系统版本及其详细信息。你看到这张表格时，数据很可能已经发生了变化，查看最新的数据可以访问 http://developer.android.com/about/dashboards/。

版本号	系统代号	API	市场占有率
2.2	Froyo	8	1.2%
2.3.3 – 2.3.7	Gingerbread	10	19.0%

3.2	Honeycomb	13	0.1%
4.0.3 – 4.0.4	Ice Cream Sandwich	15	15.2%
4.1.x	Jelly Bean	16	35.3%
4.2.x		17	17.1%
4.3		18	9.6%
4.4	KitKat	19	2.5%

从上表中可以看出，目前 4.0 以上的系统已经占据了 80%左右的 Android 市场份额，而且以后这个数字还会不断增加，因此我们本书中开发的程序也是主要面向 4.0 以上的系统，2.x 的系统就不再去兼容了。

1.1.3 Android 应用开发特色

预告一下，你马上就要开始真正的 Android 开发旅程了。不过先别急，在开始之前我们再来一起看一看，Android 系统到底提供了哪些东西，供我们可以开发出优秀的应用程序。

1. 四大组件

Android 系统四大组件分别是活动（Activity）、服务（Service）、广播接收器（Broadcast Receiver）和内容提供器（Content Provider）。其中活动是所有 Android 应用程序的门面，凡是在应用中你看得到的东西，都是放在活动中的。而服务就比较低调了，你无法看到它，但它会一直在后台默默地运行，即使用户退出了应用，服务仍然是可以继续运行的。广播接收器可以允许你的应用接收来自各处的广播消息，比如电话、短信等，当然你的应用同样也可以向外发出广播消息。内容提供器则为应用程序之间共享数据提供了可能，比如你想要读取系统电话簿中的联系人，就需要通过内容提供器来实现。

2. 丰富的系统控件

Android 系统为开发者提供了丰富的系统控件，使得我们可以很轻松地编写出漂亮的界面。当然如果你品味比较高，不满足于系统自带的控件效果，也完全可以定制属于自己的控件。

3. SQLite 数据库

Android 系统还自带了这种轻量级、运算速度极快的嵌入式关系型数据库。它不仅支持标准的 SQL 语法，还可以通过 Android 封装好的 API 进行操作，让存储和读取数据变得非常方便。

4. 地理位置定位

移动设备和 PC 相比起来，地理位置定位功能应该可以算是很大的一个亮点。现在的 Android 手机都内置有 GPS，走到哪儿都可以定位到自己的位置，发挥你的想象就可以做出创意十足的应用，如果再结合上功能强大的地图功能，LBS 这一领域潜力无限。

5. 强大的多媒体

Android 系统还提供了丰富的多媒体服务，如音乐、视频、录音、拍照、闹铃等等，这一切你都可以在程序中通过代码进行控制，让你的应用变得更加丰富多彩。

6. 传感器

Android 手机中都会内置多种传感器，如加速度传感器、方向传感器等，这也算是移动设备的一大特点。通过灵活地使用这些传感器，你可以做出很多在 PC 上根本无法实现的应用。

既然有 Android 这样出色的系统给我们提供了这么丰富的工具，你还用担心做不出优秀的应用吗？好了，纯理论的东西也就介绍到这里，我知道你已经迫不及待想要开始真正的开发之旅了，那我们就开始启程吧！

1.2 手把手带你搭建开发环境

俗话说得好，工欲善其事，必先利其器，开着记事本就想去开发 Android 程序显然不是明智之举，选择一个好的 IDE 可以极大幅度地提高你的开发效率，因此本节我就将手把手带着你把开发环境搭建起来。

1.2.1 准备所需要的软件

JDK：Java 语言的软件开发工具包；

Android SDK：谷歌推出的 Android 开发工具包，在开发 Android 程序时，需要引入该工具包，使用相关的 API；

Android Studio：2013 年谷歌推出的 Android Studio。

上述软件不需要一个个下载，谷歌已经将用到的工具集合了。

1.2.2 搭建开发环境

可以到 android 官方或者 Android Studio 中文社区下载 android-studio。双击安装，安装过程和一般的没有什么两样。下一步继续，选择安装路径：

设置了安装路径后，点击安装即可进行继续安装。等待安装过程结束，安装过程中会自动配置一些环境，后会安装完成，点击完成即可。

启动 android-studio，下边是启动后截图。android-studio 启动时提示。如果是之前安装过老版本的 android-studio，选择第一个，第一次安装默认选择即可。

1.3 创建你的第一个 Android 项目

任何一个编程语言写出的第一个程序毫无疑问都会是 Hello World，这已经是自 20 世纪 70 年代一直流传下来的传统，在编程界已成为永恒的经典，那我们当然也不会搞例外了。

1.3.1 创建 HelloWorld 项目

在 Eclipse 的导航栏中点击 File→New→Android Application Project，此时会弹出创建 Android 项目的对话框。其中 Application Name 代表应用名称，此应用安装到手机之后会在手机上显示该名称，这里我们填入 Hello World。Project Name 代表项目名称，在项目创建完成后该名称会显示在 Eclipse 中，这里我们填入 HelloWorld（项目名通常不加空格）。接着 Package Name 代表项目的包名，Android 系统就是通过包名来区分不同应用程序的，因此包名一定要有唯一性，这里我们填入 com.test.helloworld。

接下来是几个下拉选择框，Minimum Required SDK 是指程序最低兼容的版本，这里我们选择 Android 4.0。Target SDK 是指你在该目标版本上已经做过了充分的测试，系统不会再帮你在这个版本上做向前兼容的操作了，这里我们选择最高版本 Android 4.4。Compile With 是指程序将使用哪个版本的 SDK 进行编译，这里我们同样选择 Android 4.0。最后一个 Theme 是指程序 UI 所使用的主题，我个人比较喜欢选择 None。全部都选择好的界面如图 1.9 所示。

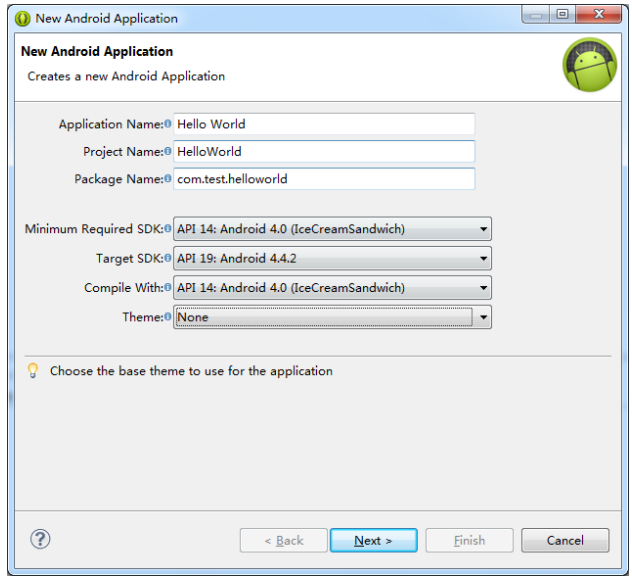

图 1.9

现在我们可以点击 Next 了，下一个界面是创建项目的一些配置，全部保持默认配置就好，如图 1.10 所示。

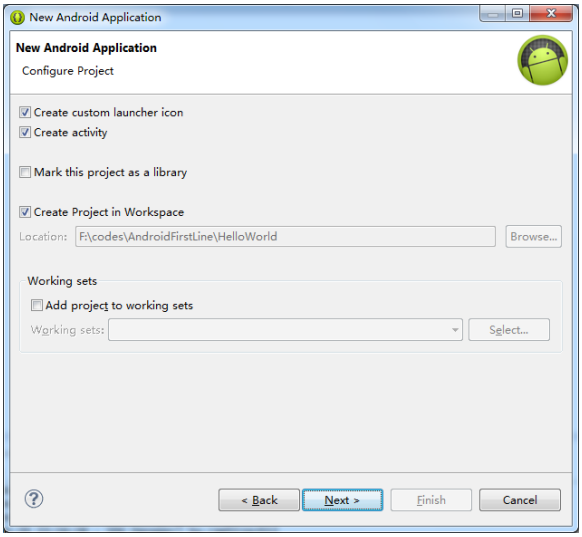

图 1.10

直接点击 Next 进入到启动图标的配置界面，在这里配置的图标就会是你的应用程序安装到手机之后显示的图标，如图 1.11 所示。

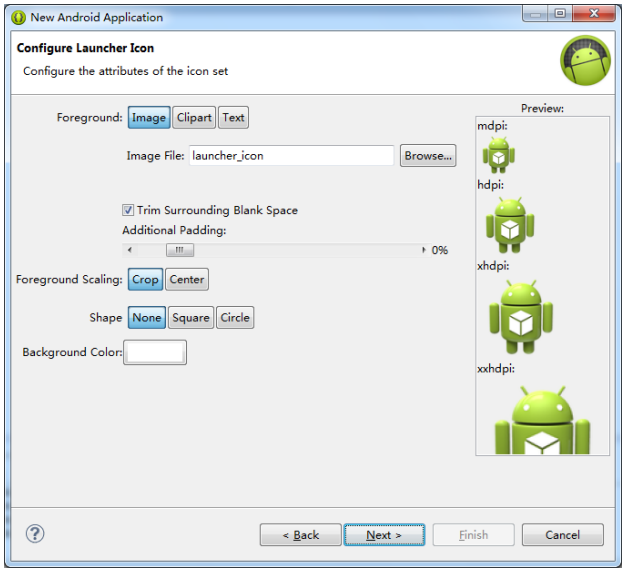

图 1.11

如果你程序的 Logo 还没设计好，别着急，在项目里面也是可以配置启动图标的，这里我们就先不配置，直接点击 Next。

然后跳转到的是创建活动界面，在这个界面你可以选择一个你想创建的活动类型，这里我们就选择 Blank Activity 了，如图 1.12 所示。

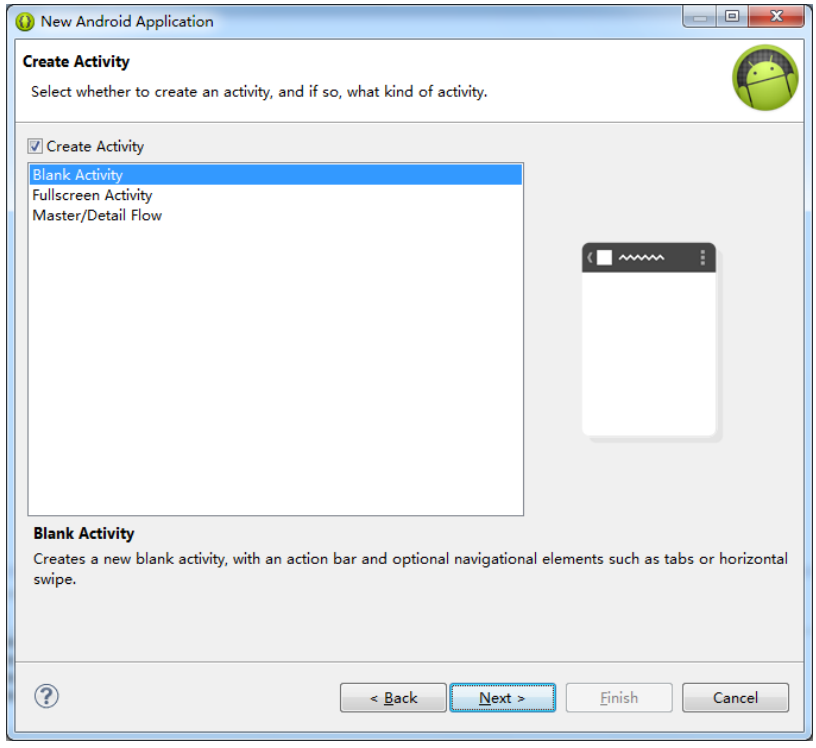

图　1.12

继续点击 Next 后，我们需要给刚刚选择的 Blank Activity 起一个名字，然后给这个活动的布局也起一个名字。Activity Name 就填入 HelloWorldActivity，Layout Name 就填入 hello_world_layout 吧，如图 1.13 所示。

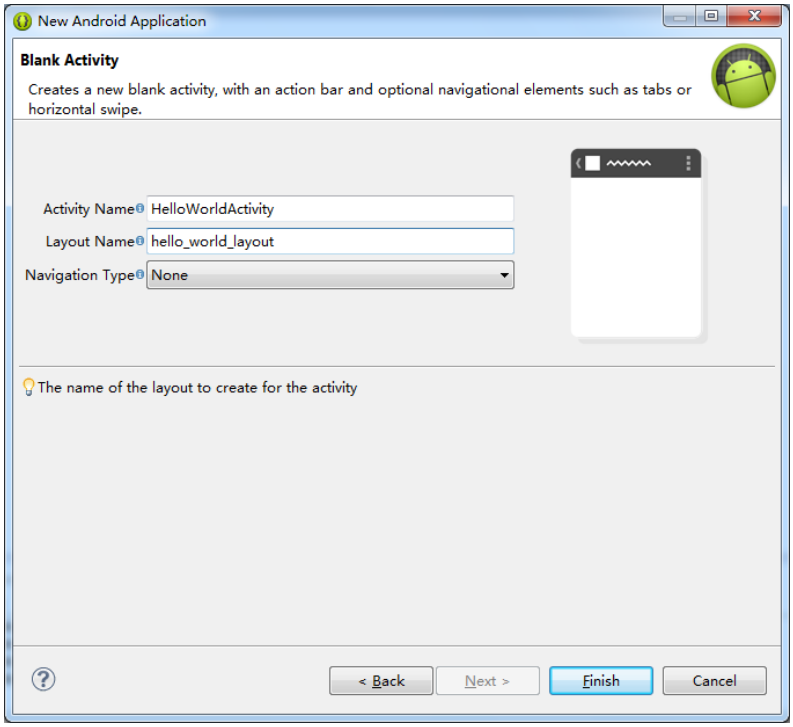

图 1.13

然后点击 Finish，项目终于创建完成了！

1.3.2 运行 HelloWorld

这个时候你的 Eclipse 中应该会显示出刚刚创建的 HelloWorld 项目，由于 ADT 已经自动为我们生成了很多东西，你现在不需要写任何代码，HelloWorld 项目就已经可以运行了。不过在运行之前，让我们先检查一下刚才的模拟器是不是还在线。

点击 Eclipse 导航栏中的 Window→Open Perspective→DDMS，这时你会进入到 DDMS 的视图中去。DDMS 中提供了很多我们开发 Android 程序时需要用到的工具，不过目前你只需要关注 Devices 窗口中有没有 Online 的设备就行了。如果你的 Devices 窗口中有一个设备显示是 Online 的，那就说明目前一切正常，你的模拟器是在线的。如果 Devices 窗口中没有设备，可能是你已经把模拟器关掉了，没关系，按照前面的步骤重新打开一次就行了。如果你的 Devices 窗口中虽然有设备，但是显示 Offline，说明你的模拟器掉线了，这种情况概率不高，但是如果出现了，你只需要点击 Reset adb 就好了，如图 1.14 所示。

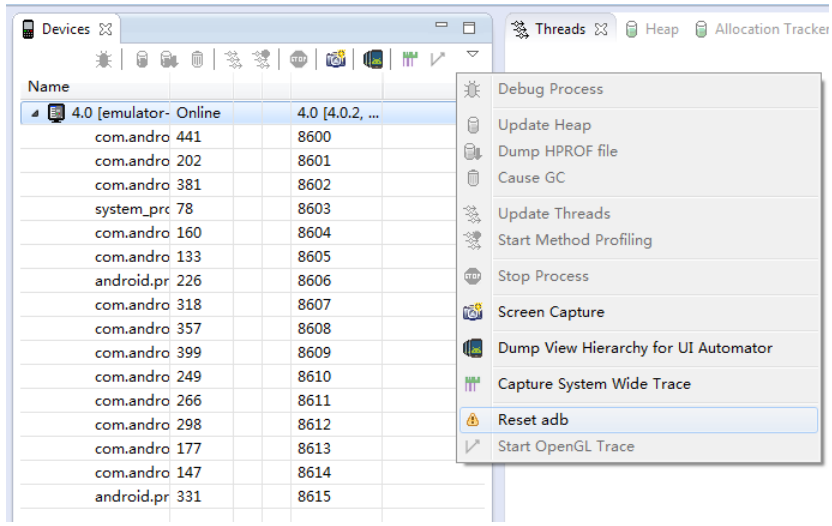

图 1.14

好了，确认完模拟器在线后，点击 Eclipse 工具栏右侧的 Java 选项，回到之前的视图，然后我们来运行一下项目吧。右击 HelloWorld 项目→Run As→Android Application。等待大约几秒钟的时间，你的项目就会运行起来了。现在快去看看你的模拟器吧，结果应该和图 1.15 中显示的是一样的。

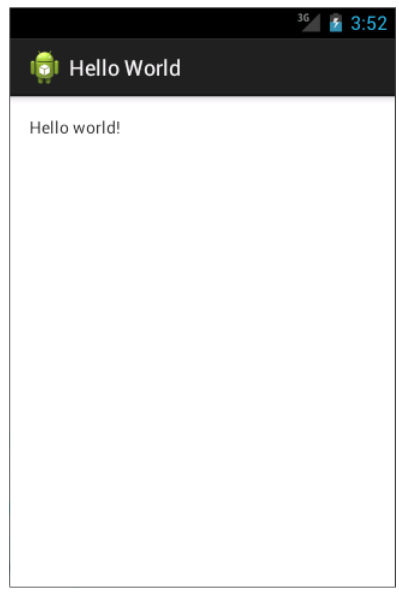

图 1.15

HelloWorld 项目运行成功！并且你会发现，你的模拟器上已经安装上 Hello World 这个应用了。打开启动器列表，如图 1.16 所示。

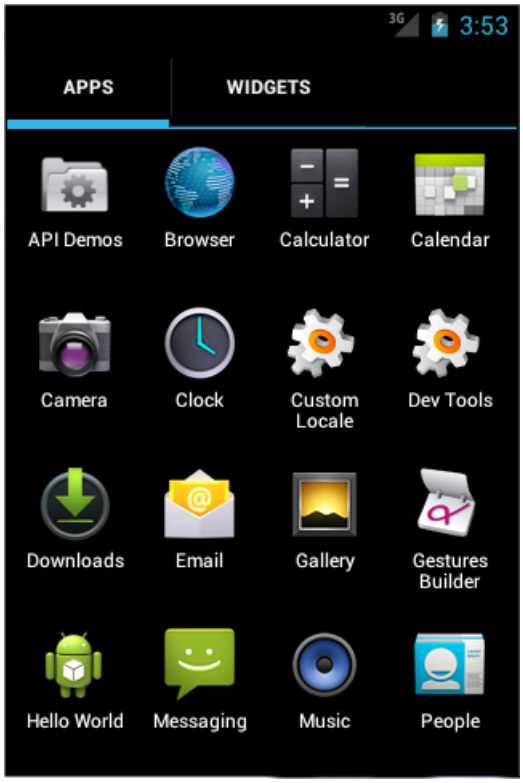

图　1.16

这个时候你可能会说我坑你了，说好的第一行代码呢？怎么一行还没写，项目就已经运行起来了？这个只能说是因为 ADT 太智能了，已经帮我们把一些简单内容都自动生成了。你也别心急，后面写代码的机会多着呢，我们先来分析一下 HelloWorld 这个项目吧。

1.3.3　分析你的第一个 Android 程序

还是回到 Eclipse 中，首先展开 HelloWorld 项目，你会看到如图 1.17 所示的目录结构。

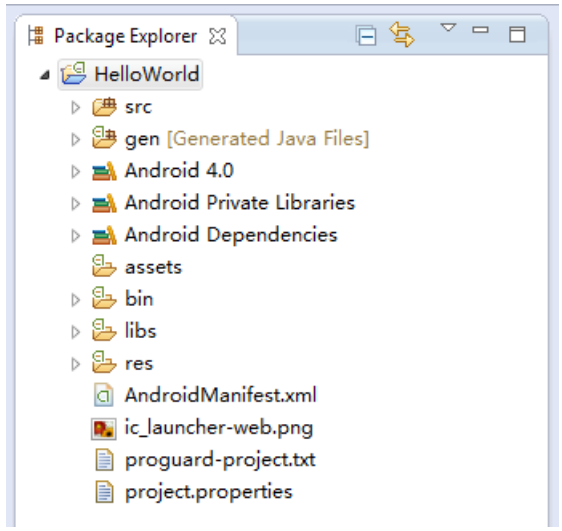

图 1.17

一开始看到这么多陌生的东西，你一定会感到有点头晕吧。别担心，我现在就对上图中的内容一一讲解，你很快再看这张图就不会感到那么吃力了。

1. src

毫无疑问，src 目录是放置我们所有 Java 代码的地方，它在这里的含义和普通 Java 项目下的 src 目录是完全一样的，展开之后你将看到我们刚才创建的 HelloWorldActivity 文件就在里面。

2. gen

这个目录里的内容都是自动生成的，主要有一个 R.java 文件，你在项目中添加的任何资源都会在其中生成一个相应的资源 id。这个文件永远不要手动去修改它。

3. assets

这个目录用得不多，主要可以存放一些随程序打包的文件，在你的程序运行时可以动态读取到这些文件的内容。另外，如果你的程序中使用到了 WebView 加载本地网页的功能，所有网页相关的文件也都存放在这个目录下。

4. bin

这个目录你也不需要过多关注，它主要包含了一些在编译时自动产生的文件。其中会有一个你当前项目编译好的安装包，展开 bin 目录你会看到 HelloWorld.apk，把这个文件拷到手机上就可以直接安装了。

5. libs

如果你的项目中使用到了第三方 Jar 包，就需要把这些 Jar 包都放在 libs 目录下，放在这个目录下的 Jar 包都会被自动添加到构建路径里去。你可以展开上图中 Android 4.0、

Android Private Libraries、Android Dependencies 这些库，其中显示的 Jar 包都是已经被添加到构建路径里的。

6. res

这个目录下的内容就有点多了，简单点说，就是你在项目中使用到的所有图片、布局、字符串等资源都要存放在这个目录下，前面提到的 R.java 中的内容也是根据这个目录下的文件自动生成的。当然这个目录下还有很多的子目录，图片放在 drawable 目录下，布局放在 layout 目录下，字符串放在 values 目录下，所以你不用担心会把整个 res 目录弄得乱糟糟的。

7. AndroidManifest.xml

这是你整个 Android 项目的配置文件，你在程序中定义的所有四大组件都需要在这个文件里注册。另外还可以在这个文件中给应用程序添加权限声明，也可以重新指定你创建项目时指定的程序最低兼容版本和目标版本。由于这个文件以后会经常用到，我们用到的时候再做详细说明。

8. project.properties

这个文件非常地简单，就是通过一行代码指定了编译程序时所使用的 SDK 版本。我们的 HelloWorld 项目使用的是 API 14，你也可以在这里改成其他版本试一试。

这样整个项目的目录结构就都介绍完了，如果你还不能完全理解的话也很正常，毕竟里面有太多的东西你都还没接触过。不用担心，这并不会影响到你后面的学习。相反，等你学完整本书后再回来看这个目录结构图时，你会觉得特别地清晰和简单。

接下来我们一起分析一下 HelloWorld 项目究竟是怎么运行起来的吧。首先打开 AndroidManifest.xml 文件，从中可以找到如下代码：

```
<activity
    android:name="com.test.helloworld.HelloWorldActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

这段代码表示对 HelloWorldActivity 这个活动进行注册，没有在 AndroidManifest.xml 里注册的活动是不能使用的。其中 intent-filter 里的两行代码非常重要，<action android:name="android.intent.action.MAIN" />和<category android:name="android.intent.category.LAUNCHER" />表示 HelloWorldActivity 是这个项目的主活动，在手机上点击应用图标，首先启动的就是这个活动。

那 HelloWorldActivity 具体又有什么作用呢？我在介绍 Android 四大组件的时候说过，

活动是 Android 应用程序的门面，凡是在应用中你看得到的东西，都是放在活动中的。因此你在图 1.15 中看到的界面，其实就是 HelloWorldActivity 这个活动。那我们快去看一下它的代码吧，打开 HelloWorldActivity，代码如下所示：

```
public class HelloWorldActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.hello_world_layout);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.hello_world, menu);
        return true;
    }

}
```

首先我们可以看到，HelloWorldActivity 是继承自 Activity 的。Activity 是 Android 系统提供的一个活动基类，我们项目中所有的活动都必须要继承它才能拥有活动的特性。然后可以看到 HelloWorldActivity 中有两个方法，onCreateOptionsMenu()这个方法是用于创建菜单的，我们可以先无视它，主要看下 onCreate()方法。onCreate()方法是一个活动被创建时必定要执行的方法，其中只有两行代码，并且没有 Hello world!的字样。那么图 1.15 中显示的 Hello world!是在哪里定义的呢？

其实 Android 程序的设计讲究逻辑和视图分离，因此是不推荐在活动中直接编写界面的，更加通用的一种做法是，在布局文件中编写界面，然后在活动中引入进来。你可以看到，在 onCreate()方法的第二行调用了 setContentView()方法，就是这个方法给当前的活动引入了一个 hello_world_layout 布局，那 Hello world!一定就是在这里定义的了！我们快打开这个文件看一看。

布局文件都是定义在 res/layout 目录下的，当你展开 layout 目录，你会看到 hello_world_layout.xml 这个文件。打开之后代码如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
```

```
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".HelloWorldActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />

</RelativeLayout>
```

现在还看不懂？没关系，后面我会对布局进行详细讲解的，你现在只需要看到上面代码中有一个 TextView，这是 Android 系统提供的一个控件，用于在布局中显示文字的。然后你终于在 TextView 中看到了 hello world 的字样，哈哈终于找到了，原来就是通过 android:text="@string/hello_world"这句代码定义的！咦？感觉不对劲啊，好像图 1.15 中显示的是 Hello world!，这感叹号怎么没了，大小写也不太一样。

其实你还是被欺骗了，真正的 Hello world!字符串也不是在布局文件中定义的。Android 不推荐在程序中对字符串进行硬编码，更好的做法一般是把字符串定义在 res/values/strings.xml 里，然后可以在布局文件或代码中引用。那我们现在打开 strings.xml 看一下，里面的内容如下：

```
<resources>
    <string name="app_name">Hello World</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
</resources>
```

这下没有什么再能逃出你的法眼了，Hello world!字符串就是定义在这个文件里的。并且字符串的定义都是使用键值对的形式，Hello world!值对应了一个叫做 hello_world 的键，因此在 hello_world_layout.xml 布局文件中就是通过引用了 hello_world 这个键，才找到了相应的值。

这个时候我无意中瞄到了这个文件中还有一个叫做 app_name 的键。你猜对了，我们还可以在这里通过修改 app_name 对应的值，来改变此应用程序的名称。那到底是哪里引用了 app_name 这个键呢？打开 AndroidManifest.xml 文件自己找找去吧！

1.3.4 详解项目中的资源

如果你展开 res 目录看一下，其实里面的东西还是挺多的，很容易让人看得眼花缭乱，如图 1.18 所示。

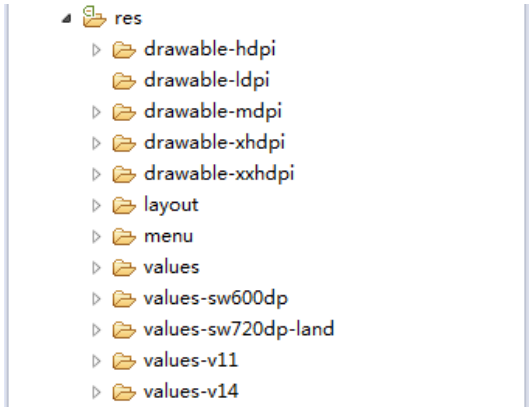

图 1.18

看到这么多的文件夹不用害怕，其实归纳一下，res 目录就变得非常简单了。所有以 drawable 开头的文件夹都是用来放图片的，所有以 values 开头的文件夹都是用来放字符串的，layout 文件夹是用来放布局文件的，menu 文件夹是用来放菜单文件的。怎么样，是不是突然感觉清晰了很多？之所以有这么多 drawable 开头的文件夹，其实主要是为了让程序能够兼容更多的设备。在制作程序的时候最好能够给同一张图片提供几个不同分辨率的副本，分别放在这些文件夹下，然后当程序运行的时候会自动根据当前运行设备分辨率的高低选择加载哪个文件夹下的图片。当然这只是理想情况，更多的时候美工只会提供给我们一份图片，这时你就把所有图片都放在 drawable-hdpi 文件夹下就好了。

知道了 res 目录下每个文件夹的含义，我们再来看一下如何去使用这些资源吧。比如刚刚在 strings.xml 中找到的 Hello world!字符串，我们有两种方式可以引用它：

1. 在代码中通过 R.string.hello_world 可以获得该字符串的引用；
2. 在 XML 中通过@string/hello_world 可以获得该字符串的引用。

基本的语法就是上面两种方式，其中 string 部分是可以替换的，如果是引用的图片资源就可以替换成 drawable，如果是引用的布局文件就可以替换成 layout，以此类推。这里就不再给出具体的例子了，因为后面你会在项目中大量地使用到各种资源，到时候例子多得是呢。另外跟你小透漏一下，HelloWorld 项目的图标就是在 AndroidManifest.xml 中通过 android:icon="@drawable/ic_launcher"来指定的，ic_launcher 这张图片就在 drawable 文件夹下，如果想要修改项目的图标应该知道怎么办了吧？

1.4 前行必备，掌握日志工具的使用

通过上一节的学习，你已经成功创建了你的第一个 Android 程序，并且对 Android 项目的目录结构和运行流程都有了一定的了解。现在本应该是你继续前行的时候，不过我想在这里给你穿插一点内容，讲解一下 Android 中日志工具的使用方法，这对你以后的 Android 开发之旅会有极大的帮助。

1.4.1 使用 Android 的日志工具 Log

既然 LogCat 已经添加完成，我们来学习一下如何使用 Android 的日志工具吧。Android 中的日志工具类是 Log（android.util.Log），这个类中提供了如下几个方法来供我们打印日志。

1. Log.v()

   这个方法用于打印那些最为琐碎的，意义最小的日志信息。对应级别 verbose，是 Android 日志里面级别最低的一种。

2. Log.d()

   这个方法用于打印一些调试信息，这些信息对你调试程序和分析问题应该是有帮助的。对应级别 debug，比 verbose 高一级。

3. Log.i()

   这个方法用于打印一些比较重要的数据，这些数据应该是你非常想看到的，可以帮你分析用户行为的那种。对应级别 info，比 debug 高一级。

4. Log.w()

   这个方法用于打印一些警告信息，提示程序在这个地方可能会有潜在的风险，最好去修复一下这些出现警告的地方。对应级别 warn，比 info 高一级。

5. Log.e()

   这个方法用于打印程序中的错误信息，比如程序进入到了 catch 语句当中。当有错误信息打印出来的时候，一般都代表你的程序出现严重问题了，必须尽快修复。对应级别 error，比 warn 高一级。

其实很简单，一共就五个方法，当然每个方法还会有不同的重载，但那对你来说肯定不是什么难理解的地方了。我们现在就在 HelloWorld 项目中试一试日志工具好不好用吧。

打开 HelloWorldActivity，在 onCreate()方法中添加一行打印日志的语句，如下所示：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.hello_world_layout);
    Log.d("HelloWorldActivity", "onCreate execute");
}
```

Log.d 方法中传入了两个参数，第一个参数是 tag，一般传入当前的类名就好，主要用于对打印信息进行过滤。第二个参数是 msg，即想要打印的具体的内容。

现在可以重新运行一下 HelloWorld 这个项目了，仍然是右击 HelloWorld 项目→Run As→Android Application。等程序运行完毕，可以看到 LogCat 中打印信息如图 1.20 所示。

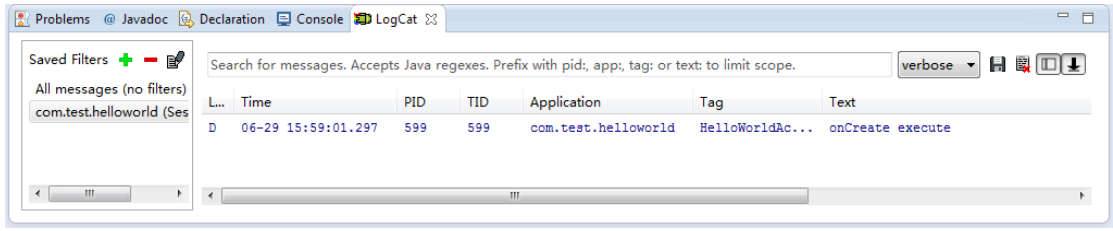

图 1.20

其中你不仅可以看到打印日志的内容和 Tag 名，就连程序的包名、打印的时间以及应用程序的进程号都可以看到。如果你的 LogCat 中并没有打印出任何信息，有可能是因为你当前的设备失去焦点了。这时你只需要进入到 DDMS 视图，在 Devices 窗口中点击一下你当前的设备，打印信息就会出来了。

另外不知道你有没有注意到，你的第一行代码已经在不知不觉中写出来了，我也总算是交差了。

1.4.2 为什么使用 Log 而不使用 System.out

我相信很多的 Java 新手都非常喜欢使用 System.out.println()方法来打印日志，不知道你是不是也喜欢这么做。不过在真正的项目开发中，是极度不建议使用 System.out.println()方法的！如果你在公司的项目中经常使用这个方法，就很有可能要挨骂了。

为什么 System.out.println()方法会这么遭大家唾弃呢？经过我仔细分析之后，发现这个方法除了使用方便一点之外，其他就一无是处了。方便在哪儿呢？在 Eclipse 中你只需要输入 syso，然后按下代码提示键，这个方法就会自动出来了，相信这也是很多 Java 新手对它钟情的原因。那缺点又在哪儿了呢？这个就太多了，比如日志打印不可控制、打印时间无法确定、不能添加过滤器、日志没有级别区分……

听我说了这些，你可能已经不太想用 System.out.println()方法了，那么 Log 就把上面所说的缺点全部都做好了吗？虽然谈不上全部，但我觉得 Log 已经做得相当不错了。我现在就来带你看看 Log 和 LogCat 配合的强大之处。

首先在 LogCat 中是可以很轻松地添加过滤器的，你可以在图 1.21 中看到我们目前所有的过滤器。

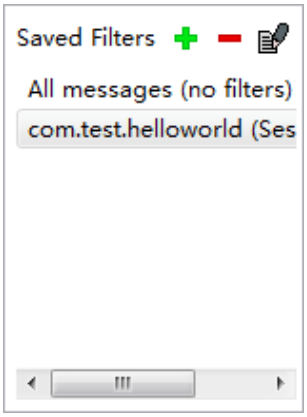

图 1.21

目前只有两个过滤器，All messages 过滤器也就相当于没有过滤器，会把所有的日志都显示出来。com.test.helloworld 过滤器是我们运行 HelloWorld 项目时自动创建的，点击这个过滤器就可以只看到 HelloWorld 程序中打印的日志。那可不可以自定义过滤器呢？当前可以，我们现在就来添加一个过滤器试试。

点击图 1.21 中的加号，会弹出一个过滤器配置界面。我们给过滤器起名叫 data，并且让它对名为 data 的 Tag 进行过滤，如图 1.22 所示。

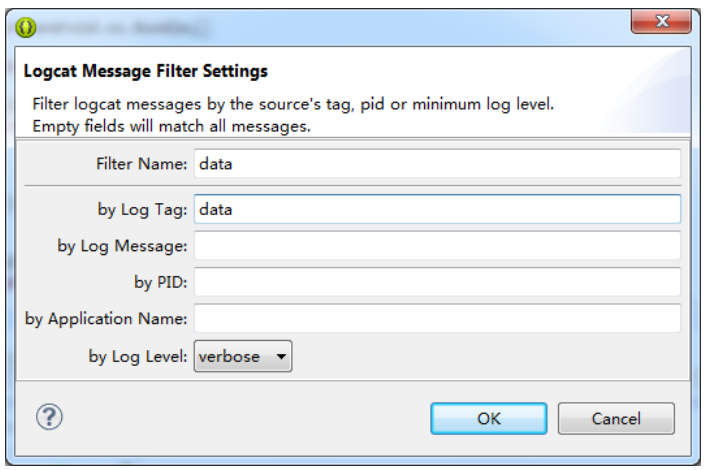

图 1.22

点击 OK，你就会发现你已经多出了一个 data 过滤器，当你点击这个过滤器的时候，你会发现刚才在 onCreate()方法里打印的日志没了，这是因为 data 这个过滤器只会显示 Tag 名称为 data 的日志。你可以尝试在 onCreate()方法中把打印日志的语句改成 Log.d("data",

"onCreate execute")，然后再次运行程序，你就会在 data 过滤器下看到这行日志了。

不知道你有没有体会到使用过滤器的好处，可能现在还没有吧。不过当你的程序打印出成百上千行日志的时候，你就会迫切地需要过滤器了。

看完了过滤器，再来看一下 LogCat 中的日志级别控制吧。LogCat 中主要有 5 个级别，分别对应着我在上一节介绍的 5 个方法，如图 1.23 所示。

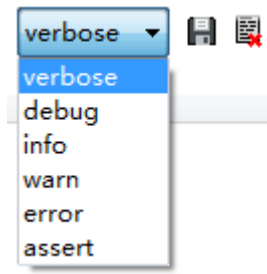

图　1.23

当前我们选中的级别是 verbose，也就是最低等级。这意味着不管我们使用哪一个方法打印日志，这条日志都一定会显示出来。而如果我们将级别选中为 debug，这时只有我们使用 debug 及以上级别方法打印的日志才会显示出来，以此类推。你可以做下试验，如果你把 LogCat 中的级别选中为 info、warn 或者 error 时，我们在 onCreate()方法中打印的语句是不会显示的，因为我们打印日志时使用的是 Log.d()方法。

日志级别控制的好处就是，你可以很快地找到你所关心的那些日志。相信如果让你从上千行日志中查找一条崩溃信息，你一定会抓狂的吧。而现在你只需要将日志级别选中为 error，那些不相干的琐碎信息就不会再干扰你的视线了。

关于 Android 中日志工具的使用我就准备讲到这里，LogCat 中其他的一些使用技巧就要靠你自己去摸索了。今天你已经学到了足够多的东西，我们来总结和梳理一下吧。

1.5　小结与点评

你现在一定会觉得很充实，甚至有点沾沾自喜。确实应该如此，因为你已经成为一名真正的 Android 开发者了。通过本章的学习，你首先对 Android 系统有了更加充足的认识，然后成功将 Android 开发环境搭建了起来，接着创建了你自己的第一个 Android 项目，并对 Android 项目的目录结构和运行流程有了一定的认识，在本章的最后还学习了 Android 日志工具的使用，这难道还不够充实吗？

不过你也别太过于满足，相信你很清楚 Android 开发者和出色的 Android 开发者还是有很大的区别的，你还需要付出更多的努力才行。即使你目前在 Java 领域已经有了不错的成

绩，我也希望在 Android 的世界你可以放下身段，以一只萌级小菜鸟的身份起飞，在后面的旅途中你会不断地成长。

现在你可以非常安心地休息一段时间，因为今天你已经做得非常不错了。储备好能量，准备进入到下一章的旅程当中。

第 2 章　探究 Activity 活动

通过上一章的学习，你已经成功创建了你的第一个 Android 项目。不过仅仅满足于此显然是不够的，是时候该学点新的东西了。作为你的导师，我有义务帮你制定好后面的学习路线，那么今天我们应该从哪儿入手呢？现在你可以想象一下，假如你已经写出了一个非常优秀的应用程序，然后推荐给你的第一个用户，你会从哪里开始介绍呢？毫无疑问，当前是从界面开始介绍了！因为即使你的程序算法再高效，架构再出色，用户根本不会在乎这些，他们一开始只会对看得到的东西感兴趣，那么我们今天的主题自然也要从看得到的入手了。

2.1　活动是什么

活动（Activity）是最容易吸引到用户的地方了，它是一种可以包含用户界面的组件，主要用于和用户进行交互。一个应用程序中可以包含零个或多个活动，但不包含任何活动的应用程序很少见，谁也不想让自己的应用永远无法被用户看到吧？

其实在上一章中，你已经和活动打过交道了，并且对活动也有了初步的认识。不过上一章我们的重点是创建你的第一个 Android 项目，对活动的介绍并不多，在本章中我将对活动进行详细的介绍。

2.2　活动的基本用法

到现在为止，你还没有手动创建过活动呢，因为上一章中的 HelloWorldActivity 是 ADT 帮我们自动创建的。手动创建活动可以加深我们的理解，因此现在是时候应该自己动手了。

首先，你需要再新建一个 Android 项目，项目名可以叫做 ActivityTest，包名我们就使用默认值 com.example.activitytest。新建项目的步骤你已经在上一章学习过了，不过图 1.12 中的那一步需要稍做修改，我们不再勾选 Create Activity 这个选项，因为这次我们准备手动创建活动，如图 2.1 所示。

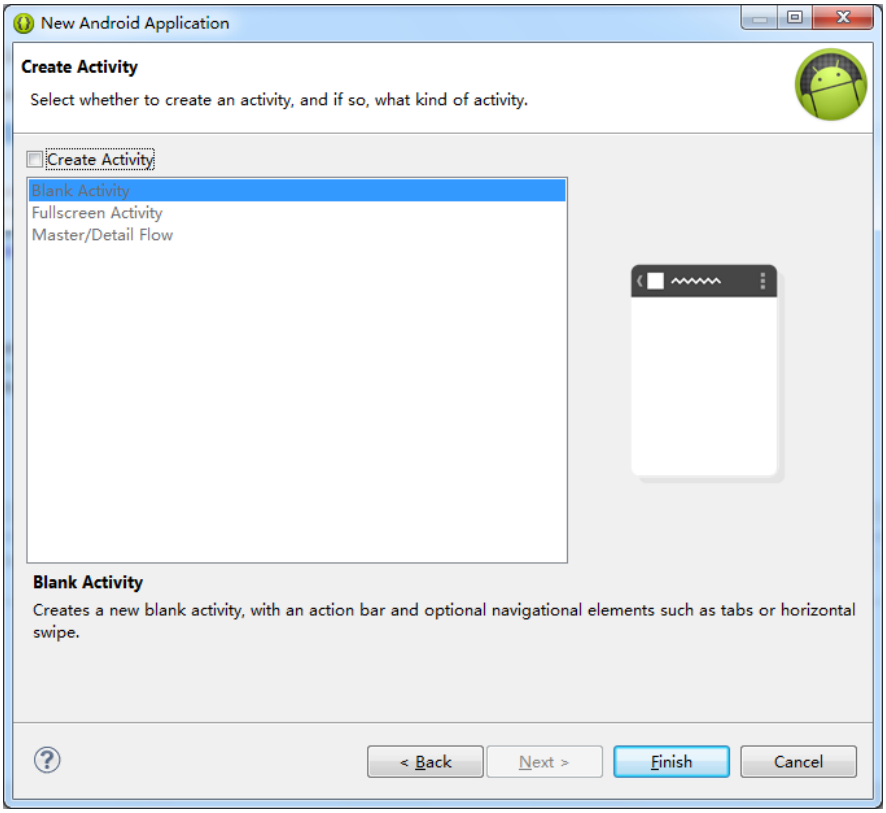

图 2.1

点击 Finish，项目就创建完成了，这时候你的 Eclipse 中应该有两个项目，ActivityTest 和 HelloWorld。极度建议你将不相干的项目关闭掉，仅打开当前工作所需要的项目，不然我保证以后你会在这方面吃亏。最好现在就右击 HelloWorld 项目→Close Project。

2.2.1 手动创建活动

目前 ActivityTest 项目的 src 目录应该是空的，你应该在 src 目录下先添加一个包。点击 Eclipse 导航栏中的 File→New→Package，在弹出窗口中填入我们新建项目时使用的默认包名 com.example.activitytest，点击 Finish。添加包之后的目录结构如图 2.2 所示。

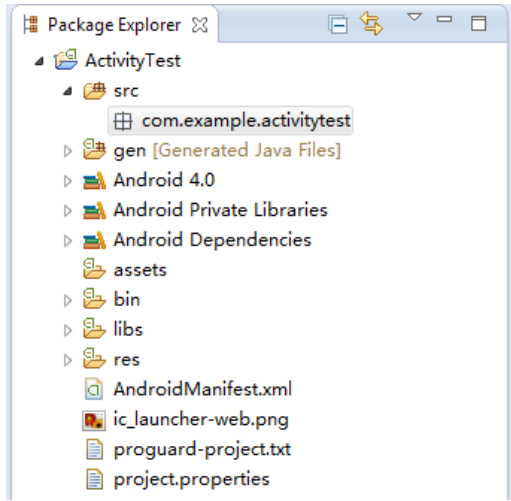

图 2.2

现在右击 com.example.activitytest 包→New→Class，会弹出新建类的对话框，我们新建一个名为 FirstActivity 的类，并让它继承自 Activity，点击 Finish 完成创建。

你需要知道，项目中的任何活动都应该重写 Activity 的 onCreate()方法，但目前我们的 FirstActivity 内部还什么代码都没有，所以首先你要做的就是在 FirstActivity 中重写 onCreate()方法，代码如下所示：

```
public class FirstActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

}
```

可以看到，onCreate()方法非常简单，就是调用了父类的 onCreate()方法。当然这只是默认的实现，后面我们还需要在里面加入很多自己的逻辑。

2.2.2 创建和加载布局

前面我们说过，Android 程序的设计讲究逻辑和视图分离，最好每一个活动都能对应一个布局，布局就是用来显示界面内容的，因此我们现在就来手动创建一个布局文件。

右击 res/layout 目录→New→Android XML File，会弹出创建布局文件的窗口。我们给这个布局文件命名为 first_layout，根元素就默认选择为 LinearLayout，如图 2.3 所示。

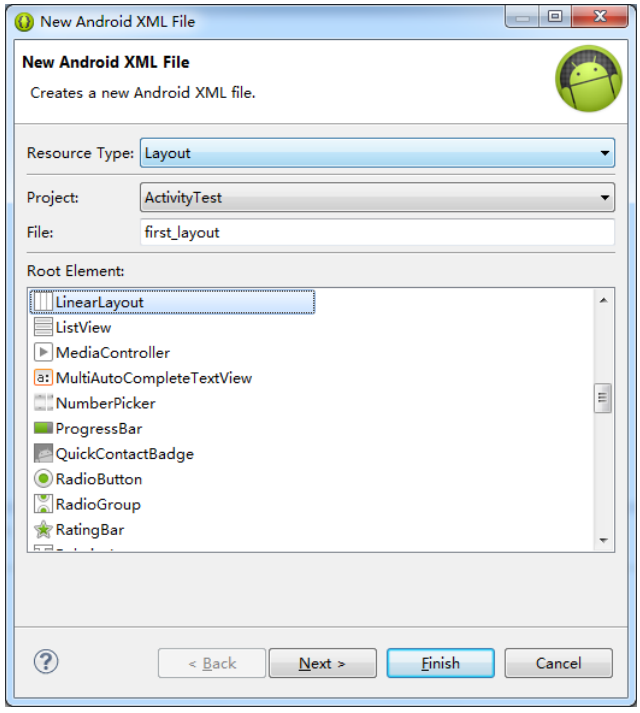

图 2.3

点击 Finish 完成布局的创建。这时候你会看到如图 2.4 所示的窗口。

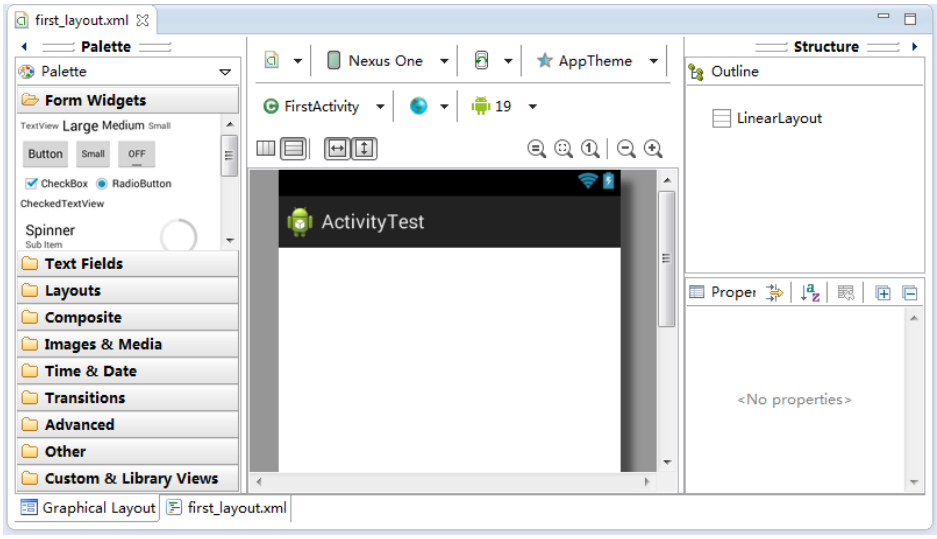

图 2.4

这是 ADT 为我们提供的可视化布局编辑器，你可以在屏幕的中央区域预览当前的布局。在窗口的最下方有两个切换卡，左边是 Graphical Layout，右边是 first_layout.xml。Graphical Layout 是当前的可视化布局编辑器，在这里你不仅可以预览当前的布局，还可以通过拖拽的方式编辑布局。而 first_layout.xml 则是通过 XML 文件的方式来编辑布局，现在点击一下 first_layout.xml 切换卡，可以看到如下代码：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
</LinearLayout>
```

由于我们刚才在创建布局文件时选择了 LinearLayout 作为根元素，因此现在布局文件中已经有一个 LinearLayout 元素了。那我们现在对这个布局稍做编辑，添加一个按钮，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button_1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 1"
        />

</LinearLayout>
```

这里添加了一个 Button 元素，并在 Button 元素的内部增加了几个属性。android:id 是给当前的元素定义一个唯一标识符，之后可以在代码中对这个元素进行操作。你可能会对@+id/button_1 这种语法感到陌生，但如果把加号去掉，变成@id/button_1，这你就会觉得有些熟悉了吧，这不就是在 XML 中引用资源的语法吗，只不过是把 string 替换成了 id。是的，如果你需要在 XML 中引用一个 id，就使用@id/id_name 这种语法，而如果你需要在 XML 中定义一个 id，则要使用@+id/id_name 这种语法。随后 android:layout_width 指定了当前元素的宽度，这里使用 match_parent 表示让当前元素和父元素一样宽。android:layout_height 指定了当前元素的高度，这里使用 wrap_content，表示当前元素的高度只要能刚好包含里面的内容就行。android:text 指定了元素中显示的文字内容。如果你还不能完全看明白，没有关系，关于编写布局的详细内容我会在下一章中重点讲解，本章只是先简单涉及一些。现在按钮已

经添加完了，你可以点回 Graphical Layout 切换卡，预览一下当前布局，如图 2.5 所示。

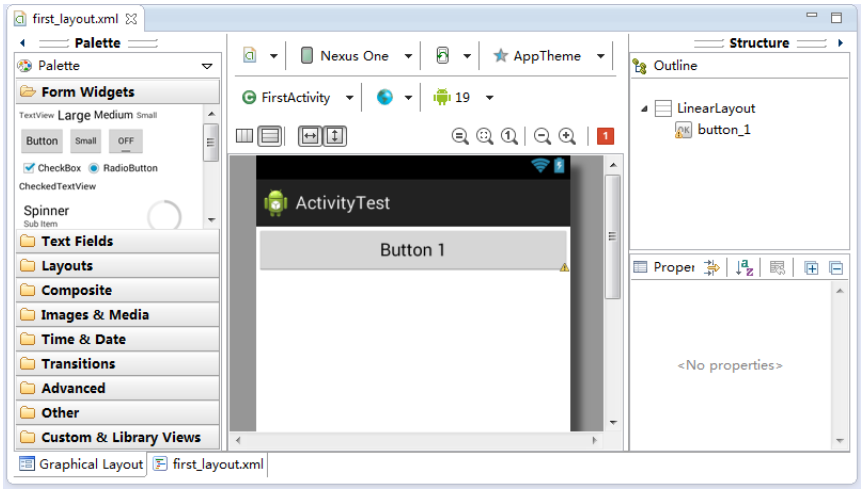

图 2.5

可以在中央的预览区域看到，按钮已经成功显示出来了，这样一个简单的布局就编写完成了。那么接下来我们要做的，就是在活动中加载这个布局。

重新回到 FirstActivity，在 onCreate()方法中加入如下代码：

```
public class FirstActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.first_layout);
    }

}
```

可以看到，这里调用了 setContentView()方法来给当前的活动加载一个布局，而在 setContentView()方法中，我们一般都会传入一个布局文件的 id。在第一章介绍 gen 目录的时候我有提到过，项目中添加的任何资源都会在 R 文件中生成一个相应的资源 id，因此我们刚才创建的 first_layout.xml 布局的 id 现在应该是已经添加到 R 文件中了。在代码中去引用布局文件的方法你也已经学过了，只需要调用 R.layout.first_layout 就可以得到 first_layout.xml 布局的 id，然后将这个值传入 setContentView()方法即可。注意这里我们使用的 R，是 com.example.activitytest 包下的 R 文件，Android SDK 还会自动提供一个 android 包下的 R 文件，千万别使用错了。

2.2.3 在 AndroidManifest 文件中注册

别忘了前面我有说过，所有的活动都要在 AndroidManifest.xml 中进行注册才能生效，那么我们现在就打开 AndroidManifest.xml 来给 FirstActivity 注册吧，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.activitytest"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="19" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".FirstActivity"
            android:label="This is FirstActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

可以看到，活动的注册声明要放在<application>标签内，这里是通过<activity>标签来对活动进行注册的。首先我们要使用 android:name 来指定具体注册哪一个活动，那么这里填入的.FirstActivity 是什么意思呢？其实这不过就是 com.example.activitytest.FirstActivity 的缩写而已。由于最外层的<manifest>标签中已经通过 package 属性指定了程序的包名是 com.example.activitytest，因此在注册活动时这一部分就可以省略了，直接使用.FirstActivity 就足够了。然后我们使用了 android:label 指定活动中标题栏的内容，标题栏是显示在活动最顶部的，待会儿运行的时候你就会看到。需要注意的是，给主活动指定的 label 不仅会成为标题栏中的内容，还会成为启动器（Launcher）中应用程序显示的名称。之后在<activity>标签的内部我们加入了<intent-filter>标签，并在这个标签里添加了<action android:name="android.intent.action.MAIN" />和<category android:name="android.intent.category.LAUNCHER" />这两句声明。这个我在前面也已经解释过了，如果你想让 FirstActivity 作为我们这个程序的

主活动，即点击桌面应用程序图标时首先打开的就是这个活动，那就一定要加入这两句声明。另外需要注意，如果你的应用程序中没有声明任何一个活动作为主活动，这个程序仍然是可以正常安装的，只是你无法在启动器中看到或者打开这个程序。这种程序一般都是作为第三方服务供其他的应用在内部进行调用的，如支付宝快捷支付服务。

好了，现在一切都已准备就绪，让我们来运行一下程序吧，结果如图 2.6 所示。

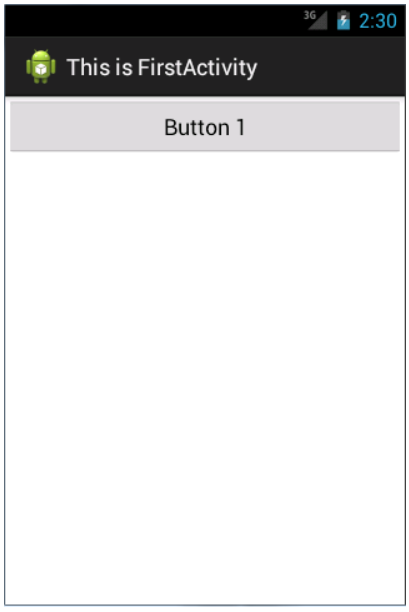

图　2.6

在界面的最顶部是一个标题栏，里面显示着我们刚才在注册活动时指定的内容。标题栏的下面就是在布局文件 first_layout.xml 中编写的界面，可以看到我们刚刚定义的按钮。现在你已经成功掌握了手动创建活动的方法，让我们继续看一看你在活动中还能做哪些更多的事情。

2.2.4　隐藏标题栏

标题栏中可以进行的操作其实还是蛮多的，尤其是在 Android 4.0 之后加入了 Action Bar 的功能。不过有些人会觉得标题栏相当占用屏幕空间，使得内容区域变小，因此也有不少的应用程序会选择将标题栏隐藏掉。

隐藏的方法非常简单，打开 FirstActivity，在 onCreate()方法中添加如下代码：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.first_layout);
    }
```

其中 requestWindowFeature(Window.FEATURE_NO_TITLE)的意思就是不在活动中显示标题栏，注意这句代码一定要在 setContentView()之前执行，不然会报错。再次运行程序，效果如图 2.7 所示。

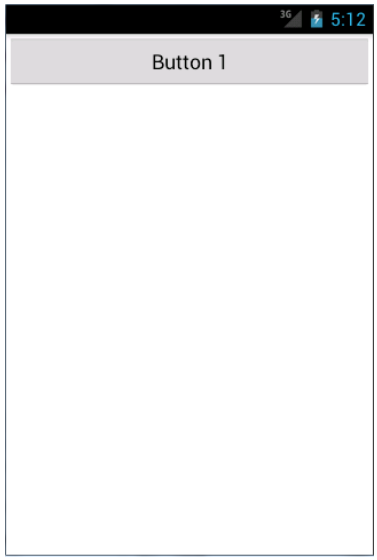

图 2.7

这样我们的活动中就不会再显示标题栏了，看起来空间大了不少吧！

2.2.5 在活动中使用 Toast

Toast 是 Android 系统提供的一种非常好的提醒方式，在程序中可以使用它将一些短小的信息通知给用户，这些信息会在一段时间后自动消失，并且不会占用任何屏幕空间，我们现在就尝试一下如何在活动中使用 Toast。

首先需要定义一个弹出 Toast 的触发点，正好界面上有个按钮，那我们就让点击这个按钮的时候弹出一个 Toast 吧。在 onCreate()方法中添加代码：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
```

```
        button1.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Toast.makeText(FirstActivity.this, "You clicked Button 1",
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
```

在活动中，可以通过 findViewById()方法获取到在布局文件中定义的元素，这里我们传入 R.id.button_1，来得到按钮的实例，这个值是刚才在 first_layout.xml 中通过 android:id 属性指定的。findViewById()方法返回的是一个 View 对象，我们需要向下转型将它转成 Button 对象。得到了按钮的实例之后，我们通过调用 setOnClickListener()方法为按钮注册一个监听器，点击按钮时就会执行监听器中的 onClick()方法。因此，弹出 Toast 的功能当然是要在 onClick()方法中编写了。

Toast 的用法非常简单，通过静态方法 makeText()创建出一个 Toast 对象，然后调用 show()将 Toast 显示出来就可以了。这里需要注意的是，makeText()方法需要传入三个参数。第一个参数是 Context，也就是 Toast 要求的上下文，由于活动本身就是一个 Context 对象，因此这里直接传入 FirstActivity.this 即可。第二个参数是 Toast 显示的文本内容，第三个参数是 Toast 显示的时长，有两个内置常量可以选择 Toast.LENGTH_SHORT 和 Toast.LENGTH_LONG。

现在重新运行程序，并点击一下按钮，效果如图 2.8 所示。

图 2.8

2.2.6 在活动中使用 Menu

不知道你还记不记得，在上一章中创建你的第一个 Android 项目时，ADT 在 HelloWorldActivity 中自动创建了一个 onCreateOptionsMenu()方法。这个方法是用于在活动中创建菜单的，由于当时我们的重点不在这里，所以直接先忽略了，现在可以来仔细分析一下了。

手机毕竟和电脑不同，它的屏幕空间非常有限，因此充分地利用屏幕空间在手机界面设计中就显得非常重要了。如果你的活动中有大量的菜单需要显示，这个时候界面设计就会比较尴尬，因为仅这些菜单就可能占用屏幕将近三分之一的空间，这该怎么办呢？不用担心，Android 给我们提供了一种方式，可以让菜单都能得到展示的同时，还能不占用任何屏幕的空间。

首先在 res 目录下新建一个 menu 文件夹，右击 res 目录→New→Folder，输入文件夹名 menu，点击 Finish。接着在这个文件夹下再新建一个名叫 main 的菜单文件，右击 menu 文件夹→New→Android XML File，如图 2.9 所示。

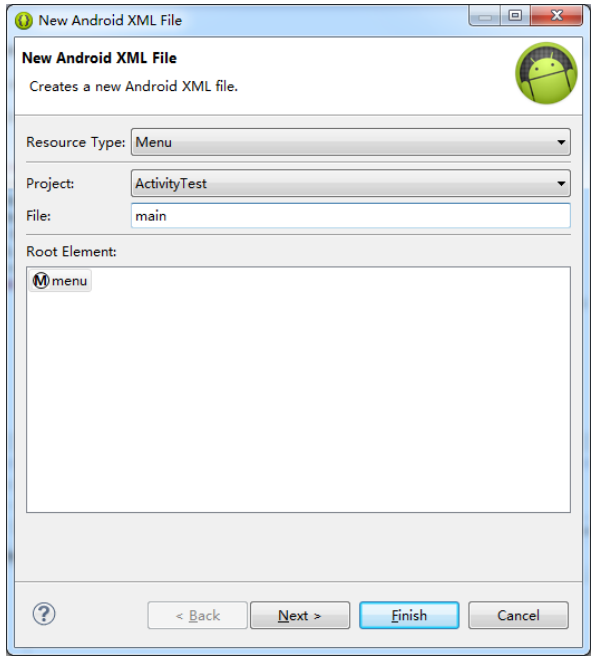

图 2.9

文件名输入 main，点击 Finish 完成创建。然后在 main.xml 中添加如下代码：

```
<menu xmlns:android="http://schemas.android.com/apk/res/android" >
```

```
    <item
        android:id="@+id/add_item"
        android:title="Add"/>
    <item
        android:id="@+id/remove_item"
        android:title="Remove"/>
</menu>
```

这里我们创建了两个菜单项，其中<item>标签就是用来创建具体的某一个菜单项，然后通过 android:id 给这个菜单项指定一个唯一标识符，通过 android:title 给这个菜单项指定一个名称。

然后打开 FirstActivity，重写 onCreateOptionsMenu()方法，代码如下所示：

```
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.main, menu);
    return true;
}
```

通过 getMenuInflater()方法能够得到 MenuInflater 对象，再调用它的 inflate()方法就可以给当前活动创建菜单了。inflate()方法接收两个参数，第一个参数用于指定我们通过哪一个资源文件来创建菜单，这里当然传入 R.menu.main，第二个参数用于指定我们的菜单项将添加到哪一个 Menu 对象当中，这里直接使用 onCreateOptionsMenu()方法中传入的 menu 参数。然后给这个方法返回 true，表示允许创建的菜单显示出来，如果返回了 false，创建的菜单将无法显示。

当然，仅仅让菜单显示出来是不够的，我们定义菜单不仅是为了看的，关键是要菜单真正可用才行，因此还要再定义菜单响应事件。在 FirstActivity 中重写 onOptionsItemSelected()方法：

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.add_item:
        Toast.makeText(this, "You clicked Add", Toast.LENGTH_SHORT).show();
        break;
    case R.id.remove_item:
        Toast.makeText(this, "You clicked Remove", Toast.LENGTH_SHORT).show();
        break;
    default:
    }
    return true;
}
```

在 onOptionsItemSelected()方法中，通过调用 item.getItemId()来判断我们点击的是哪一个菜单项，然后给每个菜单项加入自己的逻辑处理，这里我们就活学活用，弹出一个刚刚学会的 Toast。

重新运行程序，并按下 Menu 键，效果如图 2.10 所示。

图　2.10

可以看到，菜单默认是不会显示出来的，只有按下了 Menu 键，菜单才会在底部显示出来，这样我们就可以放心地使用菜单了，因为它不会占用任何活动的空间。然后点击一下 Add 菜单项，效果如图 2.11 所示。

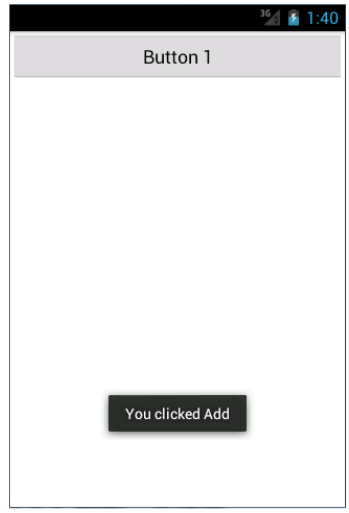

图　2.11

2.2.7 销毁一个活动

通过一整节的学习，你已经掌握了手动创建活动的方法，并学会了如何在活动中创建 Toast 和创建菜单。或许你现在心中会有个疑惑，如何销毁一个活动呢？

其实答案非常简单，只要按一下 Back 键就可以销毁当前的活动了。不过如果你不想通过按键的方式，而是希望在程序中通过代码来销毁活动，当然也可以，Activity 类提供了一个 finish()方法，我们在活动中调用一下这个方法就可以销毁当前活动了。

修改按钮监听器中的代码，如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        finish();
    }
});
```

重新运行程序，这时点击一下按钮，当前的活动就被成功销毁了，效果和按下 Back 键是一样的。

2.3 使用 Intent 在活动之间穿梭

只有一个活动的应用也太简单了吧？没错，你的追求应该更高一点。不管你想创建多少个活动，方法都和上一节中介绍的是一样的。唯一的问题在于，你在启动器中点击应用的图标只会进入到该应用的主活动，那么怎样才能由主活动跳转到其他活动呢？我们现在就来一起看一看。

2.3.1 使用显式 Intent

你应该已经对创建活动的流程比较熟悉了，那我们现在快速地在 ActivityTest 项目中再创建一个活动。新建一个 second_layout.xml 布局文件，代码如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button_2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:text="Button 2"
        />

</LinearLayout>
```

我们还是定义了一个按钮，按钮上显示 Button 2。然后新建活动 SecondActivity 继承自 Activity，代码如下：

```
public class SecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.second_layout);
    }

}
```

最后在 AndroidManifest.xml 中为 SecondActivity 进行注册。

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
        android:name=".FirstActivity"
        android:label="This is FirstActivity" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".SecondActivity" >
    </activity>
</application>
```

由于 SecondActivity 不是主活动，因此不需要配置<intent-filter>标签里的内容，注册活动的代码也是简单了许多。现在第二个活动已经创建完成，剩下的问题就是如何去启动这第二个活动了，这里我们需要引入一个新的概念，Intent。

Intent 是 Android 程序中各组件之间进行交互的一种重要方式，它不仅可以指明当前组

件想要执行的动作，还可以在不同组件之间传递数据。Intent 一般可被用于启动活动、启动服务、以及发送广播等场景，由于服务、广播等概念你暂时还未涉及，那么本章我们的目光无疑就锁定在了启动活动上面。

Intent 的用法大致可以分为两种，显式 Intent 和隐式 Intent，我们先来看一下显式 Intent 如何使用。

Intent 有多个构造函数的重载，其中一个是 Intent(Context packageContext, Class<?> cls)。这个构造函数接收两个参数，第一个参数 Context 要求提供一个启动活动的上下文，第二个参数 Class 则是指定想要启动的目标活动，通过这个构造函数就可以构建出 Intent 的“意图”。然后我们应该怎么使用这个 Intent 呢？Activity 类中提供了一个 startActivity()方法，这个方法是专门用于启动活动的，它接收一个 Intent 参数，这里我们将构建好的 Intent 传入 startActivity()方法就可以启动目标活动了。

修改 FirstActivity 中按钮的点击事件，代码如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        startActivity(intent);
    }
});
```

我们首先构建出了一个 Intent，传入 FirstActivity.this 作为上下文，传入 SecondActivity.class 作为目标活动，这样我们的“意图”就非常明显了，即在 FirstActivity 这个活动的基础上打开 SecondActivity 这个活动。然后通过 startActivity()方法来执行这个 Intent。

重新运行程序，在 FirstActivity 的界面点击一下按钮，结果如图 2.12 所示。

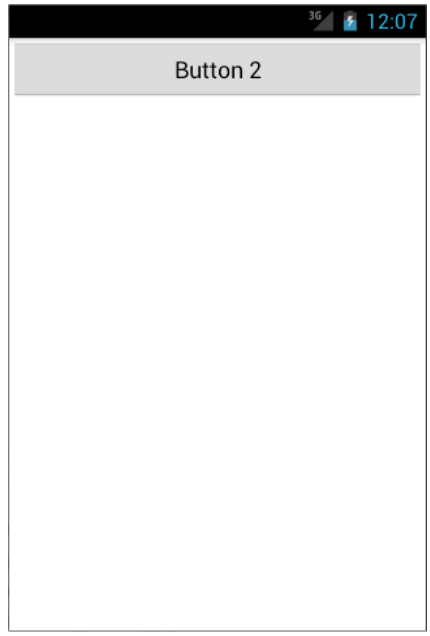

图 2.12

可以看到，我们已经成功启动 SecondActivity 这个活动了。如果你想要回到上一个活动怎么办呢？很简单，按下 Back 键就可以销毁当前活动，从而回到上一个活动了。

使用这种方式来启动活动，Intent 的“意图”非常明显，因此我们称之为显式 Intent。

2.3.2 使用隐式 Intent

相比于显式 Intent，隐式 Intent 则含蓄了许多，它并不明确指出我们想要启动哪一个活动，而是指定了一系列更为抽象的 action 和 category 等信息，然后交由系统去分析这个 Intent，并帮我们找出合适的活动去启动。

什么叫做合适的活动呢？简单来说就是可以响应我们这个隐式 Intent 的活动，那么目前 SecondActivity 可以响应什么样的隐式 Intent 呢？额，现在好像还什么都响应不了，不过很快就会有了。

通过在<activity>标签下配置<intent-filter>的内容，可以指定当前活动能够响应的 action 和 category，打开 AndroidManifest.xml，添加如下代码：

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
```

```
        </intent-filter>
    </activity>
```

在<action>标签中我们指明了当前活动可以响应 com.example.activitytest.ACTION_START 这个 action，而<category>标签则包含了一些附加信息，更精确地指明了当前的活动能够响应的 Intent 中还可能带有的 category。只有<action>和<category>中的内容同时能够匹配上 Intent 中指定的 action 和 category 时，这个活动才能响应该 Intent。

修改 FirstActivity 中按钮的点击事件，代码如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent("com.example.activitytest.ACTION_START");
        startActivity(intent);
    }
});
```

可以看到，我们使用了 Intent 的另一个构造函数，直接将 action 的字符串传了进去，表明我们想要启动能够响应 com.example.activitytest.ACTION_START 这个 action 的活动。那前面不是说要<action>和<category>同时匹配上才能响应的吗？怎么没看到哪里有指定 category 呢？这是因为 android.intent.category.DEFAULT 是一种默认的 category，在调用 startActivity()方法的时候会自动将这个 category 添加到 Intent 中。

重新运行程序，在 FirstActivity 的界面点击一下按钮，你同样成功启动 SecondActivity 了。不同的是，这次你是使用了隐式 Intent 的方式来启动的，说明我们在<activity>标签下配置的 action 和 category 的内容已经生效了！

每个 Intent 中只能指定一个 action，但却能指定多个 category。目前我们的 Intent 中只有一个默认的 category，那么现在再来增加一个吧。

修改 FirstActivity 中按钮的点击事件，代码如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent("com.example.activitytest.ACTION_START");
        intent.addCategory("com.example.activitytest.MY_CATEGORY");
        startActivity(intent);
    }
});
```

可以调用 Intent 中的 addCategory()方法来添加一个 category，这里我们指定了一个自定义的 category，值为 com.example.activitytest.MY_CATEGORY。

现在重新运行程序，在 FirstActivity 的界面点击一下按钮，你会发现，程序崩溃了！这是你第一次遇到程序崩溃，可能会有些束手无策。别紧张，其实大多数的崩溃问题都是很好解决的，只要你善于分析。在 LogCat 界面查看错误日志，你会看到如图 2.13 所示的错误信息。

```
android.content.ActivityNotFoundException: No Activity found to handle Intent □
 { act=com.example.activitytest.ACTION_START cat=[com.example.activitytest.MY □
_CATEGORY] }
```

图 2.13

错误信息中提醒我们，没有任何一个活动可以响应我们的 Intent，为什么呢？这是因为我们刚刚在 Intent 中新增了一个 category，而 SecondActivity 的<intent-filter>标签中并没有声明可以响应这个 category，所以就出现了没有任何活动可以响应该 Intent 的情况。现在我们在<intent-filter>中再添加一个 category 的声明，如下所示：

```
<activity android:name=".SecondActivity" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="com.example.activitytest.MY_CATEGORY"/>
    </intent-filter>
</activity>
```

再次重新运行程序，你就会发现一切都正常了。

2.3.3 更多隐式 Intent 的用法

上一节中，你掌握了通过隐式 Intent 来启动活动的方法，但实际上隐式 Intent 还有更多的内容需要你去了解，本节我们就来展开介绍一下。

使用隐式 Intent，我们不仅可以启动自己程序内的活动，还可以启动其他程序的活动，这使得 Android 多个应用程序之间的功能共享成为了可能。比如说你的应用程序中需要展示一个网页，这时你没有必要自己去实现一个浏览器（事实上也不太可能），而是只需要调用系统的浏览器来打开这个网页就行了。

修改 FirstActivity 中按钮点击事件的代码，如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(Intent.ACTION_VIEW);
```

```
            intent.setData(Uri.parse("http://www.baidu.com"));
            startActivity(intent);
        }
    });
```

这里我们首先指定了 Intent 的 action 是 Intent.ACTION_VIEW，这是一个 Android 系统内置的动作，其常量值为 android.intent.action.VIEW。然后通过 Uri.parse()方法，将一个网址字符串解析成一个 Uri 对象，再调用 Intent 的 setData()方法将这个 Uri 对象传递进去。

重新运行程序，在 FirstActivity 界面点击按钮就可以看到打开了系统浏览器，如图 2.14 所示。

图 2.14

上述的代码中，可能你会对 setData()部分感觉到陌生，这是我们前面没有讲到过的。这个方法其实并不复杂，它接收一个 Uri 对象，主要用于指定当前 Intent 正在操作的数据，而这些数据通常都是以字符串的形式传入到 Uri.parse()方法中解析产生的。

与此对应，我们还可以在<intent-filter>标签中再配置一个<data>标签，用于更精确地指定当前活动能够响应什么类型的数据。<data>标签中主要可以配置以下内容。

1. android:scheme
   用于指定数据的协议部分，如上例中的 http 部分。
2. android:host
   用于指定数据的主机名部分，如上例中的 www.baidu.com 部分。
3. android:port

用于指定数据的端口部分，一般紧随在主机名之后。

4. android:path

用于指定主机名和端口之后的部分，如一段网址中跟在域名之后的内容。

5. android:mimeType

用于指定可以处理的数据类型，允许使用通配符的方式进行指定。

只有<data>标签中指定的内容和 Intent 中携带的 Data 完全一致时，当前活动才能够响应该 Intent。不过一般在<data>标签中都不会指定过多的内容，如上面浏览器示例中，其实只需要指定 android:scheme 为 http，就可以响应所有的 http 协议的 Intent 了。

为了让你能够更加直观地理解，我们来自己建立一个活动，让它也能响应打开网页的 Intent。

新建 third_layout.xml 布局文件，代码如下：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button_3"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 3"
        />

</LinearLayout>
```

然后新建活动 ThirdActivity 继承自 Activity，代码如下：

```
public class ThirdActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.third_layout);
    }

}
```

最后在 AndroidManifest.xml 中为 ThirdActivity 进行注册。

```
<activity android:name=".ThirdActivity" >
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="http" />
    </intent-filter>
</activity>
```

我们在 ThirdActivity 的<intent-filter>中配置了当前活动能够响应的 action 是 Intent.ACTION_VIEW 的常量值，而 category 则毫无疑问指定了默认的 category 值，另外在<data>标签中我们通过 android:scheme 指定了数据的协议必须是 http 协议，这样 ThirdActivity 应该就和浏览器一样，能够响应一个打开网页的 Intent 了。让我们运行一下程序试试吧，在 FirstActivity 的界面点击一下按钮，结果如图 2.15 所示。

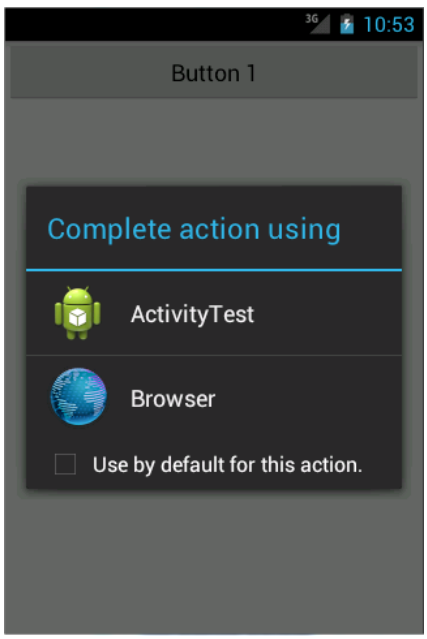

图 2.15

可以看到，系统自动弹出了一个列表，显示了目前能够响应这个 Intent 的所有程序。点击 Browser 还会像之前一样打开浏览器，并显示百度的主页，而如果点击了 ActivityTest，则会启动 ThirdActivity。需要注意的是，虽然我们声明了 ThirdActivity 是可以响应打开网页的 Intent 的，但实际上这个活动并没有加载并显示网页的功能，所以在真正的项目中尽量不要去做这种有可能误导用户的行为，不然会让用户对我们的应用产生负面的印象。

除了 http 协议外，我们还可以指定很多其他协议，比如 geo 表示显示地理位置、tel 表示拨打电话。下面的代码展示了如何在我们的程序中调用系统拨号界面。

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(Intent.ACTION_DIAL);
        intent.setData(Uri.parse("tel:10086"));
        startActivity(intent);
    }
});
```

首先指定了 Intent 的 action 是 Intent.ACTION_DIAL，这又是一个 Android 系统的内置动作。然后在 data 部分指定了协议是 tel，号码是 10086。重新运行一下程序，在 FirstActivity 的界面点击一下按钮，结果如图 2.16 所示。

图 2.16

2.3.4 向下一个活动传递数据

经过前面几节的学习，你已经对 Intent 有了一定的了解。不过到目前为止，我们都只是简单地使用 Intent 来启动一个活动，其实 Intent 还可以在启动活动的时候传递数据的，我们

来一起看一下。

在启动活动时传递数据的思路很简单，Intent 中提供了一系列 putExtra()方法的重载，可以把我们想要传递的数据暂存在 Intent 中，启动了另一个活动后，只需要把这些数据再从 Intent 中取出就可以了。比如说 FirstActivity 中有一个字符串，现在想把这个字符串传递到 SecondActivity 中，你就可以这样编写：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        String data = "Hello SecondActivity";
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        intent.putExtra("extra_data", data);
        startActivity(intent);
    }
});
```

这里我们还是使用显式 Intent 的方式来启动 SecondActivity，并通过 putExtra()方法传递了一个字符串。注意这里 putExtra()方法接收两个参数，第一个参数是键，用于后面从 Intent 中取值，第二个参数才是真正要传递的数据。

然后我们在 SecondActivity 中将传递的数据取出，并打印出来，代码如下所示：

```
public class SecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.second_layout);
        Intent intent = getIntent();
        String data = intent.getStringExtra("extra_data");
        Log.d("SecondActivity", data);
    }

}
```

首先可以通过 getIntent()方法获取到用于启动 SecondActivity 的 Intent，然后调用 getStringExtra()方法，传入相应的键值，就可以得到传递的数据了。这里由于我们传递的是字符串，所以使用 getStringExtra()方法来获取传递的数据，如果传递的是整型数据，则使用 getIntExtra()方法，传递的是布尔型数据，则使用 getBooleanExtra()方法，以此类推。

重新运行程序，在 FirstActivity 的界面点击一下按钮会跳转到 SecondActivity，查看

LogCat 打印信息，如图 2.17 所示。

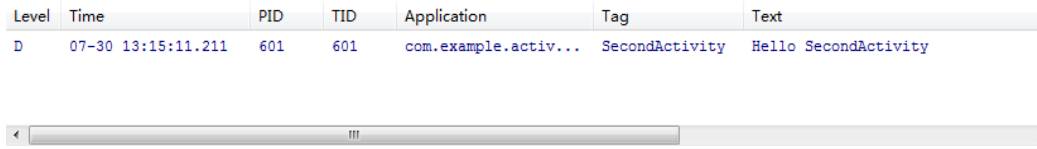

图 2.17

可以看到，我们在 SecondActivity 中成功得到了从 FirstActivity 传递过来的数据。

2.3.5 返回数据给上一个活动

既然可以传递数据给下一个活动，那么能不能够返回数据给上一个活动呢？答案是肯定的。不过不同的是，返回上一个活动只需要按一下 Back 键就可以了，并没有一个用于启动活动 Intent 来传递数据。通过查阅文档你会发现，Activity 中还有一个 startActivityForResult()方法也是用于启动活动的，但这个方法期望在活动销毁的时候能够返回一个结果给上一个活动。毫无疑问，这就是我们所需要的。

startActivityForResult()方法接收两个参数，第一个参数还是 Intent，第二个参数是请求码，用于在之后的回调中判断数据的来源。我们还是来实战一下，修改 FirstActivity 中按钮的点击事件，代码如下所示：

```
button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
        startActivityForResult(intent, 1);
    }
});
```

这里我们使用了 startActivityForResult()方法来启动 SecondActivity，请求码只要是一个唯一值就可以了，这里传入了 1。接下来我们在 SecondActivity 中给按钮注册点击事件，并在点击事件中添加返回数据的逻辑，代码如下所示：

```
public class SecondActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.second_layout);
```

```
        Button button2 = (Button) findViewById(R.id.button_2);
        button2.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent();
                intent.putExtra("data_return", "Hello FirstActivity");
                setResult(RESULT_OK, intent);
                finish();
            }
        });
    }

}
```

可以看到，我们还是构建了一个 Intent，只不过这个 Intent 仅仅是用于传递数据而已，它没有指定任何的“意图”。紧接着把要传递的数据存放在 Intent 中，然后调用了 setResult()方法。这个方法非常重要，是专门用于向上一个活动返回数据的。setResult()方法接收两个参数，第一个参数用于向上一个活动返回处理结果，一般只使用 RESULT_OK 或 RESULT_CANCELED 这两个值，第二个参数则是把带有数据的 Intent 传递回去，然后调用了 finish()方法来销毁当前活动。

由于我们是使用 startActivityForResult()方法来启动 SecondActivity 的，在 SecondActivity 被销毁之后会回调上一个活动的 onActivityResult()方法，因此我们需要在 FirstActivity 中重写这个方法来得到返回的数据，如下所示：

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
    case 1:
        if (resultCode == RESULT_OK) {
            String returnedData = data.getStringExtra("data_return");
            Log.d("FirstActivity", returnedData);
        }
        break;
    default:
    }
}
```

onActivityResult()方法带有三个参数，第一个参数 requestCode，即我们在启动活动时传入的请求码。第二个参数 resultCode，即我们在返回数据时传入的处理结果。第三个参数 data，即携带着返回数据的 Intent。由于在一个活动中有可能调用 startActivityForResult()方法去启

动很多不同的活动，每一个活动返回的数据都会回调到 onActivityResult()这个方法中，因此我们首先要做的就是通过检查 requestCode 的值来判断数据来源。确定数据是从 SecondActivity 返回的之后，我们再通过 resultCode 的值来判断处理结果是否成功。最后从 data 中取值并打印出来，这样就完成了向上一个活动返回数据的工作。

重新运行程序，在 FirstActivity 的界面点击按钮会打开 SecondActivity，然后在 SecondActivity 界面点击 Button 2 按钮会回到 FirstActivity，这时查看 LogCat 的打印信息，如图 2.18 所示。

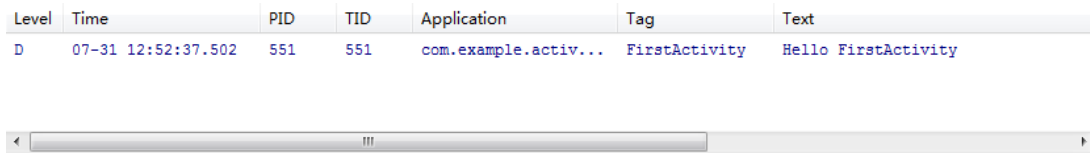

图 2.18

可以看到，SecondActivity 已经成功返回数据给 FirstActivity 了。

这时候你可能会问，如果用户在 SecondActivity 中并不是通过点击按钮，而是通过按下 Back 键回到 FirstActivity，这样数据不就没法返回了吗？没错，不过这种情况还是很好处理的，我们可以通过重写 onBackPressed()方法来解决这个问题，代码如下所示：

```
@Override
public void onBackPressed() {
    Intent intent = new Intent();
    intent.putExtra("data_return", "Hello FirstActivity");
    setResult(RESULT_OK, intent);
    finish();
}
```

这样的话，当用户按下 Back 键，就会去执行 onBackPressed()方法中的代码，我们在这里添加返回数据的逻辑就行了。

2.4 活动的生命周期

掌握活动的生命周期对任何 Android 开发者来说都非常重要，当你深入理解活动的生命周期之后，就可以写出更加连贯流畅的程序，并在如何合理管理应用资源方面，你会发挥的游刃有余。你的应用程序将会拥有更好的用户体验。

2.4.1 返回栈

经过前面几节的学习，我相信你已经发现了这一点，Android 中的活动是可以层叠的。我们每启动一个新的活动，就会覆盖在原活动之上，然后点击 Back 键会销毁最上面的活动，下面的一个活动就会重新显示出来。

其实 Android 是使用任务（Task）来管理活动的，一个任务就是一组存放在栈里的活动的集合，这个栈也被称作返回栈（Back Stack）。栈是一种后进先出的数据结构，在默认情况下，每当我们启动了一个新的活动，它会在返回栈中入栈，并处于栈顶的位置。而每当我们按下 Back 键或调用 finish()方法去销毁一个活动时，处于栈顶的活动会出栈，这时前一个入栈的活动就会重新处于栈顶的位置。系统总是会显示处于栈顶的活动给用户。

示意图 2.19 展示了返回栈是如何管理活动入栈出栈操作的。

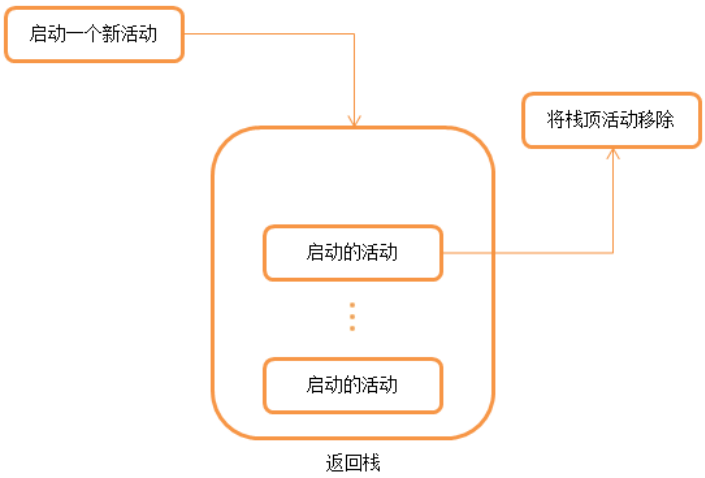

图 2.19

2.4.2 活动状态

每个活动在其生命周期中最多可能会有四种状态。

1. 运行状态

当一个活动位于返回栈的栈顶时，这时活动就处于运行状态。系统最不愿意回收的就是处于运行状态的活动，因为这会带来非常差的用户体验。

2. 暂停状态

当一个活动不再处于栈顶位置，但仍然可见时，这时活动就进入了暂停状态。你可能会觉得既然活动已经不在栈顶了，还怎么会可见呢？这是因为并不是每一个活动都会占满整个屏幕的，比如对话框形式的活动只会占用屏幕中间的部分区域，你很快就会在

后面看到这种活动。处于暂停状态的活动仍然是完全存活着的，系统也不愿意去回收这种活动（因为它还是可见的，回收可见的东西都会在用户体验方面有不好的影响），只有在内存极低的情况下，系统才会去考虑回收这种活动。

3. 停止状态

当一个活动不再处于栈顶位置，并且完全不可见的时候，就进入了停止状态。系统仍然会为这种活动保存相应的状态和成员变量，但是这并不是完全可靠的，当其他地方需要内存时，处于停止状态的活动有可能会被系统回收。

4. 销毁状态

当一个活动从返回栈中移除后就变成了销毁状态。系统会最倾向于回收处于这种状态的活动，从而保证手机的内存充足。

2.4.3 活动的生存期

Activity 类中定义了七个回调方法，覆盖了活动生命周期的每一个环节，下面我来一一介绍下这七个方法。

9. onCreate()

这个方法你已经看到过很多次了，每个活动中我们都重写了这个方法，它会在活动第一次被创建的时候调用。你应该在这个方法中完成活动的初始化操作，比如说加载布局、绑定事件等。

10. onStart()

这个方法在活动由不可见变为可见的时候调用。

11. onResume()

这个方法在活动准备好和用户进行交互的时候调用。此时的活动一定位于返回栈的栈顶，并且处于运行状态。

12. onPause()

这个方法在系统准备去启动或者恢复另一个活动的时候调用。我们通常会在这个方法中将一些消耗 CPU 的资源释放掉，以及保存一些关键数据，但这个方法的执行速度一定要快，不然会影响到新的栈顶活动的使用。

13. onStop()

这个方法在活动完全不可见的时候调用。它和 onPause()方法的主要区别在于，如果启动的新活动是一个对话框式的活动，那么 onPause()方法会得到执行，而 onStop()方法并不会执行。

14. onDestroy()

这个方法在活动被销毁之前调用，之后活动的状态将变为销毁状态。

15. onRestart()

这个方法在活动由停止状态变为运行状态之前调用，也就是活动被重新启动了。

以上七个方法中除了 onRestart()方法，其他都是两两相对的，从而又可以将活动分为三种生存期。

16. 完整生存期

活动在 onCreate()方法和 onDestroy()方法之间所经历的，就是完整生存期。一般情况下，一个活动会在 onCreate()方法中完成各种初始化操作，而在 onDestroy()方法中完成释放内存的操作。

17. 可见生存期

活动在 onStart()方法和 onStop()方法之间所经历的，就是可见生存期。在可见生存期内，活动对于用户总是可见的，即便有可能无法和用户进行交互。我们可以通过这两个方法，合理地管理那些对用户可见的资源。比如在 onStart()方法中对资源进行加载，而在 onStop()方法中对资源进行释放，从而保证处于停止状态的活动不会占用过多内存。

18. 前台生存期

活动在 onResume()方法和 onPause()方法之间所经历的，就是前台生存期。在前台生存期内，活动总是处于运行状态的，此时的活动是可以和用户进行相互的，我们平时看到和接触最多的也这个状态下的活动。

为了帮助你能够更好的理解，Android 官方提供了一张活动生命周期的示意图，如图 2.20 所示。

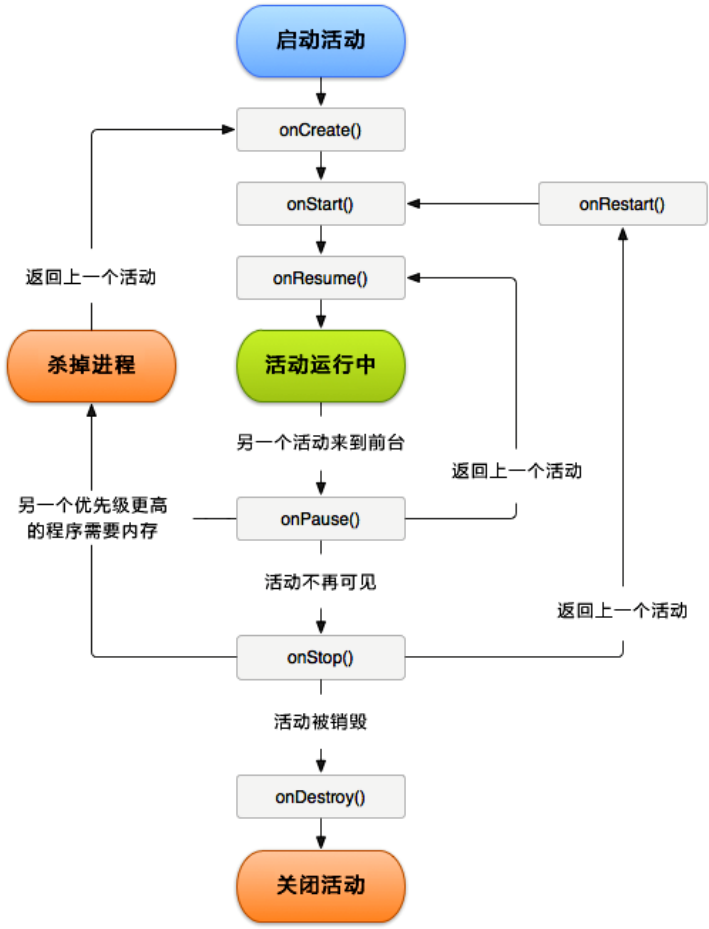

图 2.20

2.4.4 体验活动的生命周期

讲了这么多理论知识，也是时候该实战一下了，下面我们将通过一个实例，让你可以更加直观地体验活动的生命周期。

这次我们不准备在 ActivityTest 这个项目的基础上修改了，而是新建一个项目。因此，首先关闭 ActivityTest 项目，然后新建一个 ActivityLifeCycleTest 项目。新建项目的过程你应该已经非常清楚了，不需要我再进行赘述，这次我们允许 ADT 帮我们自动创建活动，这样可以省去不少工作，创建的活动名和布局名都使用默认值。

这样主活动就创建完成了，我们还需要分别再创建两个子活动，NormalActivity 和

DialogActivity，下面一步步来实现。

新建 normal_layout.xml 文件，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This is a normal activity"
        />

</LinearLayout>
```

这个布局中我们就非常简单地使用了一个 TextView，用于显示一行文字，在下一章中你将会学到更多关于 TextView 的用法。

然后同样的方法，我们再新建一个 dialog_layout.xml 文件，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This is a dialog activity"
        />

</LinearLayout>
```

两个布局文件的代码几乎没有区别，只是显示的文字不同而已。

然后新建 NormalActivity 继承自 Activity，代码如下所示：

```
public class NormalActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
```

```
        setContentView(R.layout.normal_layout);
    }

}
```

我们在 NormalActivity 中加载了 normal_layout 这个布局。

同样的方法，再新建 DialogActivity 继承自 Activity，代码如下所示：

```
public class DialogActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.dialog_layout);
    }

}
```

我们在 DialogActivity 中加载了 dialog_layout 这个布局。

其实从名字上你就可以看出，这两个活动一个是普通的活动，一个是对话框式的活动。可是现在不管怎么看，这两个活动的代码都几乎都是一模一样的，在哪里有体现出将活动设成对话框式的呢？别着急，下面我们马上开始设置。在 AndroidManifest.xml 的<activity>标签中添加如下代码：

```
    <activity android:name=".NormalActivity" >
    </activity>
    <activity android:name=".DialogActivity" android:theme="@android:style/
Theme.Dialog" >
    </activity>
```

这里分别为两个活动进行注册，但是 DialogActivity 的注册代码有些不同，它使用了一个 android:theme 属性，这是用于给当前活动指定主题的，Android 系统内置有很多主题可以选择，当然我们也可以定制自己的主题，而这里@android:style/Theme.Dialog 则毫无疑问是让 DialogActivity 使用对话框式的主题。

接下来我们修改 activity_main.xml，重新定制我们主活动的布局：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

```
    <Button
        android:id="@+id/start_normal_activity"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start NormalActivity" />

    <Button
        android:id="@+id/start_dialog_activity"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start DialogActivity" />

</LinearLayout>
```

自动生成的布局代码有些复杂，这里我们完全替换掉，仍然还是使用最熟悉的 LinearLayout，然后加入了两个按钮，一个用于启动 NormalActivity，一个用于启动 DialogActivity。

最后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    public static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);
        Button startNormalActivity = (Button) findViewById(R.id.start_
normal_activity);
        Button startDialogActivity = (Button) findViewById(R.id.start_
dialog_activity);
        startNormalActivity.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity.this,
NormalActivity.class);
                startActivity(intent);
            }
        });
```

```
        startDialogActivity.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity.this,
DialogActivity.class);
                startActivity(intent);
            }
        });
    }

    @Override
    protected void onStart() {
        super.onStart();
        Log.d(TAG, "onStart");
    }

    @Override
    protected void onResume() {
        super.onResume();
        Log.d(TAG, "onResume");
    }

    @Override
    protected void onPause() {
        super.onPause();
        Log.d(TAG, "onPause");
    }

    @Override
    protected void onStop() {
        super.onStop();
        Log.d(TAG, "onStop");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy");
    }
```

```
    @Override
    protected void onRestart() {
        super.onRestart();
        Log.d(TAG, "onRestart");
    }

}
```

在 onCreate()方法中，我们分别为两个按钮注册了点击事件，点击第一个按钮会启动 NormalActivity，点击第二个按钮会启动 DialogActivity。然后在 Activity 的七个回调方法中分别打印了一句话，这样就可以通过观察日志的方式来更直观地理解活动的生命周期。

现在运行程序，效果如图 2.21 所示。

图 2.21

这时观察 LogCat 中的打印日志，如图 2.22 所示。

Tag	Text
MainActivity	onCreate
MainActivity	onStart
MainActivity	onResume

图 2.22

可以看到，当MainActivity第一次被创建时会依次执行onCreate()、onStart()和onResume()方法。然后点击第一个按钮，启动NormalActivity，如图2.23所示。

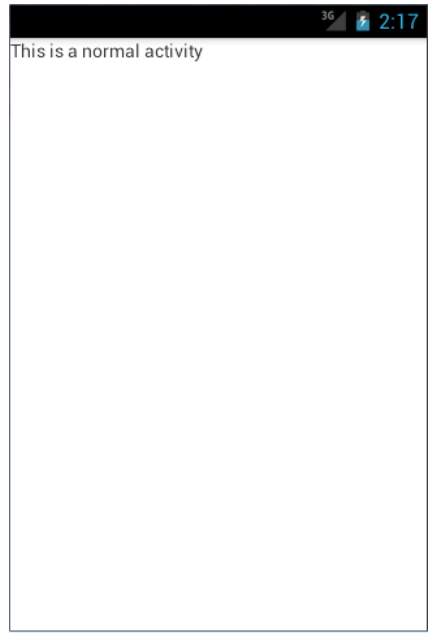

图 2.23

此时的打印信息如图2.24所示。

Tag	Text
MainActivity	onPause
MainActivity	onStop

图 2.24

由于NormalActivity已经把MainActivity完全遮挡住，因此onPause()和onStop()方法都

会得到执行。然后按下 Back 键返回 MainActivity，打印信息如图 2.25 所示。

Tag	Text
MainActivity	onRestart
MainActivity	onStart
MainActivity	onResume

图 2.25

由于之前 MainActivity 已经进入了停止状态，所以 onRestart()方法会得到执行，之后又会依次执行 onStart()和 onResume()方法。注意此时 onCreate()方法不会执行，因为 MainActivity 并没有重新创建。

然后再点击第二个按钮，启动 DialogActivity，如图 2.26 所示。

图 2.26

此时观察打印信息，如图 2.27 所示。

Tag	Text
MainActivity	onPause

图 2.27

可以看到，只有 onPause()方法得到了执行，onStop()方法并没有执行，这是因为 DialogActivity 并没有完全遮挡住 MainActivity，此时 MainActivity 只是进入了暂停状态，并没有进入停止状态。相应地，按下 Back 键返回 MainActivity 也应该只有 onResume()方法会得到执行，如图 2.28 所示。

Tag	Text
MainActivity	onResume

图 2.28

最后在 MainActivity 按下 Back 键退出程序，打印信息如图 2.29 所示。

Tag	Text
MainActivity	onPause
MainActivity	onStop
MainActivity	onDestroy

图 2.29

依次会执行 onPause()、onStop()和 onDestroy()方法，最终销毁 MainActivity。

这样活动完整的生命周期你已经体验了一遍，是不是理解得更加深刻了？

2.5 活动的启动模式

活动的启动模式对你来说应该是个全新的概念，在实际项目中我们应该根据特定的需求为每个活动指定恰当的启动模式。启动模式一共有四种，分别是 standard、singleTop、singleTask 和 singleInstance，可以在 AndroidManifest.xml 中通过给<activity>标签指定 android:launchMode 属性来选择启动模式。下面我们来逐个进行学习。

2.5.1 standard

standard 是活动默认的启动模式，在不进行显式指定的情况下，所有活动都会自动使用这种启动模式。因此，到目前为止我们写过的所有活动都是使用的 standard 模式。经过上一节的学习，你已经知道了 Android 是使用返回栈来管理活动的，在 standard 模式（即默认情况）下，每当启动一个新的活动，它就会在返回栈中入栈，并处于栈顶的位置。对于使用 standard 模式的活动，系统不会在乎这个活动是否已经在返回栈中存在，每次启动都会创建该活动的一个新的实例。

我们现在通过实践来体会一下 standard 模式，这次还是准备在 ActivityTest 项目的基础上修改，首先关闭 ActivityLifeCycleTest 项目，打开 ActivityTest 项目。

修改 FirstActivity 中 onCreate()方法的代码，如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("FirstActivity", this.toString());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
    button1.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(FirstActivity.this, FirstActivity.class);
            startActivity(intent);
        }
    });
}
```

代码看起来有些奇怪吧，在 FirstActivity 的基础上启动 FirstActivity。从逻辑上来讲这确实没什么意义，不过我们的重点在于研究 standard 模式，因此不必在意这段代码有什么实际用途。另外我们还在 onCreate()方法中添加了一行打印信息，用于打印当前活动的实例。

现在重新运行程序，然后在 FirstActivity 界面连续点击两次按钮，可以看到 LogCat 中打印信息如图 2.30 所示。

Tag	Text
FirstActivity	com.example.activitytest.FirstActivity@41068da8
FirstActivity	com.example.activitytest.FirstActivity@41076c08
FirstActivity	com.example.activitytest.FirstActivity@4107d578

图 2.30

从打印信息中我们就可以看出，每点击一次按钮就会创建出一个新的 FirstActivity 实例。此时返回栈中也会存在三个 FirstActivity 的实例，因此你需要连按三次 Back 键才能退出程序。

standard 模式的原理示意图，如图 2.31 所示。

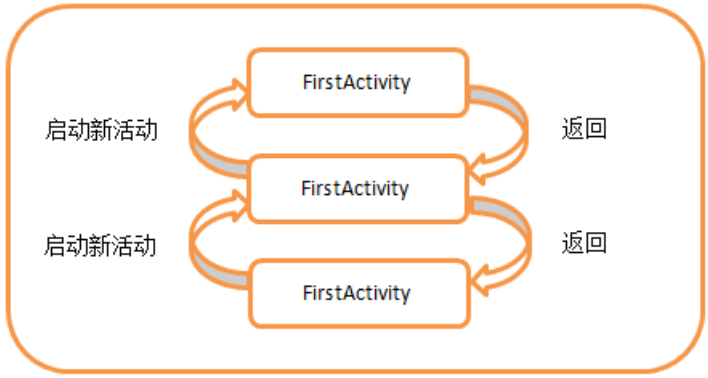

图 2.31

2.5.2 singleTop

可能在有些情况下，你会觉得 standard 模式不太合理。活动明明已经在栈顶了，为什么再次启动的时候还要创建一个新的活动实例呢？别着急，这只是系统默认的一种启动模式而已，你完全可以根据自己的需要进行修改，比如说使用 singleTop 模式。当活动的启动模式指定为 singleTop，在启动活动时如果发现返回栈的栈顶已经是该活动，则认为可以直接使用它，不会再创建新的活动实例。

我们还是通过实践来体会一下，修改 AndroidManifest.xml 中 FirstActivity 的启动模式，如下所示：

```
<activity
    android:name=".FirstActivity"
    android:launchMode="singleTop"
    android:label="This is FirstActivity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

然后重新运行程序，查看 LogCat 会看到已经创建了一个 FirstActivity 的实例，如图 2.32 所示。

Tag	Text
FirstActivity	com.example.activitytest.FirstActivity@41063f90

图 2.32

但是之后不管你点击多少次按钮都不会再有新的打印信息出现，因为目前 FirstActivity 已经处于返回栈的栈顶，每当想要再启动一个 FirstActivity 时都会直接使用栈顶的活动，因此 FirstActivity 也只会有一个实例，仅按一次 Back 键就可以退出程序。

不过当 FirstActivity 并未处于栈顶位置时，这时再启动 FirstActivity，还是会创建新的实例的。下面我们来实验一下，修改 FirstActivity 中 onCreate()方法的代码，如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("FirstActivity", this.toString());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
    button1.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(FirstActivity.this,
SecondActivity.class);
            startActivity(intent);
        }
    });
}
```

这次我们点击按钮后启动的是 SecondActivity。然后修改 SecondActivity 中 onCreate()方法的代码，如下所示：

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("SecondActivity", this.toString());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.second_layout);
    Button button2 = (Button) findViewById(R.id.button_2);
    button2.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(SecondActivity.this,
FirstActivity.class);
            startActivity(intent);
        }
```

```
        });
    }
```

我们在 SecondActivity 中的按钮点击事件里又加入了启动 FirstActivity 的代码。现在重新运行程序，在 FirstActivity 界面点击按钮进入到 SecondActivity，然后在 SecondActivity 界面点击按钮，又会重新进入到 FirstActivity。

查看 LogCat 中的打印信息，如图 2.33 所示。

Tag	Text
FirstActivity	com.example.activitytest.FirstActivity@4107c038
SecondActivity	com.example.activitytest.SecondActivity@41082530
FirstActivity	com.example.activitytest.FirstActivity@41088ec0

图 2.33

可以看到系统创建了两个不同的 FirstActivity 实例，这是由于在 SecondActivity 中再次启动 FirstActivity 时，栈顶活动已经变成了 SecondActivity，因此会创建一个新的 FirstActivity 实例。现在按下 Back 键会返回到 SecondActivity，再次按下 Back 键又会回到 FirstActivity，再按一次 Back 键才会退出程序。

singleTop 模式的原理示意图，如图 2.34 所示。

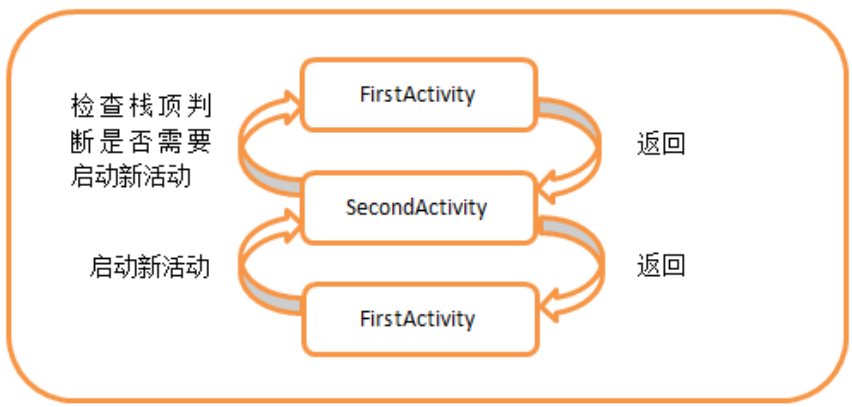

图 2.34

2.5.3 singleTask

使用 singleTop 模式可以很好地解决重复创建栈顶活动的问题，但是正如你在上一节所

看到的，如果该活动并没有处于栈顶的位置，还是可能会创建多个活动实例的。那么有没有什么办法可以让某个活动在整个应用程序的上下文中只存在一个实例呢？这就要借助 singleTask 模式来实现了。当活动的启动模式指定为 singleTask，每次启动该活动时系统首先会在返回栈中检查是否存在该活动的实例，如果发现已经存在则直接使用该实例，并把在这个活动之上的所有活动统统出栈，如果没有发现就会创建一个新的活动实例。

我们还是通过代码来更加直观地理解一下。修改 AndroidManifest.xml 中 FirstActivity 的启动模式：

```
<activity
    android:name=".FirstActivity"
    android:launchMode="singleTask"
    android:label="This is FirstActivity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

然后在 FirstActivity 中添加 onRestart()方法，并打印日志：

```
@Override
protected void onRestart() {
    super.onRestart();
    Log.d("FirstActivity", "onRestart");
}
```

最后在 SecondActivity 中添加 onDestroy()方法，并打印日志：

```
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.d("SecondActivity", "onDestroy");
}
```

现在重新运行程序，在 FirstActivity 界面点击按钮进入到 SecondActivity，然后在 SecondActivity 界面点击按钮，又会重新进入到 FirstActivity。

查看 LogCat 中的打印信息，如图 2.35 所示。

Tag	Text
FirstActivity	com.example.activitytest.FirstActivity@41067238
SecondActivity	com.example.activitytest.SecondActivity@4106e8b0
FirstActivity	onRestart
SecondActivity	onDestroy

图 2.35

其实从打印信息中就可以明显看出了，在 SecondActivity 中启动 FirstActivity 时，会发现返回栈中已经存在一个 FirstActivity 的实例，并且是在 SecondActivity 的下面，于是 SecondActivity 会从返回栈中出栈，而 FirstActivity 重新成为了栈顶活动，因此 FirstActivity 的 onRestart()方法和 SecondActivity 的 onDestroy()方法会得到执行。现在返回栈中应该只剩下一个 FirstActivity 的实例了，按一下 Back 键就可以退出程序。

singleTask 模式的原理示意图，如图 2.36 所示。

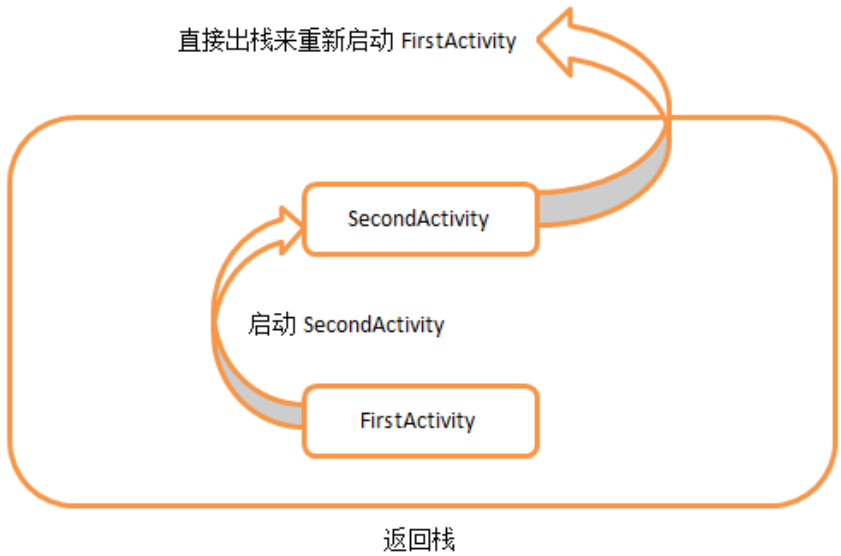

图 2.36

2.5.4 singleInstance

singleInstance 模式应该算是四种启动模式中最特殊也最复杂的一个了，你也需要多花点功夫来理解这个模式。不同于以上三种启动模式，指定为 singleInstance 模式的活动会启用一个新的返回栈来管理这个活动（其实如果 singleTask 模式指定了不同的 taskAffinity，也会启

动一个新的返回栈）。那么这样做有什么意义呢？想象以下场景，假设我们的程序中有一个活动是允许其他程序调用的，如果我们想实现其他程序和我们的程序可以共享这个活动的实例，应该如何实现呢？使用前面三种启动模式肯定是做不到的，因为每个应用程序都会有自己的返回栈，同一个活动在不同的返回栈中入栈时必然是创建了新的实例。而使用 singleInstance 模式就可以解决这个问题，在这种模式下会有一个单独的返回栈来管理这个活动，不管是哪个应用程序来访问这个活动，都共用的同一个返回栈，也就解决了共享活动实例的问题。

为了帮助你可以更好地理解这种启动模式，我们还是来实践一下。修改 AndroidManifest.xml 中 SecondActivity 的启动模式：

```
<activity
    android:name=".SecondActivity"
    android:launchMode="singleInstance" >
    <intent-filter>
        <action android:name="com.example.activitytest.ACTION_START" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="com.example.activitytest.MY_CATEGORY" />
    </intent-filter>
</activity>
```

我们先将 SecondActivity 的启动模式指定为 singleInstance，然后修改 FirstActivity 中 onCreate()方法的代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("FirstActivity", "Task id is " + getTaskId());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
    button1.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(FirstActivity.this,
SecondActivity.class);
            startActivity(intent);
        }
    });
}
```

在 onCreate()方法中打印了当前返回栈的 id。然后修改 SecondActivity 中 onCreate()方法的代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("SecondActivity", "Task id is " + getTaskId());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.second_layout);
    Button button2 = (Button) findViewById(R.id.button_2);
    button2.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent intent = new Intent(SecondActivity.this,
ThirdActivity.class);
            startActivity(intent);
        }
    });
}
```

同样在 onCreate()方法中打印了当前返回栈的 id，然后又修改了按钮点击事件的代码，用于启动 ThirdActivity。最后修改 ThirdActivity 中 onCreate()方法的代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("ThirdActivity", "Task id is " + getTaskId());
    requestWindowFeature(Window.FEATURE_NO_TITLE);
    setContentView(R.layout.third_layout);
}
```

仍然是在 onCreate()方法中打印了当前返回栈的 id。现在重新运行程序，在 FirstActivity 界面点击按钮进入到 SecondActivity，然后在 SecondActivity 界面点击按钮进入到 ThirdActivity。

查看 LogCat 中的打印信息，如图 2.37 所示。

Tag	Text
FirstActivity	Task id is 19
SecondActivity	Task id is 20
ThirdActivity	Task id is 19

图 2.37

可以看到，SecondActivity 的 Task id 不同于 FirstActivity 和 ThirdActivity，这说明 SecondActivity 确实是存放在一个单独的返回栈里的，而且这个栈中只有 SecondActivity 这一个活动。

然后我们按下 Back 键进行返回，你会发现 ThirdActivity 竟然直接返回到了 FirstActivity，再按下 Back 键又会返回到 SecondActivity，再按下 Back 键才会退出程序，这是为什么呢？其实原理很简单，由于 FirstActivity 和 ThirdActivity 是存放在同一个返回栈里的，当在 ThirdActivity 的界面按下 Back 键，ThirdActivity 会从返回栈中出栈，那么 FirstActivity 就成为了栈顶活动显示在界面上，因此也就出现了从 ThirdActivity 直接返回到 FirstActivity 的情况。然后在 FirstActivity 界面再次按下 Back 键，这时当前的返回栈已经空了，于是就显示了另一个返回栈的栈顶活动，即 SecondActivity。最后再次按下 Back 键，这时所有返回栈都已经空了，也就自然退出了程序。

singleInstance 模式的原理示意图，如图 2.38 所示。

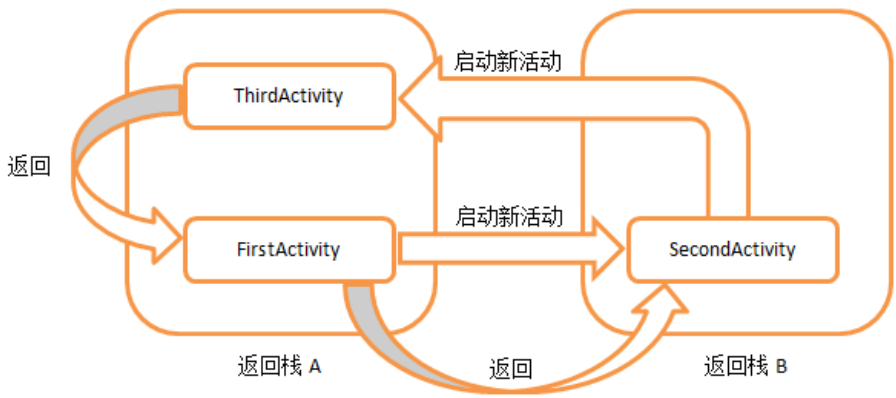

图 2.38

第 3 章　UI 用户界面开发

我一直都认为程序员在软件的审美方面普遍都比较差，至少我个人就是如此。如果说要追究其根本原因，我觉得这是由于程序员的工作性质所导致的。每当我们看到一个软件时，不会像普通用户一样仅仅是关注一下它的界面以及有哪些功能。我们总是会不自觉地思考这些功能是如何实现的，很多在普通用户看来理所应当的功能，背后可能却需要非常复杂的算法来完成。以至于当别人唾骂一句，这软件做得真丑的时候，我们还可能赞叹一句，这功能做得好牛逼啊！

不过缺乏审美观毕竟不是一件值得炫耀的事情，在软件开发过程中，界面设计和功能开发同样重要。界面美观的应用程序不仅可以大大增加用户粘性，还能帮我们吸引到更多的新用户。而 Android 也是给我们提供了大量的 UI 开发工具，只要合理地使用它们，就可以编写出各种各样漂亮的界面。

在这里，我无法教会你如何提升自己的审美观，但我可以教会你怎样使用 Android 提供的 UI 开发工具来编写程序界面。想必你在上一章中反反复复地使用那几个按钮都快要吐了吧，本章我们就来学习更多的 UI 开发方面的知识。

3.1　该如何编写程序界面

Android 中有好几种编写程序界面的方式可供你选择。比如使用 DroidDraw，这是一种可视化的界面编辑工具，允许使用拖拽控件的方式来编写布局。Eclipse 和 Android Studio 中也有相应的可视化编辑器，和 DroidDraw 用法差不多，都是可以直接拖拽控件，并能在视图上修改控件属性的。不过以上的方式我都不推荐你使用，因为使用可视化编辑工具并不利于你去真正了解界面背后的实现原理，通常这种方式制作出的界面都不具有很好的屏幕适配性，而且当需要编写较为复杂的界面时，可视化编辑工具将很难胜任。因此本书中所有的界面我们都将使用最基本的方式去实现，即编写 XML 代码。等你完全掌握了使用 XML 来编写界面的方法之后，不管是进行高复杂度的界面实现，还是分析和修改当前现有界面，对你来说都将是手到擒来。听我这么说，你可能会觉得 Eclipse 中的可视化编辑器完全就是多余的嘛！其实也不是，你还是可以使用它来进行界面预览的，毕竟你无法直接通过 XML 就看出界面的样子，而每修改一次界面就重新运行一遍程序显然又很耗时，这时你就可以好好地利用 Eclipse 的可视化编辑器了。

讲了这么多理论的东西，也是时候该学习一下到底如何编写程序界面了，我们就从

Android 中几种常见的控件开始吧。

3.2 常见控件的使用方法

Android 给我们提供了大量的 UI 控件，合理地使用这些控件就可以非常轻松地编写出相当不错的界面，下面我们就挑选几种常用的控件，详细介绍一下它们的使用方法。

首先新建一个 UIWidgetTest 项目，简单起见，我们还是允许 ADT 自动创建活动，活动名和布局名都使用默认值。别忘了将其他不相关的项目都关闭掉，始终养成这样一个良好的习惯。

3.2.1 TextView

TextView 可以说是 Android 中最简单的一个控件了，你在前面其实也已经和它打过了一些打交道。它主要用于在界面上显示一段文本信息，比如你在第一章看到的 Hello world！下面我们就来看一看关于 TextView 的更多用法。

将 activity_main.xml 中的代码改成如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="This is TextView" />

</LinearLayout>
```

外面的 LinearLayout 先忽略不看，在 TextView 中我们使用 android:id 给当前控件定义了一个唯一标识符，这个属性在上一章中已经讲解过了。然后使用 android:layout_width 指定了控件的宽度，使用 android:layout_height 指定了控件的高度。Android 中所有的控件都具有这两个属性，可选值有三种 match_parent、fill_parent 和 wrap_content，其中 match_parent 和 fill_parent 的意义相同，现在官方更加推荐使用 match_parent。match_parent 表示让当前控件的大小和父布局的大小一样，也就是由父布局来决定当前控件的大小。wrap_content 表示让当前控件的大小能够刚好包含住里面的内容，也就是由控件内容决定当前控件的大小。所以上面的代码就表示让 TextView 的宽度和父布局一样宽，也就是手机屏幕的宽度，让 TextView

的高度足够包含住里面的内容就行。当然除了使用上述值，你也可以对控件的宽和高指定一个固定的大小，但是这样做有时会在不同手机屏幕的适配方面出现问题。接下来我们通过 android:text 指定了 TextView 中显示的文本内容，现在运行程序，效果如图 3.1 所示。

图 3.1

虽然指定的文本内容是正常显示了，不过我们好像没看出来 TextView 的宽度是和屏幕一样宽的。其实这是由于 TextView 中的文字默认是居左上角对齐的，虽然 TextView 的宽度充满了整个屏幕，可是从效果上完全看不出来。现在我们修改 TextView 的文字对齐方式，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/text_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
```

```
        android:text="This is TextView" />

</LinearLayout>
```

我们使用 android:gravity 来指定文字的对齐方式，可选值有 top、bottom、left、right、center 等，可以用“|”来同时指定多个值，这里我们指定的"center"，效果等同于"center_vertical|center_horizontal"，表示文字在垂直和水平方向都居中对齐。现在重新运行程序，效果如图 3.2 所示。

图 3.2

这也说明了，TextView 的宽度确实是和屏幕宽度一样的。

另外我们还可以对 TextView 中文字的大小和颜色进行修改，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/text_view"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:textSize="24sp"
        android:textColor="#00ff00"
        android:text="This is TextView" />

</LinearLayout>
```

通过 android:textSize 属性可以指定文字的大小，通过 android:textColor 属性可以指定文字的颜色。重新运行程序，效果如图 3.3 所示。

图 3.3

当然 TextView 中还有很多其他的属性，这里我就不再一一介绍了，需要用到的时候去查阅文档就可以了。

3.2.2 Button

Button 是程序用于和用户进行交互的一个重要控件，相信你对这个控件已经是非常熟悉了，因为我们在上一章用了太多次 Button。它可配置的属性和 TextView 是差不多的，我们

可以在 activity_main.xml 中这样加入 Button：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ……

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button" />
</LinearLayout>
```

加入 Button 之后的界面如图 3.4 所示。

图　3.4

然后我们可以在 MainActivity 中为 Button 的点击事件注册一个监听器，如下所示：

```
public class MainActivity extends Activity {
```

```
    private Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 在此处添加逻辑
            }
        });
    }

}
```

这样每当点击按钮时，就会执行监听器中的 onClick()方法，我们只需要在这个方法中加入待处理的逻辑就行了。如果你不喜欢使用匿名类的方式来注册监听器，也可以使用实现接口的方式来进行注册，代码如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button button;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.button);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            // 在此处添加逻辑
            break;
        default:
```

```
                break;
            }
        }

    }
```

这两种写法都可以实现对按钮点击事件的监听，至于使用哪一种就全凭你喜好了。

3.2.3 EditText

EditText 是程序用于和用户进行交互的另一个重要控件，它允许用户在控件里输入和编辑内容，并可以在程序中对这些内容进行处理。EditText 的应用场景应该算是非常普遍了，发短信、发微博、聊 QQ 等等，在进行这些操作时，你不得不使用到 EditText。那我们来看一看如何在界面上加入 EditText 吧，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

其实看到这里，我估计你已经总结出 Android 控件的使用规律了，基本上用法都很相似，给控件定义一个 id，再指定下控件的宽度和高度，然后再适当加入些控件特有的属性就差不多了。所以使用 XML 来编写界面其实一点都不难，完全可以不用借助任何可视化工具来实现。现在重新运行一下程序，EditText 就已经在界面上显示出来了，并且我们是可以在里面输入内容的，如图 3.5 所示。

图 3.5

细心的你平时应该会留意到，一些做得比较人性化的软件会在输入框里显示一些提示性的文字，然后一旦用户输入了任何内容，这些提示性的文字就会消失。这种提示功能在 Android 里是非常容易实现的，我们甚至不需要做任何的逻辑控制，因为系统已经帮我们都处理好了。修改 activity_main.xml，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

        ......

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type something here"
        />
</LinearLayout>
```

这里使用 android:hint 属性来指定了一段提示性的文本，然后重新运行程序，效果如图 3.6 所示。

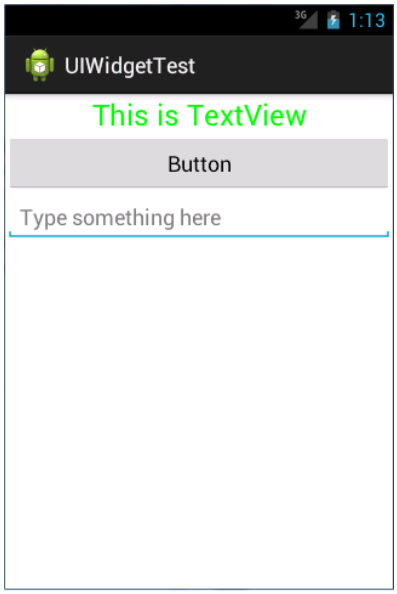

图 3.6

可以看到，EditText 中显示了一段提示性文本，然后当我们输入任何内容时，这段文本就会自动消失。

不过随着输入的内容不断增多，EditText 会被不断地拉长。这时由于 EditText 的高度指定的是 wrap_content，因此它总能包含住里面的内容，但是当输入的内容过多时，界面就会变得非常难看。我们可以使用 android:maxLines 属性来解决这个问题，修改 activity_main.xml，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <EditText
        android:id="@+id/edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:hint="Type something here"
        android:maxLines="2"
        />
</LinearLayout>
```

这里通过 android:maxLines 指定了 EditText 的最大行数为两行，这样当输入的内容超过两行时，文本就会向上滚动，而 EditText 则不会再继续拉伸，如图 3.7 所示。

图 3.7

我们还可以结合使用 EditText 与 Button 来完成一些功能，比如通过点击按钮来获取 EditText 中输入的内容。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button button;

    private EditText editText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.button);
```

```
        editText = (EditText) findViewById(R.id.edit_text);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            String inputText = editText.getText().toString();
            Toast.makeText(MainActivity.this, inputText,
Toast.LENGTH_SHORT).show();
            break;
        default:
            break;
        }
    }

}
```

首先通过 findViewById()方法得到 EditText 的实例，然后在按钮的点击事件里调用 EditText 的 getText()方法获取到输入的内容，再调用 toString()方法转换成字符串，最后仍然还是老方法，使用 Toast 将输入的内容显示出来。

重新运行程序，在 EditText 中输入一段内容，然后点击按钮，效果如图 3.8 所示。

图 3.8

3.2.4 ImageView

ImageView 是用于在界面上展示图片的一个控件，通过它可以让我们的程序界面变得更加丰富多彩。学习这个控件需要提前准备好一些图片，由于目前 drawable 文件夹下已经有一张 ic_launcher.png 图片了，那我们就先在界面上展示这张图吧，修改 activity_main.xml，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ……

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher"
        />
</LinearLayout>
```

可以看到，这里使用 android:src 属性给 ImageView 指定了一张图片，并且由于图片的宽和高都是未知的，所以将 ImageView 的宽和高都设定为 wrap_content，这样保证了不管图片的尺寸是多少都可以完整地展示出来。重新运行程序，效果如图 3.9 所示。

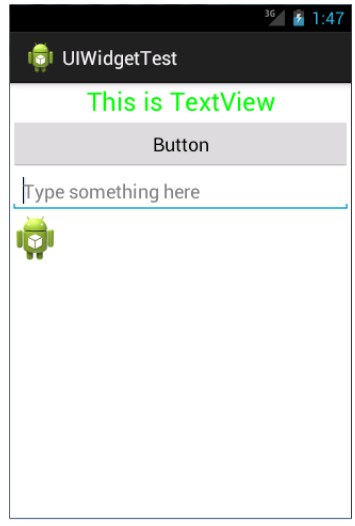

图 3.9

我们还可以在程序中通过代码动态地更改 ImageView 中的图片。这里我准备了另外一张图片，jelly_bean.png，将它复制到 res/drawable-hdpi 目录下，然后修改 MainActivity 的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button button;

    private EditText editText;

    private ImageView imageView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.button);
        editText = (EditText) findViewById(R.id.edit_text);
        imageView  = (ImageView) findViewById(R.id.image_view);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            imageView.setImageResource(R.drawable.jelly_bean);
            break;
        default:
            break;
        }
    }

}
```

在按钮的点击事件里，通过调用 ImageView 的 setImageResource()方法将显示的图片改成 jelly_bean，现在重新运行程序，然后点击一下按钮，就可以看到 ImageView 中显示的图片改变了，如图 3.10 所示。

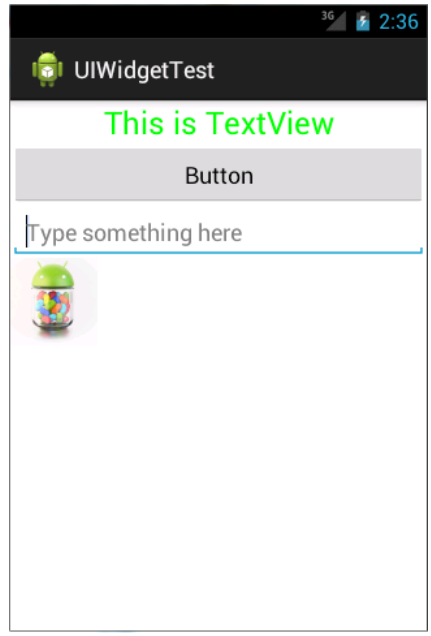

图 3.10

3.2.5 ProgressBar

ProgressBar 用于在界面上显示一个进度条，表示我们的程序正在加载一些数据。它的用法也非常简单，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <ProgressBar
        android:id="@+id/progress_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        />
</LinearLayout>
```

重新运行程序，会看到屏幕中有一个圆形进度条正在旋转，如图 3.11 所示。

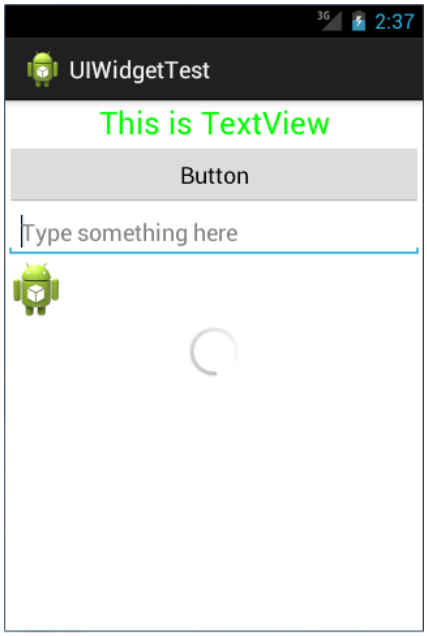

图 3.11

这时你可能会问，旋转的进度条表明我们的程序正在加载数据，那数据总会有加载完的时候吧，如何才能让进度条在数据加载完成时消失呢？这里我们就需要用到一个新的知识点，Android 控件的可见属性。所有的 Android 控件都具有这个属性，可以通过 android:visibility 进行指定，可选值有三种，visible、invisible 和 gone。visible 表示控件是可见的，这个值是默认值，不指定 android:visibility 时，控件都是可见的。invisible 表示控件不可见，但是它仍然占据着原来的位置和大小，可以理解成控件变成透明状态了。gone 则表示控件不仅不可见，而且不再占用任何屏幕空间。我们还可以通过代码来设置控件的可见性，使用的是 setVisibility()方法，可以传入 View.VISIBLE、View.INVISIBLE 和 View.GONE 三种值。

接下来我们就来尝试实现，点击一下按钮让进度条消失，再点击一下按钮让进度条出现的这种效果。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button button;

    private EditText editText;

    private ImageView imageView;
```

```
    private ProgressBar progressBar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        button = (Button) findViewById(R.id.button);
        editText = (EditText) findViewById(R.id.edit_text);
        imageView  = (ImageView) findViewById(R.id.image_view);
        progressBar = (ProgressBar) findViewById(R.id.progress_bar);
        button.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            if (progressBar.getVisibility() == View.GONE) {
                progressBar.setVisibility(View.VISIBLE);
            } else {
                progressBar.setVisibility(View.GONE);
            }
            break;
        default:
            break;
        }
    }

}
```

在按钮的点击事件中，我们通过 getVisibility()方法来判断 ProgressBar 是否可见，如果可见就将 ProgressBar 隐藏掉，如果不可见就将 ProgressBar 显示出来。重新运行程序，然后不断地点击按钮，你就会看到进度条会在显示与隐藏之间来回切换。

另外，我们还可以给 ProgressBar 指定不同的样式，刚刚是圆形进度条，通过 style 属性可以将它指定成水平进度条，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

```
    ……

    <ProgressBar
        android:id="@+id/progress_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        style="?android:attr/progressBarStyleHorizontal"
        android:max="100"
        />
</LinearLayout>
```

指定成水平进度条后，我们还可以通过 android:max 属性给进度条设置一个最大值，然后在代码中动态地更改进度条的进度。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            int progress = progressBar.getProgress();
            progress = progress + 10;
            progressBar.setProgress(progress);
            break;
        default:
            break;
        }
    }
}
```

每点击一次按钮，我们就获取进度条的当前进度，然后在现有的进度上加 10 作为更新后的进度。重新运行程序，点击数次按钮后，效果如图 3.12 所示。

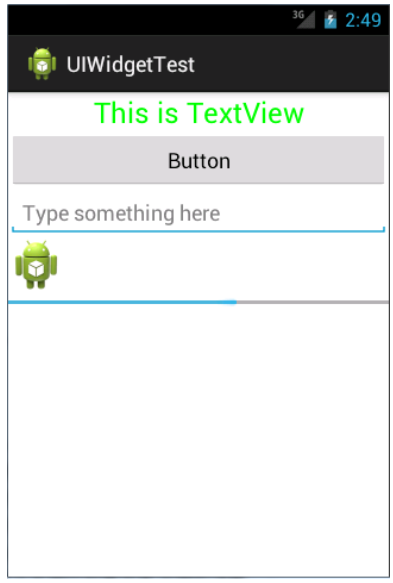

图 3.12

ProgressBar 还有几种其他的样式，你可以自己去尝试一下。

3.2.6 AlertDialog

AlertDialog 可以在当前的界面弹出一个对话框，这个对话框是置顶于所有界面元素之上的，能够屏蔽掉其他控件的交互能力，因此一般 AlertDialog 都是用于提示一些非常重要的内容或者警告信息。比如为了防止用户误删重要内容，在删除前弹出一个确认对话框。下面我们来学习一下它的用法，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ......
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            AlertDialog.Builder dialog = new AlertDialog.Builder
(MainActivity.this);
            dialog.setTitle("This is Dialog");
            dialog.setMessage("Something important.");
            dialog.setCancelable(false);
            dialog.setPositiveButton("OK", new DialogInterface.
OnClickListener() {
```

```
                @Override
                public void onClick(DialogInterface dialog, int which) {
                }
            });
            dialog.setNegativeButton("Cancel", new DialogInterface.
OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                }
            });
            dialog.show();
            break;
        default:
            break;
        }
    }
}
```

首先通过 AlertDialog.Builder 创建出一个 AlertDialog 的实例，然后可以为这个对话框设置标题、内容、可否取消等属性，接下来调用 setPositiveButton()方法为对话框设置确定按钮的点击事件，调用 setNegativeButton()方法设置取消按钮的点击事件，最后调用 show()方法将对话框显示出来。重新运行程序，点击按钮后，效果如图 3.13 所示。

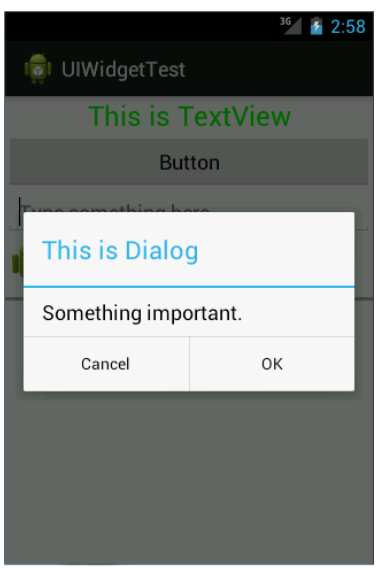

图 3.13

3.2.7 ProgressDialog

ProgressDialog 和 AlertDialog 有点类似，都可以在界面上弹出一个对话框，都能够屏蔽掉其他控件的交互能力。不同的是，ProgressDialog 会在对话框中显示一个进度条，一般是用于表示当前操作比较耗时，让用户耐心地等待。它的用法和 AlertDialog 也比较相似，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ......
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.button:
            ProgressDialog progressDialog = new ProgressDialog
(MainActivity.this);
            progressDialog.setTitle("This is ProgressDialog");
            progressDialog.setMessage("Loading...");
            progressDialog.setCancelable(true);
            progressDialog.show();
            break;
        default:
            break;
        }
    }
}
```

可以看到，这里也是先构建出一个 ProgressDialog 对象，然后同样可以设置标题、内容、可否取消等属性，最后也是通过调用 show()方法将 ProgressDialog 显示出来。重新运行程序，点击按钮后，效果如图 3.14 所示。

注意如果在 setCancelable()中传入了 false，表示 ProgressDialog 是不能通过 Back 键取消掉的，这时你就一定要在代码中做好控制，当数据加载完成后必须要调用 ProgressDialog 的 dismiss()方法来关闭对话框，否则 ProgressDialog 将会一直存在。

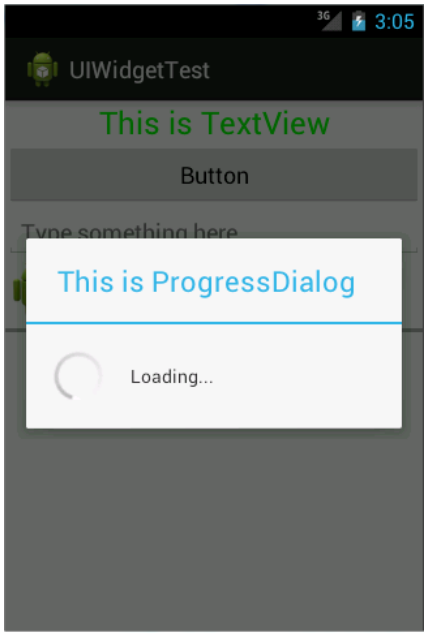

图 3.14

好了，关于 Android 控件的使用，我要讲的就只有这么多。一节内容就想覆盖 Android 控件所有的相关知识不太现实，同样一口气就想学会所有 Android 控件的使用方法也不太现实。本节所讲的内容对于你来说只是起到了一个引导的作用，你还需要在以后的学习和工作中不断地摸索，通过查阅文档以及网上搜索的方式学习更多控件的更多用法。当然，当本书后面有涉及到一些我们前面没学过的控件和相关用法时，我仍然会在相应的章节做详细的讲解。

3.3 详解四种基本布局

一个丰富的界面总是要由很多个控件组成的，那我们如何才能让各个控件都有条不紊地摆放在界面上，而不是乱糟糟的呢？这就需要借助布局来实现了。布局是一种可用于放置很多控件的容器，它可以按照一定的规律调整内部控件的位置，从而编写出精美的界面。当然，布局的内部除了放置控件外，也可以放置布局，通过多层布局的嵌套，我们就能够完成一些比较复杂的界面实现，示意图 3.15 很好地展示了它们之间的关系。

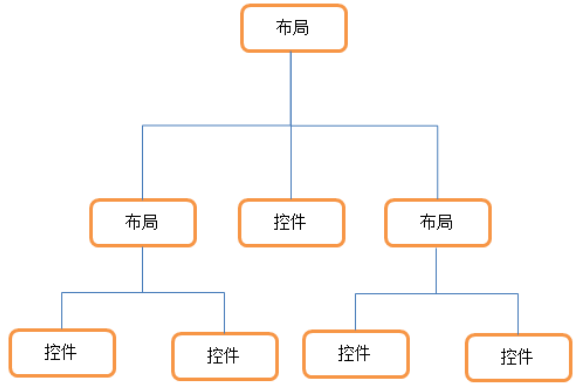

图 3.15

下面我们来详细讲解下 Android 中四种最基本的布局。先做好准备工作，新建一个 UILayoutTest 项目，并让 ADT 自动帮我们创建好活动，活动名和布局名都使用默认值。

3.3.1 LinearLayout

LinearLayout 又称作线性布局，是一种非常常用的布局。正如它名字所描述的一样，这个布局会将它所包含的控件在线性方向上依次排列。相信你之前也已经注意到了，我们在上一节中学习控件用法时，所有的控件就都是放在 LinearLayout 布局里的，因此上一节中的控件也确实是在垂直方向上线性排列的。

既然是线性排列，肯定就不仅只有一个方向，那为什么上一节中的控件都是在垂直方向排列的呢？这是由于我们通过 android:orientation 属性指定了排列方向是 vertical，如果指定的是 horizontal，控件就会在水平方向上排列了。下面我们通过实战来体会一下，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 1" />

    <Button
```

```
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button 3" />

</LinearLayout>
```

我们在 LinearLayout 中添加了三个 Button，每个 Button 的长和宽都是 wrap_content，并指定了排列方向是 vertical。现在运行一下程序，效果如图 3.16 所示。

图 3.16

然后我们修改一下 LinearLayout 的排列方向，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    ......
```

```
</LinearLayout>
```

将 android:orientation 属性的值改成了 horizontal，这就意味着要让 LinearLayout 中的控件在水平方向上依次排列，当然如果不指定 android:orientation 属性的值，默认的排列方向就是 horizontal。重新运行一下程序，效果如图 3.17 所示。

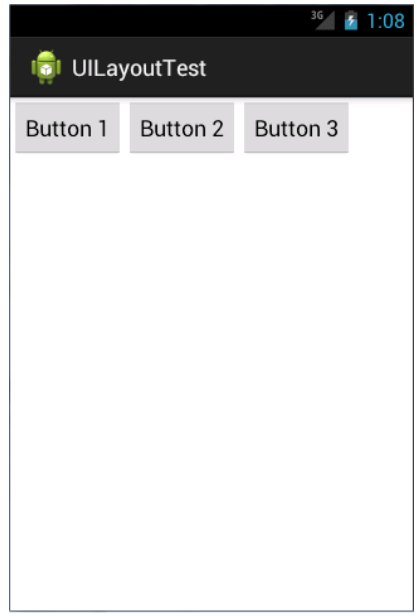

图 3.17

这里需要注意，如果 LinearLayout 的排列方向是 horizontal，内部的控件就绝对不能将宽度指定为 match_parent，因为这样的话单独一个控件就会将整个水平方向占满，其他的控件就没有可放置的位置了。同样的道理，如果 LinearLayout 的排列方向是 vertical，内部的控件就不能将高度指定为 match_parent。

了解了 LinearLayout 的排列规律，我们再来学习一下它的几个关键属性的用法吧。

首先来看 android:layout_gravity 属性，它和我们上一节中学到的 android:gravity 属性看起来有些相似，这两个属性有什么区别呢？其实从名字上就可以看出，android:gravity 是用于指定文字在控件中的对齐方式，而 android:layout_gravity 是用于指定控件在布局中的对齐方式。android:layout_gravity 的可选值和 android:gravity 差不多，但是需要注意，当 LinearLayout 的排列方向是 horizontal 时，只有垂直方向上的对齐方式才会生效，因为此时水平方向上的长度是不固定的，每添加一个控件，水平方向上的长度都会改变，因而无法指定该方向上的对齐方式。同样的道理，当 LinearLayout 的排列方向是 vertical 时，只有水平方

向上的对齐方式才会生效。修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"
        android:text="Button 3" />

</LinearLayout>
```

由于目前 LinearLayout 的排列方向是 horizontal，因此我们只能指定垂直方向上的排列方向，将第一个 Button 的对齐方式指定为 top，第二个 Button 的对齐方式指定为 center_vertical，第三个 Button 的对齐方式指定为 bottom。重新运行程序，效果如图 3.18 所示。

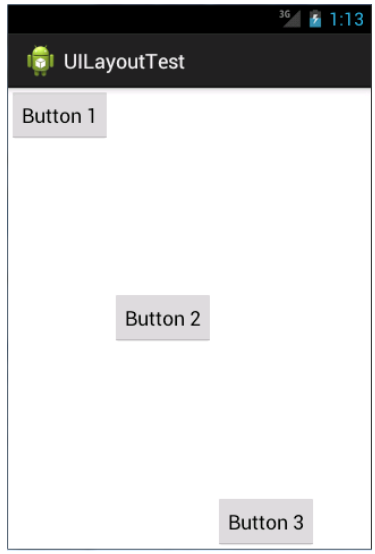

图 3.18

接下来我们学习下 LinearLayout 中的另一个重要属性，android:layout_weight。这个属性允许我们使用比例的方式来指定控件的大小，它在手机屏幕的适配性方面可以起到非常重要的作用。比如我们正在编写一个消息发送界面，需要一个文本编辑框和一个发送按钮，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <EditText
        android:id="@+id/input_message"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:hint="Type something"
        />

    <Button
        android:id="@+id/send"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

```
        android:layout_weight="1"
        android:text="Send"
        />

</LinearLayout>
```

你会发现，这里竟然将 EditText 和 Button 的宽度都指定成了 0，这样文本编辑框和按钮还能显示出来吗？不用担心，由于我们使用了 android:layout_weight 属性，此时控件的宽度就不应该再由 android:layout_width 来决定，这里指定成 0 是一种比较规范的写法。

然后我们在 EditText 和 Button 里都将 android:layout_weight 属性的值指定为 1，这表示 EditText 和 Button 将在水平方向平分宽度，重新运行下程序，你会看到如图 3.19 所示的效果。

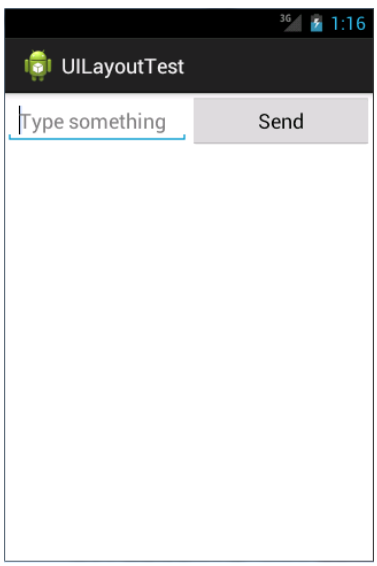

图 3.19

为什么将 android:layout_weight 属性的值同时指定为 1 就会平分屏幕宽度呢？其实原理也很简单，系统会先把 LinearLayout 下所有控件指定的 layout_weight 值相加，得到一个总值，然后每个控件所占大小的比例就是用该控件的 layout_weight 值除以刚才算出的总值。因此如果想让 EditText 占据屏幕宽度的 3/5，Button 占据屏幕宽度的 2/5，只需要将 EditText 的 layout_weight 改成 3，Button 的 layout_weight 改成 2 就可以了。

我们还可以通过指定部分控件的 layout_weight 值，来实现更好的效果。修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:orientation="horizontal" >

    <EditText
        android:id="@+id/input_message"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:hint="Type something"
        />

    <Button
        android:id="@+id/send"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Send"
        />

</LinearLayout>
```

这里我们仅指定了 EditText 的 android:layout_weight 属性，并将 Button 的宽度改回 wrap_content。这表示 Button 的宽度仍然按照 wrap_content 来计算，而 EditText 则会占满屏幕所有的剩余空间。使用这种方式编写的界面，不仅在各种屏幕的适配方面会非常好，而且看起来也更加舒服，重新运行程序，效果如图 3.20 所示。

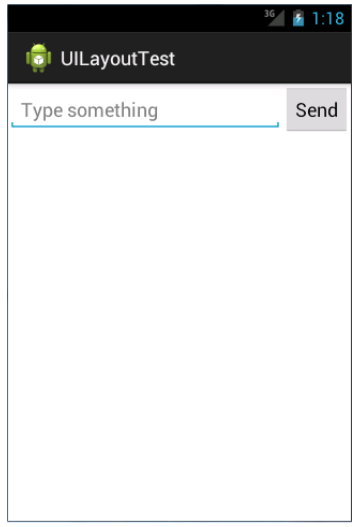

图 3.20

3.3.2 RelativeLayout

RelativeLayout 又称作相对布局，也是一种非常常用的布局。和 LinearLayout 的排列规则不同，RelativeLayout 显得更加随意一些，它可以通过相对定位的方式让控件出现在布局的任何位置。也正因为如此，RelativeLayout 中的属性非常多，不过这些属性都是有规律可循的，其实并不难理解和记忆。我们还是通过实践来体会一下，修改 activity_main.xml 中的代码，如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Button 3" />

    <Button
        android:id="@+id/button4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
```

```
        android:layout_alignParentLeft="true"
        android:text="Button 4" />

    <Button
        android:id="@+id/button5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:text="Button 5" />

</RelativeLayout>
```

我想以上代码已经不需要我再做过多解释了，因为实在是太好理解了，我们让 Button 1 和父布局的左上角对齐，Button 2 和父布局的右上角对齐，Button 3 居中显示，Button 4 和父布局的左下角对齐，Button 5 和父布局的右下角对齐。虽然 android:layout_alignParentLeft、android:layout_alignParentTop、android:layout_alignParentRight、android:layout_alignParentBottom、android:layout_centerInParent 这几个属性我们之前都没接触过，可是它们的名字已经完全说明了它们的作用。重新运行程序，效果如图 3.21 所示。

图 3.21

上面例子中的每个控件都是相对于父布局进行定位的，那控件可不可以相对于控件进行定位呢？当然是可以的，修改 activity_main.xml 中的代码，如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Button 3" />

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/button3"
        android:layout_toLeftOf="@id/button3"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/button3"
        android:layout_toRightOf="@id/button3"
        android:text="Button 2" />

    <Button
        android:id="@+id/button4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/button3"
        android:layout_toLeftOf="@id/button3"
        android:text="Button 4" />

    <Button
        android:id="@+id/button5"
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/button3"
        android:layout_toRightOf="@id/button3"
        android:text="Button 5" />

</RelativeLayout>
```

这次的代码稍微复杂一点，不过仍然是有规律可循的。android:layout_above 属性可以让一个控件位于另一个控件的上方，需要为这个属性指定相对控件 id 的引用，这里我们填入了@id/button3，表示让该控件位于 Button 3 的上方。其他的属性也都是相似的，android:layout_below 表示让一个控件位于另一个控件的下方，android:layout_toLeftOf 表示让一个控件位于另一个控件的左侧，android:layout_toRightOf 表示让一个控件位于另一个控件的右侧。注意，当一个控件去引用另一个控件的 id 时，该控件一定要定义在引用控件的后面，不然会出现找不到 id 的情况。重新运行程序，效果如图 3.22 所示。

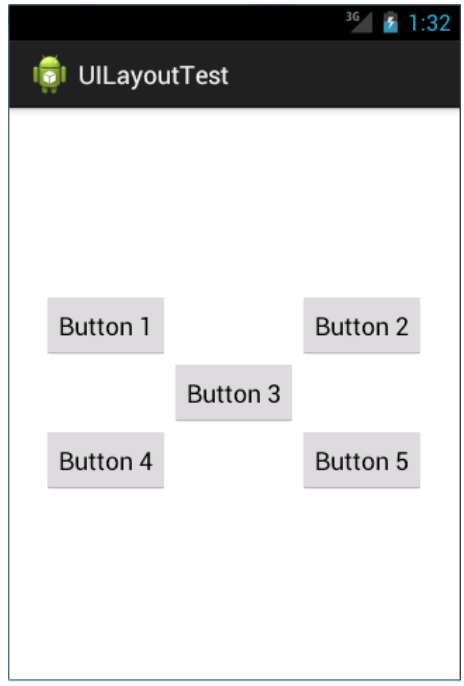

图 3.22

RelativeLayout 中还有另外一组相对于控件进行定位的属性，android:layout_alignLeft 表示让一个控件的左边缘和另一个控件的左边缘对齐，android:layout_alignRight 表示让一个控

件的右边缘和另一个控件的右边缘对齐，还有 android:layout_alignTop 和 android:layout_alignBottom，道理都是一样的，我就不再多说，这几个属性就留给你自己去尝试一下了。

好了，正如我前面所说，RelativeLayout 中的属性虽然多，但都是有规律可循的，所以学起来一点都不觉得吃力吧？

3.3.3 FrameLayout

FrameLayout 相比于前面两种布局就简单太多了，因此它的应用场景也少了很多。这种布局没有任何的定位方式，所有的控件都会摆放在布局的左上角。让我们通过例子来看一看吧，修改 activity_main.xml 中的代码，如下所示：

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        />

    <ImageView
        android:id="@+id/image_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher"
        />

</FrameLayout>
```

FrameLayout 中只是放置了一个按钮和一张图片，重新运行程序，效果如图 3.23 所示。

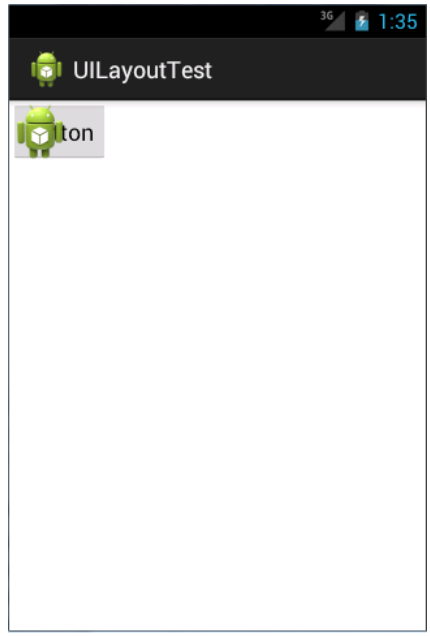

图 3.23

可以看到，按钮和图片都是位于布局的左上角。由于图片是在按钮之后添加的，因此图片压在了按钮的上面。

你可能会觉得，这个布局能有什么作用呢？确实，它的应用场景并不多，不过在下一章中介绍碎片的时候，我们还是可以用到它的。

3.3.4 TableLayout

TableLayout 允许我们使用表格的方式来排列控件，这种布局也不是很常用，你只需要了解一下它的基本用法就可以了。既然是表格，那就一定会有行和列，在设计表格时我们尽量应该让每一行都拥有相同的列数，这样的表格也是最简单的。不过有时候事情并非总会顺从我们的心意，当表格的某行一定要有不相等的列数时，就需要通过合并单元格的方式来应对。

比如我们正在设计一个登录界面，允许用户输入账号密码后登录，就可以将 activity_main.xml 中的代码改成如下所示：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
```

```
    <TableRow>

        <TextView
            android:layout_height="wrap_content"
            android:text="Account:" />

        <EditText
            android:id="@+id/account"
            android:layout_height="wrap_content"
            android:hint="Input your account" />
    </TableRow>

    <TableRow>

        <TextView
            android:layout_height="wrap_content"
            android:text="Password:" />

        <EditText
            android:id="@+id/password"
            android:layout_height="wrap_content"
            android:inputType="textPassword" />
    </TableRow>

    <TableRow>

        <Button
            android:id="@+id/login"
            android:layout_height="wrap_content"
            android:layout_span="2"
            android:text="Login" />
    </TableRow>

</TableLayout>
```

在 TableLayout 中每加入一个 TableRow 就表示在表格中添加了一行，然后在 TableRow 中每加入一个控件，就表示在该行中加入了一列，TableRow 中的控件是不能指定宽度的。这里我们将表格设计成了三行两列的格式，第一行有一个 TextView 和一个用于输入账号的 EditText，第二行也有一个 TextView 和一个用于输入密码的 EditText，我们通过将 android:inputType 属性的值指定为 textPassword，把 EditText 变为密码输入框。可是第三行只

有一个用于登录的按钮，前两行都有两列，第三行只有一列，这样的表格就会很难看，而且结构也非常不合理。这时就需要通过对单元格进行合并来解决这个问题，使用android:layout_span="2"让登录按钮占据两列的空间，就可以保证表格结构的合理性了。重新运行程序，效果如图 3.24 所示。

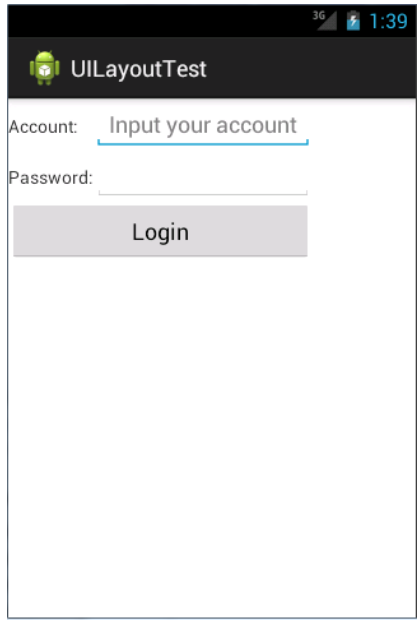

图 3.24

不过从图中可以看出，当前的登录界面并没有充分利用屏幕的宽度，右侧还空出了一块区域，这也难怪，因为在 TableRow 中我们无法指定控件的宽度。这时使用android:stretchColumns 属性就可以很好地解决这个问题，它允许将 TableLayout 中的某一列进行拉伸，以达到自动适应屏幕宽度的作用。修改 activity_main.xml 中的代码，如下所示：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
    >

    ......

</TableLayout>
```

这里将 android:stretchColumns 的值指定为 1，表示如果表格不能完全占满屏幕宽度，就

将第二列进行拉伸。没错！指定成 1 就是拉伸第二列，指定成 0 就是拉伸第一列，不要以为这里我写错了哦。重新运行程序，效果如图 3.25 所示。

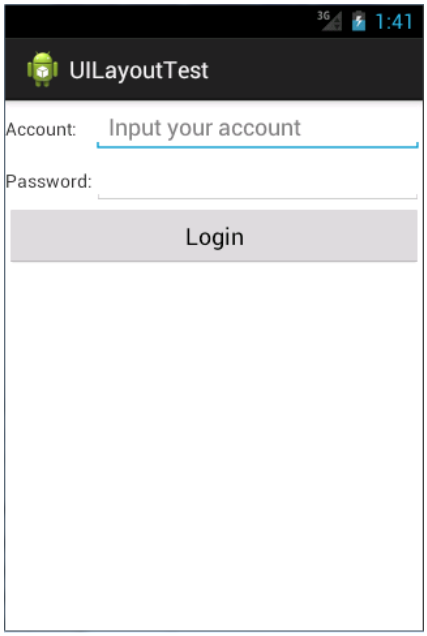

图　3.25

好了，关于布局我也就只准备讲这么多了，Android 中其实还有一个 AbsoluteLayout，不过这个布局官方已经不推荐使用了，因此我们直接将它忽略就好。

3.4.1　引入布局

如果你用过 iPhone 应该会知道，几乎每一个 iPhone 应用的界面顶部都会有一个标题栏，标题栏上会有一到两个按钮可用于返回或其他操作（iPhone 没有实体返回键）。现在很多的 Android 程序也都喜欢模仿 iPhone 的风格，在界面的顶部放置一个标题栏。虽然 Android 系统已经给每个活动提供了标题栏功能，但这里我们仍然决定不使用它，而是创建一个自定义的标题栏。

经过前面两节的学习，我想创建一个标题栏布局对你来说已经不是什么困难的事情了，只需要加入两个 Button 和一个 TextView，然后在布局中摆放好就可以了。可是这样做却存在着一个问题，一般我们的程序中可能有很多个活动都需要这样的标题栏，如果在每个活动的布局中都编写一遍同样的标题栏代码，明显就会导致代码的大量重复。这个时候我们就可以使用引入布局的方式来解决这个问题，新建一个布局 title.xml，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@drawable/title_bg" >

    <Button
        android:id="@+id/title_back"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="5dip"
        android:background="@drawable/back_bg"
        android:text="Back"
        android:textColor="#fff" />

    <TextView
        android:id="@+id/title_text"
        android:layout_width="0dip"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="1"
        android:gravity="center"
        android:text="Title Text"
        android:textColor="#fff"
        android:textSize="24sp" />

    <Button
        android:id="@+id/title_edit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="5dip"
        android:background="@drawable/edit_bg"
        android:text="Edit"
        android:textColor="#fff" />

</LinearLayout>
```

可以看到，我们在 LinearLayout 中分别加入了两个 Button 和一个 TextView，左边的 Button 可用于返回，右边的 Button 可用于编辑，中间的 TextView 则可以显示一段标题文本。上面

的代码中大多数的属性你都已经是见过的，下面我来说明一下几个之前没有讲过的属性。android:background 用于为布局或控件指定一个背景，可以使用颜色或图片来进行填充，这里我提前准备好了三张图片，title_bg.png、back_bg.png 和 edit_bg.png，分别用于作为标题栏、返回按钮和编辑按钮的背景。另外在两个 Button 中我们都使用了 android:layout_margin 这个属性，它可以指定控件在上下左右方向上偏移的距离，当然也可以使用 android:layout_marginLeft 或 android:layout_marginTop 等属性来单独指定控件在某个方向上偏移的距离。

现在标题栏布局已经编写完成了，剩下的就是如何在程序中使用这个标题栏了，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <include layout="@layout/title" />

</LinearLayout>
```

没错！我们只需要通过一行 include 语句将标题栏布局引入进来就可以了。

最后别忘了在 MainActivity 中将系统自带的标题栏隐藏掉，代码如下所示：

```
public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        setContentView(R.layout.activity_main);
    }

}
```

现在运行一下程序，效果如图 3.27 所示。

图 3.27

使用这种方式，不管有多少布局需要添加标题栏，只需一行 include 语句就可以了。

3.4.2 创建自定义控件

引入布局的技巧确实解决了重复编写布局代码的问题，但是如果布局中有一些控件要求能够响应事件，我们还是需要在每个活动中为这些控件单独编写一次事件注册的代码。比如说标题栏中的返回按钮，其实不管是在哪一个活动中，这个按钮的功能都是相同的，即销毁掉当前活动。而如果在每一个活动中都需要重新注册一遍返回按钮的点击事件，无疑又是增加了很多重复代码，这种情况最好是使用自定义控件的方式来解决。

新建 TitleLayout 继承自 LinearLayout，让它成为我们自定义的标题栏控件，代码如下所示：

```
public class TitleLayout extends LinearLayout {

    public TitleLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
        LayoutInflater.from(context).inflate(R.layout.title, this);
    }

}
```

首先我们重写了 LinearLayout 中的带有两个参数的构造函数，在布局中引入 TitleLayout 控件就会调用这个构造函数。然后在构造函数中需要对标题栏布局进行动态加载，这就要借助 LayoutInflater 来实现了。通过 LayoutInflater 的 from()方法可以构建出一个 LayoutInflater 对象，然后调用 inflate()方法就可以动态加载一个布局文件，inflate()方法接收两个参数，第一个参数是要加载的布局文件的 id，这里我们传入 R.layout.title，第二个参数是给加载好的布局再添加一个父布局，这里我们想要指定为 TitleLayout，于是直接传入 this。

现在自定义控件已经创建好了，然后我们需要在布局文件中添加这个自定义控件，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <com.example.uicustomviews.TitleLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        ></com.example.uicustomviews.TitleLayout>

</LinearLayout>
```

添加自定义控件和添加普通控件的方式基本是一样的，只不过在添加自定义控件的时候我们需要指明控件的完整类名，包名在这里是不可以省略的。

重新运行程序，你会发现此时效果和使用引入布局方式的效果是一样的。

然后我们来尝试为标题栏中的按钮注册点击事件，修改 TitleLayout 中的代码，如下所示：

```
public class TitleLayout extends LinearLayout {

    public TitleLayout(Context context, AttributeSet attrs) {
        super(context, attrs);
        LayoutInflater.from(context).inflate(R.layout.title, this);
        Button titleBack = (Button) findViewById(R.id.title_back);
        Button titleEdit = (Button) findViewById(R.id.title_edit);
        titleBack.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                ((Activity) getContext()).finish();
            }
        });
        titleEdit.setOnClickListener(new OnClickListener() {
            @Override
```

```
            public void onClick(View v) {
                Toast.makeText(getContext(), "You clicked Edit button",
                        Toast.LENGTH_SHORT).show();
            }
        });
    }

}
```

首先还是通过 findViewById()方法得到按钮的实例，然后分别调用 setOnClickListener()方法给两个按钮注册了点击事件，当点击返回按钮时销毁掉当前的活动，当点击编辑按钮时弹出一段文本。重新运行程序，点击一下编辑按钮，效果如图 3.28 所示。

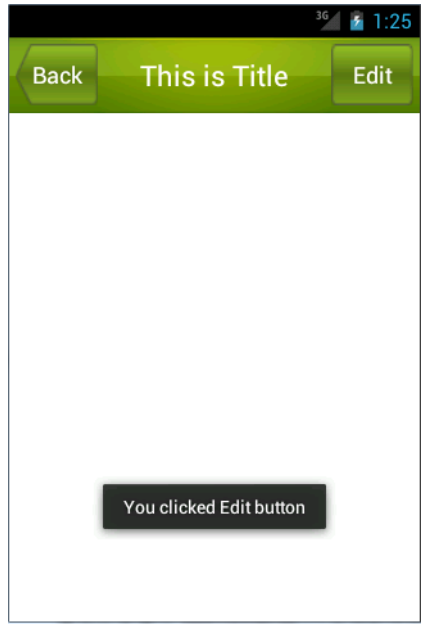

图 3.28

这样的话，每当我们在一个布局中引入 TitleLayout，返回按钮和编辑按钮的点击事件就已经自动实现好了，也是省去了很多编写重复代码的工作。

3.5 最常用和最难用的控件——ListView

ListView 绝对可以称得上是 Android 中最常用的控件之一，几乎所有的应用程序都会用到它。由于手机屏幕空间都比较有限，能够一次性在屏幕上显示的内容并不多，当我们的程

序中有大量的数据需要展示的时候，就可以借助 ListView 来实现。ListView 允许用户通过手指上下滑动的方式将屏幕外的数据滚动到屏幕内，同时屏幕上原有的数据则会滚动出屏幕。相信你其实每天都在使用这个控件，比如查看手机联系人列表，翻阅微博的最新消息等等。

不过比起前面介绍的几种控件，ListView 的用法也相对复杂了很多，因此我们就单独使用一节内容来对 ListView 进行非常详细的讲解。

3.5.1 ListView 的简单用法

首先新建一个 ListViewTest 项目，并让 ADT 自动帮我们创建好活动。然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ListView
        android:id="@+id/list_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </ListView>

</LinearLayout>
```

在布局中加入 ListView 控件还算非常简单，先为 ListView 指定了一个 id，然后将宽度和高度都设置为 match_parent，这样 ListView 也就占据了整个布局的空间。

接下来修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private String[] data = { "Apple", "Banana", "Orange", "Watermelon",
            "Pear", "Grape", "Pineapple", "Strawberry", "Cherry", "Mango" };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(
                MainActivity.this, android.R.layout.simple_list_item_1, data);
        ListView listView = (ListView) findViewById(R.id.list_view);
        listView.setAdapter(adapter);
    }
```

```
}
```

既然 ListView 是用于展示大量数据的，那我们就应该先将数据提供好。这些数据可以是从网上下载的，也可以是从数据库中读取的，应该视具体的应用程序场景来决定。这里我们就简单使用了一个 data 数组来测试，里面包含了很多水果的名称。

不过，数组中的数据是无法直接传递给 ListView 的，我们还需要借助适配器来完成。Android 中提供了很多适配器的实现类，其中我认为最好用的就是 ArrayAdapter。它可以通过泛型来指定要适配的数据类型，然后在构造函数中把要适配的数据传入即可。ArrayAdapter 有多个构造函数的重载，你应该根据实际情况选择最合适的一种。这里由于我们提供的数据都是字符串，因此将 ArrayAdapter 的泛型指定为 String，然后在 ArrayAdapter 的构造函数中依次传入当前上下文、ListView 子项布局的 id，以及要适配的数据。注意我们使用了 android.R.layout.simple_list_item_1 作为 ListView 子项布局的 id，这是一个 Android 内置的布局文件，里面只有一个 TextView，可用于简单地显示一段文本。这样适配器对象就构建好了。

最后，还需要调用 ListView 的 setAdapter()方法，将构建好的适配器对象传递进去，这样 ListView 和数据之间的关联就建立完成了。

现在运行一下程序，效果如图 3.29 所示。

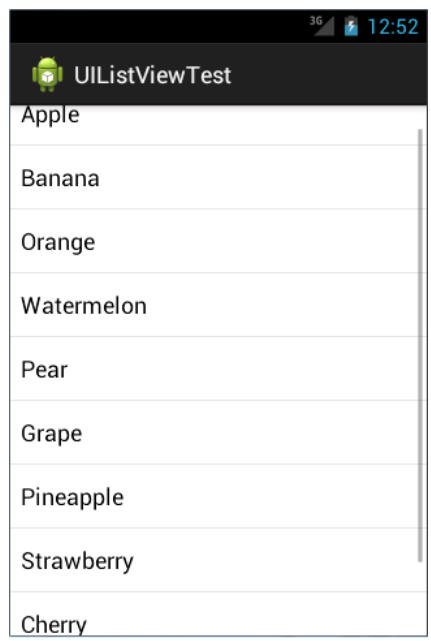

图 3.29

可以通过滚动的方式来查看屏幕外的数据。

3.5.2 定制 ListView 的界面

只能显示一段文本的 ListView 实在是太单调了，我们现在就来对 ListView 的界面进行定制，让它可以显示更加丰富的内容。

首先需要准备好一组图片，分别对应上面提供的每一种水果，待会我们要让这些水果名称的旁边都有一个图样。

接着定义一个实体类，作为 ListView 适配器的适配类型。新建类 Fruit，代码如下所示：

```
public class Fruit {

    private String name;

    private int imageId;

    public Fruit(String name, int imageId) {
        this.name = name;
        this.imageId = imageId;
    }

    public String getName() {
        return name;
    }

    public int getImageId() {
        return imageId;
    }

}
```

Fruit 类中只有两个字段，name 表示水果的名字，imageId 表示水果对应图片的资源 id。

然后需要为 ListView 的子项指定一个我们自定义的布局，在 layout 目录下新建 fruit_item.xml，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ImageView
        android:id="@+id/fruit_image"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
    <TextView
        android:id="@+id/fruit_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginLeft="10dip" />

</LinearLayout>
```

在这个布局中，我们定义了一个 ImageView 用于显示水果的图片，又定义了一个 TextView 用于显示水果的名称。

接下来需要创建一个自定义的适配器，这个适配器继承自 ArrayAdapter，并将泛型指定为 Fruit 类。新建类 FruitAdapter，代码如下所示：

```
public class FruitAdapter extends ArrayAdapter<Fruit> {

    private int resourceId;

    public FruitAdapter(Context context, int textViewResourceId,
            List<Fruit> objects) {
        super(context, textViewResourceId, objects);
        resourceId = textViewResourceId;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Fruit fruit = getItem(position); // 获取当前项的Fruit实例
        View view = LayoutInflater.from(getContext()).inflate(resourceId, null);
        ImageView fruitImage = (ImageView) view.findViewById(R.id.fruit_image);
        TextView fruitName = (TextView) view.findViewById(R.id.fruit_name);
        fruitImage.setImageResource(fruit.getImageId());
        fruitName.setText(fruit.getName());
        return view;
    }

}
```

FruitAdapter 重写了父类的一组构造函数，用于将上下文、ListView 子项布局的 id 和数据都传递进来。另外又重写了 getView()方法，这个方法在每个子项被滚动到屏幕内的时候会被调用。在 getView 方法中，首先通过 getItem()方法得到当前项的 Fruit 实例，然后使用

LayoutInflater 来为这个子项加载我们传入的布局，接着调用 View 的 findViewById()方法分别获取到 ImageView 和 TextView 的实例，并分别调用它们的 setImageResource()和 setText()方法来设置显示的图片和文字，最后将布局返回，这样我们自定义的适配器就完成了。

下面修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private List<Fruit> fruitList = new ArrayList<Fruit>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initFruits(); // 初始化水果数据
        FruitAdapter adapter = new FruitAdapter(MainActivity.this,
R.layout.fruit_item, fruitList);
        ListView listView = (ListView) findViewById(R.id.list_view);
        listView.setAdapter(adapter);
    }

    private void initFruits() {
        Fruit apple = new Fruit("Apple", R.drawable.apple_pic);
        fruitList.add(apple);
        Fruit banana = new Fruit("Banana", R.drawable.banana_pic);
        fruitList.add(banana);
        Fruit orange = new Fruit("Orange", R.drawable.orange_pic);
        fruitList.add(orange);
        Fruit watermelon = new Fruit("Watermelon", R.drawable.watermelon_pic);
        fruitList.add(watermelon);
        Fruit pear = new Fruit("Pear", R.drawable.pear_pic);
        fruitList.add(pear);
        Fruit grape = new Fruit("Grape", R.drawable.grape_pic);
        fruitList.add(grape);
        Fruit pineapple = new Fruit("Pineapple", R.drawable.pineapple_pic);
        fruitList.add(pineapple);
        Fruit strawberry = new Fruit("Strawberry", R.drawable.strawberry_pic);
        fruitList.add(strawberry);
        Fruit cherry = new Fruit("Cherry", R.drawable.cherry_pic);
        fruitList.add(cherry);
        Fruit mango = new Fruit("Mango", R.drawable.mango_pic);
```

```
            fruitList.add(mango);
        }

    }
```

可以看到，这里添加了一个 initFruits()方法，用于初始化所有的水果数据。在 Fruit 类的构造函数中将水果的名字和对应的图片 id 传入，然后把创建好的对象添加到水果列表中。接着我们在 onCreate()方法中创建了 FruitAdapter 对象，并将 FruitAdapter 作为适配器传递给了 ListView。这样定制 ListView 界面的任务就完成了。

现在重新运行程序，效果如图 3.30 所示。

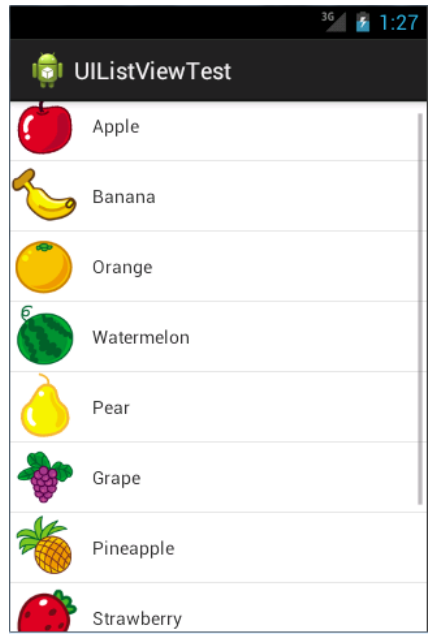

图 3.30

虽然目前我们定制的界面还是很简单，但是相信聪明的你已经领悟到了诀窍，只要修改 fruit_item.xml 中的内容，就可以定制出各种复杂的界面了。

3.5.3 提升 ListView 的运行效率

之所以说 ListView 这个控件很难用，就是因为它有很多的细节可以优化，其中运行效率就是很重要的一点。目前我们 ListView 的运行效率是很低的，因为在 FruitAdapter 的 getView()方法中每次都将布局重新加载了一遍，当 ListView 快速滚动的时候这就会成为性能的瓶颈。

仔细观察，getView()方法中还有一个 convertView 参数，这个参数用于将之前加载好的

布局进行缓存，以便之后可以进行重用。修改 FruitAdapter 中的代码，如下所示：

```
public class FruitAdapter extends ArrayAdapter<Fruit> {
    ……
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Fruit fruit = getItem(position);
        View view;
        if (convertView == null) {
            view = LayoutInflater.from(getContext()).inflate(resourceId, null);
        } else {
            view = convertView;
        }
        ImageView fruitImage = (ImageView) view.findViewById(R.id.fruit_image);
        TextView fruitName = (TextView) view.findViewById(R.id.fruit_name);
        fruitImage.setImageResource(fruit.getImageId());
        fruitName.setText(fruit.getName());
        return view;
    }
}
```

可以看到，现在我们在 getView()方法中进行了判断，如果 convertView 为空，则使用 LayoutInflater 去加载布局，如果不为空则直接对 convertView 进行重用。这样就大大提高了 ListView 的运行效率，在快速滚动的时候也可以表现出更好的性能。

不过，目前我们的这份代码还是可以继续优化的，虽然现在已经不会再重复去加载布局，但是每次在 getView()方法中还是会调用 View 的 findViewById()方法来获取一次控件的实例。我们可以借助一个 ViewHolder 来对这部分性能进行优化，修改 FruitAdapter 中的代码，如下所示：

```
public class FruitAdapter extends ArrayAdapter<Fruit> {
    ……
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Fruit fruit = getItem(position);
        View view;
        ViewHolder viewHolder;
        if (convertView == null) {
            view = LayoutInflater.from(getContext()).inflate(resourceId, null);
            viewHolder = new ViewHolder();
            viewHolder.fruitImage = (ImageView) view.findViewById
```

```
(R.id.fruit_image);
            viewHolder.fruitName = (TextView) view.findViewById
(R.id.fruit_name);
            view.setTag(viewHolder); // 将ViewHolder存储在View中
        } else {
            view = convertView;
            viewHolder = (ViewHolder) view.getTag(); // 重新获取ViewHolder
        }
        viewHolder.fruitImage.setImageResource(fruit.getImageId());
        viewHolder.fruitName.setText(fruit.getName());
        return view;
    }

    class ViewHolder {

        ImageView fruitImage;

        TextView fruitName;

    }
}
```

我们新增了一个内部类 ViewHolder，用于对控件的实例进行缓存。当 convertView 为空的时候，创建一个 ViewHolder 对象，并将控件的实例都存放在 ViewHolder 里，然后调用 View 的 setTag()方法，将 ViewHolder 对象存储在 View 中。当 convertView 不为空的时候则调用 View 的 getTag()方法，把 ViewHolder 重新取出。这样所有控件的实例都缓存在了 ViewHolder 里，就没有必要每次都通过 findViewById()方法来获取控件实例了。

通过这两步的优化之后，我们 ListView 的运行效率就已经非常不错了。

3.5.4 ListView 的点击事件

话说回来，ListView 的滚动毕竟只是满足了我们视觉上的效果，可是如果 ListView 中的子项不能点击的话，这个控件就没有什么实际的用途了。因此，本小节中我们就来学习一下 ListView 如何才能响应用户的点击事件。

修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private List<Fruit> fruitList = new ArrayList<Fruit>();
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initFruits();
        FruitAdapter adapter = new FruitAdapter(MainActivity.this,
R.layout.fruit_item, fruitList);
        ListView listView = (ListView) findViewById(R.id.list_view);
        listView.setAdapter(adapter);
        listView.setOnItemClickListener(new OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view,
                    int position, long id) {
                Fruit fruit = fruitList.get(position);
                Toast.makeText(MainActivity.this, fruit.getName(),
                        Toast.LENGTH_SHORT).show();
            }
        });
    }
    ......
}
```

可以看到，我们使用了 setOnItemClickListener()方法来为 ListView 注册了一个监听器，当用户点击了 ListView 中的任何一个子项时就会回调 onItemClick()方法，在这个方法中可以通过 position 参数判断出用户点击的是哪一个子项，然后获取到相应的水果，并通过 Toast 将水果的名字显示出来。

重新运行程序，并点击一下西瓜，效果如图 3.31 所示。

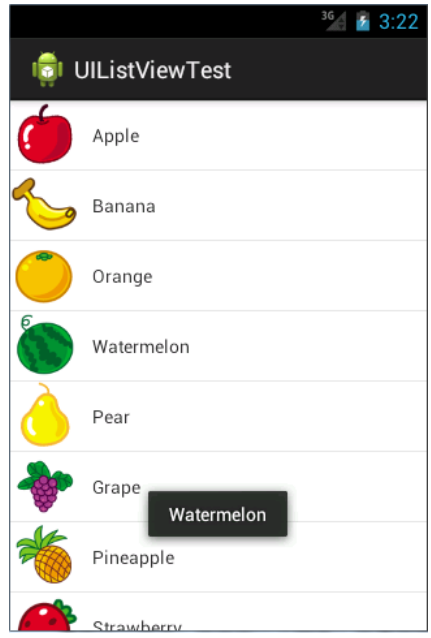
3:22
UIListViewTest
Apple
Banana
Orange
Watermelon
Pear
Grape
Watermelon
Pineapple
Strawberry

图 3.31

第 4 章　详解广播机制

记得在我上学的时候，每个班级的教室里都会装有一个喇叭，这些喇叭都是接入到学校的广播室的，一旦有什么重要的通知，就会播放一条广播来告知全校的师生。类似的工作机制其实在计算机领域也有很广泛的应用，如果你了解网络通信原理应该会知道，在一个 IP 网络范围中最大的 IP 地址是被保留作为广播地址来使用的。比如某个网络的 IP 范围是 192.168.0.XXX，子网掩码是 255.255.255.0，那么这个网络的广播地址就是 192.168.0.255。广播数据包会被发送到同一网络上的所有端口，这样在该网络中的每台主机都将会收到这条广播。

为了方便于进行系统级别的消息通知，Android 也引入了一套类似的广播消息机制。相比于我前面举出的两个例子，Android 中的广播机制会显得更加的灵活，本章就将对这一机制的方方面面进行详细的讲解。

5.1　广播机制简介

为什么说 Android 中的广播机制更加灵活呢？这是因为 Android 中的每个应用程序都可以对自己感兴趣的广播进行注册，这样该程序就只会接收到自己所关心的广播内容，这些广播可能是来自于系统的，也可能是来自于其他应用程序的。Android 提供了一套完整的 API，允许应用程序自由地发送和接收广播。发送广播的方法其实之前稍微有提到过一下，如果你记性好的话可能还会有印象，就是借助我们第 2 章学过的 Intent。而接收广播的方法则需要引入一个新的概念，广播接收器（Broadcast Receiver）。

广播接收器的具体用法将会在下一节中做介绍，这里我们先来了解一下广播的类型。Android 中的广播主要可以分为两种类型，标准广播和有序广播。

标准广播（Normal broadcasts）是一种完全异步执行的广播，在广播发出之后，所有的广播接收器几乎都会在同一时刻接收到这条广播消息，因此它们之间没有任何先后顺序可言。这种广播的效率会比较高，但同时也意味着它是无法被截断的。标准广播的工作流程如图 5.1 所示。

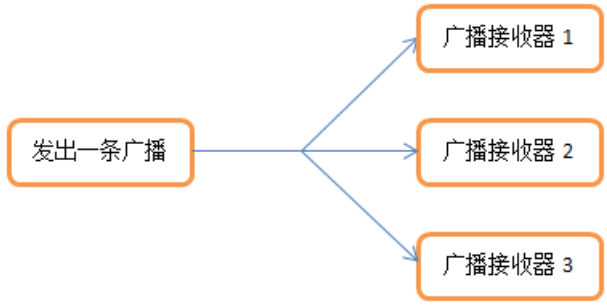

图 5.1

有序广播（Ordered broadcasts）则是一种同步执行的广播，在广播发出之后，同一时刻只会有一个广播接收器能够收到这条广播消息，当这个广播接收器中的逻辑执行完毕后，广播才会继续传递。所以此时的广播接收器是有先后顺序的，优先级高的广播接收器就可以先收到广播消息，并且前面的广播接收器还可以截断正在传递的广播，这样后面的广播接收器就无法收到广播消息了。有序广播的工作流程如图 5.2 所示。

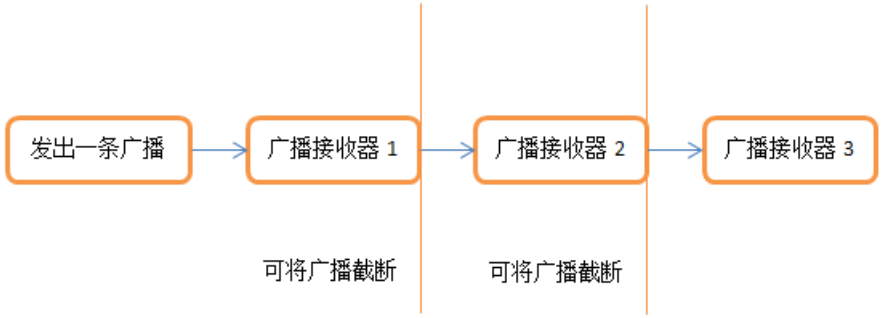

图 5.2

掌握了这些基本概念后，我们就可以来尝试一下广播的用法了，首先就从接收系统广播开始吧。

5.2 接收系统广播

Android 内置了很多系统级别的广播，我们可以在应用程序中通过监听这些广播来得到各种系统的状态信息。比如手机开机完成后会发出一条广播，电池的电量发生变化会发出一条广播，时间或时区发生改变也会发出一条广播等等。如果想要接收到这些广播，就需要使用广播接收器，下面我们就来看一下它的具体用法。

5.2.1 动态注册监听网络变化

广播接收器可以自由地对自己感兴趣的广播进行注册，这样当有相应的广播发出时，广播接收器就能够收到该广播，并在内部处理相应的逻辑。注册广播的方式一般有两种，在代码中注册和在 AndroidManifest.xml 中注册，其中前者也被称为动态注册，后者也被称为静态注册。

那么该如何创建一个广播接收器呢？其实只需要新建一个类，让它继承自 BroadcastReceiver，并重写父类的 onReceive()方法就行了。这样当有广播到来时，onReceive()方法就会得到执行，具体的逻辑就可以在这个方法中处理。

那我们就先通过动态注册的方式编写一个能够监听网络变化的程序，借此学习一下广播接收器的基本用法吧。新建一个 BroadcastTest 项目，然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private IntentFilter intentFilter;

    private NetworkChangeReceiver networkChangeReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        intentFilter = new IntentFilter();
        intentFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
        networkChangeReceiver = new NetworkChangeReceiver();
        registerReceiver(networkChangeReceiver, intentFilter);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(networkChangeReceiver);
    }

    class NetworkChangeReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
```

```
            Toast.makeText(context, "network changes",
Toast.LENGTH_SHORT).show();
        }

    }

}
```

可以看到，我们在 MainActivity 中定义了一个内部类 NetworkChangeReceiver，这个类是继承自 BroadcastReceiver 的，并重写了父类的 onReceive()方法。这样每当网络状态发生变化时，onReceive()方法就会得到执行，这里只是简单地使用 Toast 提示了一段文本信息。

然后观察 onCreate()方法，首先我们创建了一个 IntentFilter 的实例，并给它添加了一个值为 android.net.conn.CONNECTIVITY_CHANGE 的 action，为什么要添加这个值呢？因为当网络状态发生变化时，系统发出的正是一条值为 android.net.conn.CONNECTIVITY_CHANGE 的广播，也就是说我们的广播接收器想要监听什么广播，就在这里添加相应的 action 就行了。接下来创建了一个 NetworkChangeReceiver 的实例，然后调用 registerReceiver()方法进行注册，将 NetworkChangeReceiver 的实例和 IntentFilter 的实例都传了进去，这样 NetworkChangeReceiver 就会收到所有值为 android.net.conn.CONNECTIVITY_CHANGE 的广播，也就实现了监听网络变化的功能。

最后要记得，动态注册的广播接收器一定都要取消注册才行，这里我们是在 onDestroy()方法中通过调用 unregisterReceiver()方法来实现的。

整体来说，代码还是非常简单的，现在运行一下程序。首先你会在注册完成的时候收到一条广播，然后按下 Home 键回到主界面（注意不能按 Back 键，否则 onDestroy()方法会执行），接着按下 Menu 键→System settings→Data usage 进入到数据使用详情界面，然后尝试着开关 Mobile Data 来启动和禁用网络，你就会看到有 Toast 提醒你网络发生了变化。

不过只是提醒网络发生了变化还不够人性化，最好是能准确地告诉用户当前是有网络还是没有网络，因此我们还需要对上面的代码进行进一步的优化。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {
    ......
    class NetworkChangeReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            ConnectivityManager connectionManager = (ConnectivityManager)
                    getSystemService(Context.CONNECTIVITY_SERVICE);
            NetworkInfo networkInfo = connectionManager.getActiveNetworkInfo();
```

```
            if (networkInfo != null && networkInfo.isAvailable()) {
                Toast.makeText(context, "network is available",
                        Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(context, "network is unavailable",
                        Toast.LENGTH_SHORT).show();
            }
        }

    }
}
```

在 onReceive()方法中，首先通过 getSystemService()方法得到了 ConnectivityManager 的实例，这是一个系统服务类，专门用于管理网络连接的。然后调用它的 getActiveNetworkInfo()方法可以得到 NetworkInfo 的实例，接着调用 NetworkInfo 的 isAvailable()方法，就可以判断出当前是否有网络了，最后我们还是通过 Toast 的方式对用户进行提示。

另外，这里有非常重要的一点需要说明，Android 系统为了保证应用程序的安全性做了规定，如果程序需要访问一些系统的关键性信息，必须在配置文件中声明权限才可以，否则程序将会直接崩溃，比如这里查询系统的网络状态就是需要声明权限的。打开 AndroidManifest.xml 文件，在里面加入如下权限就可以查询系统网络状态了：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="19" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    ……
</manifest>
```

访问 http://developer.android.com/reference/android/Manifest.permission.html 可以查看 Android 系统所有可声明的权限。

现在重新运行程序，然后按下 Home 键→按下 Menu 键→System settings→Data usage 进入到数据使用详情界面，关闭 Mobile Data 会弹出无网络可用的提示，如图 5.3 所示。

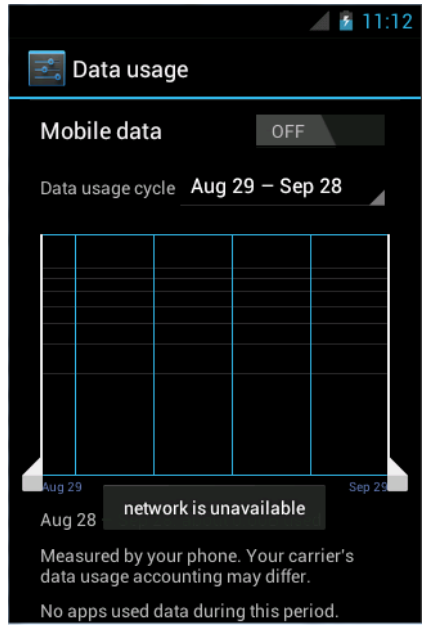

图　5.3

然后重新打开 Mobile Data 又会弹出网络可用的提示，如图 5.4 所示。

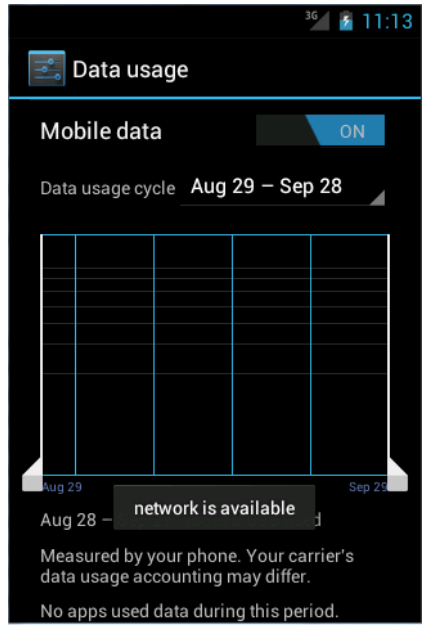

图　5.4

5.2.2 静态注册实现开机启动

动态注册的广播接收器可以自由地控制注册与注销，在灵活性方面有很大的优势，但是它也存在着一个缺点，即必须要在程序启动之后才能接收到广播，因为注册的逻辑是写在onCreate()方法中的。那么有没有什么办法可以让程序在未启动的情况下就能接收到广播呢？这就需要使用静态注册的方式了。

这里我们准备让程序接收一条开机广播，当收到这条广播时就可以在 onReceive()方法里执行相应的逻辑，从而实现开机启动的功能。新建一个 BootCompleteReceiver 继承自 BroadcastReceiver，代码如下所示：

```
public class BootCompleteReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "Boot Complete", Toast.LENGTH_LONG).show();
    }

}
```

可以看到，这里不再使用内部类的方式来定义广播接收器，因为稍后我们需要在 AndroidManifest.xml 中将这个广播接收器的类名注册进去。在 onReceive()方法中，还是简单地使用 Toast 弹出一段提示信息。

然后修改 AndroidManifest.xml 文件，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <receiver android:name=".BootCompleteReceiver" >
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
```

```
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

终于，<application>标签内出现了一个新的标签<receiver>，所有静态注册的广播接收器都是在这里进行注册的。它的用法其实和<activity>标签非常相似，首先通过 android:name 来指定具体注册哪一个广播接收器，然后在<intent-filter>标签里加入想要接收的广播就行了，由于 Android 系统启动完成后会发出一条值为 android.intent.action.BOOT_COMPLETED 的广播，因此我们在这里添加了相应的 action。

另外，监听系统开机广播也是需要声明权限的，可以看到，我们使用<uses-permission>标签又加入了一条 android.permission.RECEIVE_BOOT_COMPLETED 权限。

现在重新运行程序后，我们的程序就已经可以接收开机广播了，首先打开到应用程序管理界面来查看一下当前程序所拥有的权限。在桌面按下 Menu 键→System settings→Apps，然后点击 BroadcastTest，如图 5.5 所示。

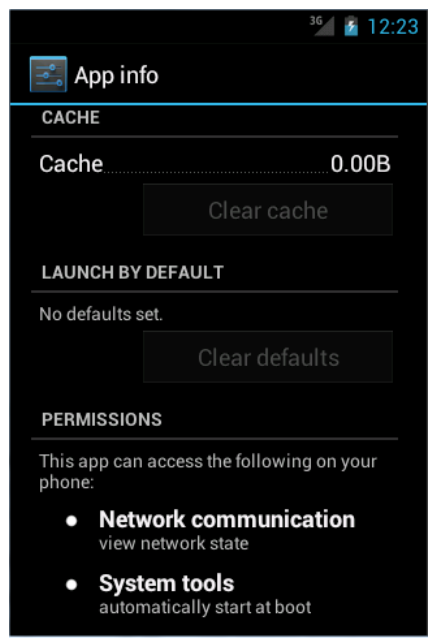

图 5.5

可以看到，我们的程序目前拥有访问网络状态和开机自动启动的权限。然后将模拟器关闭并重新启动，在启动完成之后就会收到开机广播了，如图 5.6 所示。

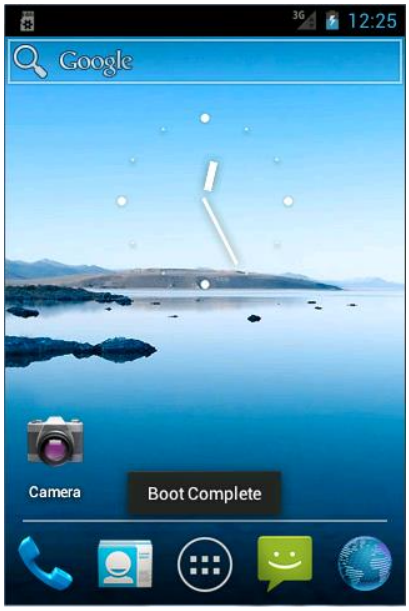

图 5.6

到目前为止，我们在广播接收器的 onReceive()方法中都只是简单地使用 Toast 提示了一段文本信息，当你真正在项目中使用到它的时候，就可以在里面编写自己的逻辑。需要注意的是，不要在 onReceive()方法中添加过多的逻辑或者进行任何的耗时操作，因为在广播接收器中是不允许开启线程的，当 onReceive()方法运行了较长时间而没有结束时，程序就会报错。因此广播接收器更多的是扮演一种打开程序其他组件的角色，比如创建一条状态栏通知，或者启动一个服务等，这几个概念我们会在后面的章节中学到。

5.3 发送自定义广播

现在你已经学会了通过广播接收器来接收系统广播，接下来我们就要学习一下如何在应用程序中发送自定义的广播。前面已经介绍过了，广播主要分为两种类型，标准广播和有序广播，在本节中我们就将通过实践的方式来看下这两种广播具体的区别。

5.3.1 发送标准广播

在发送广播之前，我们还是需要先定义一个广播接收器来准备接收此广播才行，不然发出去也是白发。因此新建一个 MyBroadcastReceiver 继承自 BroadcastReceiver，代码如下所示：

```
public class MyBroadcastReceiver extends BroadcastReceiver {
```

```
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "received in MyBroadcastReceiver",
Toast.LENGTH_SHORT).show();
    }

}
```

这里当 MyBroadcastReceiver 收到自定义的广播时，就会弹出 received in MyBroadcastReceiver 的提示。然后在 AndroidManifest.xml 中对这个广播接收器进行注册：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <receiver android:name=".MyBroadcastReceiver">
            <intent-filter>
                <action android:name="com.example.broadcasttest. MY_BROADCAST"/>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

可以看到，这里让 MyBroadcastReceiver 接收一条值为 com.example.broadcasttest. MY_BROADCAST 的广播，因此待会儿在发送广播的时候，我们就需要发出这样的一条广播。

接下来修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

   <Button
       android:id="@+id/button"
       android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:text="Send Broadcast"
        />

</LinearLayout>
```

这里在布局文件中定义了一个按钮，用于作为发送广播的触发点。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {
    ......
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent("com.example.broadcasttest.
MY_BROADCAST");
                sendBroadcast(intent);
            }
        });
        ......
    }
    ......
}
```

可以看到，我们在按钮的点击事件里面加入了发送自定义广播的逻辑。首先构建出了一个 Intent 对象，并把要发送的广播的值传入，然后调用了 Context 的 sendBroadcast()方法将广播发送出去，这样所有监听 com.example.broadcasttest.MY_BROADCAST 这条广播的广播接收器就会收到消息。此时发出去的广播就是一条标准广播。

重新运行程序，并点击一下 Send Broadcast 按钮，效果如图 5.7 所示。

图 5.7

这样我们就成功完成了发送自定义广播的功能。另外，由于广播是使用 Intent 进行传递的，因此你还可以在 Intent 中携带一些数据传递给广播接收器。

5.3.2 发送有序广播

广播是一种可以跨进程的通信方式，这一点从前面接收系统广播的时候就可以看出来了。因此在我们应用程序内发出的广播，其他的应用程序应该也是可以收到的。为了验证这一点，我们需要再新建一个 BroadcastTest2 项目。

将项目创建好之后，还需要在这个项目下定义一个广播接收器，用于接收上一小节中的自定义广播。新建 AnotherBroadcastReceiver 继承自 BroadcastReceiver，代码如下所示：

```
public class AnotherBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "received in AnotherBroadcastReceiver",
                Toast.LENGTH_SHORT).show();
    }

}
```

这里仍然是在广播接收器的 onReceive()方法中弹出了一段文本信息。然后在

AndroidManifest.xml 中对这个广播接收器进行注册，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest2"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <receiver android:name=".AnotherBroadcastReceiver" >
            <intent-filter>
                <action android:name="com.example.broadcasttest.MY_BROADCAST" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

可以看到，AnotherBroadcastReceiver 同样接收的是 com.example.broadcasttest.MY_BROADCAST 这条广播。现在运行 BroadcastTest2 项目将这个程序安装到模拟器上，然后重新回到 BroadcastTest 项目的主界面，并点击一下 Send Broadcast 按钮，就会分别弹出两次提示信息，如图 5.8 所示。

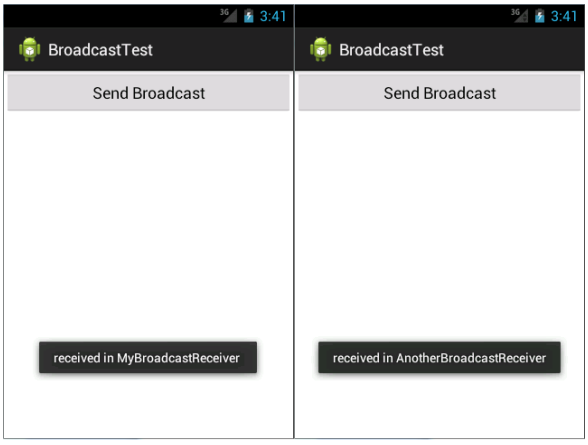

图 5.8

这样就强有力地证明了，我们的应用程序发出的广播是可以被其他的应用程序接收到的。

不过到目前为止，程序里发出的都还是标准广播，现在我们来尝试一下发送有序广播。关闭 BroadcastTest2 项目，然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {
    ……
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent("com.example.broadcasttest.
MY_BROADCAST");
                sendOrderedBroadcast(intent, null);
            }
        });
        ……
    }
    ……
}
```

可以看到，发送有序广播只需要改动一行代码，即将 sendBroadcast()方法改成 sendOrderedBroadcast()方法。sendOrderedBroadcast()方法接收两个参数，第一个参数仍然是 Intent，第二个参数是一个与权限相关的字符串，这里传入 null 就行了。现在重新运行程序，并点击 Send Broadcast 按钮，你会发现，两个应用程序仍然都可以接收到这条广播。

看上去好像和标准广播没什么区别嘛，不过别忘了，这个时候的广播接收器是有先后顺序的，而且前面的广播接收器还可以将广播截断，以阻止其继续传播。

那么该如何设定广播接收器的先后顺序呢？当然是在注册的时候进行设定的了，修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
```

```
            android:icon="@drawable/ic_launcher"
            android:label="@string/app_name"
            android:theme="@style/AppTheme" >
            ……
            <receiver android:name=".MyBroadcastReceiver">
                <intent-filter android:priority="100" >
                    <action android:name="com.example.broadcasttest.MY_BROADCAST"/>
                </intent-filter>
            </receiver>
        </application>
    </manifest>
```

可以看到，我们通过 android:priority 属性给广播接收器设置了优先级，优先级比较高的广播接收器就可以先收到广播。这里将 MyBroadcastReceiver 的优先级设成了 100，以保证它一定会在 AnotherBroadcastReceiver 之前收到广播。

既然已经获得了接收广播的优先权，那么 MyBroadcastReceiver 就可以选择是否允许广播继续传递了。修改 MyBroadcastReceiver 中的代码，如下所示：

```
public class MyBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "received in MyBroadcastReceive",
Toast.LENGTH_SHORT).show();
        abortBroadcast();
    }

}
```

如果在 onReceive()方法中调用了 abortBroadcast()方法，就表示将这条广播截断，后面的广播接收器将无法再接收到这条广播。现在重新运行程序，并点击一下 Send Broadcast 按钮，你会发现，只有 MyBroadcastReceiver 中的 Toast 信息能够弹出，说明这条广播经过 MyBroadcastReceiver 之后确实是终止传递了。

5.4 使用本地广播

前面我们发送和接收的广播全部都是属于系统全局广播，即发出的广播可以被其他任何的任何应用程序接收到，并且我们也可以接收来自于其他任何应用程序的广播。这样就很容易会引起安全性的问题，比如说我们发送的一些携带关键性数据的广播有可能被其他的应用

程序截获，或者其他的程序不停地向我们的广播接收器里发送各种垃圾广播。

为了能够简单地解决广播的安全性问题，Android 引入了一套本地广播机制，使用这个机制发出的广播只能够在应用程序的内部进行传递，并且广播接收器也只能接收来自本应用程序发出的广播，这样所有的安全性问题就都不存在了。

本地广播的用法并不复杂，主要就是使用了一个 LocalBroadcastManager 来对广播进行管理，并提供了发送广播和注册广播接收器的方法。下面我们就通过具体的实例来尝试一下它的用法，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private IntentFilter intentFilter;

    private LocalReceiver localReceiver;

    private LocalBroadcastManager localBroadcastManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        localBroadcastManager = LocalBroadcastManager.getInstance(this);
// 获取实例
        Button button = (Button) findViewById(R.id.button);
        button.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent("com.example.broadcasttest.
LOCAL_BROADCAST");
                localBroadcastManager.sendBroadcast(intent); // 发送本地广播
            }
        });
        intentFilter = new IntentFilter();
        intentFilter.addAction("com.example.broadcasttest.LOCAL_BROADCAST");
        localReceiver = new LocalReceiver();
        localBroadcastManager.registerReceiver(localReceiver, intentFilter);
// 注册本地广播监听器
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
```

```
        localBroadcastManager.unregisterReceiver(localReceiver);
    }

    class LocalReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            Toast.makeText(context, "received local broadcast",
Toast.LENGTH_SHORT).show();
        }

    }

}
```

有没有感觉这些代码很熟悉？没错，其实这基本上就和我们前面所学的动态注册广播接收器以及发送广播的代码是一样。只不过现在首先是通过 LocalBroadcastManager 的 getInstance()方法得到了它的一个实例，然后在注册广播接收器的时候调用的是 LocalBroadcastManager 的 registerReceiver()方法，在发送广播的时候调用的是 LocalBroadcastManager 的 sendBroadcast()方法，仅此而已。这里我们在按钮的点击事件里面发出了一条 com.example.broadcasttest.LOCAL_BROADCAST 广播，然后在 LocalReceiver 里去接收这条广播。重新运行程序，并点击 Send Broadcast 按钮，效果如图 5.9 所示。

图 5.9

可以看到，LocalReceiver 成功接收到了这条本地广播，并通过 Toast 提示了出来。如果你还有兴趣进行实验，可以尝试在 BroadcastTest2 中也去接收 com.example.broadcasttest.LOCAL_BROADCAST 这条广播，答案是显而易见的，肯定无法收到，因为这条广播只会在 BroadcastTest 程序内传播。

另外还有一点需要说明，本地广播是无法通过静态注册的方式来接收的。其实这也完全可以理解，因为静态注册主要就是为了让程序在未启动的情况下也能收到广播，而发送本地广播时，我们的程序肯定是已经启动了，因此也完全不需要使用静态注册的功能。

最后我们再来盘点一下使用本地广播的几点优势吧。

19. 可以明确地知道正在发送的广播不会离开我们的程序，因此不需要担心机密数据泄漏的问题。
20. 其他的程序无法将广播发送到我们程序的内部，因此不需要担心会有安全漏洞的隐患。
21. 发送本地广播比起发送系统全局广播将会更加高效。

5.5 广播的最佳实践——实现强制下线功能

本章的内容不是非常多，因此相信你也一定学得很轻松吧。现在我们就准备通过一个完整例子的实践，来综合运用一下本章中所学到的知识。

强制下线功能应该算是比较常见的了，很多的应用程序都具备这个功能，比如你的 QQ 号在别处登录了，就会将你强制挤下线。其实实现强制下线功能的思路也比较简单，只需要在界面上弹出一个对话框，让用户无法进行任何其他操作，必须要点击对话框中的确定按钮，然后回到登录界面即可。可是这样就存在着一个问题，因为我们被通知需要强制下线时可能正处于任何一个界面，难道需要在每个界面上都编写一个弹出对话框的逻辑？如果你真的这么想，那思维就偏远了，我们完全可以借助本章中所学的广播知识，来非常轻松地实现这一功能。新建一个 BroadcastBestPractice 项目，然后开始动手吧。

强制下线功能需要先关闭掉所有的活动，然后回到登录界面。如果你的反应足够快的话，应该会想到我们在第 2 章的最佳实践部分早就已经实现过关闭所有活动的功能了，因此这里只需要使用同样的方案即可。先创建一个 ActivityCollector 类用于管理所有的活动，代码如下所示：

```
public class ActivityCollector {

    public static List<Activity> activities = new ArrayList<Activity>();

    public static void addActivity(Activity activity) {
        activities.add(activity);
```

```
    }

    public static void removeActivity(Activity activity) {
        activities.remove(activity);
    }

    public static void finishAll() {
        for (Activity activity : activities) {
            if (!activity.isFinishing()) {
                activity.finish();
            }
        }
    }

}
```

然后创建 BaseActivity 类作为所有活动的父类，代码如下所示：

```
public class BaseActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityCollector.addActivity(this);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        ActivityCollector.removeActivity(this);
    }

}
```

接着需要创建一个登录界面的布局，还记得我们在 3.3.4 节里编写的登录界面吗？这里也是直接拿来用就好了，这下可省了我们不少的功夫。新建布局文件 login.xml，代码如下所示：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1" >
```

```
    <TableRow>

        <TextView
            android:layout_height="wrap_content"
            android:text="Account:" />

        <EditText
            android:id="@+id/account"
            android:layout_height="wrap_content"
            android:hint="Input your account" />
    </TableRow>

    <TableRow>

        <TextView
            android:layout_height="wrap_content"
            android:text="Password:" />

        <EditText
            android:id="@+id/password"
            android:layout_height="wrap_content"
            android:inputType="textPassword" />
    </TableRow>

    <TableRow>

        <Button
            android:id="@+id/login"
            android:layout_height="wrap_content"
            android:layout_span="2"
            android:text="Login" />
    </TableRow>

</TableLayout>
```

以上代码都是直接复用之前写好的内容，非常开心。不过从这里开始，我们又需要靠自己去动手实现了。现在登录界面的布局已经完成，那么接下来就应该去编写登录界面的活动了，新建 LoginActivity 继承自 BaseActivity，代码如下所示：

```
public class LoginActivity extends BaseActivity {

    private EditText accountEdit;

    private EditText passwordEdit;

    private Button login;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);
        accountEdit = (EditText) findViewById(R.id.account);
        passwordEdit = (EditText) findViewById(R.id.password);
        login = (Button) findViewById(R.id.login);
        login.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String account = accountEdit.getText().toString();
                String password = passwordEdit.getText().toString();
                // 如果账号是admin且密码是123456，就认为登录成功
                if (account.equals("admin") && password.equals("123456")) {
                    Intent intent = new Intent(LoginActivity.this,
MainActivity.class);
                    startActivity(intent);
                    finish();
                } else {
                    Toast.makeText(LoginActivity.this, "account or password
is invalid",
                            Toast.LENGTH_SHORT).show();
                }
            }
        });
    }

}
```

可以看到，这里我们模拟了一个非常简单的登录功能。首先使用 setContentView()方法将 login 布局加载进来，并调用 findViewById()方法分别获取到账号输入框、密码输入框以及

登录按钮的实例。然后在登录按钮的点击事件里面对输入的账号和密码进行判断，如果账号是 admin 并且密码是 123456，就认为登录成功并跳转到 MainActivity，否则就提示用户账号或密码错误。

因此，你就可以将 MainActivity 理解成是登录成功后进入的程序主界面了，这里我们并不需要在主界面里提供什么花哨的功能，只需要加入强制下线功能就可以了，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/force_offline"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send force offline broadcast" />

</LinearLayout>
```

非常简单，只有一个按钮而已。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends BaseActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button forceOffline = (Button) findViewById(R.id.force_offline);
        forceOffline.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
               Intent intent = new Intent("com.example.broadcastbestpractice.
FORCE_OFFLINE ");
                sendBroadcast(intent);
            }
        });
    }

}
```

同样非常简单，不过这里有个重点，我们在按钮的点击事件里面发送了一条广播，广播

的值为 com.example.broadcastbestpractice.FORCE_OFFLINE，这条广播就是用于通知程序强制用户下线的。也就是说强制用户下线的逻辑并不是写在 MainActivity 里的，而是应该写在接收这条广播的广播接收器里面，这样强制下线的功能就不会依附于任何的界面，不管是在程序的任何地方，只需要发出这样一条广播，就可以完成强制下线的操作了。

那么毫无疑问，接下来我们就需要创建一个广播接收器了，新建 ForceOfflineReceiver 继承自 BroadcastReceiver，代码如下所示：

```
public class ForceOfflineReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(final Context context, Intent intent) {
        AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(context);
        dialogBuilder.setTitle("Warning");
        dialogBuilder.setMessage("You are forced to be offline. Please try
to login again.");
        dialogBuilder.setCancelable(false);
        dialogBuilder.setPositiveButton("OK",
                new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        ActivityCollector.finishAll(); // 销毁所有活动
                        Intent intent = new Intent(context,
LoginActivity.class);
                        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                        context.startActivity(intent); // 重新启动LoginActivity
                    }
                });
        AlertDialog alertDialog = dialogBuilder.create();
        // 需要设置AlertDialog的类型，保证在广播接收器中可以正常弹出

    alertDialog.getWindow().setType(WindowManager.LayoutParams.TYPE_SYSTE
M_ALERT);
        alertDialog.show();
    }

}
```

这次 onReceive()方法里可不再是仅仅弹出一个 Toast 了，而是加入了较多的代码，那我们就来仔细地看一下吧。首先肯定是使用 AlertDialog.Builder 来构建一个对话框，注意这里一定要调用 setCancelable()方法将对话框设为不可取消，否则用户按一下 Back 键就可以关闭

对话框继续使用程序了。然后使用 setPositiveButton()方法来给对话框注册确定按钮，当用户点击了确定按钮时，就调用 ActivityCollector 的 finishAll()方法来销毁掉所有活动，并重新启动 LoginActivity 这个活动。另外，由于我们是在广播接收器里启动活动的，因此一定要给 Intent 加入 FLAG_ACTIVITY_NEW_TASK 这个标志。最后，还需要把对话框的类型设为 TYPE_SYSTEM_ALERT，不然它将无法在广播接收器里弹出。

这样的话，所有强制下线的逻辑就已经完成了，接下来我们还需要对 AndroidManifest.xml 文件进行配置，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcastbestpractice"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="14"
        android:targetSdkVersion="19" />

    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".LoginActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".MainActivity" >
        </activity>
        <receiver android:name=".ForceOfflineReceiver" >
            <intent-filter>
                <action android:name="com.example.broadcastbestpractice.
FORCE_OFFLINE" />
            </intent-filter>
        </receiver>
```

```
    </application>
</manifest>
```

这里有几点内容需要注意，首先由于我们在 ForceOfflineReceiver 里弹出了一个系统级别的对话框，因此必须要声明 android.permission.SYSTEM_ALERT_WINDOW 权限。然后对 LoginActivity 进行注册，并把它设置为主活动，因为肯定不能让用户启动程序就直接进入 MainActivity 吧。最后再对 ForceOfflineReceiver 进行注册，并指定它接收 com.example.broadcastbestpractice.FORCE_OFFLINE 这条广播。

好了，现在来尝试运行一下程序吧，首先会进入到登录界面，并可以在这里输入账号和密码，如图 5.10 所示。

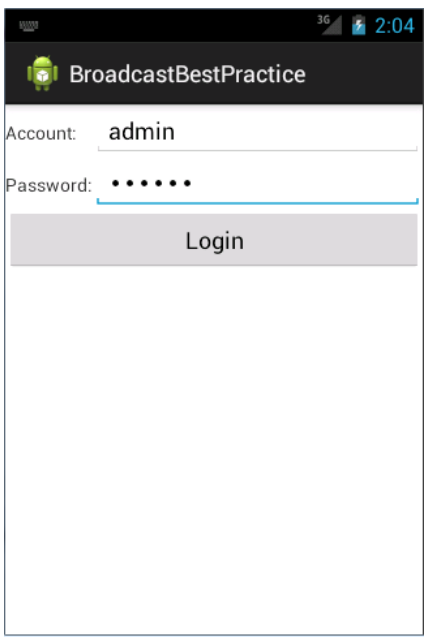

图 5.10

如果输入的账号是 admin，密码是 123456，点击登录按钮就会进入到程序的主界面，如图 5.11 所示。

图 5.11

这时点击一下发送广播的按钮，就会发出一条强制下线的广播，ForceOfflineReceiver 里收到这条广播后会弹出一个对话框提示用户已被强制下线，如图 5.12 所示。

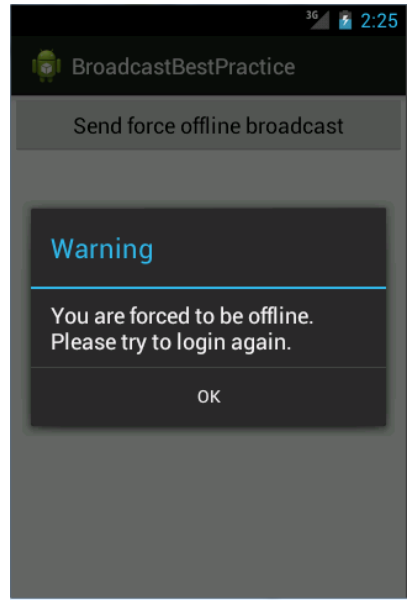

图 5.12

这时用户将无法再对界面的任何元素进行操作，只能点击确定按钮，然后会重新回到登录界面。这样，强制下线功能就已经完整地实现了。

5.7 小结与点评

本章中我们主要是对 Android 的广播机制进行了深入的研究，不仅了解了广播的理论知识、还掌握了接收广播、发送自定义广播以及本地广播的使用方法。广播接收器是属于 Android 四大组件之一，在不知不觉中你已经掌握了四大组件中的两个了。

在最佳实践环节中你一定也收获了不少，不仅运用到了本章所学的广播知识，还将前面章节所学到的技巧综合运用到了一起。经过这个例子之后，相信你对所涉及到的每个知识点都有了更深一层的认识。另外，本章还添加了一个最最特殊的环节，即 Git 时间。在这个环节中我们对 Git 这个版本控制工具进行了初步的学习，后面还会学习关于它的更多内容。

下一章我们本应该继续学习 Android 中剩余的四大组件，不过由于学习内容提供器之前需要先掌握 Android 中的持久化技术，因此下一章我们就先对这一主题展开讨论。

第 5 章　数据存储

任何一个应用程序其实说白了就是在不停地和数据打交道，我们聊 QQ、看新闻、刷微博所关心的都是里面的数据，没有数据的应用程序就变成了一个空壳子，对用户来说没有任何实际用途。那么这些数据都是从哪来的呢？现在多数的数据基本都是由用户产生的了，比如你发微博、评论新闻，其实都是在产生数据。

而我们前面章节所编写的众多例子中也有用到各种各样的数据，例如第 3 章最佳实践部分在聊天界面编写的聊天内容，第 5 章最佳实践部分在登录界面输入的账号和密码。这些数据都有一个共同点，即它们都是属于瞬时数据。那么什么是瞬时数据呢？就是指那些存储在内存当中，有可能会因为程序关闭或其他原因导致内存被回收而丢失的数据。这对于一些关键性的数据信息来说是绝对不能容忍的，谁都不希望自己刚发出去的一条微博，刷新一下就没了吧。那么怎样才能保证让一些关键性的数据不会丢失呢？这就需要用到数据持久化技术了。

5.1　持久化技术简介

数据持久化就是指将那些内存中的瞬时数据保存到存储设备中，保证即使在手机或电脑关机的情况下，这些数据仍然不会丢失。保存在内存中的数据是处于瞬时状态的，而保存在存储设备中的数据是处于持久状态的，持久化技术则是提供了一种机制可以让数据在瞬时状态和持久状态之间进行转换。

持久化技术被广泛应用于各种程序设计的领域当中，而本书中要探讨的自然是 Android 中的数据持久化技术。Android 系统中主要提供了三种方式用于简单地实现数据持久化功能，即文件存储、SharedPreference 存储以及数据库存储。当然，除了这三种方式之外，你还可以将数据保存在手机的 SD 卡中，不过使用文件、SharedPreference 或数据库来保存数据会相对更简单一些，而且比起将数据保存在 SD 卡中会更加的安全。

那么下面我就将对这三种数据持久化的方式一一进行详细的讲解。

5.2　文件存储

文件存储是 Android 中最基本的一种数据存储方式，它不对存储的内容进行任何的格式化处理，所有数据都是原封不动地保存到文件当中的，因而它比较适合用于存储一些简单的

文本数据或二进制数据。如果你想使用文件存储的方式来保存一些较为复杂的文本数据，就需要定义一套自己的格式规范，这样方便于之后将数据从文件中重新解析出来。

那么首先我们就来看一看，Android 中是如何通过文件来保存数据的。

5.2.1 将数据存储到文件中

Context 类中提供了一个 openFileOutput ()方法，可以用于将数据存储到指定的文件中。这个方法接收两个参数，第一个参数是文件名，在文件创建的时候使用的就是这个名称，注意这里指定的文件名不可以包含路径，因为所有的文件都是默认存储到/data/data/<package name>/files/目录下的。第二个参数是文件的操作模式，主要有两种模式可选，MODE_PRIVATE 和 MODE_APPEND。其中 MODE_PRIVATE 是默认的操作模式，表示当指定同样文件名的时候，所写入的内容将会覆盖原文件中的内容，而 MODE_APPEND 则表示如果该文件已存在就往文件里面追加内容，不存在就创建新文件。其实文件的操作模式本来还有另外两种，MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE，这两种模式表示允许其他的应用程序对我们程序中的文件进行读写操作，不过由于这两种模式过于危险，很容易引起应用的安全性漏洞，现已在 Android 4.2 版本中被废弃。

openFileOutput ()方法返回的是一个 FileOutputStream 对象，得到了这个对象之后就可以使用 Java 流的方式将数据写入到文件中了。以下是一段简单的代码示例，展示了如何将一段文本内容保存到文件中：

```
public void save() {
    String data = "Data to save";
    FileOutputStream out = null;
    BufferedWriter writer = null;
    try {
        out = openFileOutput("data", Context.MODE_PRIVATE);
        writer = new BufferedWriter(new OutputStreamWriter(out));
        writer.write(data);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (writer != null) {
                writer.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        }
    }
```

如果你已经比较熟悉 Java 流了，理解上面的代码一定轻而易举吧。这里通过 openFileOutput()方法能够得到一个 FileOutputStream 对象，然后再借助它构建出一个 OutputStreamWriter 对象，接着再使用 OutputStreamWriter 构建出一个 BufferedWriter 对象，这样你就可以通过 BufferedWriter 来将文本内容写入到文件中了。

下面我们就编写一个完整的例子，借此学习一下如何在 Android 项目中使用文件存储的技术。首先创建一个 FilePersistenceTest 项目，并修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <EditText
        android:id="@+id/edit"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Type something here"
        />

</LinearLayout>
```

这里只是在布局中加入了一个 EditText，用于输入文本内容。其实现在你就可以运行一下程序了，界面上肯定会有一个文本输入框。然后在文本输入框中随意输入点什么内容，再按下 Back 键，这时输入的内容肯定就已经丢失了，因为它只是瞬时数据，在活动被销毁后就会被回收。而这里我们要做的，就是在数据被回收之前，将它存储到文件当中。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private EditText edit;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        edit = (EditText) findViewById(R.id.edit);
    }

    @Override
```

```
    protected void onDestroy() {
        super.onDestroy();
        String inputText = edit.getText().toString();
        save(inputText);
    }

    public void save(String inputText) {
        FileOutputStream out = null;
        BufferedWriter writer = null;
        try {
            out = openFileOutput("data", Context.MODE_PRIVATE);
            writer = new BufferedWriter(new OutputStreamWriter(out));
            writer.write(inputText);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

可以看到，首先我们在 onCreate()方法中获取了 EditText 的实例，然后重写了 onDestroy()方法，这样就可以保证在活动销毁之前一定会调用这个方法。在 onDestroy()方法中我们获取了 EditText 中输入的内容，并调用 save()方法把输入的内容存储到文件中，文件命名为 data。save()方法中的代码和之前的示例基本相同，这里就不再做解释了。现在重新运行一下程序，并在 Editext 中输入一些内容，如图 6.1 所示。

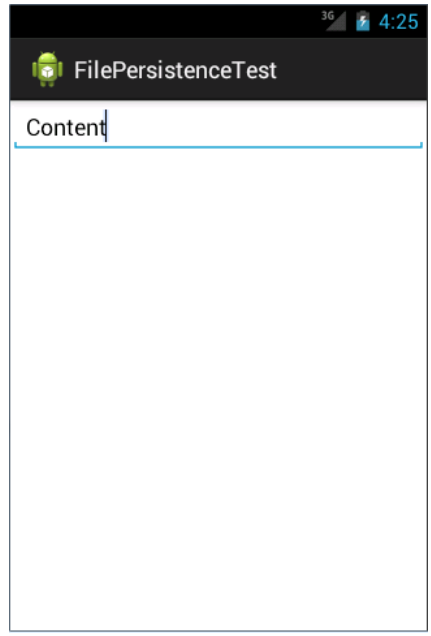

图 6.1

然后按下 Back 键关闭程序，这时我们输入的内容就已经保存到文件中了。那么如何才能证实数据确实已经保存成功了呢？我们可以借助 DDMS 的 File Explorer 来查看一下。切换到 DDMS 视图，并点击 File Explorer 切换卡，在这里进入到/data/data/com.example.filepersistencetest/files/目录下，可以看到生成了一个 data 文件，如图 6.2 所示。

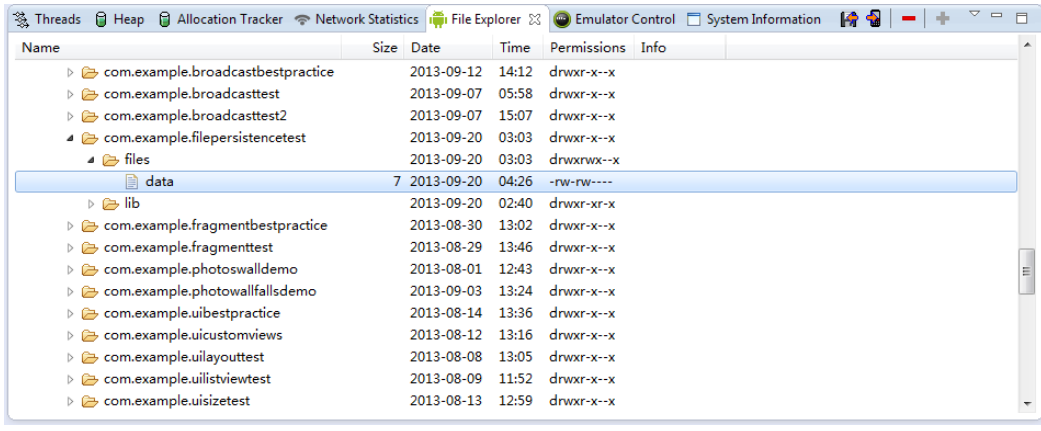

图 6.2

然后点击图 6.3 中最左边的按钮可以将这个文件导出到电脑上。

图 6.3

使用记事本打开这个文件，里面的内容如图 6.4 所示。

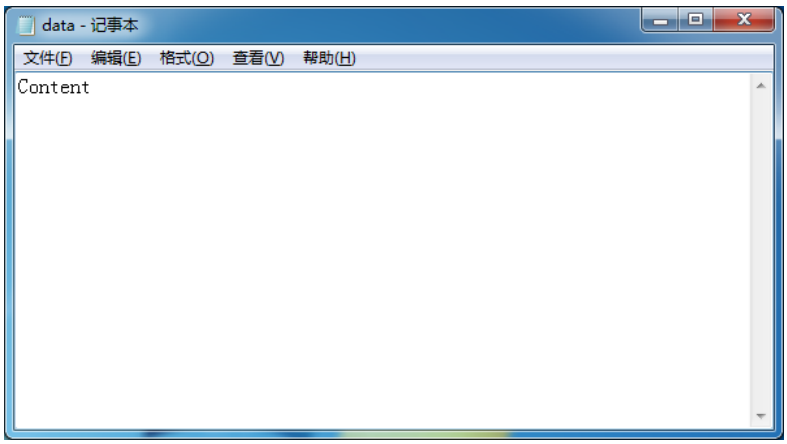

图 6.4

这样就证实了，在 EditText 中输入的内容确实已经成功保存到文件中了。

不过只是成功将数据保存下来还不够，我们还需要想办法在下次启动程序的时候让这些数据能够还原到 EditText 中，因此接下来我们就要学习一下，如何从文件中读取数据。

5.2.2 从文件中读取数据

类似于将数据存储到文件中，Context 类中还提供了一个 openFileInput()方法，用于从文件中读取数据。这个方法要比 openFileOutput()简单一些，它只接收一个参数，即要读取的文件名，然后系统会自动到/data/data/<package name>/files/目录下去加载这个文件，并返回一个 FileInputStream 对象，得到了这个对象之后再通过 Java 流的方式就可以将数据读取出来了。

以下是一段简单的代码示例，展示了如何从文件中读取文本数据：

```
public String load() {
    FileInputStream in = null;
    BufferedReader reader = null;
    StringBuilder content = new StringBuilder();
    try {
        in = openFileInput("data");
        reader = new BufferedReader(new InputStreamReader(in));
        String line = "";
```

```
            while ((line = reader.readLine()) != null) {
                content.append(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return content.toString();
    }
```

在这段代码中，首先通过 openFileInput()方法获取到了一个 FileInputStream 对象，然后借助它又构建出了一个 InputStreamReader 对象，接着再使用 InputStreamReader 构建出一个 BufferedReader 对象，这样我们就可以通过 BufferedReader 进行一行行地读取，把文件中所有的文本内容全部读取出来并存放在一个 StringBuilder 对象中，最后将读取到的内容返回就可以了。

了解了从文件中读取数据的方法，那么我们就来继续完善上一小节中的例子，使得重新启动程序时 EditText 中能够保留我们上次输入的内容。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private EditText edit;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        edit = (EditText) findViewById(R.id.edit);
        String inputText = load();
        if (!TextUtils.isEmpty(inputText)) {
            edit.setText(inputText);
            edit.setSelection(inputText.length());
            Toast.makeText(this, "Restoring succeeded",
Toast.LENGTH_SHORT).show();
```

```
        }
    }
    ......
    public String load() {
        FileInputStream in = null;
        BufferedReader reader = null;
        StringBuilder content = new StringBuilder();
        try {
            in = openFileInput("data");
            reader = new BufferedReader(new InputStreamReader(in));
            String line = "";
            while ((line = reader.readLine()) != null) {
                content.append(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (reader != null) {
                try {
                    reader.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return content.toString();
    }

}
```

可以看到，这里的思路非常简单，在 onCreate()方法中调用 load()方法来读取文件中存储的文本内容，如果读到的内容不为空，就调用 EditText 的 setText()方法将内容填充到 EditText 里，并调用 setSelection 方法将输入光标移动到文本的末尾位置以便于继续输入，然后弹出一句还原成功的提示。load()方法中的细节我们前面已经讲过了，这里就不再赘述。

注意上述代码在对字符串进行非空判断的时候使用了 TextUtils.isEmpty()方法，这是一个非常好用的方法，它可以一次性进行两种空值的判断。当传入的字符串等于 null 或者等于空字符串的时候，这个方法都会返回 true，从而使得我们不需要单独去判断这两种空值，再使用逻辑运算符连接起来了。

现在重新运行一下程序，刚才保存的 Content 字符串肯定会被填充到 EditText 中，然后

编写一点其他的内容，比如在 EditText 中输入 Hello，接着按下 Back 键退出程序，再重新启动程序，这时刚才输入的内容并不会丢失，而是还原到了 EditText 中，如图 6.5 所示。

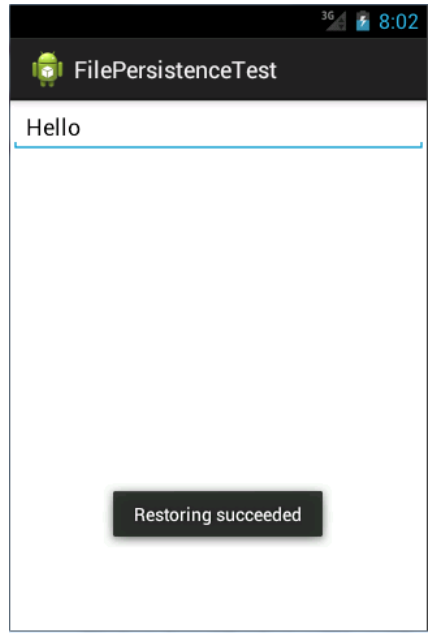

图 6.5

这样我们就已经把文件存储方面的知识学习完了，其实所用到的核心技术就是 Context 类中提供的 openFileInput()和 openFileOutput()方法，之后就是利用 Java 的各种流来进行读写操作就可以了。

不过正如我前面所说，文件存储的方式并不适合用于保存一些较为复杂的文本数据，因此，下面我们就来学习一下 Android 中另一种数据持久化的方式，它比文件存储更加简单易用，而且可以很方便地对某一指定的数据进行读写操作。

5.3 SharedPreferences 存储

不同于文件的存储方式，SharedPreferences 是使用键值对的方式来存储数据的。也就是说当保存一条数据的时候，需要给这条数据提供一个对应的键，这样在读取数据的时候就可以通过这个键把相应的值取出来。而且 SharedPreferences 还支持多种不同的数据类型存储，如果存储的数据类型是整型，那么读取出来的数据也是整型的，存储的数据是一个字符串，读取出来的数据仍然是字符串。

这样你应该就能明显地感觉到，使用 SharedPreferences 来进行数据持久化要比使用文件

方便很多，下面我们就来看一下它的具体用法吧。

5.3.1 将数据存储到 SharedPreferences 中

要想使用 SharedPreferences 来存储数据，首先需要获取到 SharedPreferences 对象。Android 中主要提供了三种方法用于得到 SharedPreferences 对象。

22. Context 类中的 getSharedPreferences()方法

此方法接收两个参数，第一个参数用于指定 SharedPreferences 文件的名称，如果指定的文件不存在则会创建一个，SharedPreferences 文件都是存放在/data/data/<package name>/shared_prefs/目录下的。第二个参数用于指定操作模式，主要有两种模式可以选择，MODE_PRIVATE 和 MODE_MULTI_PROCESS。MODE_PRIVATE 仍然是默认的操作模式，和直接传入 0 效果是相同的，表示只有当前的应用程序才可以对这个 SharedPreferences 文件进行读写。MODE_MULTI_PROCESS 则一般是用于会有多个进程中对同一个 SharedPreferences 文件进行读写的情况。类似地，MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 这两种模式已在 Android 4.2 版本中被废弃。

23. Activity 类中的 getPreferences()方法

这个方法和 Context 中的 getSharedPreferences()方法很相似，不过它只接收一个操作模式参数，因为使用这个方法时会自动将当前活动的类名作为 SharedPreferences 的文件名。

24. PreferenceManager 类中的 getDefaultSharedPreferences()方法

这是一个静态方法，它接收一个 Context 参数，并自动使用当前应用程序的包名作为前缀来命名 SharedPreferences 文件。

得到了 SharedPreferences 对象之后，就可以开始向 SharedPreferences 文件中存储数据了，主要可以分为三步实现。

25. 调用 SharedPreferences 对象的 edit()方法来获取一个 SharedPreferences.Editor 对象。
26. 向 SharedPreferences.Editor 对象中添加数据，比如添加一个布尔型数据就使用 putBoolean 方法，添加一个字符串则使用 putString()方法，以此类推。
27. 调用 commit()方法将添加的数据提交，从而完成数据存储操作。

不知不觉中已经将理论知识介绍得挺多了，那我们就赶快通过一个例子来体验一下 SharedPreferences 存储的用法吧。新建一个 SharedPreferencesTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

```
    <Button
        android:id="@+id/save_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Save data"
        />

</LinearLayout>
```

这里我们不做任何复杂的功能，只是简单地放置了一个按钮，用于将一些数据存储到SharedPreferences文件当中。然后修改MainActivity中的代码，如下所示：

```
public class MainActivity extends Activity {

    private Button saveData;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        saveData = (Button) findViewById(R.id.save_data);
        saveData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SharedPreferences.Editor editor = getSharedPreferences("data",
                        MODE_PRIVATE).edit();
                editor.putString("name", "Tom");
                editor.putInt("age", 28);
                editor.putBoolean("married", false);
                editor.commit();
            }
        });
    }

}
```

可以看到，这里首先给按钮注册了一个点击事件，然后在点击事件中通过getSharedPreferences()方法指定SharedPreferences的文件名为data，并得到了SharedPreferences.Editor对象。接着向这个对象中添加了三条不同类型的数据，最后调用commit()方法进行提交，从而完成了数据存储的操作。

很简单吧？现在就可以运行一下程序了，进入程序的主界面后，点击一下 Save data 按钮。这时的数据应该已经保存成功了，不过为了要证实一下，我们还是要借助 File Explorer 来进行查看。切换到 DDMS 视图，并点击 File Explorer 切换卡，然后进入到/data/data/com.example.sharedpreferencestest/shared_prefs /目录下，可以看到生成了一个 data.xml 文件，如图 6.6 所示。

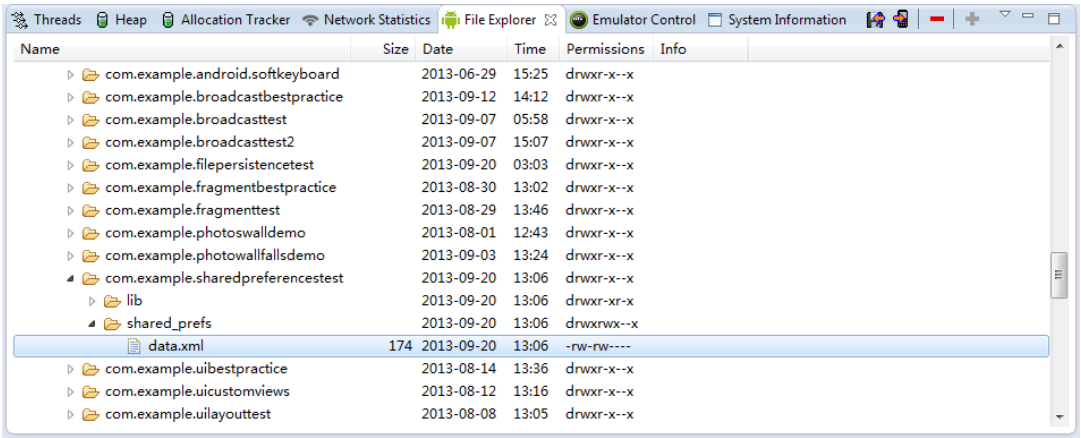

图 6.6

接下来同样是点击导出按钮将这个文件导出到电脑上，并用记事本进行查看，里面的内容如图 6.7 所示。

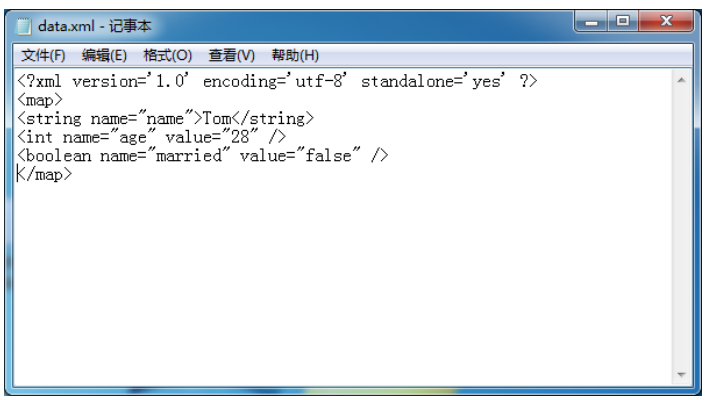

图 6.7

可以看到，我们刚刚在按钮的点击事件中添加的所有数据都已经成功保存下来了，并且 SharedPreferences 文件是使用 XML 格式来对数据进行管理的。

那么接下来我们自然要看一看，如何从 SharedPreferences 文件中去读取这些数据了。

5.3.2 从 SharedPreferences 中读取数据

你应该已经感觉到了，使用 SharedPreferences 来存储数据是非常简单的，不过下面还有更好的消息，其实从 SharedPreferences 文件中读取数据更加的简单。SharedPreferences 对象中提供了一系列的get方法用于对存储的数据进行读取，每种get方法都对应了 SharedPreferences.Editor 中的一种 put 方法，比如读取一个布尔型数据就使用 getBoolean()方法，读取一个字符串就使用 getString()方法。这些 get 方法都接收两个参数，第一个参数是键，传入存储数据时使用的键就可以得到相应的值了，第二个参数是默认值，即表示当传入的键找不到对应的值时，会以什么样的默认值进行返回。

我们还是通过例子来实际体验一下吧，仍然是在 SharedPreferencesTest 项目的基础上继续开发，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/save_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Save data"
        />

    <Button
        android:id="@+id/restore_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Restore data"
        />

</LinearLayout>
```

这里增加了一个还原数据的按钮，我们希望通过点击这个按钮来从 SharedPreferences 文件中读取数据。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private Button saveData;
```

```
    private Button restoreData;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        saveData = (Button) findViewById(R.id.save_data);
        restoreData = (Button) findViewById(R.id.restore_data);
        ......
        restoreData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SharedPreferences pref = getSharedPreferences("data",
MODE_PRIVATE);
                String name = pref.getString("name", "");
                int age = pref.getInt("age", 0);
                boolean married = pref.getBoolean("married", false);
                Log.d("MainActivity", "name is " + name);
                Log.d("MainActivity", "age is " + age);
                Log.d("MainActivity", "married is " + married);
            }
        });
    }

}
```

可以看到，我们在还原数据按钮的点击事件中首先通过 getSharedPreferences()方法得到了 SharedPreferences 对象，然后分别调用它的 getString()、getInt()和 getBoolean()方法去获取前面所存储的姓名、年龄和是否已婚，如果没有找到相应的值就会使用方法中传入的默认值来代替，最后通过 Log 将这些值打印出来。

现在重新运行一下程序，并点击界面上的 Restore data 按钮，然后查看 LogCat 中的打印信息，如图 6.8 所示。

Tag	Text
MainActivity	name is Tom
MainActivity	age is 28
MainActivity	married is false

图 6.8

所有之前存储的数据都成功读取出来了！通过这个例子，我们就把 SharedPreferences 存储的知识也学习完了。相比之下，SharedPreferences 存储确实要比文本存储简单方便了许多，应用场景也多了不少，比如很多应用程序中的偏好设置功能其实都使用到了 SharedPreferences 技术。那么下面我们就来编写一个记住密码的功能，相信通过这个例子能够加深你对 SharedPreferences 的理解。

5.3.3 实现记住密码功能

既然是实现记住密码的功能，那么我们就不需要从头去写了，因为在上一章中的最佳实践部分已经编写过一个登录界面了，有可以重用的代码为什么不用呢？那就首先打开 BroadcastBestPractice 项目，来编辑一下登录界面的布局。修改 login.xml 中的代码，如下所示：

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1" >
    ……
    <TableRow>
        <CheckBox
            android:id="@+id/remember_pass"
            android:layout_height="wrap_content" />

        <TextView
            android:layout_height="wrap_content"
            android:text="Remember password" />
    </TableRow>

    <TableRow>
        <Button
            android:id="@+id/login"
            android:layout_height="wrap_content"
            android:layout_span="2"
            android:text="Login" />
    </TableRow>
</TableLayout>
```

这里使用到了一个新控件，CheckBox。这是一个复选框控件，用户可以通过点击的方式来进行选中和取消，我们就使用这个控件来表示用户是否需要记住密码。

然后修改 LoginActivity 中的代码，如下所示：

```
public class LoginActivity extends BaseActivity {

    private SharedPreferences pref;

    private SharedPreferences.Editor editor;

    private EditText accountEdit;

    private EditText passwordEdit;

    private Button login;

    private CheckBox rememberPass;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.login);
        pref = PreferenceManager.getDefaultSharedPreferences(this);
        accountEdit = (EditText) findViewById(R.id.account);
        passwordEdit = (EditText) findViewById(R.id.password);
        rememberPass = (CheckBox) findViewById(R.id.remember_pass);
        login = (Button) findViewById(R.id.login);
        boolean isRemember = pref.getBoolean("remember_password", false);
        if (isRemember) {
            // 将账号和密码都设置到文本框中
            String account = pref.getString("account", "");
            String password = pref.getString("password", "");
            accountEdit.setText(account);
            passwordEdit.setText(password);
            rememberPass.setChecked(true);
        }
        login.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                String account = accountEdit.getText().toString();
                String password = passwordEdit.getText().toString();
                if (account.equals("admin") && password.equals("123456")) {
                    editor = pref.edit();
                    if (rememberPass.isChecked()) { // 检查复选框是否被选中
```

```
                    editor.putBoolean("remember_password", true);
                    editor.putString("account", account);
                    editor.putString("password", password);
                } else {
                    editor.clear();
                }
                editor.commit();
                Intent intent = new Intent(LoginActivity.this,
MainActivity.class);
                startActivity(intent);
                finish();
            } else {
                Toast.makeText(LoginActivity.this, "account or password
is invalid", Toast.LENGTH_SHORT).show();
            }
        }
    });
}

}
```

可以看到，这里首先在 onCreate()方法中获取到了 SharedPreferences 对象，然后调用它的 getBoolean()方法去获取 remember_password 这个键对应的值，一开始当然不存在对应的值了，所以会使用默认值 false，这样就什么都不会发生。接着在登录成功之后，会调用 CheckBox 的 isChecked()方法来检查复选框是否被选中，如果被选中了表示用户想要记住密码，这时将 remember_password 设置为 true，然后把 account 和 password 对应的值都存入到 SharedPreferences 文件当中并提交。如果没有被选中，就简单地调用一下 clear()方法，将 SharedPreferences 文件中的数据全部清除掉。

当用户选中了记住密码复选框，并成功登录一次之后，remember_password 键对应的值就是 true 了，这个时候如果再重新启动登录界面，就会从 SharedPreferences 文件中将保存的账号和密码都读取出来，并填充到文本输入框中，然后把记住密码复选框选中，这样就完成记住密码的功能了。

现在重新运行一下程序，可以看到界面上多出了一个记住密码复选框，如图 6.9 所示。

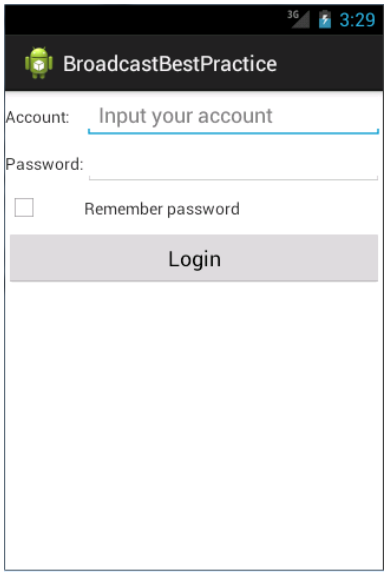

图 6.9

然后账号输入 admin，密码输入 123456，并选中记住密码复选框，点击登录，就会跳转到 MainActivity。接着在 MainActivity 中发出一条强制下线广播会让程序重新回到登录界面，此时你会发现，账号密码都已经自动填充到界面上了，如图 6.10 所示。

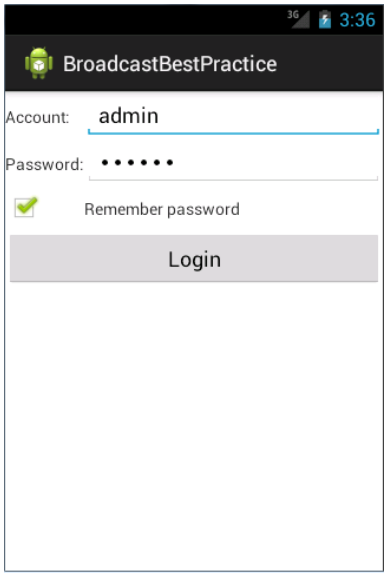

图 6.10

这样我们就使用 SharedPreferences 技术将记住密码功能成功实现了，你是不是对 SharedPreferences 理解得更加深刻了呢？

不过需要注意，这里实现的记住密码功能仍然还只是个简单的示例，并不能在实际的项目中直接使用。因为将密码以明文的形式存储在 SharedPreferences 文件中是非常不安全的，很容易就会被别人盗取，因此在正式的项目里还需要结合一定的加密算法来对密码进行保护才行。

好了，关于 SharedPreferences 的内容就讲到这里，接下来我们要学习一下本章的重头戏，Android 中的数据库技术。

5.4 SQLite 数据库存储

在刚开始接触 Android 的时候，我甚至都不敢相信，Android 系统竟然是内置了数据库的！好吧，是我太孤陋寡闻了。SQLite 是一款轻量级的关系型数据库，它的运算速度非常快，占用资源很少，通常只需要几百 K 的内存就足够了，因而特别适合在移动设备上使用。SQLite 不仅支持标准的 SQL 语法，还遵循了数据库的 ACID 事务，所以只要你以前使用过其他的关系型数据库，就可以很快地上手 SQLite。而 SQLite 又比一般的数据库要简单得多，它甚至不用设置用户名和密码就可以使用。Android 正是把这个功能极为强大的数据库嵌入到了系统当中，使得本地持久化的功能有了一次质的飞跃。

前面我们所学的文件存储和 SharedPreferences 存储毕竟只适用于去保存一些简单的数据和键值对，当需要存储大量复杂的关系型数据的时候，你就会发现以上两种存储方式很难应付得了。比如我们手机的短信程序中可能会有很多个会话，每个会话中又包含了很多条信息内容，并且大部分会话还可能各自对应了电话簿中的某个联系人。很难想象如何用文件或者 SharedPreferences 来存储这些数据量大、结构性复杂的数据吧？但是使用数据库就可以做得到。那么我们就赶快来看一看，Android 中的 SQLite 数据库到底是如何使用的。

5.4.1 创建数据库

Android 为了让我们能够更加方便地管理数据库，专门提供了一个 SQLiteOpenHelper 帮助类，借助这个类就可以非常简单地对数据库进行创建和升级。既然有好东西可以直接使用，那我们自然要尝试一下了，下面我就将对 SQLiteOpenHelper 的基本用法进行介绍。

首先你要知道 SQLiteOpenHelper 是一个抽象类，这意味着如果我们想要使用它的话，就需要创建一个自己的帮助类去继承它。SQLiteOpenHelper 中有两个抽象方法，分别是 onCreate()和 onUpgrade()，我们必须在自己的帮助类里面重写这两个方法，然后分别在这两个方法中去实现创建、升级数据库的逻辑。

SQLiteOpenHelper 中还有两个非常重要的实例方法，getReadableDatabase() 和

getWritableDatabase()。这两个方法都可以创建或打开一个现有的数据库（如果数据库已存在则直接打开，否则创建一个新的数据库），并返回一个可对数据库进行读写操作的对象。不同的是，当数据库不可写入的时候（如磁盘空间已满）getReadableDatabase()方法返回的对象将以只读的方式去打开数据库，而 getWritableDatabase()方法则将出现异常。

SQLiteOpenHelper 中有两个构造方法可供重写，一般使用参数少一点的那个构造方法即可。这个构造方法中接收四个参数，第一个参数是 Context，这个没什么好说的，必须要有它才能对数据库进行操作。第二个参数是数据库名，创建数据库时使用的就是这里指定的名称。第三个参数允许我们在查询数据的时候返回一个自定义的 Cursor，一般都是传入 null。第四个参数表示当前数据库的版本号，可用于对数据库进行升级操作。构建出 SQLiteOpenHelper 的实例之后，再调用它的 getReadableDatabase()或 getWritableDatabase()方法就能够创建数据库了，数据库文件会存放在/data/data/<package name>/databases/目录下。此时，重写的 onCreate()方法也会得到执行，所以通常会在这里去处理一些创建表的逻辑。

接下来还是让我们通过例子的方式来更加直观地体会 SQLiteOpenHelper 的用法吧，首先新建一个 DatabaseTest 项目。

这里我们希望创建一个名为 BookStore.db 的数据库，然后在这个数据库中新建一张 Book 表，表中有 id（主键）、作者、价格、页数和书名等列。创建数据库表当然还是需要用建表语句的，这里也是要考验一下你的 SQL 基本功了，Book 表的建表语句如下所示：

```
create table Book (
    id integer primary key autoincrement,
    author text,
    price real,
    pages integer,
    name text)
```

只要你对 SQL 方面的知识稍微有一些了解，上面的建表语句对你来说应该都不难吧。SQLite 不像其他的数据库拥有众多繁杂的数据类型，它的数据类型很简单，integer 表示整型，real 表示浮点型，text 表示文本类型，blob 表示二进制类型。另外，上述建表语句中我们还使用了 primary key 将 id 列设为主键，并用 autoincrement 关键字表示 id 列是自增长的。

然后需要在代码中去执行这条 SQL 语句，才能完成创建表的操作。新建 MyDatabaseHelper 类继承自 SQLiteOpenHelper，代码如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {

    public static final String CREATE_BOOK = "create table book ("
            + "id integer primary key autoincrement, "
            + "author text, "
            + "price real, "
```

```
            + "pages integer, "
            + "name text)";

    private Context mContext;

    public MyDatabaseHelper(Context context, String name, CursorFactory
factory, int version) {
        super(context, name, factory, version);
        mContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK);
        Toast.makeText(mContext, "Create succeeded", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }

}
```

可以看到，我们把建表语句定义成了一个字符串常量，然后在 onCreate()方法中又调用了 SQLiteDatabase 的 execSQL()方法去执行这条建表语句，并弹出一个 Toast 提示创建成功，这样就可以保证在数据库创建完成的同时还能成功创建 Book 表。

现在修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/create_database"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Create database"
         />
```

```
</LinearLayout>
```

布局文件很简单，就是加入了一个按钮，用于创建数据库。最后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 1);
        Button createDatabase = (Button) findViewById(R.id.create_database);
        createDatabase.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                dbHelper.getWritableDatabase();
            }
        });
    }

}
```

这里我们在 onCreate()方法中构建了一个 MyDatabaseHelper 对象，并且通过构造函数的参数将数据库名指定为 BookStore.db，版本号指定为 1，然后在 Create database 按钮的点击事件里调用了 getWritableDatabase()方法。这样当第一次点击 Create database 按钮时，就会检测到当前程序中并没有 BookStore.db 这个数据库，于是会创建该数据库并调用 MyDatabaseHelper 中的 onCreate()方法，这样 Book 表也就得到了创建，然后会弹出一个 Toast 提示创建成功。再次点击 Create database 按钮时，会发现此时已经存在 BookStore.db 数据库了，因此不会再创建一次。

现在就可以运行一下代码了，在程序主界面点击 Create database 按钮，结果如图 6.11 所示。

图 6.11

此时 BookStore.db 数据库和 Book 表应该都已经创建成功了，因为当你再次点击 Create database 按钮时不会再有 Toast 弹出。可是又回到了之前的那个老问题，怎样才能证实它们的确是创建成功了？如果还是使用 File Explorer，那么最多你只能看到 databases 目录下出现了一个 BookStore.db 文件，Book 表是无法通过 File Explorer 看到的。

5.4.2 升级数据库

如果你足够细心，一定会发现 MyDatabaseHelper 中还有一个空方法呢！没错，onUpgrade() 方法是用于对数据库进行升级的，它在整个数据库的管理工作当中起着非常重要的作用，可千万不能忽视它哟。

目前 DatabaseTest 项目中已经有一张 Book 表用于存放书的各种详细数据，如果我们想再添加一张 Category 表用于记录书籍的分类该怎么做呢？

比如 Category 表中有 id（主键）、分类名和分类代码这几个列，那么建表语句就可以写成：

```
create table Category (
    id integer primary key autoincrement,
    category_name text,
    category_code integer)
```

接下来我们将这条建表语句添加到 MyDatabaseHelper 中，代码如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {

    public static final String CREATE_BOOK = "create table Book ("
            + "id integer primary key autoincrement, "
            + "author text, "
            + "price real, "
            + "pages integer, "
            + "name text)";

    public static final String CREATE_CATEGORY = "create table Category ("
            + "id integer primary key autoincrement, "
            + "category_name text, "
            + "category_code integer)";

    private Context mContext;

    public MyDatabaseHelper(Context context, String name,
            CursorFactory factory, int version) {
        super(context, name, factory, version);
        mContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK);
        db.execSQL(CREATE_CATEGORY);
        Toast.makeText(mContext, "Create succeeded", Toast.LENGTH_SHORT).
show();
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }

}
```

看上去好像都挺对的吧，现在我们重新运行一下程序，并点击 Create database 按钮，咦？竟然没有弹出创建成功的提示。当然，你也可以通过 adb 工具到数据库中再去检查一下，这

样你会更加地确认，Category 表没有创建成功！

其实没有创建成功的原因不难思考，因为此时 BookStore.db 数据库已经存在了，之后不管我们怎样点击 Create database 按钮，MyDatabaseHelper 中的 onCreate()方法都不会再次执行，因此新添加的表也就无法得到创建了。

解决这个问题的办法也相当简单，只需要先将程序卸载掉，然后重新运行，这时 BookStore.db 数据库已经不存在了，如果再点击 Create database 按钮，MyDatabaseHelper 中的 onCreate()方法就会执行，这时 Category 表就可以创建成功了。

不过通过卸载程序的方式来新增一张表毫无疑问是很极端的做法，其实我们只需要巧妙地运用 SQLiteOpenHelper 的升级功能就可以很轻松地解决这个问题。修改 MyDatabaseHelper 中的代码，如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {
    ……
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("drop table if exists Book");
        db.execSQL("drop table if exists Category");
        onCreate(db);
    }
}
```

可以看到，我们在 onUpgrade()方法中执行了两条 DROP 语句，如果发现数据库中已经存在 Book 表或 Category 表了，就将这两张表删除掉，然后再调用 onCreate()方法去重新创建。这里先将已经存在的表删除掉，是因为如果在创建表时发现这张表已经存在了，就会直接报错。

接下来的问题就是如何让 onUpgrade()方法能够执行了，还记得 SQLiteOpenHelper 的构造方法里接收的第四个参数吗？它表示当前数据库的版本号，之前我们传入的是 1，现在只要传入一个比 1 大的数，就可以让 onUpgrade()方法得到执行了。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
```

```
        Button createDatabase = (Button) findViewById(R.id.create_database);
        createDatabase.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                dbHelper.getWritableDatabase();
            }
        });
    }

}
```

这里将数据库版本号指定为 2，表示我们对数据库进行升级了。现在重新运行程序，并点击 Create database 按钮，这时就会再次弹出创建成功的提示。为了验证一下 Category 表是不是已经创建成功了，我们在 adb shell 中打开 BookStore.db 数据库，然后键入.table 命令，结果如图 6.19 所示。

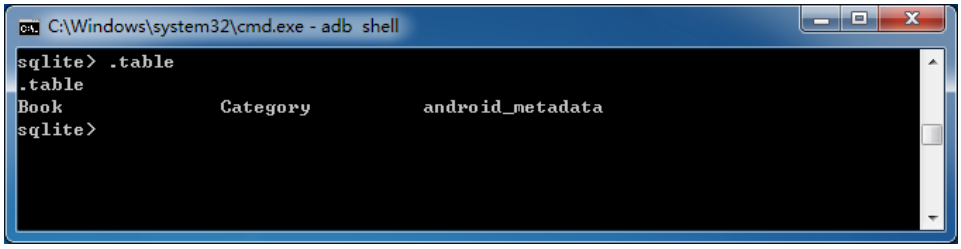

图 6.19

接着键入.schema 命令查看一下建表语句，结果如图 6.20 所示。

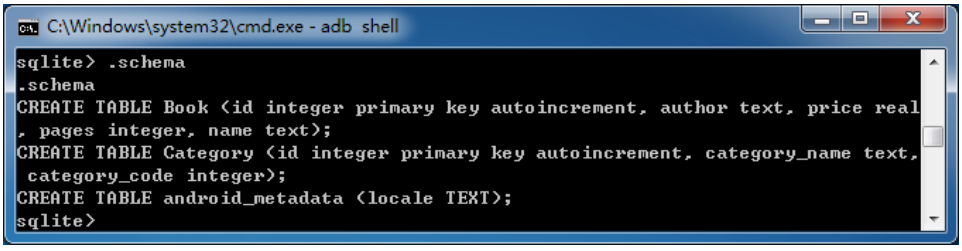

图 6.20

由此可以看出，Category 表已经创建成功了，同时也说明我们的升级功能的确起到了作用。

5.4.3 添加数据

现在你已经掌握了创建和升级数据库的方法，接下来就该学习一下如何对表中的数据进行操作了。其实我们可以对数据进行的操作也就无非四种，即 CRUD。其中 C 代表添加（Create），R 代表查询（Retrieve），U 代表更新（Update），D 代表删除（Delete）。每一种操作又各自对应了一种 SQL 命令，如果你比较熟悉 SQL 语言的话，一定会知道添加数据时使用 insert，查询数据时使用 select，更新数据时使用 update，删除数据时使用 delete。但是开发者的水平总会是参差不齐的，未必每一个人都能非常熟悉地使用 SQL 语言，因此 Android 也是提供了一系列的辅助性方法，使得在 Android 中即使不去编写 SQL 语句，也能轻松完成所有的 CRUD 操作。

前面我们已经知道，调用 SQLiteOpenHelper 的 getReadableDatabase()或 getWritableDatabase()方法是可以用于创建和升级数据库的，不仅如此，这两个方法还都会返回一个 SQLiteDatabase 对象，借助这个对象就可以对数据进行 CRUD 操作了。

那么我们一个一个功能地看，首先学习一下如何向数据库的表中添加数据吧。SQLiteDatabase 中提供了一个 insert()方法，这个方法就是专门用于添加数据的。它接收三个参数，第一个参数是表名，我们希望向哪张表里添加数据，这里就传入该表的名字。第二个参数用于在未指定添加数据的情况下给某些可为空的列自动赋值 NULL，一般我们用不到这个功能，直接传入 null 即可。第三个参数是一个 ContentValues 对象，它提供了一系列的 put()方法重载，用于向 ContentValues 中添加数据，只需要将表中的每个列名以及相应的待添加数据传入即可。

介绍完了基本用法，接下来还是让我们通过例子的方式来亲身体验一下如何添加数据吧。修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <Button
        android:id="@+id/add_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add data"
        />
</LinearLayout>
```

可以看到，我们在布局文件中又新增了一个按钮，稍后就会在这个按钮的点击事件里编写添加数据的逻辑。接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ……
        Button addData = (Button) findViewById(R.id.add_data);
        addData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                ContentValues values = new ContentValues();
                // 开始组装第一条数据
                values.put("name", "The Da Vinci Code");
                values.put("author", "Dan Brown");
                values.put("pages", 454);
                values.put("price", 16.96);
                db.insert("Book", null, values); // 插入第一条数据
                values.clear();
                // 开始组装第二条数据
                values.put("name", "The Lost Symbol");
                values.put("author", "Dan Brown");
                values.put("pages", 510);
                values.put("price", 19.95);
                db.insert("Book", null, values); // 插入第二条数据
            }
        });
    }

}
```

在添加数据按钮的点击事件里面，我们先获取到了 SQLiteDatabase 对象，然后使用 ContentValues 来对要添加的数据进行组装。如果你比较细心的话应该会发现，这里只对 Book

表里其中四列的数据进行了组装，id 那一列没并没给它赋值。这是因为在前面创建表的时候我们就将 id 列设置为自增长了，它的值会在入库的时候自动生成，所以不需要手动给它赋值了。接下来调用了 insert()方法将数据添加到表当中，注意这里我们实际上添加了两条数据，上述代码中使用 ContentValues 分别组装了两次不同的内容，并调用了两次 insert()方法。

好了，现在可以重新运行一下程序了，界面如图 6.21 所示。

图 6.21

点击一下 Add data 按钮，此时两条数据应该都已经添加成功了，不过为了证实一下，我们还是打开 BookStore.db 数据库瞧一瞧。输入 SQL 查询语句 select * from Book，结果如图 6.22 所示。

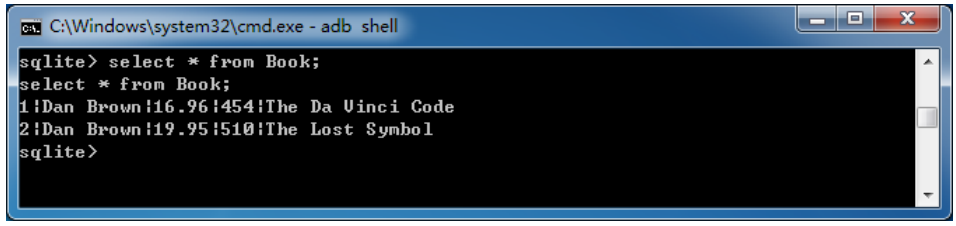

图 6.22

由此可以看出，我们刚刚组装的两条数据，都已经准确无误地添加到 Book 表中了。

5.4.4 更新数据

学习完了如何向表中添加数据，接下来我们看看怎样才能修改表中已有的数据。SQLiteDatabase 中也是提供了一个非常好用的 update()方法用于对数据进行更新，这个方法接收四个参数，第一个参数和 insert()方法一样，也是表名，在这里指定去更新哪张表里的数据。第二个参数是 ContentValues 对象，要把更新数据在这里组装进去。第三、第四个参数用于去约束更新某一行或某几行中的数据，不指定的话默认就是更新所有行。

那么接下来我们仍然是在 DatabaseTest 项目的基础上修改，看一下更新数据的具体用法。比如说刚才添加到数据库里的第一本书，由于过了畅销季，卖得不是很火了，现在需要通过降低价格的方式来吸引更多的顾客，我们应该怎么操作呢？首先修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <Button
        android:id="@+id/update_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update data"
        />
</LinearLayout>
```

布局文件中的代码就已经非常简单了，就是添加了一个用于更新数据的按钮。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ......
```

```
        Button updateData = (Button) findViewById(R.id.update_data);
        updateData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                ContentValues values = new ContentValues();
                values.put("price", 10.99);
                db.update("Book", values, "name = ?", new String[] { "The Da
Vinci Code" });
            }
        });
    }

}
```

这里在更新数据按钮的点击事件里面构建了一个 ContentValues 对象，并且只给它指定了一组数据，说明我们只是想把价格这一列的数据更新成 10.99。然后调用了 SQLiteDatabase 的 update()方法去执行具体的更新操作，可以看到，这里使用了第三、第四个参数来指定具体更新哪几行。第三个参数对应的是 SQL 语句的 where 部分，表示去更新所有 name 等于?的行，而?是一个占位符，可以通过第四个参数提供的一个字符串数组为第三个参数中的每个占位符指定相应的内容。因此上述代码想表达的意图就是，将名字是 The Da Vinci Code 的这本书的价格改成 10.99。

现在重新运行一下程序，界面如图 6.23 所示。

图 6.23

点击一下 Update data 按钮后，再次输入查询语句查看表中的数据情况，结果如图 6.24 所示。

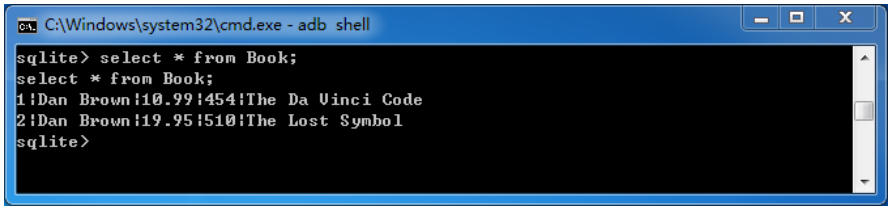

图 6.24

可以看到，The Da Vinci Code 这本书的价格已经被成功改为 10.99 了。

5.4.5 删除数据

怎么样？添加和更新数据的功能都还挺简单的吧，代码也不多，理解起来又容易，那么我们要马不停蹄地开始学习下一种操作了，即如何从表中删除数据。

删除数据对你来说应该就更简单了，因为它所需要用到的知识点你全部已经学过了。SQLiteDatabase 中提供了一个 delete()方法专门用于删除数据，这个方法接收三个参数，第一个参数仍然是表名，这个已经没什么好说的了，第二、第三个参数又是用于去约束删除某一行或某几行的数据，不指定的话默认就是删除所有行。

是不是理解起来很轻松了？那我们就继续动手实践吧，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <Button
        android:id="@+id/delete_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Delete data"
        />
</LinearLayout>
```

仍然是在布局文件中添加了一个按钮，用于删除数据。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ......
        Button deleteButton = (Button) findViewById(R.id.delete_data);
        deleteButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                db.delete("Book", "pages > ?", new String[] { "500" });
            }
        });
    }
```

```
}
```

可以看到，我们在删除按钮的点击事件里指明去删除 Book 表中的数据，并且通过第二、第三个参数来指定仅删除那些页数超过 500 页的书籍。当然这个需求很奇怪，这里也仅仅是为了做个测试。你可以先查看一下当前 Book 表里的数据，其中 The Lost Symbol 这本书的页数超过了 500 页，也就是说当我们点击删除按钮时，这条记录应该会被删除掉。

现在重新运行一下程序，界面如图 6.25 所示。

图　6.25

点击一下 Delete data 按钮后，再次输入查询语句查看表中的数据情况，结果如图 6.26 所示。

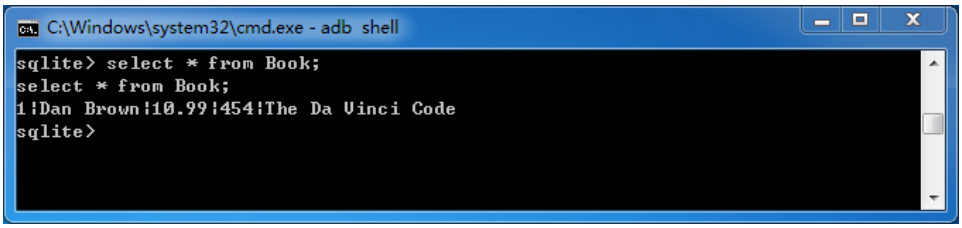

图　6.26

这样就可以明显地看出，The Lost Symbol 这本书的数据已经被删除了。

5.4.6　查询数据

终于到了最后一种操作了，掌握了查询数据的方法之后，你也就将数据库的 CRUD 操作全部学完了。不过千万不要因此而放松，因为查询数据也是在 CRUD 中最复杂的一种

操作。

我们都知道 SQL 的全称是 Structured Query Language，翻译成中文就是结构化查询语言。它的大部功能都是体现在“查”这个字上的，而“增删改”只是其中的一小部分功能。由于 SQL 查询涉及的内容实在是太多了，因此在这里我不准备对它展开来讲解，而是只会介绍 Android 上的查询功能。如果你对 SQL 语言非常感兴趣，可以找一本专门介绍 SQL 的书进行学习。

相信你已经猜到了，SQLiteDatabase 中还提供了一个 query()方法用于对数据进行查询。这个方法的参数非常复杂，最短的一个方法重载也需要传入七个参数。那我们就先来看一下这七个参数各自的含义吧，第一个参数不用说，当然还是表名，表示我们希望从哪张表中查询数据。第二个参数用于指定去查询哪几列，如果不指定则默认查询所有列。第三、第四个参数用于去约束查询某一行或某几行的数据，不指定则默认是查询所有行的数据。第五个参数用于指定需要去 group by 的列，不指定则表示不对查询结果进行 group by 操作。第六个参数用于对 group by 之后的数据进行进一步的过滤，不指定则表示不进行过滤。第七个参数用于指定查询结果的排序方式，不指定则表示使用默认的排序方式。更多详细的内容可以参考下表。其他几个 query()方法的重载其实也大同小异，你可以自己去研究一下，这里就不再进行介绍了。

query()方法参数	对应 SQL 部分	描述
table	from table_name	指定查询的表名
columns	select column1, column2	指定查询的列名
selection	where column = value	指定 where 的约束条件
selectionArgs	-	为 where 中的占位符提供具体的值
groupBy	group by column	指定需要 group by 的列
having	having column = value	对 group by 后的结果进一步约束
orderBy	order by column1, column2	指定查询结果的排序方式

虽然 query()方法的参数非常多，但是不要对它产生畏惧，因为我们不必为每条查询语句都指定上所有的参数，多数情况下只需要传入少数几个参数就可以完成查询操作了。调用 query()方法后会返回一个 Cursor 对象，查询到的所有数据都将从这个对象中取出。

下面还是让我们通过例子的方式来体验一下查询数据的具体用法，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    ......

    <Button
        android:id="@+id/query_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Query data"
        />
</LinearLayout>
```

这个已经没什么好说的了，添加了一个按钮用于查询数据。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ......
        Button queryButton = (Button) findViewById(R.id.query_data);
        queryButton.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                // 查询Book表中所有的数据
                Cursor cursor = db.query("Book", null, null, null, null, null, null);
                if (cursor.moveToFirst()) {
                    do {
                        // 遍历Cursor对象，取出数据并打印
                        String name = cursor.getString(cursor.
getColumnIndex("name"));
                        String author = cursor.getString(cursor.
```

```
getColumnIndex("author"));
                        int pages = cursor.getInt(cursor.getColumnIndex
("pages"));
                        double price = cursor.getDouble(cursor.
getColumnIndex("price"));
                        Log.d("MainActivity", "book name is " + name);
                        Log.d("MainActivity", "book author is " + author);
                        Log.d("MainActivity", "book pages is " + pages);
                        Log.d("MainActivity", "book price is " + price);
                    } while (cursor.moveToNext());
                }
                cursor.close();
            }
        });
    }

}
```

可以看到，我们首先在查询按钮的点击事件里面调用了 SQLiteDatabase 的 query()方法去查询数据。这里的 query()方法非常简单，只是使用了第一个参数指明去查询 Book 表，后面的参数全部为 null。这就表示希望查询这张表中的所有数据，虽然这张表中目前只剩下一条数据了。查询完之后就得到了一个 Cursor 对象，接着我们调用它的 moveToFirst()方法将数据的指针移动到第一行的位置，然后进入了一个循环当中，去遍历查询到的每一行数据。在这个循环中可以通过 Cursor 的 getColumnIndex()方法获取到某一列在表中对应的位置索引，然后将这个索引传入到相应的取值方法中，就可以得到从数据库中读取到的数据了。接着我们使用 Log 的方式将取出的数据打印出来，借此来检查一下读取工作有没有成功完成。最后别忘了调用 close()方法来关闭 Cursor。

好了，现在再次重新运行程序，界面如图 6.27 所示。

图 6.27

点击一下 Query data 按钮后，查看 LogCat 的打印内容，结果如图 6.28 所示。

Tag	Text
MainActivity	book name is The Da Vinci Code
MainActivity	book author is Dan Brown
MainActivity	book pages is 454
MainActivity	book price is 10.99

图 6.28

可以看到，这里已经将 Book 表中唯一的一条数据成功地读取出来了。

当然这个例子只是对查询数据的用法进行了最简单的示范，在真正的项目中你可能会遇到比这要复杂得多的查询功能，更多高级的用法还需要你自己去慢慢摸索，毕竟 query()方法中还有那么多的参数我们都还没用到呢。

5.4.7 使用 SQL 操作数据库

虽然 Android 已经给我们提供了很多非常方便的 API 用于操作数据库，不过总会有一些人不习惯去使用这些辅助性的方法，而是更加青睐于直接使用 SQL 来操作数据库。这种人

一般都是属于 SQL 大牛，如果你也是其中之一的话，那么恭喜，Android 充分考虑到了你们的编程习惯，同样提供了一系列的方法，使得可以直接通过 SQL 来操作数据库。

下面我就来简略演示一下，如何直接使用 SQL 来完成前面几小节中学过的 CRUD 操作。

添加数据的方法如下：

```
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",
            new String[] { "The Da Vinci Code", "Dan Brown", "454", "16.96" });
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",
            new String[] { "The Lost Symbol", "Dan Brown", "510", "19.95" });
```

更新数据的方法如下：

```
db.execSQL("update Book set price = ? where name = ?", new String[] { "10.99",
"The Da Vinci Code" });
```

删除数据的方法如下：

```
db.execSQL("delete from Book where pages > ?", new String[] { "500" });
```

查询数据的方法如下：

```
db.rawQuery("select * from Book", null);
```

可以看到，除了查询数据的时候调用的是 SQLiteDatabase 的 rawQuery()方法，其他的操作都是调用的 execSQL()方法。以上演示的几种方式，执行结果会和前面几小节中我们学习的 CRUD 操作的结果完全相同，选择使用哪一种方式就看你个人的喜好了。

第 6 章　探究内容提供器

在上一章中我们学了 Android 数据持久化的技术，包括文件存储、SharedPreferences 存储、以及数据库存储。不知道你有没有发现，使用这些持久化技术所保存的数据都只能在当前应用程序中访问。虽然文件和 SharedPreferences 存储中提供了 MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 这两种操作模式，用于供给其他的应用程序访问当前应用的数据，但这两种模式在 Android 4.2 版本中都已被废弃了。为什么呢？因为 Android 官方已经不再推荐使用这种方式来实现跨程序数据共享的功能，而是应该使用更加安全可靠的内容提供器技术。

可能你会有些疑惑，为什么要将我们程序中的数据共享给其他程序呢？当然，这个要视情况而定的，比如说账号和密码这样的隐私数据显然是不能共享给其他程序的，不过一些可以让其他程序进行二次开发的基础性数据，我们还是可以选择将其共享的。例如系统的电话簿程序，它的数据库中保存了很多的联系人信息，如果这些数据都不允许第三方的程序进行访问的话，恐怕很多应用的功能都要大打折扣了。除了电话簿之外，还有短信、媒体库等程序都实现了跨程序数据共享的功能，而使用的技术当然就是内容提供器了，下面我们就来对这一技术进行深入的探讨。

6.1　内容提供器简介

内容提供器（Content Provider）主要用于在不同的应用程序之间实现数据共享的功能，它提供了一套完整的机制，允许一个程序访问另一个程序中的数据，同时还能保证被访数据的安全性。目前，使用内容提供器是 Android 实现跨程序共享数据的标准方式。

不同于文件存储和 SharedPreferences 存储中的两种全局可读写操作模式，内容提供器可以选择只对哪一部分数据进行共享，从而保证我们程序中的隐私数据不会有泄漏的风险。

内容提供器的用法一般有两种，一种是使用现有的内容提供器来读取和操作相应程序中的数据，另一种是创建自己的内容提供器给我们程序的数据提供外部访问接口。那么接下来我们就一个一个开始学习吧，首先从使用现有的内容提供器开始。

6.2　访问其他程序中的数据

当一个应用程序通过内容提供器对其数据提供了外部访问接口，任何其他的应用程序就

都可以对这部分数据进行访问。Android 系统中自带的电话簿、短信、媒体库等程序都提供了类似的访问接口，这就使得第三方应用程序可以充分地利用这部分数据来实现更好的功能。下面我们就来看一看，内容提供器到底是如何使用的。

6.2.1 ContentResolver 的基本用法

对于每一个应用程序来说，如果想要访问内容提供器中共享的数据，就一定要借助 ContentResolve 类，可以通过 Context 中的 getContentResolver()方法获取到该类的实例。ContentResolver 中提供了一系列的方法用于对数据进行 CRUD 操作，其中 insert()方法用于添加数据，update()方法用于更新数据，delete()方法用于删除数据，query()方法用于查询数据。有没有似曾相识的感觉？没错，SQLiteDatabase 中也是使用的这几个方法来进行 CRUD 操作的，只不过它们在方法参数上稍微有一些区别。

不同于 SQLiteDatabase，ContentResolver 中的增删改查方法都是不接收表名参数的，而是使用一个 Uri 参数代替，这个参数被称为内容 URI。内容 URI 给内容提供器中的数据建立了唯一标识符，它主要由两部分组成，权限（authority）和路径（path）。权限是用于对不同的应用程序做区分的，一般为了避免冲突，都会采用程序包名的方式来进行命名。比如某个程序的包名是 com.example.app，那么该程序对应的权限就可以命名为 com.example.app.provider。路径则是用于对同一应用程序中不同的表做区分的，通常都会添加到权限的后面。比如某个程序的数据库里存在两张表，table1 和 table2，这时就可以将路径分别命名为/table1 和/table2，然后把权限和路径进行组合，内容 URI 就变成了 com.example.app.provider/table1 和 com.example.app.provider/table2。不过，目前还很难辨认出这两个字符串就是两个内容 URI，我们还需要在字符串的头部加上协议声明。因此，内容 URI 最标准的格式写法如下：

```
content://com.example.app.provider/table1
content://com.example.app.provider/table2
```

有没有发现，内容 URI 可以非常清楚地表达出我们想要访问哪个程序中哪张表里的数据。也正是因此，ContentResolver 中的增删改查方法才都接收 Uri 对象作为参数，因为使用表名的话系统将无法得知我们期望访问的是哪个应用程序里的表。

在得到了内容 URI 字符串之后，我们还需要将它解析成 Uri 对象才可以作为参数传入。解析的方法也相当简单，代码如下所示：

```
Uri uri = Uri.parse("content://com.example.app.provider/table1")
```

只需要调用 Uri.parse()方法，就可以将内容 URI 字符串解析成 Uri 对象了。

现在我们就可以使用这个 Uri 对象来查询 table1 表中的数据了，代码如下所示：

```
Cursor cursor = getContentResolver().query(
    uri,
```

```
        projection,
        selection,
        selectionArgs,
        sortOrder);
```

这些参数和 SQLiteDatabase 中 query()方法里的参数很像，但总体来说要简单一些，毕竟这是在访问其他程序中的数据，没必要构建过于复杂的查询语句。下表对使用到的这部分参数进行了详细的解释。

query()方法参数	对应 SQL 部分	描述
uri	from table_name	指定查询某个应用程序下的某一张表
projection	select column1, column2	指定查询的列名
selection	where column = value	指定 where 的约束条件
selectionArgs	-	为 where 中的占位符提供具体的值
orderBy	order by column1, column2	指定查询结果的排序方式

查询完成后返回的仍然是一个 Cursor 对象，这时我们就可以将数据从 Cursor 对象中逐个读取出来了。读取的思路仍然是通过移动游标的位置来遍历 Cursor 的所有行，然后再取出每一行中相应列的数据，代码如下所示：

```
if (cursor != null) {
    while (cursor.moveToNext()) {
        String column1 = cursor.getString(cursor.getColumnIndex("column1"));
        int column2 = cursor.getInt(cursor.getColumnIndex("column2"));
    }
    cursor.close();
}
```

掌握了最难的查询操作，剩下的增加、修改、删除操作就更不在话下了。我们先来看看如何向 table1 表中添加一条数据，代码如下所示：

```
ContentValues values = new ContentValues();
values.put("column1", "text");
values.put("column2", 1);
getContentResolver().insert(uri, values);
```

可以看到，仍然是将待添加的数据组装到 ContentValues 中，然后调用 ContentResolver 的 insert()方法，将 Uri 和 ContentValues 作为参数传入即可。

现在如果我们想要更新这条新添加的数据，把 column1 的值清空，可以借助 ContentResolver 的 update()方法实现，代码如下所示：

```
    ContentValues values = new ContentValues();
    values.put("column1", "");
    getContentResolver().update(uri, values, "column1 = ? and column2 = ?", new
String[] {"text", "1"});
```

注意上述代码使用了 selection 和 selectionArgs 参数来对想要更新的数据进行约束，以防止所有的行都会受影响。

最后，可以调用 ContentResolver 的 delete()方法将这条数据删除掉，代码如下所示：

```
getContentResolver().delete(uri, "column2 = ?", new String[] { "1" });
```

到这里为止，我们就把 ContentResolver 中的增删改查方法全部学完了。是不是感觉非常简单？因为这些知识早在上一章中学习 SQLiteDatabase 的时候你就已经掌握了，所需特别注意的就只有 uri 这个参数而已。那么接下来，我们就利用目前所学的知识，看一看如何读取系统电话簿中的联系人信息。

6.2.2 读取系统联系人

由于我们之前一直使用的都是模拟器，电话簿里面并没有联系人存在，所以现在需要自己手动添加几个，以便稍后进行读取。打开电话簿程序，界面如图 6.1 所示。

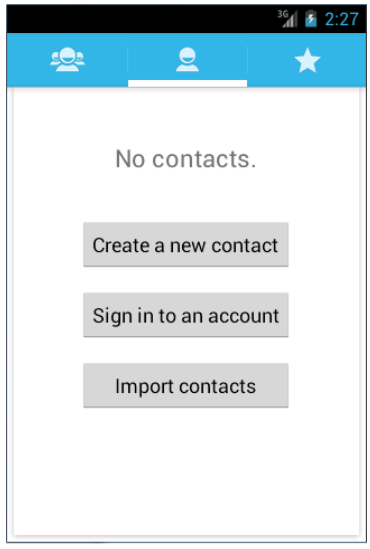

图 6.1

可以看到，目前电话簿里是没有任何联系人的，我们可以通过点击 Create a new contact 按钮来对联系人进行创建。这里就先创建两个联系人吧，分别填入他们的姓名和手机号，如

图 6.2 所示。

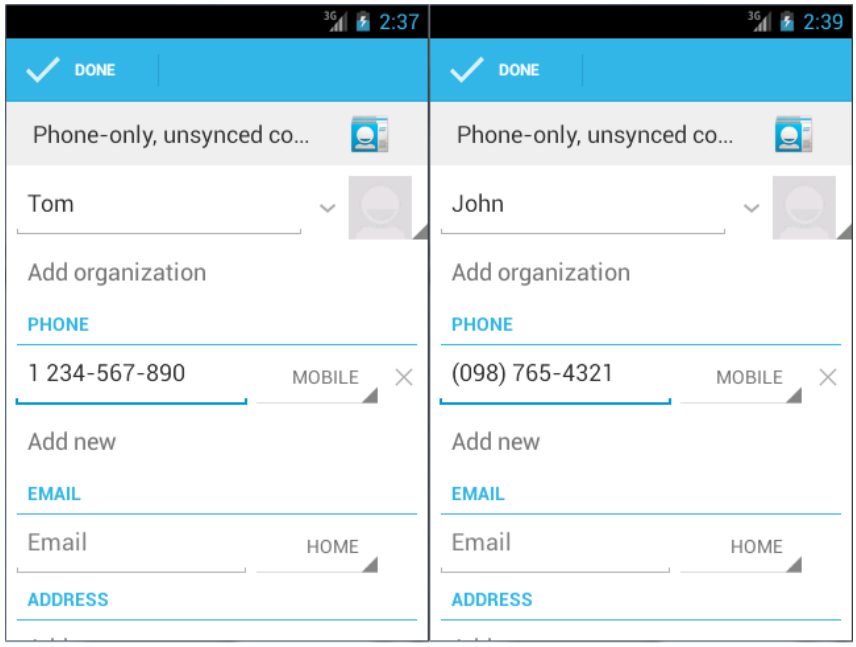

图 6.2

这样准备工作就做好了，现在新建一个 ContactsTest 项目，让我们开始动手吧。

首先还是来编写一下布局文件，这里我们希望读取出来的联系人信息能够在 ListView 中显示，因此，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ListView
        android:id="@+id/contacts_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" >
    </ListView>

</LinearLayout>
```

简单起见，LinearLayout 里就只放置了一个 ListView。接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    ListView contactsView;

    ArrayAdapter<String> adapter;

    List<String> contactsList = new ArrayList<String>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        contactsView = (ListView) findViewById(R.id.contacts_view);
        adapter = new ArrayAdapter<String>(this, android.R.layout.
simple_list_item_1, contactsList);
        contactsView.setAdapter(adapter);
        readContacts();
    }

    private void readContacts() {
        Cursor cursor = null;
        try {
            // 查询联系人数据
            cursor = getContentResolver().query(
                    ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                    null, null, null, null);
            while (cursor.moveToNext()) {
                // 获取联系人姓名
                String displayName = cursor.getString(cursor.getColumnIndex(
                        ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
                // 获取联系人手机号
                String number = cursor.getString(cursor.getColumnIndex(
                        ContactsContract.CommonDataKinds.Phone.NUMBER));
                contactsList.add(displayName + "\n" + number);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (cursor != null) {
                cursor.close();
```

```
                }
            }
        }

    }
```

在 onCreate()方法中，我们首先获取了 ListView 控件的实例，并给它设置好了适配器，然后就去调用 readContacts()方法。下面重点看下 readContacts()方法，可以看到，这里使用了 ContentResolver 的 query()方法来查询系统的联系人数据。不过传入的 Uri 参数怎么有些奇怪啊，为什么没有调用 Uri.parse()方法去解析一个内容 URI 字符串呢？这是因为 ContactsContract.CommonDataKinds.Phone 类已经帮我们做好了封装，提供了一个 CONTENT_URI 常量，而这个常量就是使用 Uri.parse()方法解析出来的结果。接着我们对 Cursor 对象进行遍历，将联系人姓名和手机号这些数据逐个取出，联系人姓名这一列对应的常量是 ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME，联系人手机号这一列对应的常量是 ContactsContract.CommonDataKinds.Phone.NUMBER。两个数据都取出之后，将它们进行拼接，并且中间加上换行符，然后将拼接后的数据添加到 ListView 里。最后千万不要忘记将 Cursor 对象关闭掉。

这样就结束了吗？还差一点点，读取系统联系人也是需要声明权限的，因此修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.contactstest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <uses-permission android:name="android.permission.READ_CONTACTS" />
    ……
</manifest>
```

加入了 android.permission.READ_CONTACTS 权限，这样我们的程序就可以访问到系统的联系人数据了。现在才算是大功告成，让我们来运行一下程序吧，效果如图 6.3 所示。

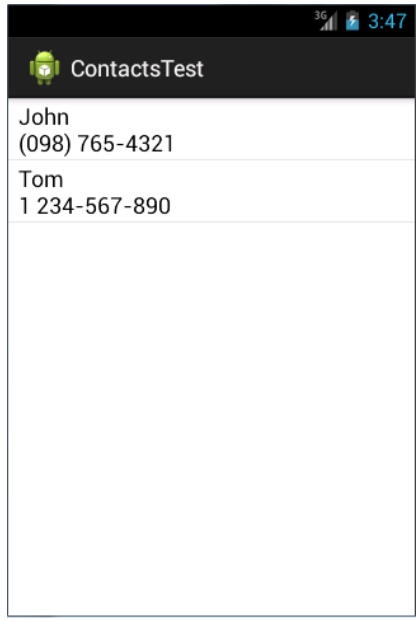

图 6.3

刚刚添加的两个联系人的数据都成功读取出来了！说明跨程序访问数据的功能确实是实现了。

6.3 创建自己的内容提供器

在上一节当中，我们学习了如何在自己的程序中访问其他应用程序的数据。总体来说思路还是非常简单的，只需要获取到该应用程序的内容 URI，然后借助 ContentResolver 进行 CRUD 操作就可以了。可是你有没有想过，那些提供外部访问接口的应用程序都是如何实现这种功能的呢？它们又是怎样保证数据的安全性，使得隐私数据不会泄漏出去？学习完本节的知识后，你的疑惑将会被一一解开。

6.3.1 创建内容提供器的步骤

前面已经提到过，如果想要实现跨程序共享数据的功能，官方推荐的方式就是使用内容提供器，可以通过新建一个类去继承 ContentProvider 的方式来创建一个自己的内容提供器。ContentProvider 类中有六个抽象方法，我们在使用子类继承它的时候，需要将这六个方法全部重写。新建 MyProvider 继承自 ContentProvider，代码如下所示：

```
public class MyProvider extends ContentProvider {

    @Override
    public boolean onCreate() {
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
        return 0;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }

    @Override
    public String getType(Uri uri) {
        return null;
    }

}
```

在这六个方法中，相信大多数你都已经非常熟悉了，我再来简单介绍一下吧。

28. onCreate()

初始化内容提供器的时候调用。通常会在这里完成对数据库的创建和升级等操作，返回 true 表示内容提供器初始化成功，返回 false 则表示失败。注意，只有当存在

ContentResolver 尝试访问我们程序中的数据时，内容提供器才会被初始化。

29. query()

从内容提供器中查询数据。使用 uri 参数来确定查询哪张表，projection 参数用于确定查询哪些列，selection 和 selectionArgs 参数用于约束查询哪些行，sortOrder 参数用于对结果进行排序，查询的结果存放在 Cursor 对象中返回。

30. insert()

向内容提供器中添加一条数据。使用 uri 参数来确定要添加到的表，待添加的数据保存在 values 参数中。添加完成后，返回一个用于表示这条新记录的 URI。

31. update()

更新内容提供器中已有的数据。使用 uri 参数来确定更新哪一张表中的数据，新数据保存在 values 参数中，selection 和 selectionArgs 参数用于约束更新哪些行，受影响的行数将作为返回值返回。

32. delete()

从内容提供器中删除数据。使用 uri 参数来确定删除哪一张表中的数据，selection 和 selectionArgs 参数用于约束删除哪些行，被删除的行数将作为返回值返回。

33. getType()

根据传入的内容 URI 来返回相应的 MIME 类型。

可以看到，几乎每一个方法都会带有 Uri 这个参数，这个参数也正是调用 ContentResolver 的增删改查方法时传递过来的。而现在，我们需要对传入的 Uri 参数进行解析，从中分析出调用方期望访问的表和数据。

回顾一下，一个标准的内容 URI 写法是这样的：

```
content://com.example.app.provider/table1
```

这就表示调用方期望访问的是 com.example.app 这个应用的 table1 表中的数据。除此之外，我们还可以在这个内容 URI 的后面加上一个 id，如下所示：

```
content://com.example.app.provider/table1/1
```

这就表示调用方期望访问的是 com.example.app 这个应用的 table1 表中 id 为 1 的数据。

内容 URI 的格式主要就只有以上两种，以路径结尾就表示期望访问该表中所有的数据，以 id 结尾就表示期望访问该表中拥有相应 id 的数据。我们可以使用通配符的方式来分别匹配这两种格式的内容 URI，规则如下。

34. *：表示匹配任意长度的任意字符

35. #：表示匹配任意长度的数字

所以，一个能够匹配任意表的内容 URI 格式就可以写成：

```
content://com.example.app.provider/*
```

而一个能够匹配 table1 表中任意一行数据的内容 URI 格式就可以写成：

```
content://com.example.app.provider/table1/#
```

接着，我们再借助 UriMatcher 这个类就可以轻松地实现匹配内容 URI 的功能。UriMatcher 中提供了一个 addURI()方法，这个方法接收三个参数，可以分别把权限、路径和一个自定义代码传进去。这样，当调用 UriMatcher 的 match()方法时，就可以将一个 Uri 对象传入，返回值是某个能够匹配这个 Uri 对象所对应的自定义代码，利用这个代码，我们就可以判断出调用方期望访问的是哪张表中的数据了。修改 MyProvider 中的代码，如下所示：

```
public class MyProvider extends ContentProvider {

    public static final int TABLE1_DIR = 0;

    public static final int TABLE1_ITEM = 1;

    public static final int TABLE2_DIR = 2;

    public static final int TABLE2_ITEM = 3;

    private static UriMatcher uriMatcher;

    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("com.example.app.provider", "table1", TABLE1_DIR);
        uriMatcher.addURI("com.example.app.provider ", "table1/#", TABLE1_ITEM);
        uriMatcher.addURI("com.example.app.provider ", "table2", TABLE2_ITEM);
        uriMatcher.addURI("com.example.app.provider ", "table2/#", TABLE2_ITEM);
    }
    ......
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
        switch (uriMatcher.match(uri)) {
        case TABLE1_DIR:
            // 查询table1表中的所有数据
            break;
        case TABLE1_ITEM:
            // 查询table1表中的单条数据
            break;
        case TABLE2_DIR:
```

```
            // 查询table2表中的所有数据
            break;
        case TABLE2_ITEM:
            // 查询table2表中的单条数据
            break;
        default:
            break;
        }
        ......
    }
    ......
}
```

可以看到，MyProvider 中新增了四个整型常量，其中 TABLE1_DIR 表示访问 table1 表中的所有数据，TABLE1_ITEM 表示访问 table1 表中的单条数据，TABLE2_DIR 表示访问 table2 表中的所有数据，TABLE2_ITEM 表示访问 table2 表中的单条数据。接着在静态代码块里我们创建了 UriMatcher 的实例，并调用 addURI()方法，将期望匹配的内容 URI 格式传递进去，注意这里传入的路径参数是可以使用通配符的。然后当 query()方法被调用的时候，就会通过 UriMatcher 的 match()方法对传入的 Uri 对象进行匹配，如果发现 UriMatcher 中某个内容 URI 格式成功匹配了该 Uri 对象，则会返回相应的自定义代码，然后我们就可以判断出调用方期望访问的到底是什么数据了。

上述代码只是以 query()方法为例做了个示范，其实 insert()、update()、delete()这几个方法的实现也是差不多的，它们都会携带 Uri 这个参数，然后同样利用 UriMatcher 的 match()方法判断出调用方期望访问的是哪张表，再对该表中的数据进行相应的操作就可以了。

除此之外，还有一个方法你会比较陌生，即 getType()方法。它是所有的内容提供器都必须提供的一个方法，用于获取 Uri 对象所对应的 MIME 类型。一个内容 URI 所对应的 MIME 字符串主要由三部分组分，Android 对这三个部分做了如下格式规定。

36. 必须以 vnd 开头。
37. 如果内容 URI 以路径结尾，则后接 android.cursor.dir/，如果内容 URI 以 id 结尾，则后接 android.cursor.item/。
38. 最后接上 vnd.<authority>.<path>。

所以，对于 content://com.example.app.provider/table1 这个内容 URI，它所对应的 MIME 类型就可以写成：

```
vnd.android.cursor.dir/vnd.com.example.app.provider.table1
```

对于 content://com.example.app.provider/table1/1 这个内容 URI，它所对应的 MIME 类型就可以写成：

```
vnd.android.cursor.item/vnd. com.example.app.provider.table1
```

现在我们可以继续完善 MyProvider 中的内容了，这次来实现 getType()方法中的逻辑，代码如下所示：

```
public class MyProvider extends ContentProvider {
    ......
    @Override
    public String getType(Uri uri) {
        switch (uriMatcher.match(uri)) {
        case TABLE1_DIR:
            return "vnd.android.cursor.dir/vnd.com.example.app.provider.
table1";
        case TABLE1_ITEM:
            return "vnd.android.cursor.item/vnd.com.example.app.provider.
table1";
        case TABLE2_DIR:
            return "vnd.android.cursor.dir/vnd.com.example.app.provider.
table2";
        case TABLE2_ITEM:
            return "vnd.android.cursor.item/vnd.com.example.app.provider.
table2";
        default:
            break;
        }
        return null;
    }
}
```

到这里，一个完整的内容提供器就创建完成了，现在任何一个应用程序都可以使用 ContentResolver 来访问我们程序中的数据。那么前面所提到的，如何才能保证隐私数据不会泄漏出去呢？其实多亏了内容提供器的良好机制，这个问题在不知不觉中已经被解决了。因为所有的 CRUD 操作都一定要匹配到相应的内容 URI 格式才能进行的，而我们当然不可能向 UriMatcher 中添加隐私数据的 URI，所以这部分数据根本无法被外部程序访问到，安全问题也就不存在了。

好了，创建内容提供器的步骤你也已经清楚了，下面就来实战一下，真正体验一回跨程序数据共享的功能。

6.3.2 实现跨程序数据共享

简单起见，我们还是在上一章中 DatabaseTest 项目的基础上继续开发，通过内容提供器来给它加入外部访问接口。打开 DatabaseTest 项目，首先将 MyDatabaseHelper 中使用 Toast 弹出创建数据库成功的提示去除掉，因为跨程序访问时我们不能直接使用 Toast。然后添加一个 DatabaseProvider 类，代码如下所示：

```
public class DatabaseProvider extends ContentProvider {

    public static final int BOOK_DIR = 0;

    public static final int BOOK_ITEM = 1;

    public static final int CATEGORY_DIR = 2;

    public static final int CATEGORY_ITEM = 3;

    public static final String AUTHORITY = "com.example.databasetest.provider";

    private static UriMatcher uriMatcher;

    private MyDatabaseHelper dbHelper;

    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI(AUTHORITY, "book", BOOK_DIR);
        uriMatcher.addURI(AUTHORITY, "book/#", BOOK_ITEM);
        uriMatcher.addURI(AUTHORITY, "category", CATEGORY_DIR);
        uriMatcher.addURI(AUTHORITY, "category/#", CATEGORY_ITEM);
    }

    @Override
    public boolean onCreate() {
        dbHelper = new MyDatabaseHelper(getContext(), "BookStore.db", null, 2);
        return true;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder) {
```

```
        // 查询数据
        SQLiteDatabase db = dbHelper.getReadableDatabase();
        Cursor cursor = null;
        switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            cursor = db.query("Book", projection, selection, selectionArgs,
null, null, sortOrder);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            cursor = db.query("Book", projection, "id = ?", new String[]
{ bookId }, null, null, sortOrder);
            break;
        case CATEGORY_DIR:
            cursor = db.query("Category", projection, selection,
selectionArgs, null, null, sortOrder);
            break;
        case CATEGORY_ITEM:
            String categoryId = uri.getPathSegments().get(1);
            cursor = db.query("Category", projection, "id = ?", new String[]
{ categoryId }, null, null, sortOrder);
            break;
        default:
            break;
        }
        return cursor;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        // 添加数据
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        Uri uriReturn = null;
        switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
        case BOOK_ITEM:
            long newBookId = db.insert("Book", null, values);
            uriReturn = Uri.parse("content://" + AUTHORITY + "/book/" +
newBookId);
            break;
```

```
        case CATEGORY_DIR:
        case CATEGORY_ITEM:
            long newCategoryId = db.insert("Category", null, values);
            uriReturn = Uri.parse("content://" + AUTHORITY + "/category/" +
newCategoryId);
            break;
        default:
            break;
        }
        return uriReturn;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
String[] selectionArgs) {
        // 更新数据
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        int updatedRows = 0;
        switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            updatedRows = db.update("Book", values, selection, selectionArgs);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            updatedRows = db.update("Book", values, "id = ?", new String[]
{ bookId });
            break;
        case CATEGORY_DIR:
            updatedRows = db.update("Category", values, selection,
selectionArgs);
            break;
        case CATEGORY_ITEM:
            String categoryId = uri.getPathSegments().get(1);
            updatedRows = db.update("Category", values, "id = ?", new String[]
{ categoryId });
            break;
        default:
            break;
        }
```

```
        return updatedRows;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        // 删除数据
        SQLiteDatabase db = dbHelper.getWritableDatabase();
        int deletedRows = 0;
        switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            deletedRows = db.delete("Book", selection, selectionArgs);
            break;
        case BOOK_ITEM:
            String bookId = uri.getPathSegments().get(1);
            deletedRows = db.delete("Book", "id = ?", new String[] { bookId });
            break;
        case CATEGORY_DIR:
            deletedRows = db.delete("Category", selection, selectionArgs);
            break;
        case CATEGORY_ITEM:
            String categoryId = uri.getPathSegments().get(1);
            deletedRows = db.delete("Category", "id = ?", new String[]
{ categoryId });
            break;
        default:
            break;
        }
        return deletedRows;
    }

    @Override
    public String getType(Uri uri) {
        switch (uriMatcher.match(uri)) {
        case BOOK_DIR:
            return "vnd.android.cursor.dir/vnd.com.example.databasetest.
provider.book";
        case BOOK_ITEM:
            return "vnd.android.cursor.item/vnd.com.example.databasetest.
provider.book";
```

```
            case CATEGORY_DIR:
                return "vnd.android.cursor.dir/vnd.com.example.databasetest.
    provider.category";
            case CATEGORY_ITEM:
                return "vnd.android.cursor.item/vnd.com.example.databasetest.
    provider.category";
            }
            return null;
        }

    }
```

代码虽然很长，不过不用担心，这些内容都非常容易理解，因为使用到的全部都是上一小节中我们学到的知识。首先在类的一开始，同样是定义了四个常量，分别用于表示访问 Book 表中的所有数据、访问 Book 表中的单条数据、访问 Category 表中的所有数据和访问 Category 表中的单条数据。然后在静态代码块里对 UriMatcher 进行了初始化操作，将期望匹配的几种 URI 格式添加了进去。

接下来就是每个抽象方法的具体实现了，先来看下 onCreate()方法，这个方法的代码很短，就是创建了一个 MyDatabaseHelper 的实例，然后返回 true 表示内容提供器初始化成功，这时数据库就已经完成了创建或升级操作。

接着看一下 query()方法，在这个方法中先获取到了 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要访问哪张表，再调用 SQLiteDatabase 的 query()进行查询，并将 Cursor 对象返回就好了。注意当访问单条数据的时候有一个细节，这里调用了 Uri 对象的 getPathSegments()方法，它会将内容 URI 权限之后的部分以“/”符号进行分割，并把分割后的结果放入到一个字符串列表中，那这个列表的第 0 个位置存放的就是路径，第 1 个位置存放的就是 id 了。得到了 id 之后，再通过 selection 和 selectionArgs 参数进行约束，就实现了查询单条数据的功能。

再往后就是 insert()方法，同样它也是先获取到了 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要往哪张表里添加数据，再调用 SQLiteDatabase 的 insert()方法进行添加就可以了。注意 insert()方法要求返回一个能够表示这条新增数据的 URI，所以我们还需要调用 Uri.parse()方法来将一个内容 URI 解析成 Uri 对象，当然这个内容 URI 是以新增数据的 id 结尾的。

接下来就是 update()方法了，相信这个方法中的代码已经完全难不倒你了。也是先获取 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要更新哪张表里的数据，再调用 SQLiteDatabase 的 update()方法进行更新就好了，受影响的行数将作为返回值返回。

下面是 delete()方法，是不是感觉越到后面越轻松了？因为你已经渐入佳境，真正地找

到窍门了。这里仍然是先获取到 SQLiteDatabase 的实例，然后根据传入的 Uri 参数判断出用户想要删除哪张表里的数据，再调用 SQLiteDatabase 的 delete()方法进行删除就好了，被删除的行数将作为返回值返回。

最后是 getType()方法，这个方法中的代码完全是按照上一节中介绍的格式规则编写的，相信已经没有什么解释的必要了。

这样我们就将内容提供器中的代码全部编写完了，不过离实现跨程序数据共享的功能还差了一小步，因为还需要将内容提供器在 AndroidManifest.xml 文件中注册才可以，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.databasetest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <provider
            android:name="com.example.databasetest.DatabaseProvider"
            android:authorities="com.example.databasetest.provider" >
        </provider>
    </application>
</manifest>
```

可以看到，这里我们使用了<provider>标签来对 DatabaseProvider 这个内容提供器进行注册，在 android:name 属性中指定了该类的全名，又在 android:authorities 属性中指定了该内容提供器的权限。

现在 DatabaseTest 这个项目就已经拥有了跨程序共享数据的功能了，我们赶快来尝试一下。首先需要将 DatabaseTest 程序从模拟器中删除掉，以防止上一章中产生的遗留数据对我们造成干扰。然后运行一下项目，将 DatabaseTest 程序重新安装在模拟器上了。接着关闭掉 DatabaseTest 这个项目，并创建一个新项目 ProviderTest，我们就将通过这个程序去访问 DatabaseTest 中的数据。

还是先来编写一下布局文件吧，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/add_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add To Book" />

    <Button
        android:id="@+id/query_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Query From Book" />

    <Button
        android:id="@+id/update_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update Book" />

    <Button
        android:id="@+id/delete_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Delete From Book" />

</LinearLayout>
```

布局文件很简单，里面放置了四个按钮，分别用于添加、查询、修改和删除数据的。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private String newId;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        Button addData = (Button) findViewById(R.id.add_data);
        addData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 添加数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book");
                ContentValues values = new ContentValues();
                values.put("name", "A Clash of Kings");
                values.put("author", "George Martin");
                values.put("pages", 1040);
                values.put("price", 22.85);
                Uri newUri = getContentResolver().insert(uri, values);
                newId = newUri.getPathSegments().get(1);
            }
        });
        Button queryData = (Button) findViewById(R.id.query_data);
        queryData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 查询数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book");
                Cursor cursor = getContentResolver().query(uri, null, null,
null, null);
                if (cursor != null) {
                    while (cursor.moveToNext()) {
                        String name = cursor.getString(cursor.
getColumnIndex("name"));
                        String author = cursor.getString(cursor.
getColumnIndex("author"));
                        int pages = cursor.getInt(cursor.getColumnIndex
("pages"));
                        double price = cursor.getDouble(cursor.
getColumnIndex("price"));
                        Log.d("MainActivity", "book name is " + name);
                        Log.d("MainActivity", "book author is " + author);
                        Log.d("MainActivity", "book pages is " + pages);
                        Log.d("MainActivity", "book price is " + price);
                    }
```

```
                    cursor.close();
                }
            }
        });
        Button updateData = (Button) findViewById(R.id.update_data);
        updateData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 更新数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book/" + newId);
                ContentValues values = new ContentValues();
                values.put("name", "A Storm of Swords");
                values.put("pages", 1216);
                values.put("price", 24.05);
                getContentResolver().update(uri, values, null, null);
            }
        });
        Button deleteData = (Button) findViewById(R.id.delete_data);
        deleteData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 删除数据
                Uri uri = Uri.parse("content://com.example.databasetest.
provider/book/" + newId);
                getContentResolver().delete(uri, null, null);
            }
        });
    }

}
```

可以看到，我们分别在这四个按钮的点击事件里面处理了增删改查的逻辑。添加数据的时候，首先调用了 Uri.parse()方法将一个内容 URI 解析成 Uri 对象，然后把要添加的数据都存放到 ContentValues 对象中，接着调用 ContentResolver 的 insert()方法执行添加操作就可以了。注意 insert()方法会返回一个 Uri 对象，这个对象中包含了新增数据的 id，我们通过 getPathSegments()方法将这个 id 取出，稍后会用到它。

查询数据的时候，同样是调用了 Uri.parse()方法将一个内容 URI 解析成 Uri 对象，然后

调用 ContentResolver 的 query()方法去查询数据，查询的结果当然还是存放在 Cursor 对象中的。之后对 Cursor 进行遍历，从中取出查询结果，并一一打印出来。

更新数据的时候，也是先将内容 URI 解析成 Uri 对象，然后把想要更新的数据存放到 ContentValues 对象中，再调用 ContentResolver 的 update()方法执行更新操作就可以了。注意这里我们为了不想让 Book 表中其他的行受到影响，在调用 Uri.parse()方法时，给内容 URI 的尾部增加了一个 id，而这个 id 正是添加数据时所返回的。这就表示我们只希望更新刚刚添加的那条数据，Book 表中的其他行都不会受影响。

删除数据的时候，也是使用同样的方法解析了一个以 id 结尾的内容 URI，然后调用 ContentResolver 的 delete()方法执行删除操作就可以了。由于我们在内容 URI 里指定了一个 id，因此只会删掉拥有相应 id 的那行数据，Book 表中的其他数据都不会受影响。

现在运行一下 ProviderTest 项目，会显示如图 6.4 所示的界面。

图 6.4

点击一下 Add To Book 按钮，此时数据就应该已经添加到 DatabaseTest 程序的数据库中了，我们可以通过点击 Query From Book 按钮来检查一下，打印日志如图 6.5 所示。

Tag	Text
MainActivity	book name is A Clash of Kings
MainActivity	book author is George Martin
MainActivity	book pages is 1040
MainActivity	book price is 22.85

图 6.5

然后点击一下 Update Book 按钮来更新数据，再点击一下 Query From Book 按钮进行检查，结果如图 6.6 所示。

Tag	Text
MainActivity	book name is A Storm of Swords
MainActivity	book author is George Martin
MainActivity	book pages is 1216
MainActivity	book price is 24.05

图 6.6

最后点击 Delete From Book 按钮删除数据，此时再点击 Query From Book 按钮就查询不到数据了。

由此可以看出，我们的跨程序共享数据功能已经成功实现了！现在不仅是 ProviderTest 程序，任何一个程序都可以轻松访问 DatabaseTest 中的数据，而且我们还丝毫不用担心隐私数据泄漏的问题。

到这里，与内容提供器相关的重要内容就基本全部介绍完了，下面就让我们再次进入本书的特殊环节，学习一下关于 Git 更多的用法。

6.5 小结与点评

本章的内容比较少，而且很多时候都是在使用上一章中学习的数据库知识，所以理解这部分内容对你来说应该是比较轻松的吧。在本章中，我们主要学习了内容提供器的相关内容，以实现跨程序数据共享的功能。现在你不仅知道了如何去访问其他程序中的数据，还学会了怎样创建自己的内容提供器来共享数据，收获还是挺大的吧。

不过每次在创建内容提供器的时候，你都需要提醒一下自己，我是不是应该这么做？因为只有真正需要将数据共享出去的时候我们才应该创建内容提供器，仅仅是用于程序内部访问的数据就没有必要这么做，所以千万别对它进行滥用。

在连续学了几章系统机制方面的内容之后是不是感觉有些枯燥？那么下一章中我们就来换换口味，学习一下 Android 多媒体方面的知识吧。

第 7 章　运用手机多媒体

在过去，手机的功能都比较单调，仅仅就是用来打电话和发短信的。而如今，手机在我们生活中正扮演着越来越重要的角色，各种娱乐方式都可以在手机上进行。上班的路上太无聊，可以带着耳机听音乐。外出旅行的时候，可以在手机上看电影。无论走到哪里，遇到喜欢的事物都可以随手拍下来。

众多的娱乐方式少不了强大的多媒体功能的支持，而 Android 在这一方面也是做得非常出色。它提供了一系列的 API，使得我们可以在程序中调用很多手机的多媒体资源，从而编写出更加丰富多彩的应用程序。本章我们就将对 Android 中一些常用的多媒体功能的使用技巧进行学习。

7.1　使用通知

通知（Notification）是 Android 系统中比较有特色的一个功能，当某个应用程序希望向用户发出一些提示信息，而该应用程序又不在前台运行时，就可以借助通知来实现。发出一条通知后，手机最上方的状态栏中会显示一个通知的图标，下拉状态栏后可以看到通知的详细内容。Android 的通知功能获得了大量用户的认可和喜爱，就连 iOS 系统也在 5.0 版本之后加入了类似的功能。

7.1.1　通知的基本用法

了解了通知的基本概念，下面我们就来看一下通知的使用方法吧。通知的用法还是比较灵活的，既可以在活动里创建，也可以在广播接收器里创建，当然还可以在下一章中我们即将学习的服务里创建。相比于广播接收器和服务，在活动里创建通知的场景还是比较少的，因为一般只有当程序进入到后台的时候我们才需要使用通知。

不过，无论是在哪里创建通知，整体的步骤都是相同的，下面我们就来学习一下创建通知的详细步骤。首先需要一个 NotificationManager 来对通知进行管理，可以调用 Context 的 getSystemService()方法获取到。getSystemService()方法接收一个字符串参数用于确定获取系统的哪个服务，这里我们传入 Context.NOTIFICATION_SERVICE 即可。因此，获取 NotificationManager 的实例就可以写成：

```
NotificationManager manager = (NotificationManager)
```

```
getSystemService(Context.NOTIFICATION_SERVICE);
```

接下来需要创建一个 Notification 对象，这个对象用于存储通知所需的各种信息，我们可以使用它的有参构造函数来进行创建。Notification 的有参构造函数接收三个参数，第一个参数用于指定通知的图标，比如项目的 res/drawable 目录下有一张 icon.png 图片，那么这里就可以传入 R.drawable.icon。第二个参数用于指定通知的 ticker 内容，当通知刚被创建的时候，它会在系统的状态栏一闪而过，属于一种瞬时的提示信息。第三个参数用于指定通知被创建的时间，以毫秒为单位，当下拉系统状态栏时，这里指定的时间会显示在相应的通知上。因此，创建一个 Notification 对象就可以写成：

```
Notification notification = new Notification(R.drawable.icon, "This is ticker text",
                                    System.currentTimeMillis());
```

创建好了 Notification 对象后，我们还需要对通知的布局进行设定，这里只需要调用 Notification 的 setLatestEventInfo()方法就可以给通知设置一个标准的布局。这个方法接收四个参数，第一个参数是 Context，这个没什么好解释的。第二个参数用于指定通知的标题内容，下拉系统状态栏就可以看到这部分内容。第三个参数用于指定通知的正文内容，同样下拉系统状态栏就可以看到这部分内容。第四个参数我们暂时还用不到，可以先传入 null。因此，对通知的布局进行设定就可以写成：

```
notification.setLatestEventInfo(context, "This is content title", "This is
content text", null);
```

以上工作都完成之后，只需要调用 NotificationManager 的 notify()方法就可以让通知显示出来了。notify()方法接收两个参数，第一个参数是 id，要保证为每个通知所指定的 id 都是不同的。第二个参数则是 Notification 对象，这里直接将我们刚刚创建好的 Notification 对象传入即可。因此，显示一个通知就可以写成：

```
manager.notify(1, notification);
```

到这里就已经把创建通知的每一个步骤都分析完了，下面就让我们通过一个具体的例子来看一看通知到底是长什么样的。

新建一个 NotificationTest 项目，并修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/send_notice"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send notice"
        />

</LinearLayout>
```

布局文件非常简单，里面只有一个 Send notice 按钮，用于发出一条通知。接下来修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button sendNotice;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sendNotice = (Button) findViewById(R.id.send_notice);
        sendNotice.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.send_notice:
            NotificationManager manager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
            Notification notification = new Notification(R.drawable.
ic_launcher, "This is ticker text", System.currentTimeMillis());
            notification.setLatestEventInfo(this, "This is content title",
"This is content text", null);
            manager.notify(1, notification);
            break;
        default:
            break;
        }
    }

}
```

可以看到，我们在 Send notice 按钮的点击事件里面完成了通知的创建工作，创建的过程正如前面所描述的一样。现在就可以来运行一下程序了，点击 Send notice 按钮，就会看到有

一条通知在系统状态栏显示出来，如图 7.1 所示。

图　7.1

下拉系统状态栏可以看到该通知的详细信息，如图 7.2 所示。

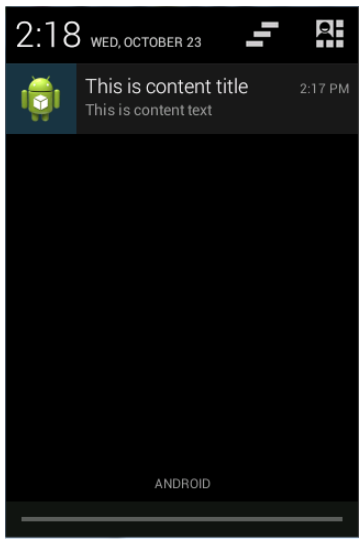

图　7.2

如果你使用过 Android 手机，此时应该会下意识地认为这条通知是可以点击的。但是当你去点击它的时候，你会发现没有任何效果。不对啊，好像每条通知点击之后都应该会有反

应的呀？其实要想实现通知的点击效果，我们还需要在代码中进行相应的设置，这就涉及到了一个新的概念，PendingIntent。

PendingIntent 从名字上看起来就和 Intent 有些类似，它们之间也确实存在着不少共同点。比如它们都可以去指明某一个“意图”，都可以用于启动活动、启动服务以及发送广播等。不同的是，Intent 更加倾向于去立即执行某个动作，而 PendingIntent 更加倾向于在某个合适的时机去执行某个动作。所以，也可以把 PendingIntent 简单地理解为延迟执行的 Intent。

PendingIntent 的用法同样很简单，它主要提供了几个静态方法用于获取 PendingIntent 的实例，可以根据需求来选择是使用 getActivity()方法、getBroadcast()方法、还是 getService()方法。这几个方法所接收的参数都是相同的，第一个参数依旧是 Context，不用多做解释。第二个参数一般用不到，通常都是传入 0 即可。第三个参数是一个 Intent 对象，我们可以通过这个对象构建出 PendingIntent 的“意图”。第四个参数用于确定 PendingIntent 的行为，有 FLAG_ONE_SHOT、FLAG_NO_CREATE、FLAG_CANCEL_CURRENT 和 FLAG_UPDATE_CURRENT 这四种值可选，每种值的含义你可以查看文档，我就不一一进行解释了。

对 PendingIntent 有了一定的了解后，我们再回过头来看一下 Notification 的 setLatestEventInfo()方法。刚才我们将 setLatestEventInfo()方法的第四个参数忽略掉了，直接传入了 null，现在仔细观察一下，发现第四个参数正是一个 PendingIntent 对象。因此，这里就可以通过 PendingIntent 构建出一个延迟执行的“意图”，当用户点击这条通知时就会执行相应的逻辑。

现在我们来优化一下 NotificationTest 项目，给刚才的通知加上点击功能，让用户点击它的时候可以启动另一个活动。

首先需要准备好另一个活动，这里新建布局文件 notification_layout.xml，代码如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:textSize="24sp"
        android:text="This is notification layout"
        />

</RelativeLayout>
```

布局文件的内容非常简单，只有一个居中显示的 TextView，用于展示一段文本信息。然

后新建 NotificationActivity 继承自 Activity，在这里加载刚才定义的布局文件，代码如下所示：

```
public class NotificationActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notification_layout);
    }

}
```

接着修改 AndroidManifest.xml 中的代码，在里面加入 NotificationActivity 的注册声明，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.notificationtest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <activity android:name=".NotificationActivity" >
        </activity>
    </application>

</manifest>
```

这样就把 NotificationActivity 这个活动准备好了，下面我们修改 MainActivity 中的代码，给通知加入点击功能，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.send_notice:
            NotificationManager manager = (NotificationManager)
```

```
getSystemService(NOTIFICATION_SERVICE);
            Notification notification = new Notification(R.drawable.
ic_launcher, "This is ticker text", System.currentTimeMillis());
            Intent intent = new Intent(this, NotificationActivity.class);
            PendingIntent pi = PendingIntent.getActivity(this, 0, intent,
PendingIntent.FLAG_CANCEL_CURRENT);
            notification.setLatestEventInfo(this, "This is content title",
"This is content text", pi);
            manager.notify(1, notification);
            break;
        default:
            break;
        }
    }

}
```

可以看到，这里先是使用 Intent 表达出我们想要启动 NotificationActivity 的“意图”，然后将构建好的 Intent 对象传入到 PendingIntent 的 getActivity()方法里，以得到 PendingIntent 的实例，接着把它作为第四个参数传入到 Notification 的 setLatestEventInfo()方法中。

现在重新运行一下程序，并点击 Send notice 按钮，依旧会发出一条通知。然后下拉系统状态栏，点击一下该通知，就会看到 NotificationActivity 这个活动的界面了，如图 7.3 所示。

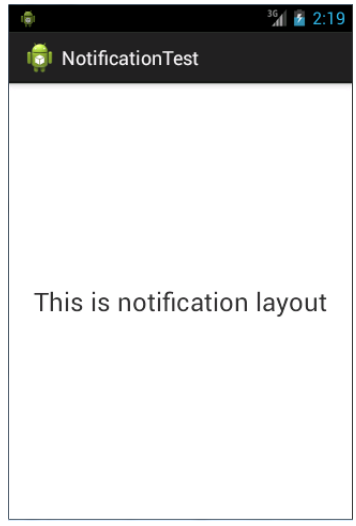

图 7.3

咦？怎么系统状态上的通知图标还没有消失呢？是这样的，如果我们没有在代码中对该通知进行取消，它就会一直显示在系统的状态栏上显示。解决的方法也很简单，调用 NotificationManager 的 cancel()方法就可以取消通知了。修改 NotificationActivity 中的代码，如下所示：

```
public class NotificationActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notification_layout);
        NotificationManager manager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
        manager.cancel(1);
    }

}
```

可以看到，这里我们在 cancel()方法中传入了 1，这个 1 是什么意思呢？还记得在创建通知的时候给每条通知指定的 id 吗？当时我们给这条通知设置的 id 就是 1。因此，如果你想要取消哪一条通知，就在 cancel()方法中传入该通知的 id 就行了。

7.1.2 通知的高级技巧

现在你已经掌握了创建和取消通知的方法，并且知道了如何去响应通知的点击事件。不过通知的用法并不仅仅是这些呢，那么本节中我们就来探究一下通知更多的高级技巧。

观察 Notification 这个类，你会发现里面还有很多我们没有使用过的属性。先来看看 sound 这个属性吧，它可以在通知发出的时候播放一段音频，这样就能够更好地告知用户有通知到来。sound 这个属性是一个 Uri 对象，所以在指定音频文件的时候还需要先获取到音频文件对应的 URI。比如说，我们手机的/system/media/audio/ringtones 目录下有一个 Basic_tone.ogg 音频文件，那么在代码中这样就可以这样指定：

```
    Uri soundUri = Uri.fromFile(new File("/system/media/audio/ringtones/
Basic_tone.ogg"));
    notification.sound = soundUri;
```

除了允许播放音频外，我们还可以在通知到来的时候让手机进行振动，使用的是 vibrate 这个属性。它是一个长整型的数组，用于设置手机静止和振动的时长，以毫秒为单位。下标为 0 的值表示手机静止的时长，下标为 1 的值表示手机振动的时长，下标为 2 的值又表示手机静止的时长，以此类推。所以，如果想要让手机在通知到来的时候立刻振动 1 秒，然后静

止 1 秒，再振动 1 秒，代码就可以写成：

```
long[] vibrates = {0, 1000, 1000, 1000};
notification.vibrate = vibrates;
```

不过，想要控制手机振动还需要声明权限的。因此，我们还得编辑 AndroidManifest.xml 文件，加入如下声明：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.notificationtest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <uses-permission android:name="android.permission.VIBRATE" />
    ……
</manifest>
```

学会了控制通知的声音和振动，下面我们来看一下如何在通知到来时控制手机 LED 灯的显示。

现在的手机基本上都会前置一个 LED 灯，当有未接电话或未读短信，而此时手机又处于锁屏状态时，LED 灯就会不停地闪烁，提醒用户去查看。我们可以使用 ledARGB、ledOnMS、ledOffMS 以及 flags 这几个属性来实现这种效果。ledARGB 用于控制 LED 灯的颜色，一般有红绿蓝三种颜色可选。ledOnMS 用于指定 LED 灯亮起的时长，以毫秒为单位。ledOffMS 用于指定 LED 灯暗去的时长，也是以毫秒为单位。flags 可用于指定通知的一些行为，其中就包括显示 LED 灯这一选项。所以，当通知到来时，如果想要实现 LED 灯以绿色的灯光一闪一闪的效果，就可以写成：

```
notification.ledARGB = Color.GREEN;
notification.ledOnMS = 1000;
notification.ledOffMS = 1000;
notification.flags = Notification.FLAG_SHOW_LIGHTS;
```

当然，如果你不想进行那么多繁杂的设置，也可以直接使用通知的默认效果，它会根据当前手机的环境来决定播放什么铃声，以及如何振动，写法如下：

```
notification.defaults = Notification.DEFAULT_ALL;
```

注意，以上所涉及的这些高级技巧都要在手机上运行才能看得到效果，模拟器是无法表现出振动、以及 LED 灯闪烁等功能的。

7.2 接收和发送短信

收发短信应该是每个手机最基本的功能之一了，即使是许多年前的老手机也都会具备这项功能，而 Android 作为出色的智能手机操作系统，自然也少不了在这方面的支持。每个 Android 手机都会内置一个短信应用程序，使用它就可以轻松地完成收发短信的操作，如图 7.4 所示。

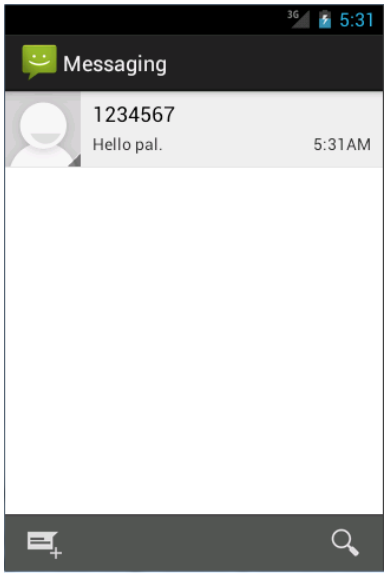

图 7.4

不过作为一名开发者，仅仅满足于此显然是不够的。你要知道，Android 还提供了一系列的 API，使得我们甚至可以在自己的应用程序里接收和发送短信。也就是说，只要你有足够的信心，完全可以自己实现一个短信应用来替换掉 Android 系统自带的短信应用。那么下面我们就来看一看，如何才能在自己的应用程序里接收和发送短信。

7.2.1 接收短信

其实接收短信主要是利用了我们在第 5 章学习过的广播机制。当手机接收到一条短信的时候，系统会发出一条值为 android.provider.Telephony.SMS_RECEIVED 的广播，这条广播里携带着与短信相关的所有数据。每个应用程序都可以在广播接收器里对它进行监听，收到广播时再从中解析出短信的内容即可。

让我们通过一个具体的例子来实践一下吧，新建一个 SMSTest 项目，首先修改 activity_

main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="50dp" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:padding="10dp"
            android:text="From:" />

        <TextView
            android:id="@+id/sender"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="50dp" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:padding="10dp"
            android:text="Content:" />

        <TextView
            android:id="@+id/content"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```
            android:layout_gravity="center_vertical" />
    </LinearLayout>

</LinearLayout>
```

这个布局文件里，我们在根元素下面放置了两个 LinearLayout，用于显示两行数据。第一个 LinearLayout 中有两个 TextView，用于显示短信的发送方。第二个 LinearLayout 中也有两个 TextView，用于显示短信的内容。

接着修改 MainActivity 中的代码，在 onCreate()方法中获取到两个 TextView 的实例，如下所示：

```
public class MainActivity extends Activity {

    private TextView sender;

    private TextView content;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        sender = (TextView) findViewById(R.id.sender);
        content = (TextView) findViewById(R.id.content);
    }

}
```

然后我们需要创建一个广播接收器来接收系统发出的短信广播。在 MainActivity 中新建 MessageReceiver 内部类继承自 BroadcastReceiver，并在 onReceive()方法中编写获取短信数据的逻辑，代码如下所示：

```
public class MainActivity extends Activity {

    ......

    class MessageReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            Bundle bundle = intent.getExtras();
            Object[] pdus = (Object[]) bundle.get("pdus"); // 提取短信消息
```

```
            SmsMessage[] messages = new SmsMessage[pdus.length];
            for (int i = 0; i < messages.length; i++) {
                messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
            }
            String address = messages[0].getOriginatingAddress(); // 获取发
送方号码
            String fullMessage = "";
            for (SmsMessage message : messages) {
                fullMessage += message.getMessageBody(); // 获取短信内容
            }
            sender.setText(address);
            content.setText(fullMessage);
        }

    }
}
```

可以看到，首先我们从 Intent 参数中取出了一个 Bundle 对象，然后使用 pdu 密钥来提取一个 SMS pdus 数组，其中每一个 pdu 都表示一条短信消息。接着使用 SmsMessage 的 createFromPdu()方法将每一个 pdu 字节数组转换为 SmsMessage 对象，调用这个对象的 getOriginatingAddress()方法就可以获取到短信的发送方号码，调用 getMessageBody()方法就可以获取到短信的内容，然后将每一个 SmsMessage 对象中的短信内容拼接起来，就组成了一条完整的短信。最后将获取到的发送方号码和短信内容显示在 TextView 上。

完成了 MessageReceiver 之后，我们还需要对它进行注册才能让它接收到短信广播，代码如下所示：

```
public class MainActivity extends Activity {

    private TextView sender;

    private TextView content;

    private IntentFilter receiveFilter;

    private MessageReceiver messageReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        sender = (TextView) findViewById(R.id.sender);
        content = (TextView) findViewById(R.id.content);
        receiveFilter = new IntentFilter();
        receiveFilter.addAction("android.provider.Telephony.SMS_RECEIVED");
        messageReceiver = new MessageReceiver();
        registerReceiver(messageReceiver, receiveFilter);
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(messageReceiver);
    }
    ……
}
```

这些代码你应该都已经非常熟悉了，使用的就是动态注册广播的技术。在 onCreate()方法中对 MessageReceiver 进行注册，在 onDestroy()方法中再对它取消注册。

代码到这里就已经完成得差不多了，不过最后我们还需要给程序声明一个接收短信的权限才行，修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.smstest"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ……
</manifest>
```

现在可以来运行一下程序了，界面如图 7.5 所示。

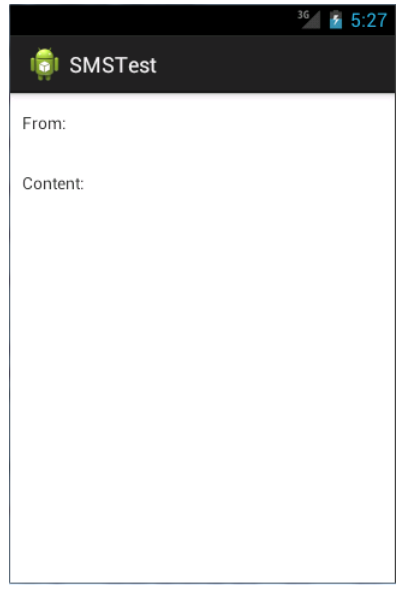

图 7.5

当有短信到来时，短信的发送方和内容就会显示在界面上。不过话说回来，我们使用的是模拟器，模拟器上怎么可能会收得到短信呢？不用担心，DDMS 提供了非常充分的模拟环境，使得我们不需要支付真正的短信费用也可以模拟收发短信的场景。将 Eclipse 切换到 DDMS 视图下，然后点击 Emulator Control 切换卡，在这里就可以向模拟器发送短信了，如图 7.6 所示。

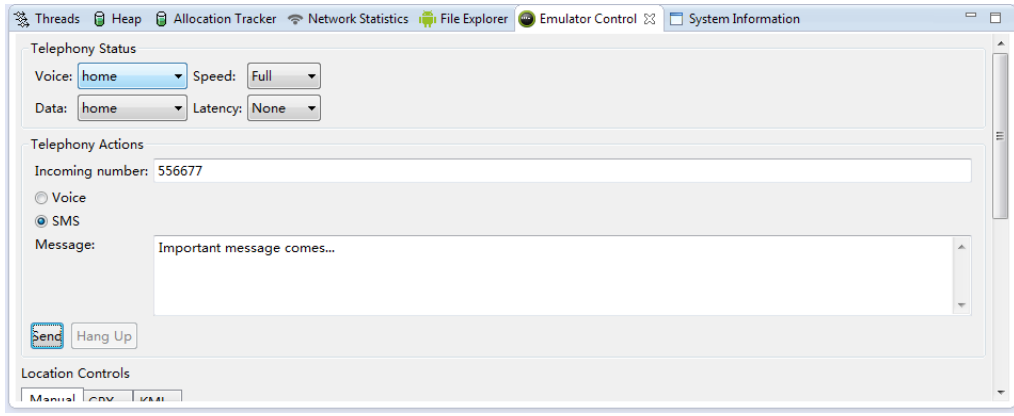

图 7.6

可以看到，我们指定发送方的号码是 556677，并填写了一段短信内容，然后点击 Send

按钮，这样短信就发送成功了。接着我们立马查看一下 SMSTest 这个程序，结果如图 7.7 所示。

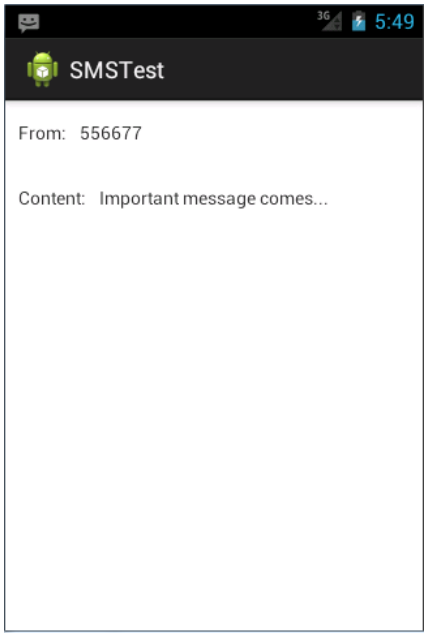

图 7.7

可以看到，短信的发送方号码和短信内容都显示到界面上了，说明接收短信的功能成功实现了。

7.2.2 拦截短信

仔细观察图 7.7，你会发现在系统状态栏出现了一个通知图标，这个通知图标是由 Android 自带的短信程序产生的。也就是说当短信到来时，不仅我们的程序会接收到这条短信，系统的短信程序同样也会收到。同样一条短信被重复接收两遍就会造成比较差的用户体验，那么有没有什么办法可以屏蔽系统短信程序的接收功能呢？

在前面 5.3.2 节学习有序广播的时候我们就已经知道，有序广播的传递是可以截断的，而系统发出的短信广播正是一条有序广播，因此这里我们的答案是肯定的。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {
    ……
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ……
```

```
        receiveFilter = new IntentFilter();
        receiveFilter.addAction("android.provider.Telephony.SMS_RECEIVED");
        receiveFilter.setPriority(100);
        messageReceiver = new MessageReceiver();
        registerReceiver(messageReceiver, receiveFilter);
    }
    ……
    class MessageReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            ……
            abortBroadcast();
        }

    }
}
```

可以看到，关键性的步骤只有两步。一是提高 MessageReceiver 的优先级，让它能够先于系统短信程序接收到短信广播。二是在 onReceive()方法中调用 abortBroadcast()方法，中止掉广播的继续传递。

现在重新运行程序，再向模拟器发送一条短信，这时只有我们自己的程序才能收到这条短信了。按下 Back 键将程序关闭后，系统的短信程序又会重新拥有接收短信的功能。

注意这个功能一定要慎用，随意拦截短信有可能会造成重要数据的丢失，所以你在拦截之前一定要先想清楚这种功能是不是你想要的。

7.2.3 发送短信

下面我们继续对 SMSTest 项目进行扩展，给它加上发送短信的功能。那么还是先来编写一下布局文件吧，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    ……
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="50dp" >
        <TextView
```

```
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:padding="10dp"
            android:text="To:" />

        <EditText
            android:id="@+id/to"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:layout_weight="1" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="50dp" >
        <EditText
            android:id="@+id/msg_input"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:layout_weight="1" />

        <Button
            android:id="@+id/send"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:text="Send" />
    </LinearLayout>

</LinearLayout>
```

这里我们又新增了两个 LinearLayout，分别处于第三和第四行的位置。第三行中放置了一个 EditText，用于输入接收方的手机号码。第四行中放置了一个 EditText 和一个 Button，分别用于输入短信内容和发送短信。

然后修改 MainActivity 中的代码，在里面加入发送短信的处理逻辑，代码如下所示：

```
public class MainActivity extends Activity {
```

```
    ……
    private EditText to;

    private EditText msgInput;

    private Button send;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        to = (EditText) findViewById(R.id.to);
        msgInput = (EditText) findViewById(R.id.msg_input);
        send = (Button) findViewById(R.id.send);
        send.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SmsManager smsManager = SmsManager.getDefault();
                smsManager.sendTextMessage(to.getText().toString(), null,
                                    msgInput.getText().toString(), null, null);
            }
        });
    }
    ……
}
```

可以看到，首先我们获取到了布局文件中新增控件的实例，然后在 Send 按钮的点击事件里面处理了发送短信的具体逻辑。当 Send 按钮被点击时，会先调用 SmsManager 的 getDefault()方法获取到 SmsManager 的实例，然后再调用它的 sendTextMessage()方法就可以去发送短信了。sendTextMessage()方法接收五个参数，其中第一个参数用于指定接收人的手机号码，第三个参数用于指定短信的内容，其他的几个参数我们暂时用不到，直接传入 null 就可以了。

接下来也许你已经猜到了，发送短信也是需要声明权限的，因此修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.smstest"
    android:versionCode="1"
    android:versionName="1.0" >
```

```
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <uses-permission android:name="android.permission. SEND_SMS" />
    ……
</manifest>
```

现在重新运行程序之后，SMSTest 就拥有了发送短信的能力。不过点击 Send 按钮虽然可以将短信发送出去，但是我们并不知道到底发送成功了没有，这个时候就可以利用 sendTextMessage()方法的第四个参数来对短信的发送状态进行监控。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {
    ……
    private IntentFilter sendFilter;

    private SendStatusReceiver sendStatusReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        sendFilter = new IntentFilter();
        sendFilter.addAction("SENT_SMS_ACTION");
        sendStatusReceiver = new SendStatusReceiver();
        registerReceiver(sendStatusReceiver, sendFilter);
        send.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                SmsManager smsManager = SmsManager.getDefault();
                Intent sentIntent = new Intent("SENT_SMS_ACTION");
                PendingIntent pi = PendingIntent.getBroadcast
(MainActivity.this, 0, sentIntent, 0);
                smsManager.sendTextMessage(to.getText().toString(), null,
msgInput.getText().toString(), pi, null);
            }
        });
    }

    @Override
    protected void onDestroy() {
```

```
        super.onDestroy();
        unregisterReceiver(messageReceiver);
        unregisterReceiver(sendStatusReceiver);
    }
    ……
    class SendStatusReceiver extends BroadcastReceiver {

        @Override
        public void onReceive(Context context, Intent intent) {
            if (getResultCode() == RESULT_OK) {
                // 短信发送成功
                Toast.makeText(context, "Send succeeded",
Toast.LENGTH_LONG).show();
            } else {
                // 短信发送失败
                Toast.makeText(context, "Send failed",
Toast.LENGTH_LONG).show();
            }
        }

    }
}
```

可以看到，在 Send 按钮的点击事件里面我们调用了 PendingIntent 的 getBroadcast()方法获取到了一个 PendingIntent 对象，并将它作为第四个参数传递到 sendTextMessage()方法中。然后又注册了一个新的广播接收器 SendStatusReceiver，这个广播接收器就是专门用于监听短信发送状态的，当 getResultCode()的值等于 RESULT_OK 就会提示发送成功，否则提示发送失败。

现在重新运行一下程序，在文本输入框里输入接收方的手机号码以及短信内容，然后点击 Send 按钮，结果如图 7.7 所示。

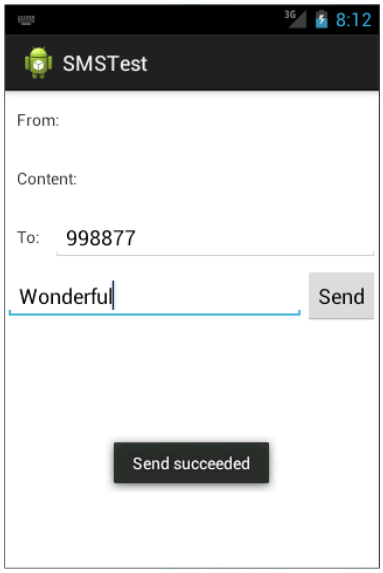

图 7.7

注意，这里虽然提示发送成功了，但实际上使用模拟器来发送短信对方是不可能收得到的，只有把这个项目运行在手机上，才能真正地实现发送短信的功能。

另外，根据国际标准，每条短信的长度不得超过 160 个字符，如果想要发送超出这个长度的短信，则需要将这条短信分割成多条短信来发送，使用 SmsManager 的 sendMultipartTextMessage()方法就可以实现上述功能。它的用法和 sendTextMessage()方法也基本类似，感兴趣的话你可以自己研究一下，这里就不再展开讲解了。

7.3 调用摄像头和相册

前面两节所学习的知识当中，都涉及到了一些必须要在真正的 Android 手机上运行才看得到效果的功能。本节即将学习的知识也是，比如模拟器对摄像头的支持并不友好。你会发现，当涉及到多媒体这一领域的时候，模拟器多多少少会有些力不从心。因此，下面我们就先来学习一下，如何使用 Android 手机来运行程序。

7.3.1 将程序运行到手机上

不必我多说，首先你需要拥有一部 Android 手机。现在 Android 手机早就不是什么稀罕物，几乎已经是人手一部了，如果你还没有话，抓紧去购买吧。

想要将程序运行到手机上，我们需要先通过数据线把手机连接到电脑上。然后进入到设置→开发者选项界面，并在这个界面中勾选中 USB 调试选项，如图 7.9 所示。注意从 Android

4.2 版本开始，系统默认是把开发者选项隐藏掉的，你需要先进入到关于手机界面，然后对着最下面的版本号那一栏连击四次，就会让开发者选项显示出来。

图　7.9

然后如果你使用的是 Windows 操作系统，还需要在电脑上安装手机的驱动。一般借助 91 手机助手或豌豆荚等工具都可以快速地进行安装，安装完成后就可以看到手机已经连接到电脑上了，如图 7.10 所示。

图 7.10

现在进入到 Eclipse 的 DDMS 视图，你会发现当前是有两个设备在线的，一个是我们一直使用的模拟器，另外一个则是刚刚连接上的手机了，如图 7.11 所示。

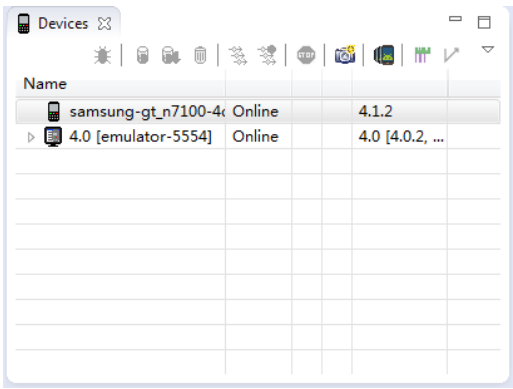

图 7.11

然后，对着 Eclipse 中的任何一个项目右击→Run As→Android Application，这时不会直接将程序运行到模拟器或者手机上，而是会弹出一个对话框让你进行选择，如图 7.12 所示。

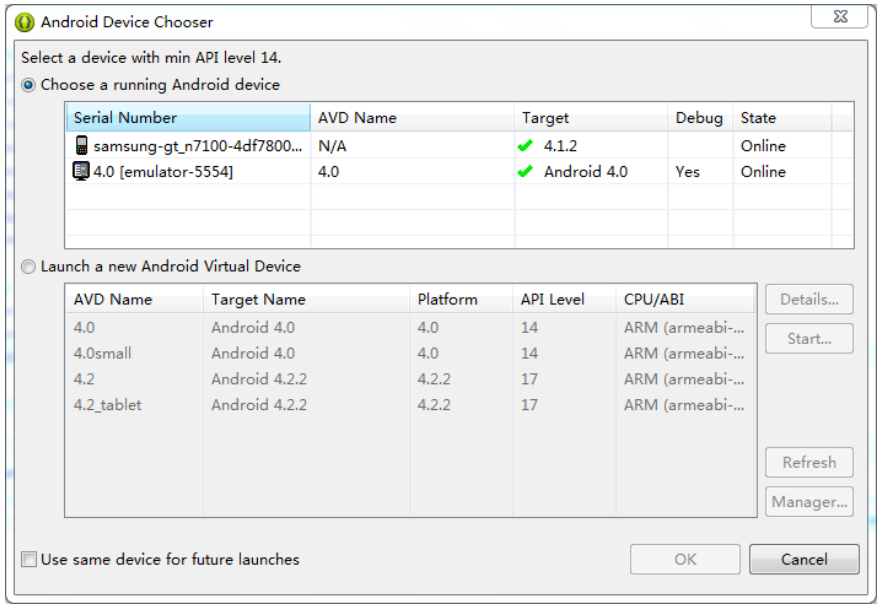

图 7.12

选择第一行的那个设备后点击 OK，就会将程序运行到手机上了。

7.3.2 调用摄像头拍照

很多应用程序都可能会使用到调用摄像头拍照的功能，比如说程序里需要上传一张图片作为用户的头像，这时打开摄像头拍张照是最简单快捷的。下面就让我们通过一个例子来学习一下，如何才能在应用程序里调用手机的摄像头进行拍照。

新建一个 ChoosePicTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/take_photo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Take Photo" />

    <ImageView
```

```
        android:id="@+id/picture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal" />

</LinearLayout>
```

可以看到，布局文件中只有两个控件，一个 Button 和一个 ImageView。Button 是用于打开摄像头进行拍照的，而 ImageView 则是用于将拍到的图片显示出来。

然后开始编写调用摄像头的具体逻辑，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    public static final int TAKE_PHOTO = 1;

    public static final int CROP_PHOTO = 2;

    private Button takePhoto;

    private ImageView picture;

    private Uri imageUri;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        takePhoto = (Button) findViewById(R.id.take_photo);
        picture = (ImageView) findViewById(R.id.picture);
        takePhoto.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 创建File对象，用于存储拍照后的图片
                File outputImage = new File(Environment.
getExternalStorageDirectory(), "tempImage.jpg");
                try {
                    if (outputImage.exists()) {
                        outputImage.delete();
                    }
                    outputImage.createNewFile();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
                imageUri = Uri.fromFile(outputImage);
                Intent intent = new Intent("android.media.action. IMAGE_CAPTURE");
                intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
                startActivityForResult(intent, TAKE_PHOTO); // 启动相机程序
            }
        });
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        switch (requestCode) {
        case TAKE_PHOTO:
            if (resultCode == RESULT_OK) {
                Intent intent = new Intent("com.android.camera.action.CROP");
                intent.setDataAndType(imageUri, "image/*");
                intent.putExtra("scale", true);
                intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
                startActivityForResult(intent, CROP_PHOTO); // 启动裁剪程序
            }
            break;
        case CROP_PHOTO:
            if (resultCode == RESULT_OK) {
                try {
                    Bitmap bitmap = BitmapFactory.decodeStream
(getContentResolver()
                            .openInputStream(imageUri));
                    picture.setImageBitmap(bitmap); // 将裁剪后的照片显示出来
                } catch (FileNotFoundException e) {
                    e.printStackTrace();
                }
            }
            break;
        default:
            break;
        }
    }
```

```
}
```

上述代码稍微有点复杂，我们来仔细地分析一下。在 MainActivity 中要做的第一件事自然是分别获取到 Button 和 ImageView 的实例，并给 Button 注册上点击事件，然后在 Button 的点击事件里开始处理调用摄像头的逻辑，我们重点看下这部分代码。

首先这里创建了一个 File 对象，用于存储摄像头拍下的图片，这里我们把图片命名为 output_image.jpg，并将它存放在手机 SD 卡的根目录下，调用 Environment 的 getExternalStorageDirectory()方法获取到的就是手机 SD 卡的根目录。然后再调用 Uri 的 fromFile()方法将 File 对象转换成 Uri 对象，这个 Uri 对象标识着 output_image.jpg 这张图片的唯一地址。接着构建出一个 Intent 对象，并将这个 Intent 的 action 指定为 android.media.action.IMAGE_CAPTURE，再调用 Intent 的 putExtra()方法指定图片的输出地址，这里填入刚刚得到的 Uri 对象，最后调用 startActivityForResult()来启动活动。由于我们使用的是一个隐式 Intent，系统会找出能够响应这个 Intent 的活动去启动，这样照相机程序就会被打开，拍下的照片将会输出到 output_image.jpg 中。

注意刚才我们是使用 startActivityForResult()来启动活动的，因此拍完照后会有结果返回到 onActivityResult()方法中。如果发现拍照成功，则会再次构建出一个 Intent 对象，并把它的 action 指定为 com.android.camera.action.CROP。这个 Intent 是用于对拍出的照片进行裁剪的，因为摄像头拍出的照片都比较大，而我们可能只希望截取其中的一小部分。然后给这个 Intent 设置上一些必要的属性，并再次调用 startActivityForResult()来启动裁剪程序。裁剪后的照片同样会输出到 output_image.jpg 中。

裁剪操作完成之后，程序又会回调到 onActivityResult()方法中，这个时候我们就可以调用 BitmapFactory 的 decodeStream()方法将 output_image.jpg 这张照片解析成 Bitmap 对象，然后把它设置到 ImageView 中显示出来。

由于这个项目涉及到了向 SD 卡中写数据的操作，因此我们还需要在 AndroidManifest.xml 中声明权限：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.choosepictest"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ……
</manifest>
```

这样代码就都编写完了，现在将程序运行到手机上，然后点击 Take Photo 按钮就可以进行拍照了，如图 7.13 所示。

图 7.13

拍照完成后点击确定则可以对照片进行裁剪，如图 7.14 所示。

图 7.14

点击完成，就会回到我们程序的界面。同时，裁剪后的照片当然也会显示出来了，如图 7.15 所示。

图 7.15

7.3.3 从相册中选择照片

虽然调用摄像头拍照既方便又快捷，但并不是每一次我们都需要去当场拍一张照片的。因为每个人的手机相册里应该都会存有许许多多张照片，直接从相册里选取一张现有的照片会比打开相机拍一张照片更加常用。一个优秀的应用程序应该将这两种选择方式都提供给用户，由用户来决定使用哪一种。下面我们就来看一下，如何才能实现从相册中选择照片的功能。

还是在 ChoosePicTest 项目的基础上进行修改，首先编辑 activity_main.xml 文件，在布局中添加一个按钮用于从相册中选择照片，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/take_photo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
        android:text="Take Photo" />

    <Button
        android:id="@+id/choose_from_album"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Choose From Album" />

    <ImageView
        android:id="@+id/picture"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal" />

</LinearLayout>
```

然后修改 MainActivity 中的代码，加入从相册选择照片的逻辑，代码如下所示：

```
public class MainActivity extends Activity {
    ……
    private Button chooseFromAlbum;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        chooseFromAlbum = (Button) findViewById(R.id.choose_from_album);
        chooseFromAlbum.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                // 创建File对象，用于存储选择的照片
                File outputImage = new File(Environment.
getExternalStorageDirectory(), "output_image.jpg");
                try {
                    if (outputImage.exists()) {
                        outputImage.delete();
                    }
                    outputImage.createNewFile();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
                imageUri = Uri.fromFile(outputImage);
                Intent intent = new Intent("android.intent.action.
GET_CONTENT");
                intent.setType("image/*");
                intent.putExtra("crop", true);
                intent.putExtra("scale", true);
                intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
                startActivityForResult(intent, CROP_PHOTO);
            }
        });
    }
    ......
}
```

可以看到，在 Choose From Album 按钮的点击事件里我们同样创建了一个 File 对象，用于存储从相册中选择的图片。然后构建出一个 Intent 对象，并将它的 action 指定为 android.intent.action.GET_CONTENT。接着给这个 Intent 对象设置一些必要的参数，包括是否允许缩放和裁剪、图片的输出位置等。最后调用 startActivityForResult()方法，就可以打开相册程序选择照片了。

注意在调用 startActivityForResult()方法的时候，我们给第二个参数传入的值仍然是 CROP_PHOTO 常量，这样的好处就是从相册选择好照片之后，会直接进入到 CROP_PHOTO 的 case 下将图片显示出来，这样就可以复用之前写好的显示图片的逻辑，不用再编写一遍了。

现在将程序重新运行到手机上，然后点击一下 Choose From Album 按钮，就会打开相册程序，如图 7.16 所示。

图 7.16

然后随意选择一张照片就可以对它进行裁剪，如图 7.17 所示。

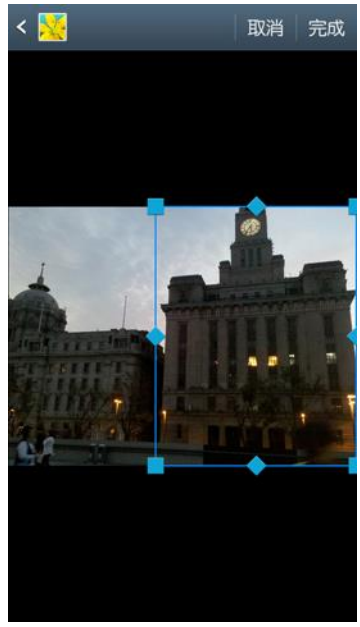

图 7.17

最后点击完成，回到我们程序的界面，就会看到裁剪后的图片已经显示出来了，如图7.17所示。

图　7.17

调用摄像头拍照以及从相册中选择照片是很多 Android 应用都会带有的功能，现在你已经将这两种技术都学会了，将来在工作中如果需要开发类似的功能，相信你一定能轻松完成吧。不过目前我们的实现还不算完美，因为某些照片即使经过裁剪后体积仍然很大，直接加载到内存中有可能会导致程序崩溃。更好的做法是根据项目的需求先对照片进行适当的压缩，然后再加载到内存中。至于如何对照片进行压缩，就要考验你查阅资料的能力了，这里就不再展开进行讲解。

7.4　播放多媒体文件

手机上最常见的休闲方式毫无疑问就是听音乐和看电影了，随着移动设备的普及，越来越多人都可以随时享受优美的音乐，以及观看精彩的电影。而 Android 在播放音频和视频方面也是做了相当不错的支持，它提供了一套较为完整的 API，使得开发者可以很轻松地编写出一个简易的音频或视频播放器，下面我们就来具体地学习一下。

7.4.1 播放音频

在 Android 中播放音频文件一般都是使用 MediaPlayer 类来实现的，它对多种格式的音频文件提供了非常全面的控制方法，从而使得播放音乐的工作变得十分简单。下表列出了 MediaPlayer 类中一些较为常用的控制方法。

方法名	功能描述
setDataSource()	设置要播放的音频文件的位置。
prepare()	在开始播放之前调用这个方法完成准备工作。
start()	开始或继续播放音频。
pause()	暂停播放音频。
reset()	将 MediaPlayer 对象重置到刚刚创建的状态。
seekTo()	从指定的位置开始播放音频。
stop()	停止播放音频。调用这个方法后的 MediaPlayer 对象无法再播放音频。
release()	释放掉与 MediaPlayer 对象相关的资源。
isPlaying()	判断当前 MediaPlayer 是否正在播放音频。
getDuration()	获取载入的音频文件的时长。

简单了解了上述方法后，我们再来梳理一下 MediaPlayer 的工作流程。首先需要创建出一个 MediaPlayer 对象，然后调用 setDataSource()方法来设置音频文件的路径，再调用 prepare()方法使 MediaPlayer 进入到准备状态，接下来调用 start()方法就可以开始播放音频，调用 pause()方法就会暂停播放，调用 reset()方法就会停止播放。

下面就让我们通过一个具体的例子来学习一下吧，新建一个 PlayAudioTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/play"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Play" />

    <Button
```

```
        android:id="@+id/pause"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Pause" />

    <Button
        android:id="@+id/stop"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Stop" />

</LinearLayout>
```

布局文件中横向放置了三个按钮，分别用于对音频文件进行播放、暂停和停止操作。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button play;

    private Button pause;

    private Button stop;

    private MediaPlayer mediaPlayer = new MediaPlayer();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        play = (Button) findViewById(R.id.play);
        pause = (Button) findViewById(R.id.pause);
        stop = (Button) findViewById(R.id.stop);
        play.setOnClickListener(this);
        pause.setOnClickListener(this);
        stop.setOnClickListener(this);
        initMediaPlayer(); // 初始化MediaPlayer
    }
```

```
    private void initMediaPlayer() {
        try {
            File file = new File(Environment.getExternalStorageDirectory(),
"music.mp3");
            mediaPlayer.setDataSource(file.getPath()); // 指定音频文件的路径
            mediaPlayer.prepare(); // 让MediaPlayer进入到准备状态
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.play:
            if (!mediaPlayer.isPlaying()) {
                mediaPlayer.start(); // 开始播放
            }
            break;
        case R.id.pause:
            if (mediaPlayer.isPlaying()) {
                mediaPlayer.pause(); // 暂停播放
            }
            break;
        case R.id.stop:
            if (mediaPlayer.isPlaying()) {
                mediaPlayer.reset(); // 停止播放
                initMediaPlayer();
            }
            break;
        default:
            break;
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mediaPlayer != null) {
            mediaPlayer.stop();
```

```
                mediaPlayer.release();
            }
        }

    }
```

可以看到，在类初始化的时候我们就创建了一个 MediaPlayer 的实例，然后在 onCreate()方法中调用了 initMediaPlayer()方法为 MediaPlayer 对象进行初始化操作。在 initMediaPlayer()方法中，首先是通过创建一个 File 对象来指定音频文件的路径，从这里可以看出，我们需要事先在 SD 卡的根目录下放置一个名为 music.mp3 的音频文件。后面依次调用了 setDataSource()方法和 prepare()方法为 MediaPlayer 做好了播放前的准备。

接下来我们看一下各个按钮的点击事件中的代码。当点击 Play 按钮时会进行判断，如果当前 MediaPlayer 没有正在播放音频，则调用 start()方法开始播放。当点击 Pause 按钮时会判断，如果当前 MediaPlayer 正在播放音频，则调用 pause()方法暂停播放。当点击 Stop 按钮时会判断，如果当前 MediaPlayer 正在播放音频，则调用 reset()方法将 MediaPlayer 重置为刚刚创建的状态，然后重新调用一遍 initMediaPlayer()方法。

最后在 onDestroy()方法中，我们还需要分别调用 stop()和 release()方法，将与 MediaPlayer 相关的资源释放掉。

这样一个简易版的音乐播放器就完成了，现在将程序运行到手机上，界面如图 7.19 所示。

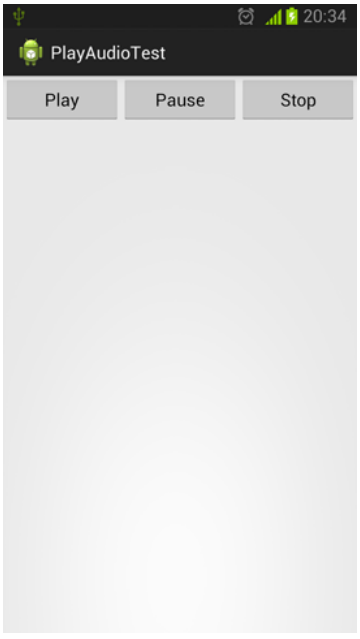

图 7.19

点击一下 Play 按钮就可以听到优美的音乐了，然后点击 Pause 按钮声音会停住，再次点击 Play 按钮会接着暂停之前的位置继续播放。这时如果点击一下 Stop 按钮声音也会停住，但是再次点击 Play 按钮时，音乐就会重头开始播放了。

7.4.2 播放视频

播放视频文件其实并不比播放音频文件复杂，主要是使用 VideoView 类来实现的。这个类将视频的显示和控制集于一身，使得我们仅仅借助它就可以完成一个简易的视频播放器。VideoView 的用法和 MediaPlayer 也比较类似，主要有以下常用方法：

方法名	功能描述
setVideoPath()	设置要播放的视频文件的位置。
start()	开始或继续播放视频。
pause()	暂停播放视频。
resume()	将视频重头开始播放。
seekTo()	从指定的位置开始播放视频。
isPlaying()	判断当前是否正在播放视频。
getDuration()	获取载入的视频文件的时长。

那么我们还是通过一个实际的例子来学习一下吧，新建 PlayVideoTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <VideoView
        android:id="@+id/video_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <Button
            android:id="@+id/play"
```

```
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Play" />

        <Button
            android:id="@+id/pause"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Pause" />

        <Button
            android:id="@+id/replay"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Replay" />
    </LinearLayout>

</LinearLayout>
```

在这个布局文件中，首先是放置了一个 VideoView，稍后的视频就将在这里显示。然后在 VideoView 的下面又放置了三个按钮，分别用于控制视频的播放、暂停和重新播放。

接下来修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private VideoView videoView;

    private Button play;

    private Button pause;

    private Button replay;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        play = (Button) findViewById(R.id.play);
        pause = (Button) findViewById(R.id.pause);
        replay = (Button) findViewById(R.id.replay);
        videoView = (VideoView) findViewById(R.id.video_view);
        play.setOnClickListener(this);
        pause.setOnClickListener(this);
        replay.setOnClickListener(this);
        initVideoPath();
    }

    private void initVideoPath() {
        File file = new File(Environment.getExternalStorageDirectory(),
"movie.3gp");
        videoView.setVideoPath(file.getPath()); // 指定视频文件的路径
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.play:
            if (!videoView.isPlaying()) {
                videoView.start(); // 开始播放
            }
            break;
        case R.id.pause:
            if (videoView.isPlaying()) {
                videoView.pause(); // 暂时播放
            }
            break;
        case R.id.replay:
            if (videoView.isPlaying()) {
                videoView.resume(); // 重新播放
            }
            break;
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
```

```
            if (videoView != null) {
                videoView.suspend();
            }
        }

    }
```

这部分代码相信你理解起来会很轻松，因为它和前面播放音频的代码非常类似。首先在onCreate()方法中仍然是去获取一些控件的实例，然后调用了 initVideoPath()方法来设置视频文件的路径，这里我们需要事先在 SD 卡的根目录下放置一个名为 movie.3gp 的视频文件。

下面看一下各个按钮的点击事件中的代码。当点击 Play 按钮时会进行判断，如果当前并没有正在播放音频，则调用 start()方法开始播放。当点击 Pause 按钮时会判断，如果当前视频正在播放，则调用 pause()方法暂时播放。当点击 Replay 按钮时会判断，如果当前视频正在播放，则调用 resume()方法重头播放视频。

最后在 onDestroy()方法中，我们还需要调用一下 suspend()方法，将 VideoView 所占用的资源释放掉。

现在将程序运行到手机上，然后点击一下 Play 按钮，就可以看到视频已经开始播放了，如图 7.20 所示。

图 7.20

点击 Pause 按钮可以暂停视频的播放，点击 Replay 按钮可以重头播放视频。

这样的话，你就已经将 VideoView 的基本用法掌握得差不多了。不过，为什么它的用法和 MediaPlayer 这么相似呢？其实 VideoView 只是帮我们做了一个很好的封装而已，它的背后仍然是使用 MediaPlayer 来对视频文件进行控制的。另外需要注意，VideoView 并不是一个万能的视频播放工具类，它在视频格式的支持以及播放效率方面都存在着较大的不足。所以，如果想要仅仅使用 VideoView 就编写出一个功能非常强大的视频播放器是不太现实的。但是如果只是用于播放一些游戏的片头动画，或者某个应用的视频宣传，使用 VideoView 还是绰绰有余的。

好了，关于 Android 多媒体方面的知识你已经学得足够多了，下面就让我们一起来总结一下本章所学的内容吧。

7.5 小结与点评

本章我们主要对 Android 系统中的各种多媒体技术进行了学习，其中包括通知的使用技巧、调用摄像头拍照、从相册中选取照片，以及播放音频和视频文件。由于所涉及的多媒体技术在模拟器上很难看得到效果，因此本章中还特意讲解了在 Android 手机上调试程序的方法。此外，我们还学习了如何使用 Android 提供的 API 来接收、发送和拦截短信，这使得我们甚至可以编写一个自己的短信程序来替换掉系统的短信程序。

又是充实饱满的一章啊！现在多媒体方面的知识已经学得足够多了，我希望你可以很好地将它们消化掉，尤其是与通知相关的内容，因为在下一章的学习当中我们还会用到它。在进行了一章多媒体相关知识的学习之后，你是否想起来 Android 四大组件中还剩一个没有学过呢，那么下面就让我们进入到 Android 服务的学习旅程之中。

第 8 章　使用网络技术

如果你在玩手机的时候不能上网，那你一定会感到特别的枯燥乏味。没错，现在早已不是玩单机的时代了，无论是 PC、手机、平板、还是电视几乎都会具备上网的功能，到未来甚至是手表、眼镜、拖鞋等等设备也可能会逐个加入到这个行列，21 世纪的确是互联网的时代。

那么不用多说，Android 手机肯定也是可以上网的，所以作为开发者的我们就需要考虑如何利用网络来编写出更加出色的应用程序，像 QQ、微博、新闻等常见的应用都会大量地使用到网络技术。本章主要会讲述如何在手机端使用 HTTP 协议和服务器端进行网络交互，并对服务器返回的数据进行解析，这也是 Android 中最常使用到的网络技术了，下面就让我们一起来学习一下吧。

8.1　WebView 的用法

有时候我们可能会碰到一些比较特殊的需求，比如说要求在应用程序里展示一些网页。相信每个人都知道，加载和显示网页通常都是浏览器的任务，但是需求里又明确指出，不允许打开系统浏览器，而我们当然也不可能自己去编写一个浏览器出来，这时应该怎么办呢？

不用担心，Android 早就已经考虑到了这种需求，并提供了一个 WebView 控件，借助它我们就可以在自己的应用程序里嵌入一个浏览器，从而非常轻松地展示各种各样的网页。

WebView 的用法也是相当简单，下面我们就通过一个例子来学习一下吧。新建一个 WebViewTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <WebView
        android:id="@+id/web_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

可以看到，我们在布局文件中使用到了一个新的控件，WebView。这个控件当然也就是用来显示网页的了，这里的写法很简单，给它设置了一个 id，并让它充满整个屏幕。

然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity {

    private WebView webView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        webView = (WebView) findViewById(R.id.web_view);
        webView.getSettings().setJavaScriptEnabled(true);
        webView.setWebViewClient(new WebViewClient() {
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String
url) {
                view.loadUrl(url); // 根据传入的参数再去加载新的网页
                return true; // 表示当前WebView可以处理打开新网页的请求，不用借助
系统浏览器
            }
        });
        webView.loadUrl("http://www.baidu.com");
    }

}
```

MainActivity 中的代码也很短，首先使用 findViewById()方法获取到了 WebView 的实例，然后调用 WebView 的 getSettings()方法可以去设置一些浏览器的属性，这里我们并不去设置过多的属性，只是调用了 setJavaScriptEnabled()方法来让 WebView 支持 JavaScript 脚本。

接下来是非常重要的一个部分，我们调用了 WebView 的 setWebViewClient()方法，并传入了 WebViewClient 的匿名类作为参数，然后重写了 shouldOverrideUrlLoading()方法。这就表明当需要从一个网页跳转到另一个网页时，我们希望目标网页仍然在当前 WebView 中显示，而不是打开系统浏览器。

最后一步就非常简单了，调用 WebView 的 loadUrl()方法，并将网址传入，即可展示相应网页的内容，这里就让我们看一看百度的首页是长什么样的吧。

另外还需要注意，由于本程序使用到了网络功能，而访问网络是需要声明权限的，因此我们还得修改 AndroidManifest.xml 文件，并加入权限声明，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.webviewtest"
```

```
    android:versionCode="1"
    android:versionName="1.0" >
    ......
    <uses-permission android:name="android.permission.INTERNET" />
    ......
</manifest>
```

在开始运行之前，首先需要保证你的手机或模拟器是联网的，如果你使用的是模拟器，只需保证电脑能正常上网即可。然后就可以运行一下程序了，效果如图 8.1 所示。

图　8.1

可以看到，WebViewTest 这个程序现在已经具备了一个简易浏览器的功能，不仅成功将百度的首页展示了出来，还可以通过点击链接浏览更多的网页。

当然，WebView 还有很多更加高级的使用技巧，我们就不再继续进行探讨了，因为那不是本章的重点。这里先介绍了一下 WebView 的用法，只是希望你能对 HTTP 协议的使用有一个最基本的认识，接下来我们就要利用这个协议来做一些真正的网络开发工作了。

8.2 使用 HTTP 协议访问网络

如果真的说要去深入分析 HTTP 协议，可能需要花费整整一本书的篇幅。这里我当然不会这么干，因为毕竟你是跟着我学习 Android 开发的，而不是网站开发。对于 HTTP 协议，你只需要稍微了解一些就足够了，它的工作原理特别的简单，就是客户端向服务器发出一条 HTTP 请求，服务器收到请求之后会返回一些数据给客户端，然后客户端再对这些数据进行解析和处理就可以了。是不是非常简单？一个浏览器的基本工作原理也就是如此了。比如说上一节中使用到的 WebView 控件，其实也就是我们向百度的服务器发起了一条 HTTP 请求，接着服务器分析出我们想要访问的是百度的首页，于是会把该网页的 HTML 代码进行返回，然后 WebView 再调用手机浏览器的内核对返回的 HTML 代码进行解析，最终将页面展示出来。

简单来说，WebView 已经在后台帮我们处理好了发送 HTTP 请求、接收服务响应、解析返回数据，以及最终的页面展示这几步工作，不过由于它封装得实在是太好了，反而使得我们不能那么直观地看出 HTTP 协议到底是如何工作的。因此，接下来就让我们通过手动发送 HTTP 请求的方式，来更加深入地理解一下这个过程。

8.2.1 使用 HttpURLConnection

在 Android 上发送 HTTP 请求的方式一般有两种，HttpURLConnection 和 HttpClient，本小节我们先来学习一下 HttpURLConnection 的用法。

首先需要获取到 HttpURLConnection 的实例，一般只需 new 出一个 URL 对象，并传入目标的网络地址，然后调用一下 openConnection()方法即可，如下所示：

```
URL url = new URL("http://www.baidu.com");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
```

得到了 HttpURLConnection 的实例之后，我们可以设置一下 HTTP 请求所使用的方法。常用的方法主要有两个，GET 和 POST。GET 表示希望从服务器那里获取数据，而 POST 则表示希望提交数据给服务器。写法如下：

```
connection.setRequestMethod("GET");
```

接下来就可以进行一些自由的定制了，比如设置连接超时、读取超时的毫秒数，以及服务器希望得到的一些消息头等。这部分内容根据自己的实际情况进行编写，示例写法如下：

```
connection.setConnectTimeout(8000);
connection.setReadTimeout(8000);
```

之后再调用 getInputStream()方法就可以获取到服务器返回的输入流了，剩下的任务就是对输入流进行读取，如下所示：

```
InputStream in = connection.getInputStream();
```

最后可以调用 disconnect()方法将这个 HTTP 连接关闭掉，如下所示：

```
connection.disconnect();
```

下面就让我们通过一个具体的例子来真正体验一下 HttpURLConnection 的用法。新建一个 NetworkTest 项目，首先修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/send_request"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Send Request" />

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <TextView
            android:id="@+id/response"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </ScrollView>

</LinearLayout>
```

注意这里我们使用了一个新的控件，ScrollView，它是用来做什么的呢？由于手机屏幕的空间一般都比较小，有些时候过多的内容一屏是显示不下的，借助 ScrollView 控件的话就可以允许我们以滚动的形式查看屏幕外的那部分内容。另外，布局中还放置了一个 Button 和一个 TextView，Button 用于发送 HTTP 请求，TextView 用于将服务器返回的数据显示出来。

接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    public static final int SHOW_RESPONSE = 0;

    private Button sendRequest;
```

```
private TextView responseText;

private Handler handler = new Handler() {

    public void handleMessage(Message msg) {
        switch (msg.what) {
        case SHOW_RESPONSE:
            String response = (String) msg.obj;
            // 在这里进行UI操作，将结果显示到界面上
            responseText.setText(response);
        }
    }

};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    sendRequest = (Button) findViewById(R.id.send_request);
    responseText = (TextView) findViewById(R.id.response_text);
    sendRequest.setOnClickListener(this);
}

@Override
public void onClick(View v) {
    if (v.getId() == R.id.send_request) {
        sendRequestWithHttpURLConnection();
    }
}

private void sendRequestWithHttpURLConnection() {
    // 开启线程来发起网络请求
    new Thread(new Runnable() {
        @Override
        public void run() {
            HttpURLConnection connection = null;
            try {
                URL url = new URL("http://www.baidu.com");
                connection = (HttpURLConnection) url.openConnection();
```

```
                    connection.setRequestMethod("GET");
                    connection.setConnectTimeout(8000);
                    connection.setReadTimeout(8000);
                    InputStream in = connection.getInputStream();
                    // 下面对获取到的输入流进行读取
                    BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
                    StringBuilder response = new StringBuilder();
                    String line;
                    while ((line = reader.readLine()) != null) {
                        response.append(line);
                    }
                    Message message = new Message();
                    message.what = SHOW_RESPONSE;
                    // 将服务器返回的结果存放到Message中
                    message.obj = response.toString();
                    handler.sendMessage(message);
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    if (connection != null) {
                        connection.disconnect();
                    }
                }
            }
        }).start();
    }

}
```

可以看到，我们在 Send Request 按钮的点击事件里调用了 sendRequestWithHttpURL-Connection()方法，在这个方法中先是开启了一个子线程，然后在子线程里使用 HttpURLConnection 发出一条 HTTP 请求，请求的目标地址就是百度的首页。接着利用 BufferedReader 对服务器返回的流进行读取，并将结果存放到了一个 Message 对象中。这里为什么要使用 Message 对象呢？当然是因为子线程中无法对 UI 进行操作了。我们希望可以将服务器返回的内容显示到界面上，所以就创建了一个 Message 对象，并使用 Handler 将它发送出去。之后又在 Handler 的 handleMessage()方法中对这条 Message 进行处理，最终取出结果并设置到 TextView 上。

完整的一套流程就是这样，不过在开始运行之前，仍然别忘了要声明一下网络权限。修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.networktest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <uses-permission android:name="android.permission.INTERNET" />
    ……
</manifest>
```

好了，现在运行一下程序，并点击 Send Request 按钮，结果如图 8.2 所示。

图 8.2

是不是看得头晕眼花？没错，服务器返回给我们的就是这种 HTML 代码，只是通常情况下浏览器都会将这些代码解析成漂亮的网页后再展示出来。

那么如果是想要提交数据给服务器应该怎么办呢？其实也不复杂，只需要将 HTTP 请求的方法改成 POST，并在获取输入流之前把要提交的数据写出即可。注意每条数据都要以键值对的形式存在，数据与数据之间用&符号隔开，比如说我们想要向服务器提交用户名和密码，就可以这样写：

```
connection.setRequestMethod("POST");
DataOutputStream out = new DataOutputStream(connection.getOutputStream());
out.writeBytes("username=admin&password=123456");
```

好了，相信你已经将 HttpURLConnection 的用法很好地掌握了，下面我们来学习一下 HttpClient 的用法吧。

8.2.2 使用 HttpClient

HttpClient 是 Apache 提供的 HTTP 网络访问接口，从一开始的时候就被引入到了 Android API 中。它可以完成和 HttpURLConnection 几乎一模一样的效果，但两者之间的用法却有较大的差别，那么我们自然要看一下 HttpClient 是如何使用的了。

首先你需要知道，HttpClient 是一个接口，因此无法创建它的实例，通常情况下都会创建一个 DefaultHttpClient 的实例，如下所示：

```
HttpClient httpClient = new DefaultHttpClient();
```

接下来如果想要发起一条 GET 请求，就可以创建一个 HttpGet 对象，并传入目标的网络地址，然后调用 HttpClient 的 execute()方法即可：

```
HttpGet httpGet = new HttpGet("http://www.baidu.com");
httpClient.execute(httpGet);
```

如果是发起一条 POST 请求会比 GET 稍微复杂一点，我们需要创建一个 HttpPost 对象，并传入目标的网络地址，如下所示：

```
HttpPost httpPost = new HttpPost("http://www.baidu.com");
```

然后通过一个 NameValuePair 集合来存放待提交的参数，并将这个参数集合传入到一个 UrlEncodedFormEntity 中，然后调用 HttpPost 的 setEntity()方法将构建好的 UrlEncodedFormEntity 传入，如下所示：

```
List<NameValuePair> params = new ArrayList<NameValuePair>();
params.add(new BasicNameValuePair("username", "admin"));
params.add(new BasicNameValuePair("password", "123456"));
UrlEncodedFormEntity entity = new UrlEncodedFormEntity(params, "utf-8");
httpPost.setEntity(entity);
```

接下来的操作就和 HttpGet 一样了，调用 HttpClient 的 execute()方法，并将 HttpPost 对象传入即可：

```
httpClient.execute(httpPost);
```

执行 execute()方法之后会返回一个 HttpResponse 对象，服务器所返回的所有信息就会包含在这里面。通常情况下我们都会先取出服务器返回的状态码，如果等于 200 就说明请求和

响应都成功了，如下所示：

```
if (httpResponse.getStatusLine().getStatusCode() == 200) {
    // 请求和响应都成功了
}
```

接下来在这个 if 判断的内部取出服务返回的具体内容，可以调用 getEntity()方法获取到一个 HttpEntity 实例，然后再用 EntityUtils.toString()这个静态方法将 HttpEntity 转换成字符串即可，如下所示：

```
HttpEntity entity = httpResponse.getEntity();
String response = EntityUtils.toString(entity);
```

注意如果服务器返回的数据是带有中文的，直接调用 EntityUtils.toString()方法进行转换会有乱码的情况出现，这个时候只需要在转换的时候将字符集指定成 utf-8 就可以了，如下所示：

```
String response = EntityUtils.toString(entity, "utf-8");
```

好了，基本的用法就是如此，接下来就让我们把 NetworkTest 这个项目改用 HttpClient 的方式再实现一遍吧。

由于布局部分完全不用改动，所以现在直接修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ......
    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.send_request) {
            sendRequestWithHttpClient();
        }
    }

    private void sendRequestWithHttpClient() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    HttpClient httpClient = new DefaultHttpClient();
                    HttpGet httpGet = new HttpGet("http://www.baidu.com");
                    HttpResponse httpResponse = httpClient.execute(httpGet);
                    if (httpResponse.getStatusLine().getStatusCode() == 200) {
                        // 请求和响应都成功了
                        HttpEntity entity = httpResponse.getEntity();
```

```
                        String response = EntityUtils.toString(entity,
"utf-8");
                        Message message = new Message();
                        message.what = SHOW_RESPONSE;
                        // 将服务器返回的结果存放到Message中
                        message.obj = response.toString();
                        handler.sendMessage(message);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
    ......
}
```

这里我们并没有做太多的改动，只是添加了一个 sendRequestWithHttpClient()方法，并在 Send Request 按钮的点击事件里去调用这个方法。在这个方法中同样还是先开启了一个子线程，然后在子线程里使用 HttpClient 发出一条 HTTP 请求，请求的目标地址还是百度的首页，HttpClient 的用法也正如前面所介绍的一样。然后为了能让结果在界面上显示出来，这里仍然是将服务器返回的数据存放到了 Message 对象中，并用 Handler 将 Message 发送出去。

仅仅只是改了这么多代码，现在我们可以重新运行一下程序了。点击 Send Request 按钮后，你会看到和上一小节中同样的运行结果，由此证明，使用 HttpClient 来发送 HTTP 请求的功能也已经成功实现了。

这样的话，相信你就已经把 HttpURLConnection 和 HttpClient 的基本用法都掌握得差不多了。

8.3 解析 XML 格式数据

通常情况下，每个需要访问网络的应用程序都会有一个自己的服务器，我们可以向服务器提交数据，也可以从服务器上获取数据。不过这个时候就出现了一个问题，这些数据到底要以什么样的格式在网络上传输呢？随便传递一段文本肯定是不行的，因为另一方根本就不会知道这段文本的用途是什么。因此，一般我们都会在网络上传输一些格式化后的数据，这种数据会有一定的结构规格和语义，当另一方收到数据消息之后就可以按照相同的结构规格进行解析，从而取出他想要的那部分内容。

在网络上传输数据时最常用的格式有两种，XML 和 JSON，下面我们就来一个个地进行

学习，本节首先学一下如何解析 XML 格式的数据。

在开始之前我们还需要先解决一个问题，就是从哪儿才能获取一段 XML 格式的数据呢？这里我准备教你搭建一个最简单的 Web 服务器，在这个服务器上提供一段 XML 文本，然后我们在程序里去访问这个服务器，再对得到的 XML 文本进行解析。

搭建 Web 服务器其实非常简单，有很多的服务器类型可供选择，这里我准备使用 Apache 服务器。首先你需要去下载一个 Apache 服务器的安装包，下载地址是：http://httpd.apache.org/download.cgi。

下载完成后双击就可以进行安装了，如图 8.3 所示。

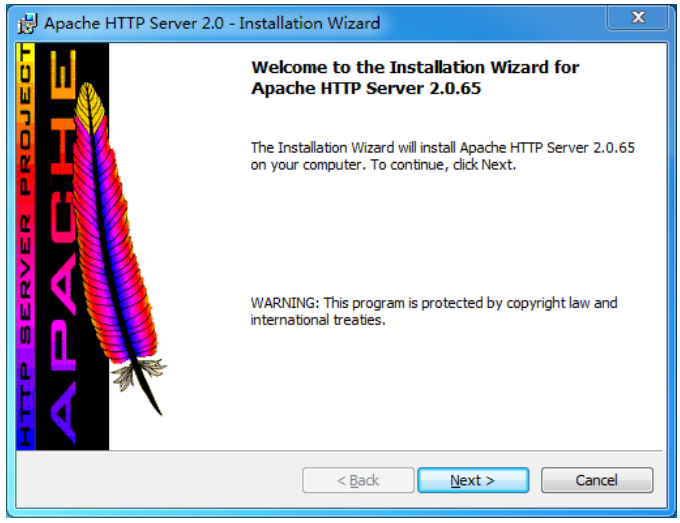

图 8.3

然后一直点击 Next，会提示让你输入自己的域名，我们随便填一个域名就可以了，如图 8.4 所示。

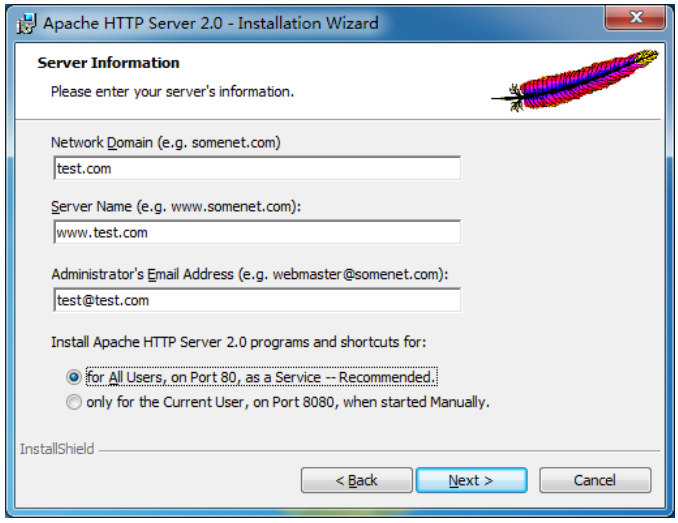

图 8.4

接着继续一直点击 Next 会提示让你选择程序安装的路径，这里我选择安装到 C:\Apache 目录下，之后再继续点击 Next 就可以完成安装了。安装成功后服务器会自动启动起来，你可以打开电脑的浏览器来验证一下。在地址栏输入 127.0.0.1，如果出现了如图 8.5 所示的界面，就说明服务器已经启动成功了。

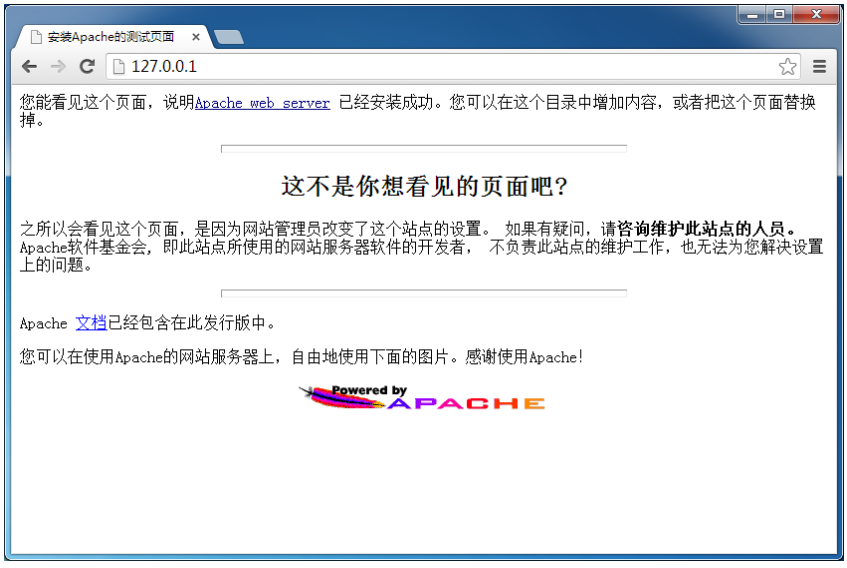

图 8.5

接下来进入到 C:\Apache\Apache2\htdocs 目录下，在这里新建一个名为 get_data.xml 的文件，然后编辑这个文件，并加入如下 XML 格式的内容。

```
<apps>
    <app>
        <id>1</id>
        <name>Google Maps</name>
        <version>1.0</version>
    </app>
    <app>
        <id>2</id>
        <name>Chrome</name>
        <version>2.1</version>
    </app>
    <app>
        <id>3</id>
        <name>Google Play</name>
        <version>2.3</version>
    </app>
</apps>
```

这时在浏览器中访问 http://127.0.0.1/get_data.xml 这个网址，就应该出现如图 8.6 所示的内容。

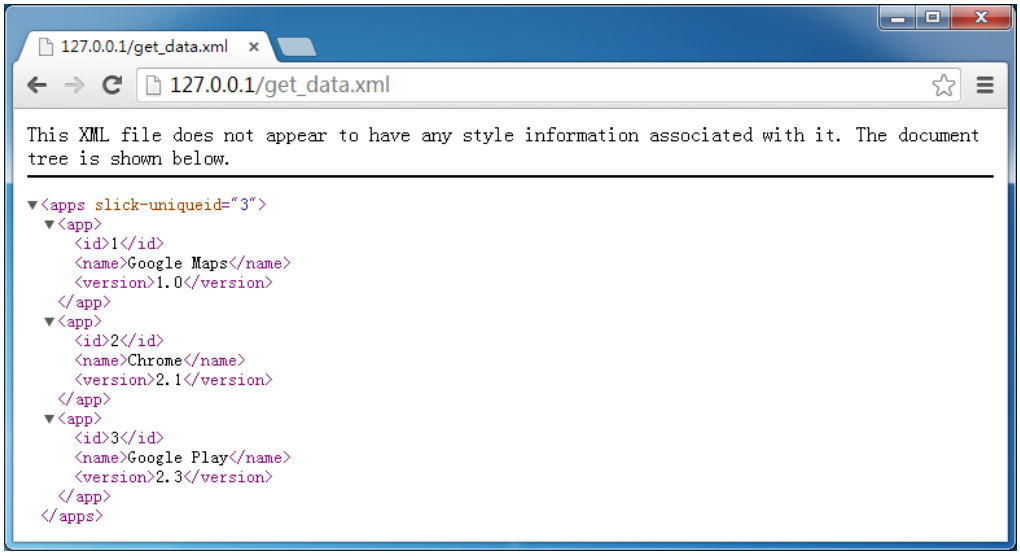

图　8.6

好了，准备工作到此结束，接下来就让我们在 Android 程序里去获取并解析这段 XML 数据吧。

8.3.1 Pull 解析方式

解析 XML 格式的数据其实也有挺多种方式的，本节中我们学习比较常用的两种，Pull 解析和 SAX 解析。那么简单起见，这里仍然是在 NetworkTest 项目的基础上继续开发，这样我们就可以重用之前网络通信部分的代码，从而把工作的重心放在 XML 数据解析上。

既然 XML 格式的数据已经提供好了，现在要做的就是从中解析出我们想要得到的那部分内容。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ......
    private void sendRequestWithHttpClient() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    HttpClient httpClient = new DefaultHttpClient();
                    // 指定访问的服务器地址是电脑本机
                    HttpGet httpGet = new HttpGet("http://8.0.2.2/get_data.xml");
                    HttpResponse httpResponse = httpClient.execute(httpGet);
                    if (httpResponse.getStatusLine().getStatusCode() == 200) {
                        // 请求和响应都成功了
                        HttpEntity entity = httpResponse.getEntity();
                        String response = EntityUtils.toString(entity,
"utf-8");
                        parseXMLWithPull(response);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    private void parseXMLWithPull(String xmlData) {
        try {
            XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
            XmlPullParser xmlPullParser = factory.newPullParser();
```

```
                xmlPullParser.setInput(new StringReader(xmlData));
                int eventType = xmlPullParser.getEventType();
                String id = "";
                String name = "";
                String version = "";
                while (eventType != XmlPullParser.END_DOCUMENT) {
                    String nodeName = xmlPullParser.getName();
                    switch (eventType) {
                    // 开始解析某个结点
                    case XmlPullParser.START_TAG: {
                        if ("id".equals(nodeName)) {
                            id = xmlPullParser.nextText();
                        } else if ("name".equals(nodeName)) {
                            name = xmlPullParser.nextText();
                        } else if ("version".equals(nodeName)) {
                            version = xmlPullParser.nextText();
                        }
                        break;
                    }
                    // 完成解析某个结点
                    case XmlPullParser.END_TAG: {
                        if ("app".equals(nodeName)) {
                            Log.d("MainActivity", "id is " + id);
                            Log.d("MainActivity", "name is " + name);
                            Log.d("MainActivity", "version is " + version);
                        }
                        break;
                    }
                    default:
                        break;
                    }
                    eventType = xmlPullParser.next();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

可以看到，这里首先是将 HTTP 请求的地址改成了 http://8.0.2.2/get_data.xml，8.0.2.2 对

于模拟器来说就是电脑本机的 IP 地址。在得到了服务器返回的数据后，我们并不再去发送一条消息，而是调用了 parseXMLWithPull()方法来解析服务器返回的数据。

下面就来仔细看下 parseXMLWithPull()方法中的代码吧。这里首先要获取到一个 XmlPullParserFactory 的实例，并借助这个实例得到 XmlPullParser 对象，然后调用 XmlPullParser 的 setInput()方法将服务器返回的 XML 数据设置进去就可以开始解析了。解析的过程也是非常简单，通过 getEventType()可以得到当前的解析事件，然后在一个 while 循环中不断地进行解析，如果当前的解析事件不等于 XmlPullParser.END_DOCUMENT，说明解析工作还没完成，调用 next()方法后可以获取下一个解析事件。

在 while 循环中，我们通过 getName()方法得到当前结点的名字，如果发现结点名等于 id、name 或 version，就调用 nextText()方法来获取结点内具体的内容，每当解析完一个 app 结点后就将获取到的内容打印出来。

好了，整体的过程就是这么简单，下面就让我们来测试一下吧。运行 NetworkTest 项目，然后点击 Send Request 按钮，观察 LogCat 中的打印日志，如图 8.7 所示。

Tag	Text
MainActivity	id is 1
MainActivity	name is Google Maps
MainActivity	version is 1.0
MainActivity	id is 2
MainActivity	name is Chrome
MainActivity	version is 2.1
MainActivity	id is 3
MainActivity	name is Google Play
MainActivity	version is 2.3

图 8.7

可以看到，我们已经将 XML 数据中的指定内容成功解析出来了。

8.3.2 SAX 解析方式

Pull 解析方式虽然非常的好用，但它并不是我们唯一的选择。SAX 解析也是一种特别常用的 XML 解析方式，虽然它的用法比 Pull 解析要复杂一些，但在语义方面会更加的清楚。

通常情况下我们都会新建一个类继承自 DefaultHandler，并重写父类的五个方法，如下所示：

```
public class MyHandler extends DefaultHandler {
```

```
    @Override
    public void startDocument() throws SAXException {
    }

    @Override
    public void startElement(String uri, String localName, String qName,
Attributes attributes) throws SAXException {
    }

    @Override
    public void characters(char[] ch, int start, int length) throws
SAXException {
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws
SAXException {
    }

    @Override
    public void endDocument() throws SAXException {
    }

}
```

这五个方法一看就很清楚吧？startDocument()方法会在开始 XML 解析的时候调用，startElement()方法会在开始解析某个结点的时候调用，characters()方法会在获取结点中内容的时候调用，endElement()方法会在完成解析某个结点的时候调用，endDocument()方法会在完成整个 XML 解析的时候调用。其中，startElement()、characters()和 endElement()这三个方法是有参数的，从 XML 中解析出的数据就会以参数的形式传入到这些方法中。需要注意的是，在获取结点中的内容时，characters()方法可能会被调用多次，一些换行符也被当作内容解析出来，我们需要针对这种情况在代码中做好控制。

那么下面就让我们尝试用 SAX 解析的方式来实现和上一小节中同样的功能吧。新建一个 ContentHandler 类继承自 DefaultHandler，并重写父类的五个方法，如下所示：

```
public class ContentHandler extends DefaultHandler {

    private String nodeName;
```

```
    private StringBuilder id;

    private StringBuilder name;

    private StringBuilder version;

    @Override
    public void startDocument() throws SAXException {
        id = new StringBuilder();
        name = new StringBuilder();
        version = new StringBuilder();
    }

    @Override
    public void startElement(String uri, String localName, String qName,
Attributes attributes) throws SAXException {
        // 记录当前结点名
        nodeName = localName;
    }

    @Override
    public void characters(char[] ch, int start, int length) throws
SAXException {
        // 根据当前的结点名判断将内容添加到哪一个StringBuilder对象中
        if ("id".equals(nodeName)) {
            id.append(ch, start, length);
        } else if ("name".equals(nodeName)) {
            name.append(ch, start, length);
        } else if ("version".equals(nodeName)) {
            version.append(ch, start, length);
        }
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws
SAXException {
        if ("app".equals(localName)) {
            Log.d("ContentHandler", "id is " + id.toString().trim());
            Log.d("ContentHandler", "name is " + name.toString().trim());
```

```
            Log.d("ContentHandler", "version is " + version.toString().trim());
            // 最后要将StringBuilder清空掉
            id.setLength(0);
            name.setLength(0);
            version.setLength(0);
        }
    }

    @Override
    public void endDocument() throws SAXException {
    }

}
```

可以看到，我们首先给 id、name 和 version 结点分别定义了一个 StringBuilder 对象，并在 startDocument()方法里对它们进行了初始化。每当开始解析某个结点的时候，startElement()方法就会得到调用，其中 localName 参数记录着当前结点的名字，这里我们把它记录下来。接着在解析结点中具体内容的时候就会调用 characters()方法，我们会根据当前的结点名进行判断，将解析出的内容添加到哪一个 StringBuilder 对象中。最后在 endElement()方法中进行判断，如果 app 结点已经解析完成，就打印出 id、name 和 version 的内容。需要注意的是，目前 id、name 和 version 中都可能是包括回车或换行符的，因此在打印之前我们还需要调用一下 trim()方法，并且打印完成后还要将 StringBuilder 的内容清空掉，不然的话会影响下一次内容的读取。

接下来的工作就非常简单了，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    private void sendRequestWithHttpClient() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    HttpClient httpClient = new DefaultHttpClient();
                    // 指定访问的服务器地址是电脑本机
                    HttpGet httpGet = new HttpGet("http://8.0.2.2:8080/
get_data.xml");
                    HttpResponse httpResponse = httpClient.execute(httpGet);
                    if (httpResponse.getStatusLine().getStatusCode() == 200) {
                        // 请求和响应都成功了
```

```
                        HttpEntity entity = httpResponse.getEntity();
                        String response = EntityUtils.toString(entity,
"utf-8");
                        parseXMLWithSAX(response);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
    ......
    private void parseXMLWithSAX(String xmlData) {
        try {
            SAXParserFactory factory = SAXParserFactory.newInstance();
            XMLReader xmlReader = factory.newSAXParser().getXMLReader();
            ContentHandler handler = new ContentHandler();
            // 将ContentHandler的实例设置到XMLReader中
            xmlReader.setContentHandler(handler);
            // 开始执行解析
            xmlReader.parse(new InputSource(new StringReader(xmlData)));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在得到了服务器返回的数据后，我们这次去调用 parseXMLWithSAX()方法来解析 XML 数据。parseXMLWithSAX()方法中先是创建了一个 SAXParserFactory 的对象，然后再获取到 XMLReader 对象，接着将我们编写的 ContentHandler 的实例设置到 XMLReader 中，最后调用 parse()方法开始执行解析就好了。

现在重新运行一下程序，点击 Send Request 按钮后观察 LogCat 中的打印日志，你会看到和图 8.7 中一样的结果。

除了 Pull 解析和 SAX 解析之外，其实还有一种 DOM 解析方式也算挺常用的，不过这里我们就不再展开进行讲解了，感兴趣的话你可以自己去查阅一下相关资料。

8.4 解析 JSON 格式数据

现在你已经掌握了 XML 格式数据的解析方式，那么接下来我们要去学习一下如何解析 JSON 格式的数据了。比起 XML，JSON 的主要优势在于它的体积更小，在网络上传输的时候可以更省流量。但缺点在于，它的语义性较差，看起来不如 XML 直观。

在开始之前，我们还需要在 C:\Apache\Apache2\htdocs 目录中新建一个 get_data.json 的文件，然后编辑这个文件，并加入如下 JSON 格式的内容：

```
[{"id":"5","version":"5.5","name":"Angry Birds"},
{"id":"6","version":"7.0","name":"Clash of Clans"},
{"id":"7","version":"3.5","name":"Hey Day"}]
```

这时在浏览器中访问 http://127.0.0.1/get_data.json 这个网址，就应该出现如图 8.8 所示的内容。

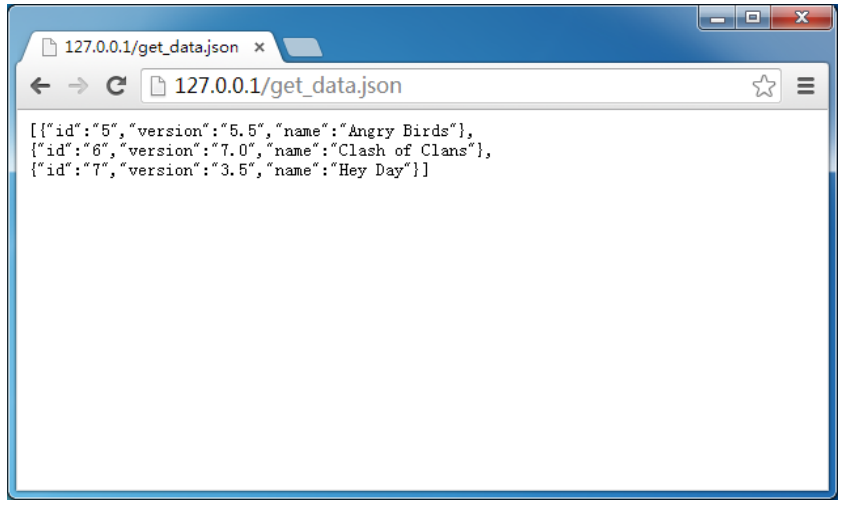

图 8.8

好了，这样我们把 JSON 格式的数据也准备好了，下面就开始学习如何在 Android 程序中解析这些数据吧。

8.4.1 使用 JSONObject

类似地，解析 JSON 数据也有很多种方法，可以使用官方提供的 JSONObject，也可以使用谷歌的开源库 GSON。另外，一些第三方的开源库如 Jackson、FastJSON 等也非常不错。本节中我们就来学习一下前两种解析方式的用法。

修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    private void sendRequestWithHttpClient() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    HttpClient httpClient = new DefaultHttpClient();
                    // 指定访问的服务器地址是电脑本机
                    HttpGet httpGet = new HttpGet("http://8.0.2.2/
get_data.json");
                    HttpResponse httpResponse = httpClient.execute(httpGet);
                    if (httpResponse.getStatusLine().getStatusCode() == 200) {
                        // 请求和响应都成功了
                        HttpEntity entity = httpResponse.getEntity();
                        String response = EntityUtils.toString(entity,
"utf-8");
                        parseJSONWithJSONObject(response);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
    ……
    private void parseJSONWithJSONObject(String jsonData) {
        try {
            JSONArray jsonArray = new JSONArray(jsonData);
            for (int i = 0; i < jsonArray.length(); i++) {
                JSONObject jsonObject = jsonArray.getJSONObject(i);
                String id = jsonObject.getString("id");
                String name = jsonObject.getString("name");
                String version = jsonObject.getString("version");
                Log.d("MainActivity", "id is " + id);
                Log.d("MainActivity", "name is " + name);
                Log.d("MainActivity", "version is " + version);
            }
```

```
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

首先记得要将 HTTP 请求的地址改成 http://8.0.2.2/get_data.json，然后在得到了服务器返回的数据后调用 parseJSONWithJSONObject()方法来解析数据。可以看到，解析 JSON 的代码真的是非常简单，由于我们在服务器中定义的是一个 JSON 数组，因此这里首先是将服务器返回的数据传入到了一个 JSONArray 对象中。然后循环遍历这个 JSONArray，从中取出的每一个元素都是一个 JSONObject 对象，每个 JSONObject 对象中又会包含 id、name 和 version 这些数据。接下来只需要调用 getString()方法将这些数据取出，并打印出来即可。

好了，就是这么简单！现在重新运行一下程序，并点击 Send Request 按钮，结果如图 8.9 所示。

Tag	Text
MainActivity	id is 5
MainActivity	name is Angry Birds
MainActivity	version is 5.5
MainActivity	id is 6
MainActivity	name is Clash of Clans
MainActivity	version is 7.0
MainActivity	id is 7
MainActivity	name is Hey Day
MainActivity	version is 3.5

图 8.9

8.4.2 使用 GSON

如何你认为使用 JSONObject 来解析 JSON 数据已经非常简单了，那你就太容易满足了。谷歌提供的 GSON 开源库可以让解析 JSON 数据的工作简单到让你不敢想象的地步，那我们肯定是不能错过这个学习机会的。

不过 GSON 并没有被添加到 Android 官方的 API 中，因此如果想要使用这个功能的话，则必须要在项目中添加一个 GSON 的 Jar 包。首先我们需要将 GSON 的资源压缩包下载下来，下载地址是：http://code.google.com/p/google-gson/downloads/list。

然后将资源包进行解压，会看到如图 8.8 所示的几个文件。

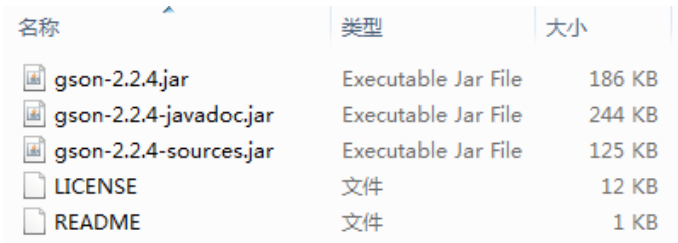

图 8.8

其中 gson-2.2.4.jar 这个文件就是我们所需要的了，现在将它拷贝到 NetworkTest 项目的 libs 目录下，GSON 库就会自动添加到 NetworkTest 项目中了，如图 8.11 所示。

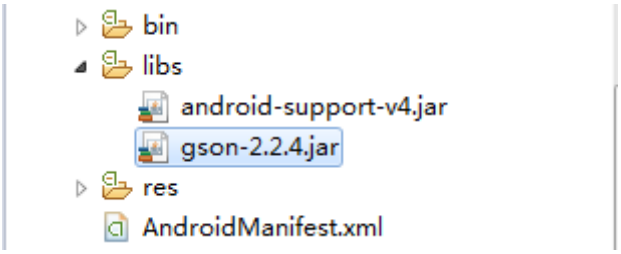

图 8.11

那么 GSON 库究竟是神奇在哪里呢？其实它主要就是可以将一段 JSON 格式的字符串自动映射成一个对象，从而不需要我们再手动去编写代码进行解析了。

比如说一段 JSON 格式的数据如下所示：

```
{"name":"Tom","age":20}
```

那我们就可以定义一个 Person 类，并加入 name 和 age 这两个字段，然后只需简单地调用如下代码就可以将 JSON 数据自动解析成一个 Person 对象了：

```
Gson gson = new Gson();
Person person = gson.fromJson(jsonData, Person.class);
```

如果需要解析的是一段 JSON 数组会稍微麻烦一点，我们需要借助 TypeToken 将期望解析成的数据类型传入到 fromJson()方法中，如下所示：

```
List<Person> people = gson.fromJson(jsonData, new TypeToken<List<Person>>()
{}.getType());
```

好了，基本的用法就是这样，下面就让我们来真正地尝试一下吧。首先新增一个 App 类，并加入 id、name 和 version 这三个字段，如下所示：

```
public class App {
```

```
    private String id;

    private String name;

    private String version;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

}
```

然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    private void sendRequestWithHttpClient() {
        new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                try {
                    HttpClient httpClient = new DefaultHttpClient();
                    // 指定访问的服务器地址是电脑本机
                    HttpGet httpGet = new HttpGet("http://8.0.2.2/
get_data.json");
                    HttpResponse httpResponse = httpClient.execute(httpGet);
                    if (httpResponse.getStatusLine().getStatusCode() == 200) {
                        // 请求和响应都成功了
                        HttpEntity entity = httpResponse.getEntity();
                        String response = EntityUtils.toString(entity,
"utf-8");
                        parseJSONWithGSON(response);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
    ......
    private void parseJSONWithGSON(String jsonData) {
        Gson gson = new Gson();
        List<App> appList = gson.fromJson(jsonData, new
TypeToken<List<App>>() {}.getType());
        for (App app : appList) {
            Log.d("MainActivity", "id is " + app.getId());
            Log.d("MainActivity", "name is " + app.getName());
            Log.d("MainActivity", "version is " + app.getVersion());
        }
    }
}
```

现在重新运行程序，点击 Send Request 按钮后观察 LogCat 中的打印日志，你会看到和图 8.9 中一样的结果。

好了，这样我们就算是把 XML 和 JSON 这两种数据格式最常用的几种解析方法都学习完了，在网络数据的解析方面，你已经成功毕业了。

8.5 网络编程的最佳实践

目前你已经掌握了 HttpURLConnection 和 HttpClient 的用法，知道了如何发起 HTTP 请求，以及解析服务器返回的数据，但也许你还没有发现，之前我们的写法其实是很有问题的。因为一个应用程序很可能会在许多地方都使用到网络功能，而发送 HTTP 请求的代码基本都是相同的，如果我们每次都去编写一遍发送 HTTP 请求的代码，这显然是非常差劲的做法。

没错，通常情况下我们都应该将这些通用的网络操作提取到一个公共的类里，并提供一个静态方法，当想要发起网络请求的时候只需简单地调用一下这个方法即可。比如使用如下的写法：

```
public class HttpUtil {

    public static String sendHttpRequest(String address) {
        HttpURLConnection connection = null;
        try {
            URL url = new URL(address);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.setConnectTimeout(8000);
            connection.setReadTimeout(8000);
            connection.setDoInput(true);
            connection.setDoOutput(true);
            InputStream in = connection.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
            StringBuilder response = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
            return response.toString();
        } catch (Exception e) {
            e.printStackTrace();
            return e.getMessage();
        } finally {
            if (connection != null) {
                connection.disconnect();
            }
```

```
            }
        }

    }
```

以后每当需要发起一条 HTTP 请求的时候就可以这样写：

```
String address = "http://www.baidu.com";
String response = HttpUtil.sendHttpRequest(address);
```

在获取到服务器响应的数据后我们就可以对它进行解析和处理了。但是需要注意，网络请求通常都是属于耗时操作，而 sendHttpRequest()方法的内部并没有开启线程，这样就有可能导致在调用 sendHttpRequest()方法的时候使得主线程被阻塞住。

你可能会说，很简单嘛，在 sendHttpRequest()方法内部开启一个线程不就解决这个问题了吗？其实不是像你想象中的那么容易，因为如果我们在 sendHttpRequest()方法中开启了一个线程来发起 HTTP 请求，那么服务器响应的数据是无法进行返回的，所有的耗时逻辑都是在子线程里进行的，sendHttpRequest()方法会在服务器还来得及响应的时候就执行结束了，当然也就无法返回响应的数据了。

那么遇到这种情况应该怎么办呢？其实解决方法并不难，只需要使用 Java 的回调机制就可以了，下面就让我们来学习一下回调机制到底是如何使用的。

首先需要定义一个接口，比如将它命名成 HttpCallbackListener，代码如下所示：

```
public interface HttpCallbackListener {

    void onFinish(String response);

    void onError(Exception e);

}
```

可以看到，我们在接口中定义了两个方法，onFinish()方法表示当服务器成功响应我们请求的时候调用，onError()表示当进行网络操作出现错误的时候调用。这两个方法都带有参数，onFinish()方法中的参数代表着服务器返回的数据，而 onError()方法中的参数记录着错误的详细信息。

接着修改 HttpUtil 中的代码，如下所示：

```
public class HttpUtil {

    public static void sendHttpRequest(final String address, final
HttpCallbackListener listener) {
        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                HttpURLConnection connection = null;
                try {
                    URL url = new URL(address);
                    connection = (HttpURLConnection) url.openConnection();
                    connection.setRequestMethod("GET");
                    connection.setConnectTimeout(8000);
                    connection.setReadTimeout(8000);
                    connection.setDoInput(true);
                    connection.setDoOutput(true);
                    InputStream in = connection.getInputStream();
                    BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
                    StringBuilder response = new StringBuilder();
                    String line;
                    while ((line = reader.readLine()) != null) {
                        response.append(line);
                    }
                    if (listener != null) {
                        // 回调onFinish()方法
                        listener.onFinish(response.toString());
                    }
                } catch (Exception e) {
                    if (listener != null) {
                        // 回调onError()方法
                        listener.onError(e);
                    }
                } finally {
                    if (connection != null) {
                        connection.disconnect();
                    }
                }
            }
        }).start();
    }

}
```

我们首先给 sendHttpRequest()方法添加了一个 HttpCallbackListener 参数，并在方法的内

部开启了一个子线程，然后在子线程里去执行具体的网络操作。注意子线程中是无法通过return 语句来返回数据的，因此这里我们将服务器响应的数据传入了 HttpCallbackListener 的onFinish()方法中，如果出现了异常就将异常原因传入到 onError()方法中。

现在 sendHttpRequest()方法接收两个参数了，因此我们在调用它的时候还需要将HttpCallbackListener 的实例传入，如下所示：

```
HttpUtil.sendHttpRequest(address, new HttpCallbackListener() {
    @Override
    public void onFinish(String response) {
        // 在这里根据返回内容执行具体的逻辑
    }

    @Override
    public void onError(Exception e) {
        // 在这里对异常情况进行处理
    }
});
```

这样的话，当服务器成功响应的时候我们就可以在 onFinish()方法里对响应数据进行处理了，类似地，如果出现了异常，就可以在 onError()方法里对异常情况进行处理。如此一来，我们就巧妙地利用回调机制将响应数据成功返回给调用方了。

另外需要注意的是，onFinish()方法和 onError()方法最终还是在子线程中运行的，因此我们不可以在这里执行任何的 UI 操作，如果需要根据返回的结果来更新 UI，则仍然要使用上一章中我们学习的异步消息处理机制。

8.6 小结与点评

本章中我们主要学习了在 Android 中使用 HTTP 协议来进行网络交互的知识，虽然Android 中支持的网络通信协议有很多种，但 HTTP 协议无疑是最常用的一种。通常我们有两种方式来发送 HTTP 请求，分别是 HttpURLConnection 和 HttpClient，相信这两种方式你都已经很好地掌握了吧？

接着我们又学习了 XML 和 JSON 格式数据的解析方式，因为服务器响应给我们的数据一般都是属于这两种格式的。无论是 XML 还是 JSON，它们各自又拥有多种的解析方式，这里我们只是学习了最常用的几种，如果以后你的工作中还需要用到其他的解析方式，可以自行去学习。

本章的最后同样是最佳实践环节，在这次的最佳实践中，我们主要学习了如何利用 Java的回调机制来将服务器响应的数据进行返回。其实除此之外，还有很多地方都可以使用到 Java

的回调机制，希望你能举一反三，以后在其他地方需要用到回调机制时都能够灵活地使用。

好了，关于 Android 网络编程部分的内容就学习这么多，下一章中我们该去学习一下 Android 特色开发的相关内容了。

第 9 章　探究 service 服务

记得在几年前，iPhone 属于少数人才拥有的稀有物品，Android 甚至还没面世，那个时候全球的手机市场是由诺基亚统治着的。当时我觉得诺基亚的 Symbian 操作系统做得特别出色，因为比起一般的手机，它可以支持后台功能。那个时候能够一边打着电话、听着音乐，一边在后台挂着 QQ 是件非常酷的事情。所以我也曾经单纯地认为，支持后台的手机就是智能手机。

而如今，Symbian 已经风光不再，Android 和 iOS 占据了大部分的智能市场份额，Windows Phone 也占据了一部分，目前已是三分天下的局面。在这三大智能手机操作系统中，iOS 是不支持后台的，当应用程序不在前台运行时就会进入到挂起状态。Android 则是沿用了 Symbian 的老习惯，加入了后台功能，这使得应用程序即使在关闭的情况下仍然可以在后台继续运行。而 Windows Phone 则是经历了一个由不支持到支持后台的过程，目前 Windows Phone 8 系统也是具备后台功能的。这里我们不会花时间去辩论到底谁的方案更好，既然 Android 提供了这个功能，而且是一个非常重要的组件，那我们自然要去学习一下它的用法了。

9.1　服务是什么

服务（Service）是 Android 中实现程序后台运行的解决方案，它非常适合用于去执行那些不需要和用户交互而且还要求长期运行的任务。服务的运行不依赖于任何用户界面，即使当程序被切换到后台，或者用户打开了另外一个应用程序，服务仍然能够保持正常运行。

不过需要注意的是，服务并不是运行在一个独立的进程当中的，而是依赖于创建服务时所在的应用程序进程。当某个应用程序进程被杀掉时，所有依赖于该进程的服务也会停止运行。

另外，也不要被服务的后台概念所迷惑，实际上服务并不会自动开启线程，所有的代码都是默认运行在主线程当中的。也就是说，我们需要在服务的内部手动创建子线程，并在这里执行具体的任务，否则就有可能出现主线程被阻塞住的情况。那么本章的第一堂课，我们就先来学习一下关于 Android 多线程编程的知识。

9.2　Android 多线程编程

熟悉 Java 的你，对多线程编程一定不会陌生吧。当我们需要执行一些耗时操作，比如

说发起一条网络请求时，考虑到网速等其他原因，服务器未必会立刻响应我们的请求，如果不将这类操作放在子线程里去运行，就会导致主线程被阻塞住，从而影响用户对软件的正常使用。那么就让我们从线程的基本用法开始学习吧。

9.2.1 线程的基本用法

Android 多线程编程其实并不比 Java 多线程编程特珠，基本都是使用相同的语法。比如说，定义一个线程只需要新建一个类继承自 Thread，然后重写父类的 run()方法，并在里面编写耗时逻辑即可，如下所示：

```
class MyThread extends Thread {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}
```

那么该如何启动这个线程呢？其实也很简单，只需要 new 出 MyThread 的实例，然后调用它的 start()方法，这样 run()方法中的代码就会在子线程当中运行了，如下所示：

```
new MyThread().start();
```

当然，使用继承的方式耦合性有点高，更多的时候我们都会选择使用实现 Runnable 接口的方式来定义一个线程，如下所示：

```
class MyThread implements Runnable {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}
```

如果使用了这种写法，启动线程的方法也需要进行相应的改变，如下所示：

```
MyThread myThread = new MyThread();
new Thread(myThread).start();
```

可以看到，Thread 的构造函数接收一个 Runnable 参数，而我们 new 出的 MyThread 正是一个实现了 Runnable 接口的对象，所以可以直接将它传入到 Thread 的构造函数里。接着调

用 Thread 的 start()方法，run()方法中的代码就会在子线程当中运行了。

当然，如果你不想专门再定义一个类去实现 Runnable 接口，也可以使用匿名类的方式，这种写法更为常见，如下所示：

```
new Thread(new Runnable() {

    @Override
    public void run() {
        // 处理具体的逻辑
    }

}).start();
```

以上几种线程的使用方式相信你都不会感到陌生，因为在 Java 中创建和启动线程也是使用同样的方式。了解了线程的基本用法后，下面我们来看一下 Android 多线程编程与 Java 多线程编程不同的地方。

9.2.2 在子线程中更新 UI

和许多其他的 GUI 库一样，Android 的 UI 也是线程不安全的。也就是说，如果想要更新应用程序里的 UI 元素，则必须在主线程中进行，否则就会出现异常。

眼见为实，让我们通过一个具体的例子来验证一下吧。新建一个 AndroidThreadTest 项目，然后修改 activity_main.xml 中的代码，如下所示：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <Button
        android:id="@+id/change_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Change Text" />

    <TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Hello world"
```

```
        android:textSize="20sp" />

</RelativeLayout>
```

布局文件中定义了两个控件，TextView 用于在屏幕的正中央显示一个 Hello world 字符串，Button 用于改变 TextView 中显示的内容，我们希望在点击 Button 后可以把 TextView 中显示的字符串改成 Nice to meet you。

接下来修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private TextView text;

    private Button changeText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = (TextView) findViewById(R.id.text);
        changeText = (Button) findViewById(R.id.change_text);
        changeText.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.change_text:
            new Thread(new Runnable() {
                @Override
                public void run() {
                    text.setText("Nice to meet you");
                }
            }).start();
            break;
        default:
            break;
        }
    }

}
```

可以看到，我们在 Change Text 按钮的点击事件里面开启了一个子线程，然后在子线程中调用 TextView 的 setText()方法将显示的字符串改成 Nice to meet you。代码的逻辑非常简单，只不过我们是在子线程中更新 UI 的。现在运行一下程序，并点击 Change Text 按钮，你会发现程序果然崩溃了，如图 9.1 所示。

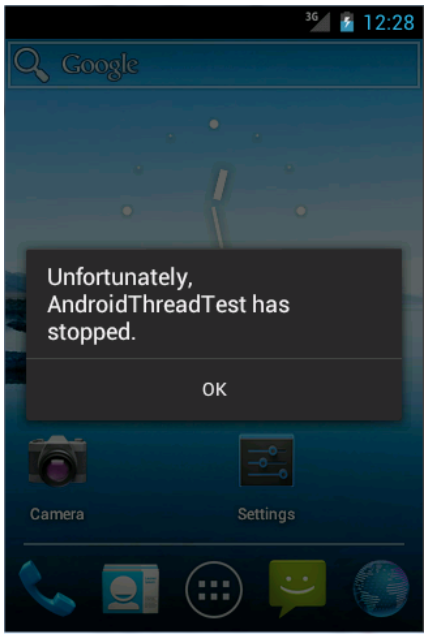

图　9.1

然后观察 LogCat 中的错误日志，可以看出是由于在子线程中更新 UI 所导致的，如图 9.2 所示。

```
android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that
created a view hierarchy can touch its views.
```

图　9.2

由此证实了 Android 确实是不允许在子线程中进行 UI 操作的。但是有些时候，我们必须在子线程里去执行一些耗时任务，然后根据任务的执行结果来更新相应的 UI 控件，这该如何是好呢？

对于这种情况，Android 提供了一套异步消息处理机制，完美地解决了在子线程中进行 UI 操作的问题。本小节中我们先来学习一下异步消息处理的使用方法，下一小节中再去分析它的原理。

修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    public static final int UPDATE_TEXT = 1;

    private TextView text;

    private Button changeText;

    private Handler handler = new Handler() {

        public void handleMessage(Message msg) {
            switch (msg.what) {
            case UPDATE_TEXT:
                // 在这里可以进行UI操作
                text.setText("Nice to meet you");
                break;
            default:
                break;
            }
        }

    };
    ……
    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.change_text:
            new Thread(new Runnable() {
                @Override
                public void run() {
                    Message message = new Message();
                    message.what = UPDATE_TEXT;
                    handler.sendMessage(message); // 将Message对象发送出去
                }
            }).start();
            break;
        default:
            break;
```

```
            }
        }

    }
```

这里我们先是定义了一个整型常量 UPDATE_TEXT，用于表示更新 TextView 这个动作。然后新增一个 Handler 对象，并重写父类的 handleMessage 方法，在这里对具体的 Message 进行处理。如果发现 Message 的 what 字段的值等于 UPDATE_TEXT，就将 TextView 显示的内容改成 Nice to meet you。

下面再来看一下 Change Text 按钮的点击事件中的代码。可以看到，这次我们并没有在子线程里直接进行 UI 操作，而是创建了一个 Message（android.os.Message）对象，并将它的 what 字段的值指定为 UPDATE_TEXT，然后调用 Handler 的 sendMessage()方法将这条 Message 发送出去。很快，Handler 就会收到这条 Message，并在 handleMessage()方法中对它进行处理。注意此时 handleMessage()方法中的代码就是在主线程当中运行的了，所以我们可以放心地在这里进行 UI 操作。接下来对 Message 携带的 what 字段的值进行判断，如果等于 UPDATE_TEXT，就将 TextView 显示的内容改成 Nice to meet you。

现在重新运行程序，可以看到屏幕的正中央显示着 Hello world。然后点击一下 Change Text 按钮，显示的内容着就被替换成 Nice to meet you，如图 9.3 所示。

图 9.3

这样你就已经掌握了 Android 异步消息处理的基本用法，使用这种机制就可以出色地解决掉在子线程中更新 UI 的问题。不过恐怕你对它的工作原理还不是很清楚，下面我们就来分析一下 Android 异步消息处理机制到底是如何工作的。

9.2.3 解析异步消息处理机制

Android 中的异步消息处理主要由四个部分组成，Message、Handler、MessageQueue 和 Looper。其中 Message 和 Handler 在上一小节中我们已经接触过了，而 MessageQueue 和 Looper 对于你来说还是全新的概念，下面我就对这四个部分进行一下简要的介绍。

39. Message

Message 是在线程之间传递的消息，它可以在内部携带少量的信息，用于在不同线程之间交换数据。上一小节中我们使用到了 Message 的 what 字段，除此之外还可以使用 arg1 和 arg2 字段来携带一些整型数据，使用 obj 字段携带一个 Object 对象。

40. Handler

Handler 顾名思义也就是处理者的意思，它主要是用于发送和处理消息的。发送消息一般是使用 Handler 的 sendMessage()方法，而发出的消息经过一系列地辗转处理后，最终会传递到 Handler 的 handleMessage()方法中。

41. MessageQueue

MessageQueue 是消息队列的意思，它主要用于存放所有通过 Handler 发送的消息。这部分消息会一直存在于消息队列中，等待被处理。每个线程中只会有一个 MessageQueue 对象。

42. Looper

Looper 是每个线程中的 MessageQueue 的管家，调用 Looper 的 loop()方法后，就会进入到一个无限循环当中，然后每当发现 MessageQueue 中存在一条消息，就会将它取出，并传递到 Handler 的 handleMessage()方法中。每个线程中也只会有一个 Looper 对象。

了解了 Message、Handler、MessageQueue 以及 Looper 的基本概念后，我们再来对异步消息处理的整个流程梳理一遍。首先需要在主线程当中创建一个 Handler 对象，并重写 handleMessage()方法。然后当子线程中需要进行 UI 操作时，就创建一个 Message 对象，并通过 Handler 将这条消息发送出去。之后这条消息会被添加到 MessageQueue 的队列中等待被处理，而 Looper 则会一直尝试从 MessageQueue 中取出待处理消息，最后分发回 Handler 的 handleMessage()方法中。由于 Handler 是在主线程中创建的，所以此时 handleMessage()方法中的代码也会在主线程中运行，于是我们在这里就可以安心地进行 UI 操作了。整个异步消息处理机制的流程示意图如图 9.4 所示。

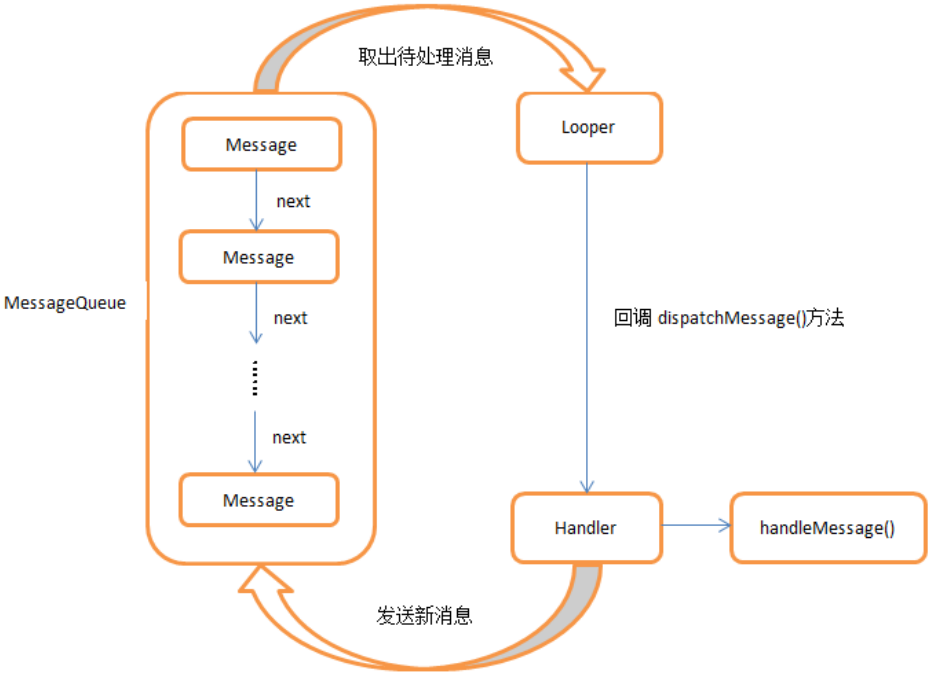

图 9.4

一条 Message 经过这样一个流程的辗转调用后，也就从子线程进入到了主线程，从不能更新 UI 变成了可以更新 UI，整个异步消息处理的核心思想也就是如此。

9.3 服务的基本用法

了解了 Android 多线程编程的技术之后，下面就让我们进入到本章的正题，开始对服务的相关内容进行学习。作为 Android 四大组件之一，服务也少不了有很多非常重要的知识点，那我们自然要从最基本的用法开始学习了。

9.3.1 定义一个服务

首先看一下如何在项目中定义一个服务。新建一个 ServiceTest 项目，然后在这个项目中新增一个名为 MyService 的类，并让它继承自 Service，完成后的代码如下所示：

```
public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
```

```
        return null;
    }

}
```

目前 MyService 中可以算是空空如也，但有一个 onBind()方法特别醒目。这个方法是 Service 中唯一的一个抽象方法，所以必须要在子类里实现。我们会在后面的小节中使用到 onBind()方法，目前可以暂时将它忽略掉。

既然是定义一个服务，自然应该在服务中去处理一些事情了，那处理事情的逻辑应该写在哪里呢？这时就可以重写 Service 中的另外一些方法了，如下所示：

```
public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
    }

}
```

可以看到，这里我们又重写了 onCreate()、onStartCommand()和 onDestroy()这三个方法，它们是每个服务中最常用到的三个方法了。其中 onCreate()方法会在服务创建的时候调用，onStartCommand()方法会在每次服务启动的时候调用，onDestroy()方法会在服务销毁的时候调用。

通常情况下，如果我们希望服务一旦启动就立刻去执行某个动作，就可以将逻辑写在

onStartCommand()方法里。而当服务销毁时，我们又应该在 onDestroy()方法中去回收那些不再使用的资源。

另外需要注意，每一个服务都需要在 AndroidManifest.xml 文件中进行注册才能生效，不知道你有没有发现，这是 Android 四大组件共有的特点。于是我们还应该修改 AndroidManifest.xml 文件，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <service android:name=".MyService" >
        </service>
    </application>
</manifest>
```

这样的话，就已经将一个服务完全定义好了。

9.3.2 启动和停止服务

定义好了服务之后，接下来就应该考虑如何去启动以及停止这个服务。启动和停止的方法当然你也不会陌生，主要是借助 Intent 来实现的，下面就让我们在 ServiceTest 项目中尝试去启动以及停止 MyService 这个服务。

首先修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button
        android:id="@+id/start_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start Service" />
```

```
    <Button
        android:id="@+id/stop_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Stop Service" />

</LinearLayout>
```

这里我们在布局文件中加入了两个按钮，分别是用于启动服务和停止服务的。
然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button startService;

    private Button stopService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        startService = (Button) findViewById(R.id.start_service);
        stopService = (Button) findViewById(R.id.stop_service);
        startService.setOnClickListener(this);
        stopService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        case R.id.start_service:
            Intent startIntent = new Intent(this, MyService.class);
            startService(startIntent); // 启动服务
            break;
        case R.id.stop_service:
            Intent stopIntent = new Intent(this, MyService.class);
            stopService(stopIntent); // 停止服务
            break;
        default:
            break;
```

```
        }
    }

}
```

可以看到，这里在 onCreate()方法中分别获取到了 Start Service 按钮和 Stop Service 按钮的实例，并给它们注册了点击事件。然后在 Start Service 按钮的点击事件里，我们构建出了一个 Intent 对象，并调用 startService()方法来启动 MyService 这个服务。在 Stop Serivce 按钮的点击事件里，我们同样构建出了一个 Intent 对象，并调用 stopService()方法来停止 MyService 这个服务。startService()和 stopService()方法都是定义在 Context 类中的，所以我们在活动里可以直接调用这两个方法。注意，这里完全是由活动来决定服务何时停止的，如果没有点击 Stop Service 按钮，服务就会一直处于运行状态。那服务有没有什么办法让自己停止下来呢？当然可以，只需要在 MyService 的任何一个位置调用 stopSelf()方法就能让这个服务停止下来了。

那么接下来又有一个问题需要思考了，我们如何才能证实服务已经成功启动或者停止了呢？最简单的方法就是在 MyService 的几个方法中加入打印日志，如下所示：

```
public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public void onCreate() {
        super.onCreate();
        Log.d("MyService", "onCreate executed");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d("MyService", "onStartCommand executed");
        return super.onStartCommand(intent, flags, startId);
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
```

```
        Log.d("MyService", "onDestroy executed");
    }

}
```

现在可以运行一下程序来进行测试了，程序的主界面如图 9.5 所示。

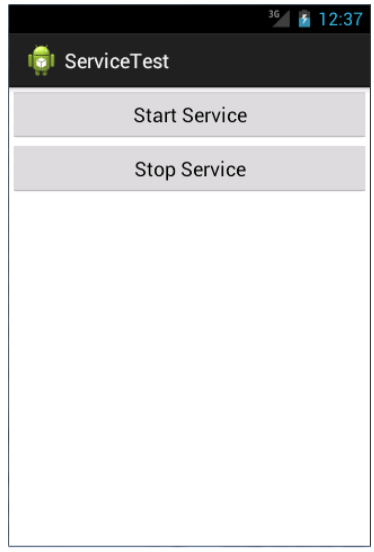

图 9.5

点击一下 Start Service 按钮，观察 LogCat 中的打印日志如图 9.6 所示。

Tag	Text
MyService	onCreate executed
MyService	onStartCommand executed

图 9.6

MyService 中的 onCreate()和 onStartCommand()方法都执行了，说明这个服务确实已经启动成功了，并且你还可以在正在运行的服务列表中找到它，如图 9.7 所示。

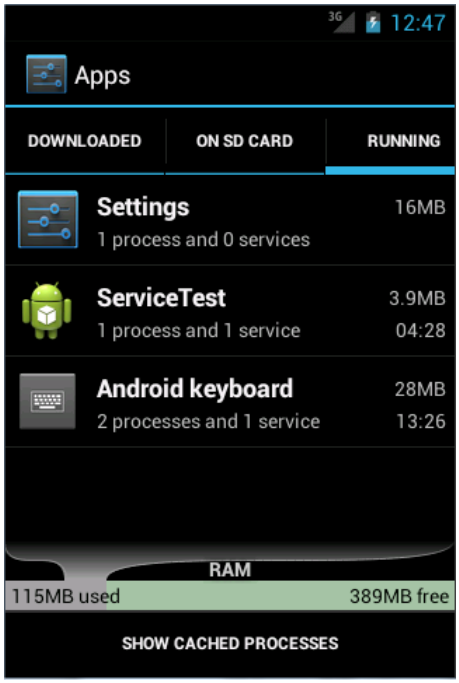

图 9.7

然后再点击一下 Stop Service 按钮，观察 LogCat 中的打印日志如图 9.8 所示。

Tag	Text
MyService	onDestroy executed

图 9.8

由此证明，MyService 确实已经成功停止下来了。

话说回来，虽然我们已经学会了启动服务以及停止服务的方法，不知道你心里现在有没有一个疑惑，那就是 onCreate()方法和 onStartCommand()到底有什么区别呢？因为刚刚点击 Start Service 按钮后两个方法都执行了。

其实 onCreate()方法是在服务第一次创建的时候调用的，而 onStartCommand()方法则在每次启动服务的时候都会调用，由于刚才我们是第一次点击 Start Service 按钮，服务此时还未创建过，所以两个方法都会执行，之后如果你再连续多点击几次 Start Service 按钮，你就会发现只有 onStartCommand()方法可以得到执行了。

9.3.3 活动和服务进行通信

上一小节中我们学习了启动和停止服务的方法，不知道你有没有发现，虽然服务是在活动里启动的，但在启动了服务之后，活动与服务基本就没有什么关系了。确实如此，我们在活动里调用了 startService()方法来启动 MyService 这个服务，然后 MyService 的 onCreate()和 onStartCommand()方法就会得到执行。之后服务会一直处于运行状态，但具体运行的是什么逻辑，活动就控制不了了。这就类似于活动通知了服务一下：“你可以启动了!”然后服务就去忙自己的事情了，但活动并不知道服务到底去做了什么事情，以及完成的如何。

那么有没有什么办法能让活动和服务的关系更紧密一些呢？例如在活动中指挥服务去干什么，服务就去干什么。当然可以，这就需要借助我们刚刚忽略的 onBind()方法了。

比如说目前我们希望在 MyService 里提供一个下载功能，然后在活动中可以决定何时开始下载，以及随时查看下载进度。实现这个功能的思路是创建一个专门的 Binder 对象来对下载功能进行管理，修改 MyService 中的代码，如下所示：

```
public class MyService extends Service {

    private DownloadBinder mBinder = new DownloadBinder();

    class DownloadBinder extends Binder {

        public void startDownload() {
            Log.d("MyService", "startDownload executed");
        }

        public int getProgress() {
            Log.d("MyService", "getProgress executed");
            return 0;
        }

    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }

    ……

}
```

可以看到，这里我们新建了一个 DownloadBinder 类，并让它继承自 Binder，然后在它的内部提供了开始下载以及查看下载进度的方法。当然这只是两个模拟方法，并没有实现真正的功能，我们在这两个方法中分别打印了一行日志。

接着，在 MyService 中创建了 DownloadBinder 的实例，然后在 onBind()方法里返回了这个实例，这样 MyService 中的工作就全部完成了。

下面就要看一看，在活动中如何去调用服务里的这些方法了。首先需要在布局文件里新增两个按钮，修改 activity_main.xml 中的代码，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    ……
    <Button
        android:id="@+id/bind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Bind Service" />

    <Button
        android:id="@+id/unbind_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Unbind Service" />
</LinearLayout>
```

这两个按钮分别是用于绑定服务和取消绑定服务的，那到底谁需要去和服务绑定呢？当然就是活动了。当一个活动和服务绑定了之后，就可以调用该服务里的 Binder 提供的方法了。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {

    private Button startService;

    private Button stopService;

    private Button bindService;

    private Button unbindService;
```

```
private MyService.DownloadBinder downloadBinder;

private ServiceConnection connection = new ServiceConnection() {

    @Override
    public void onServiceDisconnected(ComponentName name) {
    }

    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        downloadBinder = (MyService.DownloadBinder) service;
        downloadBinder.startDownload();
        downloadBinder.getProgress();
    }
};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ......
    bindService = (Button) findViewById(R.id.bind_service);
    unbindService = (Button) findViewById(R.id.unbind_service);
    bindService.setOnClickListener(this);
    unbindService.setOnClickListener(this);
}

@Override
public void onClick(View v) {
    switch (v.getId()) {
    ......
    case R.id.bind_service:
        Intent bindIntent = new Intent(this, MyService.class);
        bindService(bindIntent, connection, BIND_AUTO_CREATE); // 绑定服务
        break;
    case R.id.unbind_service:
        unbindService(connection); // 解绑服务
        break;
    default:
        break;
```

```
            }
        }

    }
```

可以看到，这里我们首先创建了一个 ServiceConnection 的匿名类，在里面重写了 onServiceConnected()方法和 onServiceDisconnected()方法，这两个方法分别会在活动与服务成功绑定以及解除绑定的时候调用。在 onServiceConnected()方法中，我们又通过向下转型得到了 DownloadBinder 的实例，有了这个实例，活动和服务之间的关系就变得非常紧密了。现在我们可以在活动中根据具体的场景来调用 DownloadBinder 中的任何 public 方法，即实现了指挥服务干什么，服务就去干什么的功能。这里仍然只是做了个简单的测试，在 onServiceConnected()方法中调用了 DownloadBinder 的 startDownload()和 getProgress()方法。

当然，现在活动和服务其实还没进行绑定呢，这个功能是在 Bind Service 按钮的点击事件里完成的。可以看到，这里我们仍然是构建出了一个 Intent 对象，然后调用 bindService()方法将 MainActivity 和 MyService 进行绑定。bindService()方法接收三个参数，第一个参数就是刚刚构建出的 Intent 对象，第二个参数是前面创建出的 ServiceConnection 的实例，第三个参数则是一个标志位，这里传入 BIND_AUTO_CREATE 表示在活动和服务进行绑定后自动创建服务。这会使得 MyService 中的 onCreate()方法得到执行，但 onStartCommand()方法不会执行。

然后如果我们想解除活动和服务之间的绑定该怎么办呢？调用一下 unbindService()方法就可以了，这也是 Unbind Service 按钮的点击事件里实现的功能。

现在让我们重新运行一下程序吧，界面如图 9.9 所示。

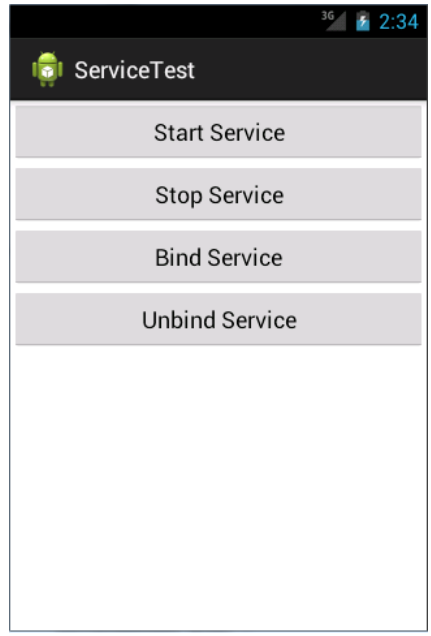

图 9.9

点击一下 Bind Service 按钮，然后观察 LogCat 中的打印日志如图 9.10 所示：

Tag	Text
MyService	onCreate executed
MyService	startDownload executed
MyService	getProgress executed

图 9.10

可以看到，首先是 MyService 的 onCreate()方法得到了执行，然后 startDownload()和 getProgress()方法都得到了执行，说明我们确实已经在活动里成功调用了服务里提供的方法了。

另外需要注意，任何一个服务在整个应用程序范围内都是通用的，即 MyService 不仅可以和 MainActivity 绑定，还可以和任何一个其他的活动进行绑定，而且在绑定完成后它们都可以获取到相同的 DownloadBinder 实例。

9.4 服务的生命周期

之前章节我们学习过了活动以及碎片的生命周期。类似地，服务也有自己的生命周期，前面我们使用到的 onCreate()、onStartCommand()、onBind()和 onDestroy()等方法都是在服务

的生命周期内可能回调的方法。

一旦在项目的任何位置调用了 Context 的 startService()方法，相应的服务就会启动起来，并回调 onStartCommand()方法。如果这个服务之前还没有创建过，onCreate()方法会先于 onStartCommand()方法执行。服务启动了之后会一直保持运行状态，直到 stopService()或 stopSelf()方法被调用。注意虽然每调用一次 startService()方法，onStartCommand()就会执行一次，但实际上每个服务都只会存在一个实例。所以不管你调用了多少次 startService()方法，只需调用一次 stopService()或 stopSelf()方法，服务就会停止下来了。

另外，还可以调用 Context 的 bindService()来获取一个服务的持久连接，这时就会回调服务中的 onBind()方法。类似地，如果这个服务之前还没有创建过，onCreate()方法会先于 onBind()方法执行。之后，调用方可以获取到 onBind()方法里返回的 IBinder 对象的实例，这样就能自由地和服务进行通信了。只要调用方和服务之间的连接没有断开，服务就会一直保持运行状态。

当调用了 startService()方法后，又去调用 stopService()方法，这时服务中的 onDestroy()方法就会执行，表示服务已经销毁了。类似地，当调用了 bindService()方法后，又去调用 unbindService()方法，onDestroy()方法也会执行，这两种情况都很好理解。但是需要注意，我们是完全有可能对一个服务既调用了 startService()方法，又调用了 bindService()方法的，这种情况下该如何才能让服务销毁掉呢？根据 Android 系统的机制，一个服务只要被启动或者被绑定了之后，就会一直处于运行状态，必须要让以上两种条件同时不满足，服务才能被销毁。所以，这种情况下要同时调用 stopService()和 unbindService()方法，onDestroy()方法才会执行。

这样你就已经把服务的生命周期完整地走了一遍。

9.5 服务的更多技巧

以上所学的都是关于服务最基本的一些用法和概念，当然也是最常用的。不过，仅仅满足于此显然是不够的，服务的更多高级使用技巧还在等着我们呢，下面就赶快去看一看吧。

9.5.1 使用前台服务

服务几乎都是在后台运行的，一直以来它都是默默地做着辛苦的工作。但是服务的系统优先级还是比较低的，当系统出现内存不足的情况时，就有可能会回收掉正在后台运行的服务。如果你希望服务可以一直保持运行状态，而不会由于系统内存不足的原因导致被回收，就可以考虑使用前台服务。前台服务和普通服务最大的区别就在于，它会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，非常类似于通知的效果。当然有时候你也可能不仅仅是为了防止服务被回收掉才使用前台服务的，有些项目由

于特殊的需求会要求必须使用前台服务，比如说墨迹天气，它的服务在后台更新天气数据的同时，还会在系统状态栏一直显示当前的天气信息，如图 9.11 所示。

图 9.11

那么我们就来看一下如何才能创建一个前台服务吧，其实并不复杂，修改 MyService 中的代码，如下所示：

```
public class MyService extends Service {
    ……
    @Override
    public void onCreate() {
        super.onCreate();
        Notification notification = new Notification(R.drawable.ic_launcher,
"Notification comes", System. currentTimeMillis());
        Intent notificationIntent = new Intent(this, MainActivity.class);
        PendingIntent pendingIntent = PendingIntent.getActivity(this, 0,
notificationIntent, 0);
        notification.setLatestEventInfo(this, "This is title", "This is
content", pendingIntent);
        startForeground(1, notification);
        Log.d("MyService", "onCreate executed");
```

```
        }
        ......
    }
```

可以看到，这里只是修改了 onCreate()方法中的代码，相信这部分的代码你会非常眼熟。没错！这就是我们在上一章中学习的创建通知的方法。只不过这次在构建出 Notification 对象后并没有使用 NotificationManager 来将通知显示出来，而是调用了 startForeground()方法。这个方法接收两个参数，第一个参数是通知的 id，类似于 notify()方法的第一个参数，第二个参数则是构建出的 Notification 对象。调用 startForeground()方法后就会让 MyService 变成一个前台服务，并在系统状态栏显示出来。

现在重新运行一下程序，并点击 Start Service 或 Bind Service 按钮，MyService 就会以前台服务的模式启动了，并且在系统状态栏会显示一个通知图标，下拉状态栏后可以看到该通知的详细内容，如图 9.12 所示。

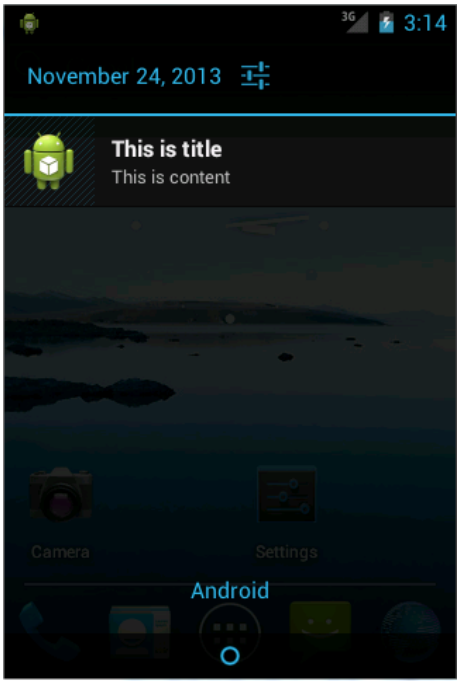

图 9.12

前台服务的用法就这么简单，只要你在上一章中将通知的用法掌握好了，学习本节的知识一定会特别轻松。

9.5.2 使用 IntentService

话说回来，在本章一开始的时候我们就已经知道，服务中的代码都是默认运行在主线程当中的，如果直接在服务里去处理一些耗时的逻辑，就很容易出现 ANR（Application Not Responding）的情况。

所以这个时候就需要用到 Android 多线程编程的技术了，我们应该在服务的每个具体的方法里开启一个子线程，然后在这里去处理那些耗时的逻辑。因此，一个比较标准的服务就可以写成如下形式：

```
public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 处理具体的逻辑
            }
        }).start();
        return super.onStartCommand(intent, flags, startId);
    }

}
```

但是，这种服务一旦启动之后，就会一直处于运行状态，必须调用 stopService()或者 stopSelf()方法才能让服务停止下来。所以，如果想要实现让一个服务在执行完毕后自动停止的功能，就可以这样写：

```
public class MyService extends Service {

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
```

```
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 处理具体的逻辑
                stopSelf();
            }
        }).start();
        return super.onStartCommand(intent, flags, startId);
    }

}
```

虽说这种写法并不复杂，但是总会有一些程序员忘记开启线程，或者忘记调用 stopSelf()方法。为了可以简单地创建一个异步的、会自动停止的服务，Android 专门提供了一个 IntentService 类，这个类就很好地解决了前面所提到的两种尴尬，下面我们就来看一下它的用法。

新建一个 MyIntentService 类继承自 IntentService，代码如下所示：

```
public class MyIntentService extends IntentService {

    public MyIntentService() {
        super("MyIntentService"); // 调用父类的有参构造函数
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        // 打印当前线程的id
        Log.d("MyIntentService", "Thread id is " + Thread.currentThread().
getId());
    }

    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("MyIntentService", "onDestroy executed");
    }

}
```

这里首先是要提供一个无参的构造函数，并且必须在其内部调用父类的有参构造函数。然后要在子类中去实现 onHandleIntent()这个抽象方法，在这个方法中可以去处理一些具体的逻辑，而且不用担心 ANR 的问题，因为这个方法已经是在子线程中运行的了。这里为了证实一下，我们在 onHandleIntent()方法中打印了当前线程的 id。另外根据 IntentService 的特性，这个服务在运行结束后应该是会自动停止的，所以我们又重写了 onDestroy()方法，在这里也打印了一行日志，以证实服务是不是停止掉了。

接下来修改 activity_main.xml 中的代码，加入一个用于启动 MyIntentService 这个服务的按钮，如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    ……
    <Button
        android:id="@+id/start_intent_service"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Start IntentService" />
</LinearLayout>
```

然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
    ……
    private Button startIntentService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ……
        startIntentService = (Button) findViewById(R.id.start_intent_service);
        startIntentService.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
        ……
```

```
            case R.id.start_intent_service:
                // 打印主线程的id
                Log.d("MainActivity", "Thread id is " + Thread.currentThread().
    getId());
                Intent intentService = new Intent(this, MyIntentService.class);
                startService(intentService);
                break;
            default:
                break;
            }
        }
    }
```

可以看到，我们在 Start IntentService 按钮的点击事件里面去启动 MyIntentService 这个服务，并在这里打印了一下主线程的 id，稍后用于和 IntentService 进行比对。你会发现，其实 IntentService 的用法和普通的服务没什么两样。

最后仍然不要忘记，服务都是需要在 AndroidManifest.xml 里注册的，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest"
    android:versionCode="1"
    android:versionName="1.0" >
    ……
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        ……
        <service android:name=".MyIntentService"></service>
    </application>
</manifest>
```

现在重新运行一下程序，界面如图 9.13 所示。

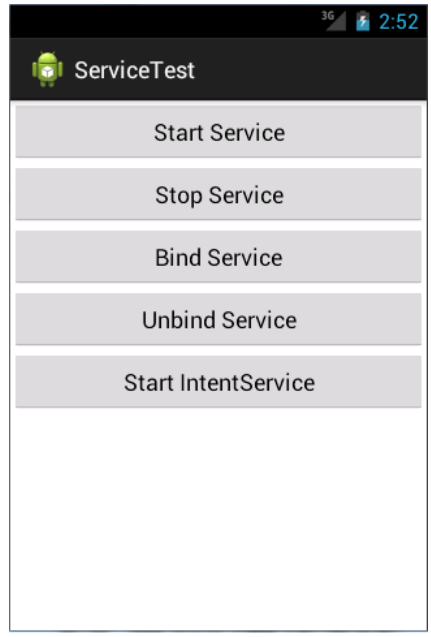

图 9.13

点击 Start IntentService 按钮后，观察 LogCat 中的打印日志，如图 9.14 所示。

Tag	Text
MainActivity	Thread id is 1
MyIntentService	Thread id is 97
MyIntentService	onDestroy executed

图 9.14

可以看到，不仅 MyIntentService 和 MainActivity 所在的线程 id 不一样，而且 onDestroy() 方法也得到了执行，说明 MyIntentService 在运行完毕后确实自动停止了。集开启线程和自动停止于一身，IntentService 还是博得了不少程序员的喜爱。

好了，关于服务的知识点你已经学得够多了，下面就让我们进入到本章的最佳实践环节吧。

9.7 小结与点评

在本章中，我们学习了很多与服务相关的重要知识点，包括 Android 多线程编程、服务的基本用法、服务的生命周期、前台服务和 IntentService 等。这些内容已经覆盖了大部分在

你日常开发中可能用到的服务技术，再加上最佳实践部分学习的后台定时任务的技巧，相信以后不管遇到什么样的服务难题，你都能从容解决吧。

另外，本章同样是有里程碑式的纪念意义的，因为我们已经将 Android 中的四大组件全部学完，并且本书的内容也学习过半了。对于你来说，现在你已经脱离了 Android 初级开发者的身份，并应该具备了独立完成很多功能的能力了。

那么后面我们应该再接再厉，争取进一步地提升自身的能力，所以现在还不是放松的时候。目前我们所学的所有东西都仅仅是在本地上进行的，而实际上几乎市场上的每个应用都会涉及到网络交互的部分，所以下一章中我们就来学习一下 Android 网络编程方面的内容。