

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 文件进行修改，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcastbestpractice">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
        </activity>
        <activity android:name=".LoginActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

这里只需要对一处代码进行修改，就是将主活动设置为 LoginActivity 而不再是 MainActivity，因为你肯定不希望用户在没登录的情况下就能直接进入到程序主界面吧？

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。

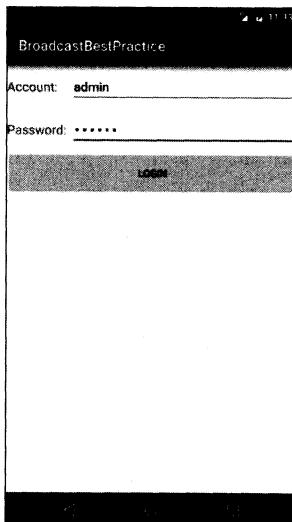


图 5.10 登录界面

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。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。

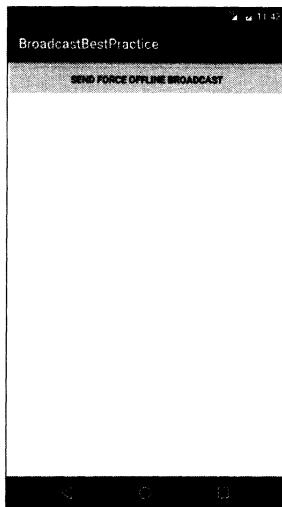


图 5.11 主界面



图 5.12 强制下线提示

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。

结束了本章的最佳实践部分，接下来我们要进入一个特殊的环节。相信你一定也知道，几乎所有出色的项目都不会是由一个人单枪匹马完成的，而是由一个团队共同合作开发完成的。这个时候多人之间代码同步的问题就显得异常重要，因此版本控制工具也就应运而生了。常见的版本控制工具主要有 svn 和 Git，本书中将会对 Git 的使用方法进行全面的讲解，并且讲解的内容是穿插于一些章节当中的。那么今天，我们就先来看一看关于 Git 最基本的用法。

5.6 Git 时间——初识版本控制工具

Git 是一个开源的分布式版本控制工具，它的开发者就是鼎鼎大名的 Linux 操作系统的作者 Linus Torvalds。Git 被开发出来的初衷是为了更好地管理 Linux 内核，而现在却早已被广泛应用于全球各种大中小型的项目中。今天是我们关于 Git 的第一堂课，主要是讲解一下它最基本的方法，那么就从安装 Git 开始吧。

5.6.1 安装 Git

由于 Git 和 Linux 操作系统都是同一个作者，因此不用我说，你也应该猜到 Git 在 Linux 上的

安装是最简单方便的。比如你使用的是 Ubuntu 系统，只需要打开 shell 界面，并输入：

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

按下回车后输入密码，即可完成 Git 的安装。

不过我相信你更有可能使用的还是 Windows 操作系统，因此本小节的重点是教会你如何在 Windows 上安装 Git。不同于 Linux，Windows 上可无法通过一行命令就完成安装了，我们需要先把 Git 的安装包下载下来。访问网址 <https://git-for-windows.github.io/>，可以看到如图 5.13 所示的页面。



图 5.13 git for windows 主页

目前最新的 git for windows 版本是 2.8.1，我就准备使用这一版本了，如果你下载的时候发现又有新的版本，可以尝试一下最新版本的 Git。点击 Download 按钮可以开始下载，下载完成后双击安装包进行安装，之后一直点击“下一步”就可以完成安装了。

5.6.2 创建代码仓库

虽然在 Windows 上安装的 Git 是可以在图形界面上进行操作的，并且 Android Studio 也支持以图形化的形式操作 Git，但是这里我并不建议你这样做，因为 Git 的各种命令才是你应该掌握的核心技能，不管你是在哪个操作系统中，使用命令来操作 Git 肯定都是通用的。而图形化的操作应该是在你能熟练掌握命令用法的前提下，进一步提升你工作效率的手段。

那么我们现在就来尝试一下如何通过命令来使用 Git。如果你使用的是 Linux 系统，就先打开 shell 界面，如果使用的是 Windows 系统，就从开始里找到 Git Bash 并打开。

首先应该配置一下你的身份，这样在提交代码的时候 Git 就可以知道是谁提交的了，命令如下所示：

```
git config --global user.name "Tony"
git config --global user.email "tony@gmail.com"
```

配置完成后你还可以使用同样的命令来查看是否配置成功，只需要将最后的名字和邮箱地址去掉即可，如图 5.14 所示。



图 5.14 查看 git 用户名和邮箱

然后我们就可以开始创建代码仓库了，仓库（Repository）是用于保存版本管理所需信息的地方，所有本地提交的代码都会被提交到代码仓库中，如果有需要还可以再推送到远程仓库中。

这里我们尝试着给 BroadcastBestPractice 项目建立一个代码仓库。先进入到 BroadcastBestPractice 项目的目录下面，如图 5.15 所示。



图 5.15 切换到 BroadcastBestPractice 项目目录下

然后在这个目录下面输入如下命令：

```
git init
```

很简单吧！只需要一行命令就可以完成创建代码仓库的操作，如图 5.16 所示。



图 5.16 创建代码仓库

仓库创建完成后，会在 BroadcastBestPractice 项目的根目录下生成一个隐藏的.git 文件夹，这个文件夹就是用来记录本地所有的 Git 操作的，可以通过 `ls -al` 命令来查看一下，如图 5.17 所示。

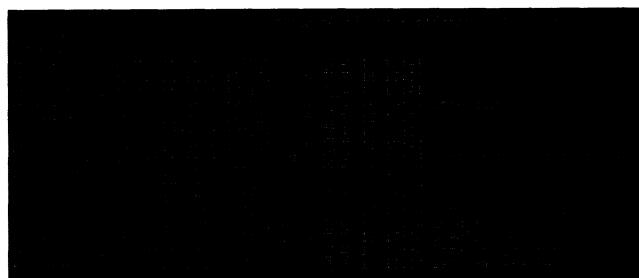


图 5.17 查看.git 文件

如果你想要删除本地仓库，只需要删除这个文件夹就行了。

5.6.3 提交本地代码

代码仓库建立完之后就可以提交代码了，其实提交代码的方法也非常简单，只需要使用 `add` 和 `commit` 命令就可以了。`add` 用于把想要提交的代码先添加进来，而 `commit` 则是真正地去执行提交操作。比如我们想添加 `build.gradle` 文件，就可以输入如下命令：

```
git add build.gradle
```

这是添加单个文件的方法，那如果我们想添加某个目录呢？其实只需要在 `add` 后面加上目录名就可以了。比如将整个 `app` 目录下的所有文件都进行添加，就可以输入如下命令：

```
git add app
```

可是这样一个个地添加感觉还是有些复杂，有没有什么办法可以一次性就把所有的文件都添加好呢？当然可以，只需要在 `add` 的后面加上一个点，就表示添加所有的文件了，命令如下所示：

```
git add .
```

现在 `BroadcastBestPractice` 项目下所有的文件都已经添加好了，我们可以来提交一下了，输入如下命令：

```
git commit -m "First commit."
```

注意，在 `commit` 命令的后面，我们一定要通过 `-m` 参数来加上提交的描述信息，没有描述信息的提交被认为是不合法的。这样所有的代码就已经成功提交了！

好了，关于 Git 的内容，今天我们就学到这里，虽然内容并不多，但是你已经将 Git 最基本的用法都掌握了，不是吗？在本书后面的章节，还会穿插一些 Git 的讲解，到时候你将学会更多关于 Git 的使用技巧，现在就让我们来总结一下吧。

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 四大组件中的内容提供器，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 6 章

数据存储全方案——详解持久化技术

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，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。

6.1 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 3 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 3 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 卡中会更加地安全。

那么下面我就将对这 3 种数据持久化的方式一一进行详细的讲解。

6.2 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，这样可以方便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。

6.2.1 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，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:orientation="vertical"
    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 AppCompatActivity {

    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 `EditText` 中输入一些内容，如图 6.1 所示。

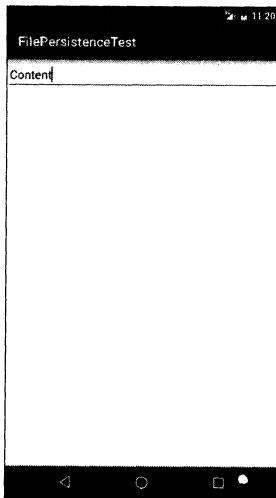


图 6.1 在 `EditText` 中随意输入点内容

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 `Android Device Monitor` 工具来查看一下。点击 `Android Studio` 导航栏中的 `Tools→Android`，会看到如图 6.2 所示的工具列表。

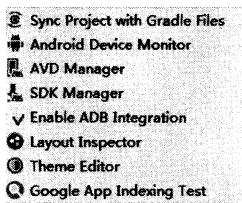


图 6.2 Android 工具列表

点击 Android Device Monitor 就可以打开 Android Device Monitor 工具了，然后进入 File Explorer 标签页，在这里找到 /data/data/com.example.filepersistencetest/files/ 目录，可以看到生成了一个 data 文件，如图 6.3 所示。

Name	Size	Date	Time	Permissions	Info
com.example.broadcastbestpracti		2016-04-16	09:46	drwxr-x--x	
com.example.broadcasttest		2016-04-16	03:40	drwxr-x--x	
com.example.broadcasttest2		2016-04-16	07:51	drwxr-x--x	
com.example.filepersistencetest		2016-04-24	11:18	drwxr-x--x	
cache		2016-04-24	11:18	drwxrwx--x	
code_cache		2016-04-24	11:18	drwxrwx--x	
files		2016-04-24	11:24	drwx-----	
data	7	2016-04-24	11:24	r--w--r--	
instant-run		2016-04-24	11:18	drwx-----	
com.example.fragmentbestpractice		2016-04-10	08:50	drwxr-x--x	
com.example.fragmenttest		2016-04-09	15:20	drwxr-x--x	
com.example.listviewtest		2016-04-01	12:26	drwxr-x--x	
com.example.recycleviewtest		2016-04-02	11:47	drwxr-x--x	
com.example.ulibestpractice		2016-04-04	08:13	drwxr-x--x	
com.example.uicustomviews		2016-03-28	14:14	drwxr-x--x	
com.example.ulayouttest		2016-03-27	12:01	drwxr-x--x	

图 6.3 生成的 data 文件

然后点击图 6.4 中左边的按钮可以将这个文件导出到电脑上。



图 6.4 导入导出按钮

使用记事本打开这个文件，里面的内容如图 6.5 所示。

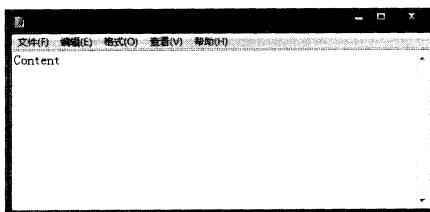


图 6.5 data 文件中的内容

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。

6.2.2 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 AppCompatActivity {

    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`null`，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()`方法中的细节我们在前面已经讲过，这里就不再赘述了。

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先单独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 6.6 所示。

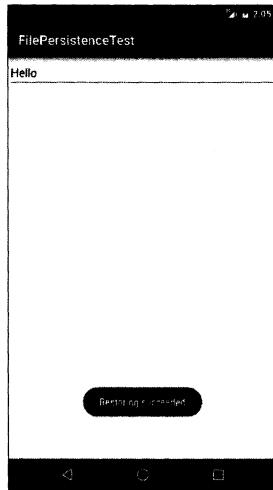


图 6.6 成功还原保存的内容

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。

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。

6.3 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；如果存储的数据是一个字符串，那么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。

6.3.1 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3种方法用于得到 SharedPreferences 对象。

1. 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，目前只有 MODE_PRIVATE 这一种模式可选，它是默认的操作模式，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。其他几种操作模式均已被废弃，MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 这两种模式是在 Android 4.2 版本中被废弃的，MODE_MULTI_PROCESS 模式是在 Android 6.0 版本中被废弃的。

2. 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。

3. 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3步实现。

(1) 调用 SharedPreferences 对象的 edit() 方法来获取一个 SharedPreferences.Editor 对象。

(2) 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。

(3) 调用 apply() 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 AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button saveData = (Button) findViewById(R.id.save_data);
        saveData.setOnClickListener(new View.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.apply();
            }
        });
    }
}
```

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 3 条不同类型的数据，最后调用 `apply()` 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，不过为了证实一下，我们还是要借助 File Explorer 来进行查看。打开 Android Device Monitor，并点击 File Explorer 标签页，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 6.7 所示。

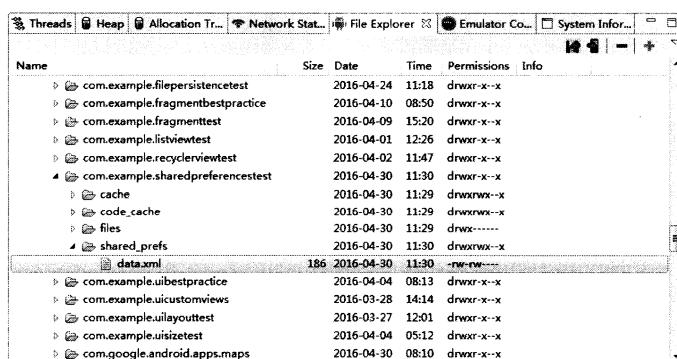


图 6.7 生成的 `data.xml` 文件

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 6.8 所示。

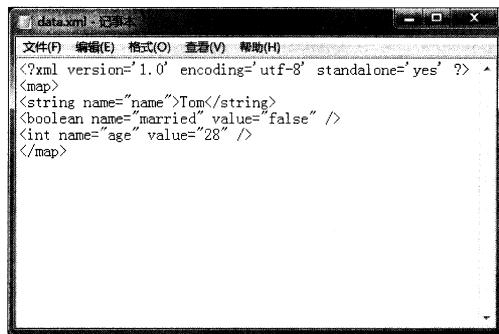


图 6.8 data.xml 文件中的内容

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。

6.3.2 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 文件中读取数据会更加地简单。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 AppCompatActivity {

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

        Button restoreData = (Button) findViewById(R.id.restore_data);
        restoreData.setOnClickListener(new View.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 6.9 所示。

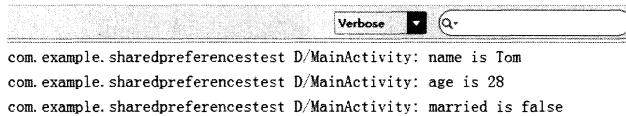


图 6.9 打印 data.xml 中存储的内容

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。

6.3.3 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 Broadcast-BestPractice 项目，来编辑一下登录界面的布局。修改 activity_login.xml 中的代码，如下所示：

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

    ...

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <CheckBox
            android:id="@+id/remember_pass"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="18sp"
            android:text="Remember password" />
    </LinearLayout>

    <Button
        android:id="@+id/login"
        android:layout_width="match_parent"
        android:layout_height="60dp"
        android:text="Login" />
</LinearLayout>
```

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.activity_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 View.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")) {
                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.apply();
                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，则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 6.10 所示。

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 6.11 所示。

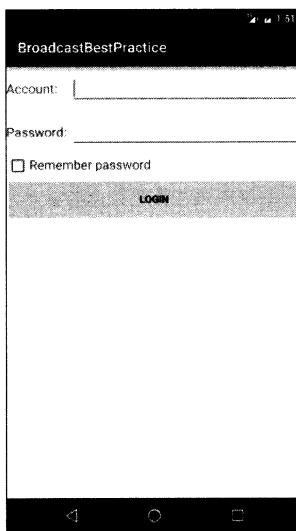


图 6.10 带记住密码复选框的登录界面

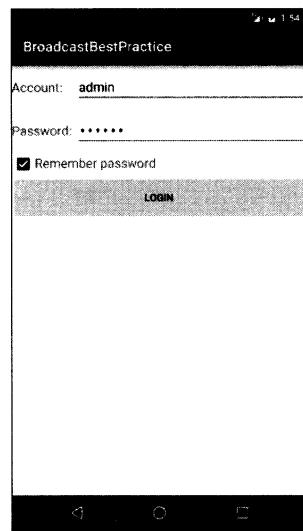


图 6.11 实现记住账号密码功能

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。

6.4 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 KB 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。

6.4.1 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 4 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,
                           SQLiteDatabase.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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    <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 AppCompatActivity {

    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 View.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 6.12 所示。

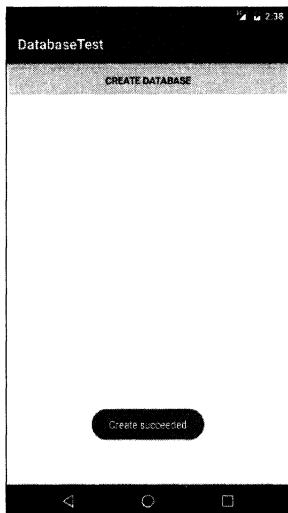


图 6.12 创建数据库成功

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。因此这次我们准备换一种查看方式，使用 `adb shell` 来对数据库和表的创建情况进行检查。

`adb` 是 `Android SDK` 中自带的一个调试工具，使用这个工具可以直接对连接在电脑上的手机或模拟器进行调试操作。它存放在 `sdk` 的 `platform-tools` 目录下，如果想要在命令行中使用这个工具，就需要先把它的路径配置到环境变量里。

如果你使用的是 `Windows` 系统，可以右击计算机→属性→高级系统设置→环境变量，然后在系统变量里找到 `Path` 并点击编辑，将 `platform-tools` 目录配置进去，如图 6.13 所示。

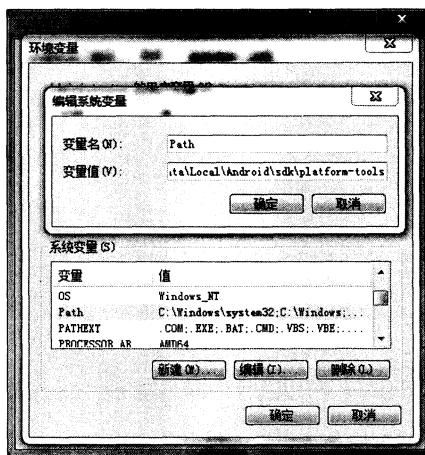


图 6.13 Windows 下配置环境变量

如果你使用的是 Linux 或 Mac 系统，可以在 home 路径下编辑.bashrc 文件，将 platform-tools 目录配置进去即可，如图 6.14 所示。

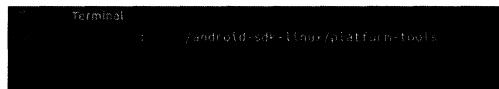


图 6.14 Linux 或 Mac 下配置环境变量

配置好了环境变量之后，就可以使用 adb 工具了。打开命令行界面，输入 adb shell，就会进入到设备的控制台，如图 6.15 所示。



图 6.15 进入设备的控制台

然后使用 cd 命令进入到 /data/data/com.example.databasetest/databases/ 目录下，并使用 ls 命令查看到该目录里的文件，如图 6.16 所示。

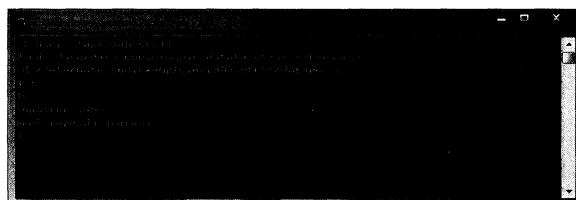


图 6.16 查看数据库文件

这个目录下出现了两个数据库文件，一个正是我们创建的 BookStore.db，而另一个 BookStore.db-journal 则是为了让数据库能够支持事务而产生的临时日志文件，通常情况下这个文件的大小都是 0 字节。

接下来我们就要借助 sqlite 命令来打开数据库了，只需要键入 sqlite3，后面加上数据库名即可，如图 6.17 所示。

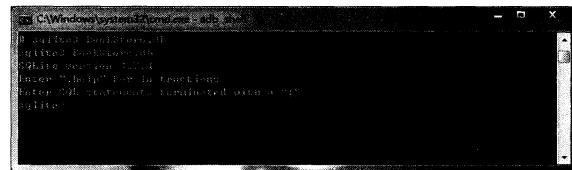


图 6.17 打开 BookStore.db 数据库

这时就已经打开了 BookStore.db 数据库，现在就可以对这个数据库中的表进行管理了。首先来看一下目前数据库中有哪些表，键入.table 命令，如图 6.18 所示。



图 6.18 查看表

可以看到，此时数据库中有两张表，`android_metadata` 表是每个数据库中都会自动生成的，不用管它，而另外一张 `Book` 表就是我们在 `MyDatabaseHelper` 中创建的了。这里还可以通过.schema 命令来查看它们的建表语句，如图 6.19 所示。



图 6.19 查看建表语句

由此证明，`BookStore.db` 数据库和 `Book` 表确实已经创建成功了。之后键入.exit 或.quit 命令可以退出数据库的编辑，再键入 exit 命令就可以退出设备控制台了。

6.4.2 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 表用于记录图书的分类，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,
                           SQLiteDatabase.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()` 方法重新创建。这里先将已经存在的表删除掉，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 AppCompatActivity {
    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 View.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 6.20 所示。



图 6.20 查看新增表

接着键入 `.schema` 命令查看一下建表语句，结果如图 6.21 所示。



图 6.21 查看新增建表语句

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。

6.4.3 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无非有 4 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。

那么下面我们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 3 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    ...
    <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 AppCompatActivity {

    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 View.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 6.22 所示。

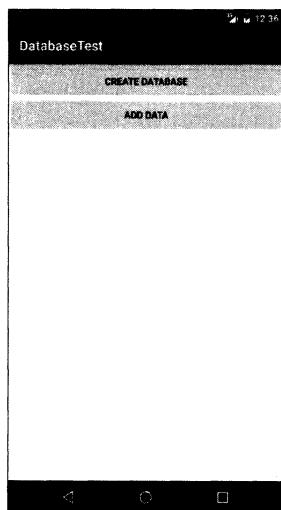


图 6.22 加入添加数据按钮

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 6.23 所示。



图 6.23 查看添加的数据

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。

6.4.4 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。SQLite-Database 中也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 4 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    ...

    <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 AppCompatActivity {

    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 View.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 6.24 所示。

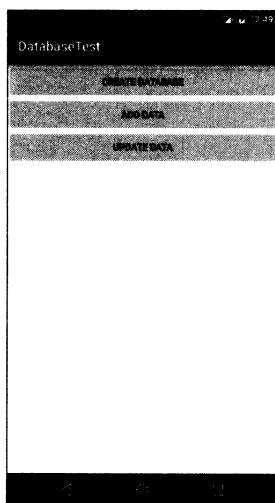


图 6.24 加入更新数据按钮

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 6.25 所示。



图 6.25 查看更新后的数据

可以看到，`The Da Vinci Code` 这本书的价格已经被成功改为了 10.99 了。

6.4.5 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 3 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    ...

    <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 AppCompatActivity {

    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 View.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。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 6.26 所示。

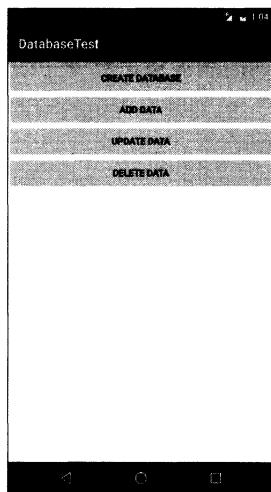


图 6.26 加入删除数据按钮

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 6.27 所示。



图 6.27 查看删除后的数据

6.4.6 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，因为查询数据是 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。它的大部分功能都体现在“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 7 个参数。那我们就先来看一下这 7 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、第四个参数用于约束查询某一行或某几行的数据，不指定则默认查询所有行的数据。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	<code>from table_name</code>	指定查询的表名
columns	<code>select column1, column2</code>	指定查询的列名
selection	<code>where column = value</code>	指定where的约束条件
selectionArgs	-	为where中的占位符提供具体的值
groupBy	<code>group by column</code>	指定需要group by的列
having	<code>having column = value</code>	对group by后的结果进一步约束
orderBy	<code>order by column1, column2</code>	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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >

    ...

    <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 AppCompatActivity {

    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 View.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 6.28 所示。

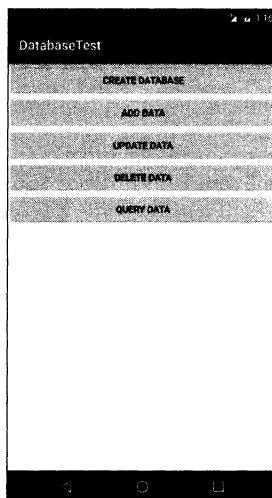


图 6.28 加入查询数据按钮

点击一下 Query data 按钮后，查看 logcat 的打印内容，结果如图 6.29 所示。

```
Verbose
com.example.databasetest D/MainActivity: book name is The Da Vinci Code
com.example.databasetest D/MainActivity: book author is Dan Brown
com.example.databasetest D/MainActivity: book pages is 454
com.example.databasetest D/MainActivity: book price is 10.99
```

图 6.29 打印查询到的数据

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。

6.4.7 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。

6.5 使用 LitePal 操作数据库

上一节中我们学习了使用 SQLiteDatabase 来操作 SQLite 数据库的方法，你觉得好用吗？每个人的回答可能会不一样。但我相信，等学完了本节的内容之后，你将再也不想去碰 SQLiteDatabase 了。到底是什么东西这么神奇？新建一个 LitePalTest 项目，然后开始我们本节的学习之旅吧。

6.5.1 LitePal 简介

如今，Android 的学习环境比起我当年学习的时候已经好太多了。当时国内做 Android 的人并不多，各种学习资料也比较欠缺，一个项目中几乎所有的功能都要完全靠自己从头来实现，开发效率之低下可想而知。

而现在开源的热潮让所有 Android 开发者都大大受益，GitHub 上面有成百上千的优秀 Android 开源项目，很多之前我们要写很久才能实现的功能，使用开源库可能短短几分钟就能实现了。除此之外，公司里的代码非常强调稳定性，而我们自己写出的代码往往越复杂就越容易出问题。相反，开源项目的代码都是经过时间验证的，通常比我们自己的代码要稳定得多。因此，现在有很多公司为了追求开发效率以及项目稳定性，都会选择使用开源库。

本书中我们将会学习多个开源库的使用方法，而现在你将正式开始接触第一个开源库——LitePal。

LitePal 是一款开源的 Android 数据库框架，它采用了对象关系映射（ORM）的模式，并将我们平时开发最常用到的一些数据库功能进行了封装，使得不用编写一行 SQL 语句就可以完成各种建表和增删改查的操作。LitePal 的项目主页上也有详细的使用文档，地址是：<https://github.com/LitePalFramework/LitePal>。

6.5.2 配置 LitePal

那么怎样才能在项目中使用开源库呢？过去的方式比较复杂，通常需要下载开源库的 Jar 包或者源码，然后再集成到我们的项目当中。而现在就简单得多了，大多数的开源项目都会将版本提交到 jcenter 上，我们只需要在 app/build.gradle 文件中声明该开源库的引用就可以了。

因此，要使用 LitePal 的第一步，就是编辑 app/build.gradle 文件，在 dependencies 闭包中添加如下内容：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:23.2.0'
    testCompile 'junit:junit:4.12'
    compile 'org.litepal.android:core:1.3.2'
}
```

添加的这一行声明中，前面部分是固定的，最后的 1.3.2 是版本号的意思，最新的版本号可以到 LitePal 的项目主页上去查看。

这样我们就把 LitePal 成功引入到当前项目中了，接下来需要配置 litpal.xml 文件。右击 app/src/main 目录→New→Directory，创建一个 assets 目录，然后在 assets 目录下再新建一个 litpal.xml 文件，接着编辑 litpal.xml 文件中的内容，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<litepal>
    <dbname value="BookStore" ></dbname>

    <version value="1" ></version>

    <list>
    </list>
</litepal>
```

其中，**<dbname>**标签用于指定数据库名，**<version>**标签用于指定数据库版本号，**<list>**标签用于指定所有的映射模型，我们稍后就会用到。

最后还需要再配置一下 LitePalApplication，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.litepaltest">
    <application
        android:name="org.litepal.LitePalApplication"
        android:allowBackup="true"
```

```
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    ...
</application>
</manifest>
```

这里我们将项目的 `application` 配置为 `org.litepal.LitePalApplication`, 这样才能让 LitePal 的所有功能都可以正常工作。关于 `application` 的作用, 我们之前并没有进行过详细的讲解, 现在你只需要知道必须这么写就行了, 我们将会在第 13 章中学习 `application` 的更多内容。

现在 LitePal 的配置工作已经全部结束了, 下面我们开始正式使用它吧。

6.5.3 创建和升级数据库

我们之前创建数据库是通过自定义一个类继承自 `SQLiteOpenHelper`, 然后在 `onCreate()` 方法中编写建表语句来实现的, 而使用 LitePal 就不用再这么麻烦了。本节中我们会使用 LitePal 来逐一完成上一节中所学的所有功能, 以此来对比它们之间的差距, 那么为了方便测试, 我们先将 `activity_main.xml` 布局文件从 `DatabaseTest` 项目复制到 `LitePalTest` 项目中来。

刚才在介绍的时候已经说过, LitePal 采取的是对象关系映射 (ORM) 的模式, 那么什么是对象关系映射呢? 简单点说, 我们使用的编程语言是面向对象语言, 而使用的数据库则是关系型数据库, 那么将面向对象的语言和面向关系的数据库之间建立一种映射关系, 这就是对象关系映射了。

不过你可千万不要小看对象关系映射模式, 它赋予了我们一个强大的功能, 就是可以用面向对象的思维来操作数据库, 而不用再和 SQL 语句打交道了, 不信的话我们现在就来体验一下。比如在 6.4.1 小节中, 为了创建一张 `Book` 表, 需要先分析表中应该包含哪些列, 然后再编写出一条建表语句, 最后在自定义的 `SQLiteOpenHelper` 中去执行这条建表语句。但是使用 LitePal, 你就可以用面向对象的思维来实现同样的功能了, 定义一个 `Book` 类, 代码如下所示:

```
public class Book {
    private int id;
    private String author;
    private double price;
    private int pages;
    private String name;
    public int getId() {
        return id;
    }
}
```

```

public void setId(int id) {
    this.id = id;
}

public String getAuthor() {
    return author;
}

public void setAuthor(String author) {
    this.author = author;
}

public double getPrice() {
    return price;
}

public void setPrice(double price) {
    this.price = price;
}

public int getPages() {
    return pages;
}

public void setPages(int pages) {
    this.pages = pages;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}

```

这是一个典型的 Java bean，在 Book 类中我们定义了 `id`、`author`、`price`、`pages`、`name` 这几个字段，并生成了相应的 `getter` 和 `setter` 方法。^① 相应你已经能猜到了，Book 类就会对应数据库中的 Book 表，而类中的每一个字段分别对应了表中的每一个列，这就是对象关系映射最直观的体验，现在你能够理解得更加清楚了吧。

接下来我们还需要将 Book 类添加到映射模型列表当中，修改 `litepal.xml` 中的代码，如下所示：

```

<litepal>
    <dbname value="BookStore" ></dbname>

```

^① 生成 `getter` 和 `setter` 方法的快捷方式是，先将类中的字段定义好，然后按下 `Alt + Insert` 键（Mac 系统是 `command + N`），在弹出菜单中选择 `Getter and Setter`，接着使用 `Shift` 键将所有字段都选中，最后点击 `OK`。

```

<version value="1" ></version>

<list>
    <mapping class="com.example.litepaltest.Book"></mapping>
</list>
</litepal>

```

这里使用`<mapping>`标签来声明我们要配置的映射模型类，注意一定要使用完整的类名。不管有多少模型类需要映射，都使用同样的方式配置在`<list>`标签下即可。

没错，这样就已经把所有工作都完成了，现在只要进行任意一次数据库的操作，`BookStore.db`数据库应该就会自动创建出来。那么我们修改 `MainActivity` 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button createDatabase = (Button) findViewById(R.id.create_database);
        createDatabase.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Connector.getDatabase();
            }
        });
    }
}

```

其中，调用 `Connector.getDatabase()`方法就是一次最简单的数据库操作，只要点击一下按钮，数据库就会自动创建完成了。运行一下程序，然后点击 `Create database` 按钮，接着通过 `adb shell` 查看一下数据库创建情况，如图 6.30 所示。



图 6.30 查看数据库文件

非常棒！数据库文件已经创建成功了。接下来我们使用 `sqlite3` 命令打开 `BookStore.db` 文件，然后再使用`.schema`命令来查看建表语句，如图 6.31 所示。

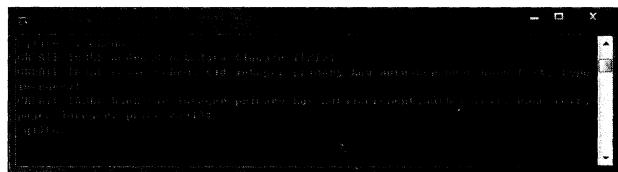


图 6.31 查看建表语句

可以看到，这里有 3 张表的建表语句，其中 `android_metadata` 表仍然不用管，`table_schema` 表是 LitePal 内部使用的，我们也可以直接忽视，`book` 表就是根据我们定义的 `Book` 类以及类中的字段来自动生成的了。

怎么样，是不是很神奇？但不用太吃惊，因为更加神奇的还在后面呢。6.4.2 节中我们体验了使用 `SQLiteOpenHelper` 来升级数据库的方式，虽说功能是实现了，但你有没有发现一个问题，就是升级数据库的时候我们需要先把之前的表 `drop` 掉，然后再重新创建才行。这其实是一个非常严重的问题，因为这样会造成数据丢失，每当升级一次数据库，之前表中的数据就全没了。

当然如果你是非常有经验的程序员，也可以通过复杂的逻辑控制来避免这种情况，但是维护成本很高。而有了 LitePal，这些就都不是问题了，使用 LitePal 来升级数据库非常非常简单，你完全不用思考任何的逻辑，只需要改你想改的任何内容，然后将版本号加 1 就行了。

比如我们想要向 `Book` 表中添加一个 `press`（出版社）列，直接修改 `Book` 类中的代码，添加一个 `press` 字段即可，如下所示：

```
public class Book {
    ...
    private String press;
    ...
    public String getPress() {
        return press;
    }
    public void setPress(String press) {
        this.press = press;
    }
}
```

与此同时，我们还想再添加一张 `Category` 表，那么只需要新建一个 `Category` 类就可以了，代码如下所示：

```
public class Category {
    private int id;
```

```

private String categoryName;

private int categoryCode;

public void setId(int id) {
    this.id = id;
}

public void setCategoryName(String categoryName) {
    this.categoryName = categoryName;
}

public void setCategoryCode(int categoryCode) {
    this.categoryCode = categoryCode;
}

}

```

改完了所有我们想改的东西，只需要记得将版本号加 1 就行了。当然由于这里还添加了一个新的模型类，因此也需要将它添加到映射模型列表中。修改 `litepal.xml` 中的代码，如下所示：

```

<litepal>
    <dbname value="BookStore" ></dbname>

    <version value="2" ></version>

    <list>
        <mapping class="com.example.litepaltest.Book"></mapping>
        <mapping class="com.example.litepaltest.Category"></mapping>
    </list>
</litepal>

```

现在重新运行一下程序，然后点击 `Create database` 按钮，再查看一下最新的建表语句，结果如图 6.32 所示。



图 6.32 升级数据库后的建表语句

可以看到，`book` 表中新增了一个 `press` 列，`category` 表也创建成功了，当然 LitePal 还自动帮我们做了一项非常重要的工作，就是保留之前表中的所有数据，这样就再也不用担心数据丢失的问题了。

6.5.4 使用 LitePal 添加数据

体验了使用 LitePal 来创建和升级数据库，是不是感觉已经有一些小震撼了呢？不过 LitePal 所提供的强大功能还远不止于此，接下来我们就学习一下如何使用它来向数据库的表中添加数据吧。

首先回顾一下之前添加数据的方法，我们需要创建出一个 `ContentValues` 对象，然后将所有要添加的数据 `put` 到这个 `ContentValues` 对象当中，最后再调用 `SQLiteDatabase` 的 `insert()` 方法将数据添加到数据库表当中。

而使用 LitePal 来添加数据，这些操作可以简单到让你惊叹！我们只需要创建出模型类的实例，再将所有要存储的数据设置好，最后调用一下 `save()` 方法就可以了。

下面开始来动手实现，观察现有的模型类，你会发现它们都是没有继承结构的。没错，因为 LitePal 进行表管理操作时不需要模型类有任何的继承结构，但是进行 CRUD 操作时就不行了，必须要继承自 `DataSupport` 类才行，因此这里我们需要先把继承结构给加上。修改 `Book` 类中的代码，如下所示：

```
public class Book extends DataSupport {  
    ...  
}
```

接着我们开始向 `Book` 表中添加数据，修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {  
  
    @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 View.OnClickListener() {  
            @Override  
            public void onClick(View v) {  
                Book book = new Book();  
                book.setName("The Da Vinci Code");  
                book.setAuthor("Dan Brown");  
                book.setPages(454);  
                book.setPrice(16.96);  
                book.setPress("Unknow");  
                book.save();  
            }  
        });  
    }  
}
```

这段代码非常神奇，我们来仔细阅读一下。在添加数据按钮的点击事件里面，首先是创建出

了一个 Book 的实例，然后调用 Book 类中的各种 set 方法对数据进行设置，最后再调用 book.save()方法就能完成数据添加操作了。那么这个 save()方法是从哪儿来的呢？当然是从 DataSupport 类中继承而来的了。除了 save()方法之外，DataSupport 类还给我们提供了丰富的 CRUD 方法，这些我们在后面都会学到。

现在重新运行程序，点击一下 Add data 按钮，此时数据应该已经添加成功了，我们打开 BookStore.db 数据库瞧一瞧。输入 SQL 查询语句 select * from Book，结果如图 6.33 所示。



图 6.33 查看添加的数据

可以看到，作者、书名、页数、价格、出版社，这些数据全部精确无误地添加成功了。

6.5.5 使用 LitePal 更新数据

学习完了如何使用 LitePal 添加数据，接下来我们看看怎样使用 LitePal 更新数据。更新数据要比添加数据稍微复杂一点，因为它的 API 接口比较多，这里我们只介绍最常用的几种更新方式。

首先，最简单的一种更新方式就是对已存储的对象重新设值，然后重新调用 save()方法即可。那么这里我们就要了解一个概念，什么是已存储的对象？

对于 LitePal 来说，对象是否已存储就是根据调用 model.isSaved()方法的结果来判断的，返回 true 就表示已存储，返回 false 就表示未存储。那么接下来的问题就是，什么情况下会返回 true，什么情况下会返回 false 呢？

实际上只有在两种情况下 model.isSaved()方法才会返回 true，一种情况是已经调用过 model.save()方法去添加数据了，此时 model 会被认为是已存储的对象。另一种情况是 model 对象是通过 LitePal 提供的查询 API 查出来的，由于是从数据库中查到的对象，因此也会被认为 是已存储的对象。

由于查询 API 我们暂时还没学到，因此只能先通过第一种情况进行验证。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        Button updateData = (Button) findViewById(R.id.update_data);
        updateData.setOnClickListener(new View.OnClickListener() {
            @Override
            ...
        });
    }
}
```

```

public void onClick(View v) {
    Book book = new Book();
    book.setName("The Lost Symbol");
    book.setAuthor("Dan Brown");
    book.setPages(510);
    book.setPrice(19.95);
    book.setPress("Unknow");
    book.save();
    book.setPrice(10.99);
    book.save();
}
});
}
}

```

在更新数据按钮的点击事件里面，我们先是通过上一小节中学习的知识添加了一条 Book 数据，然后调用 `setPrice()` 方法将这本书的价格进行了修改，之后再次调用了 `save()` 方法。此时 LitePal 会发现当前的 Book 对象是已存储的，因此不会再向数据库中去添加一条新数据，而是会直接更新当前的数据。

现在重新运行一下程序，然后点击 Update data 按钮，我们再次输入查询语句查看表中的数据情况，结果如图 6.34 所示。



图 6.34 查看更新后的数据

可以看到，Book 表中新增了一条书的数据，但这本书的价格并不是一开始设置的 19.95，而是 10.99，说明我们的更新操作确实生效了。

但是这种更新方式只能对已存储的对象进行操作，限制性比较大，接下来我们学习另外一种更加灵巧的更新方式。修改 MainActivity 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity {

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

        Button updateData = (Button) findViewById(R.id.update_data);
        updateData.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Book book = new Book();
                book.setPrice(14.95);
            }
        });
    }
}

```

```
        book.setPress("Anchor");
        book.updateAll("name = ? and author = ?", "The Lost Symbol", "Dan
                      Brown");
    }
});
```

可以看到，这里我们首先 new 出了一个 Book 的实例，然后直接调用 `setPrice()` 和 `setPress()` 方法来设置要更新的数据，最后再调用 `updateAll()` 方法去执行更新操作。注意 `updateAll()` 方法中可以指定一个条件约束，和 `SQLiteDatabase` 中 `update()` 方法的 `where` 参数部分有点类似，但更加简洁，如果不指定条件语句的话，就表示更新所有数据。这里我们指定将所有书名是 The Lost Symbol 并且作者是 Dan Brown 的书价格更新为 14.95，出版社更新为 Anchor。

现在重新运行程序并点击 `Update data` 按钮，我们再次查询一下表中的数据情况，结果如图 6.35 所示。



图 6.35 再次查看更新后的数据

意料之中，第二本书的价格被更新成了 14.95，出版社被更新成了 Anchor。怎么样？LitePal 的更新 API 是不是明显比 SQLiteDatabase 的 update() 方法要好用多了？

不过，在使用 `updateAll()`方法时，还有一个非常重要的知识点是你需要知晓的，就是当你想把一个字段的值更新成默认值时，是不可以使用上面的方式来 `set` 数据的。我们都应该，在Java中任何一种数据类型的字段都会有默认值，例如 `int` 类型的默认值是 0，`boolean` 类型的默认值是 `false`，`String` 类型的默认值是 `null`。那么当 `new` 出一个 `Book` 对象时，其实所有字段都已经被初始化成默认值了，比如说 `pages` 字段的值就是 0。因此，如果我们想把数据库表中的 `pages` 列更新成 0，直接调用 `book.setPages(0)` 是不可以的，因为即使不调用这行代码，`pages` 字段本身也是 0，LitePal 此时是不会对这个列进行更新的。对于所有想要将为数据更新成默认值的操作，LitePal 统一提供了一个 `setDefault()` 方法，然后传入相应的列名就可以了实现了。比如我们可以这样写：

```
Book book = new Book();
book.setToDefault("pages");
book.updateAll();
```

这段代码的意思是，将所有书的页数都更新为 0，因为 `updateAll()` 方法中没有指定约束条件，因此更新操作对所有数据都生效了。

6.5.6 使用 LitePal 删除数据

使用 LitePal 删除数据的方式主要有两种，第一种比较简单，就是直接调用已存储对象的 `delete()` 方法就可以了，对于已存储对象的概念，我们在上一小节中已经学习过了。也就是说，调用过 `save()` 方法的对象，或者是通过 LitePal 提供的查询 API 查出来的对象，都是可以直接使用 `delete()` 方法来删除数据的。这种方式比较简单，我们就不进行代码演示了，下面直接来看另外一种删除数据的方式。

修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        Button deleteButton = (Button) findViewById(R.id.delete_data);
        deleteButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                DataSupport.deleteAll(Book.class, "price < ?", "15");
            }
        });
    }
}
```

这里调用了 `DataSupport.deleteAll()` 方法来删除数据，其中 `deleteAll()` 方法的第一个参数用于指定删除哪张表中的数据，`Book.class` 就意味着删除 `Book` 表中的数据，后面的参数用于指定约束条件，应该不难理解。那么这行代码的意思就是，删除 `Book` 表中价格低于 15 的书，正好目前 `Book` 表中有两本书，一本价格是 16.96，一本价格是 14.95，刚好可以看出效果。

现在重新运行程序，并点击一下 `Delete data` 按钮，然后查询表中的数据情况，如图 6.36 所示。



图 6.36 查看删除后的数据

可以看到，价格低于 15 的那本书已经被删除掉了。

另外，`deleteAll()` 方法如果不指定约束条件，就意味着你要删除表中的所有数据，这一点和 `updateAll()` 方法是比较相似的。

6.5.7 使用 LitePal 查询数据

终于又到了最复杂的查询数据部分了，不过这个“最复杂”只是相对于过去而言，因为使用 LitePal 来查询数据一点都不复杂。我一直都认为 LitePal 在查询 API 方面的设计极为人性化，想想之前我们所使用的 `query()` 方法，冗长的参数列表让人看得头疼，即使多数参数都是用不到的，也不得不传入 `null`，如下所示：

```
Cursor cursor = db.query("Book", null, null, null, null, null, null);
```

像这样的代码恐怕是没人会喜欢的。为此 LitePal 在查询 API 方面做了非常多的优化，基本上可以满足绝大多数场景的查询需求，并且代码十分整洁，下面我们就来一起学习一下。

首先分析一下上述代码，`query()` 方法中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。那么使用 LitePal 如何完成同样的功能呢？非常简单，只需要这样写：

```
List<Book> books = DataSupport.findAll(Book.class);
```

怎么样，代码是不是简单易懂多了？没有冗长的参数列表，只需要调用一下 `findAll()` 方法，然后通过 `Book.class` 参数指定查询 Book 表就可以。另外，`findAll()` 方法的返回值是一个 `Book` 类型的 `List` 集合，也就是说，我们不用像之前那样再通过 `Cursor` 对象一行行去取值了，LitePal 已经自动帮我们完成了赋值操作。

下面通过一个完整的例子来实践一下吧，修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        Button queryButton = (Button) findViewById(R.id.query_data);
        queryButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                List<Book> books = DataSupport.findAll(Book.class);
                for (Book book: books) {
                    Log.d("MainActivity", "book name is " + book.getName());
                    Log.d("MainActivity", "book author is " + book.getAuthor());
                    Log.d("MainActivity", "book pages is " + book.getPages());
                    Log.d("MainActivity", "book price is " + book.getPrice());
                    Log.d("MainActivity", "book press is " + book.getPress());
                }
            }
        });
    }
}
```

查询的那段代码刚刚已经解释过了，接下来就是遍历 List 集合中的 Book 对象，并将其中的信息全部打印出来。现在重新运行一下程序，点击 Query data 按钮，然后查看 logcat 的打印内容，结果如图 6.37 所示。

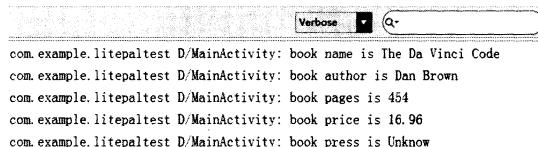


图 6.37 打印查询到的数据

Book 表中只剩下一条数据，由此可见，我们已经将这条数据成功查询出来了。

除了 `findAll()` 方法之外，LitePal 还提供了很多其他非常有用的查询 API。比如我们想要查询 Book 表中的第一条数据就可以这样写：

```
Book firstBook = DataSupport.findFirst(Book.class);
```

查询 Book 表中的最后一条数据就可以这样写：

```
Book lastBook = DataSupport.findLast(Book.class);
```

我们还可以通过连缀查询来定制更多的查询功能。

❑ `select()` 方法用于指定查询哪几列的数据，对应了 SQL 当中的 `select` 关键字。比如只查 `name` 和 `author` 这两列的数据，就可以这样写：

```
List<Book> books = DataSupport.select("name", "author").find(Book.class);
```

❑ `where()` 方法用于指定查询的约束条件，对应了 SQL 当中的 `where` 关键字。比如只查页数大于 400 的数据，就可以这样写：

```
List<Book> books = DataSupport.where("pages > ?", "400").find(Book.class);
```

❑ `order()` 方法用于指定结果的排序方式，对应了 SQL 当中的 `order by` 关键字。比如将查询结果按照书价从高到低排序，就可以这样写：

```
List<Book> books = DataSupport.order("price desc").find(Book.class);
```

其中 `desc` 表示降序排列，`asc` 或者不写表示升序排列。

❑ `limit()` 方法用于指定查询结果的数量，比如只查表中的前 3 条数据，就可以这样写：

```
List<Book> books = DataSupport.limit(3).find(Book.class);
```

❑ `offset()` 方法用于指定查询结果的偏移量，比如查询表中的第 2 条、第 3 条、第 4 条数据，就可以这样写：

```
List<Book> books = DataSupport.limit(3).offset(1).find(Book.class);
```

由于 `limit(3)` 查询到的是前 3 条数据，这里我们再加上 `offset(1)` 进行一个位置的偏移，就能实现查询第 2 条、第 3 条、第 4 条数据的功能了。`limit()` 和 `offset()` 方法共同对应了 SQL 当中的 `limit` 关键字。

当然，你还可以对这 5 个方法进行任意的连缀组合，来完成一个比较复杂的查询操作：

```
List<Book> books = DataSupport.select("name", "author", "pages")
    .where("pages > ?", "400")
    .order("pages")
    .limit(10)
    .offset(10)
    .find(Book.class);
```

这段代码就表示，查询 Book 表中第 11~20 条满足页数大于 400 这个条件的 `name`、`author` 和 `pages` 这 3 列数据，并将查询结果按照页数升序排列。

怎么样？是不是感觉 LitePal 的查询功能非常强大，并且代码明显更加简洁？我们需要用到一个方法的时候直接连缀一下就可以了，不需要的话就可以不写，而不是像之前的 `query()` 方法，不管需不需要用到，都必须要传固定的参数进去才行。

关于 LitePal 的查询 API 差不多就介绍到这里，这些 API 已经足够我们应对绝大多数场景的查询需求了。当前，如果你实在有一些特殊需求，上述的 API 都满足不了你的时候，LitePal 仍然支持使用原生的 SQL 来进行查询：

```
Cursor c = DataSupport.findBySQL("select * from Book where pages > ? and price < ?",
    "400", "20");
```

调用 `DataSupport.findBySQL()` 方法来进行原生查询，其中第一个参数用于指定 SQL 语句，后面的参数用于指定占位符的值。注意 `findBySQL()` 方法返回的是一个 `Cursor` 对象，接下来你还需要通过之前所学的老方式将数据一一取出才行。

6.6 小结与点评

经过了这一章漫长的学习，我们终于可以缓解一下疲劳，对本章所学的知识进行梳理和总结了。本章主要是对 Android 常用的数据持久化方式进行了详细的讲解，包括文件存储、`SharedPreferences` 存储以及数据库存储。其中文件适用于存储一些简单的文本数据或者二进制数据，`SharedPreferences` 适用于存储一些键值对，而数据库则适用于存储那些复杂的关系型数据。虽然目前你已经掌握了这 3 种数据持久化方式的用法，但是能够根据项目的需求来选择最合适的方式也是你未来需要继续探索的。

那么正如上一章小结里提到的，既然现在我们已经掌握了 Android 中的数据持久化技术，接下来就应该继续学习 Android 中剩余的四大组件了。放松一下自己，然后一起踏上内容提供器的学习之旅吧。

第 7 章

跨程序共享数据——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，这个是要视情况而定的，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。

7.1 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。

不过在正式开始学习内容提供器之前，我们需要先掌握另外一个非常重要的知识——Android 运行时权限，因为待会的内容提供器示例中会使用到运行时权限的功能。当然不光是

内容提供器，以后我们的开发过程中也会经常使用到运行时权限，因此你必须能够牢牢掌握它才行。

7.2 运行时权限

Android 的权限机制并不是什么新鲜事物，从系统的第一版开始就已经存在了。但其实之前 Android 的权限机制在保护用户安全和隐私等方面起到的作用比较有限，尤其是一些大家都离不开的常用软件，非常容易“店大欺客”。为此，Android 开发团队在 Android 6.0 系统中引入了运行时权限这个功能，从而更好地保护了用户的安全和隐私，那么本节我们就来详细学习一下这个 6.0 系统中引入的新特性。

7.2.1 Android 权限机制详解

首先来回顾一下过去 Android 的权限机制是什么样的。我们在第 5 章写 BroadcastTest 项目的时候第一次接触了 Android 权限相关的内容，当时为了要访问系统的网络状态以及监听开机广播，于是在 AndroidManifest.xml 文件中添加了这样两句权限声明：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.broadcasttest">  
  
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />  
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />  
    ...  
</manifest>
```

因为访问系统的网络状态以及监听开机广播涉及了用户设备的安全性，因此必须在 AndroidManifest.xml 中加入权限声明，否则我们的程序就会崩溃。

那么现在问题来了，加入了这两句权限声明后，对于用户来说到底有什么影响呢？为什么这样就可以保护用户设备的安全性了呢？

其实用户主要在以下两个方面得到了保护，一方面，如果用户在低于 6.0 系统的设备上安装该程序，会在安装界面给出如图 7.1 所示的提醒。这样用户就可以清楚地知晓该程序一共申请了哪些权限，从而决定是否要安装这个程序。

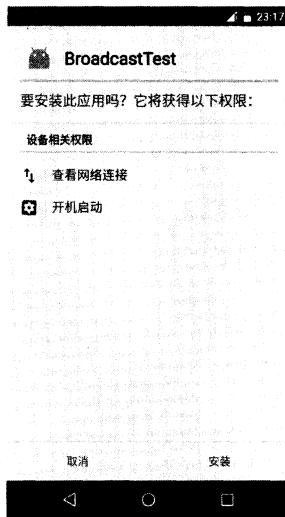


图 7.1 安装界面的权限提醒

另一方面，用户可以随时在应用程序管理界面查看任意一个程序的权限申请情况，如图 7.2 所示。这样该程序申请的所有权限就尽收眼底，什么都瞒不过用户的眼睛，以此保证应用程序不会出现各种滥用权限的情况。



图 7.2 管理界面的权限展示

这种权限机制的设计思路其实非常简单，就是用户如果认可你所申请的权限，那么就会安装你的程序，如果不认可你所申请的权限，那么拒绝安装就可以了。

但是理想是美好的，现实却很残酷，因为很多我们所离不开的常用软件普遍存在着滥用权限

的情况，不管到底用不用得到，反正先把权限申请了再说。比如说微信所申请的权限列表如图 7.3 所示。

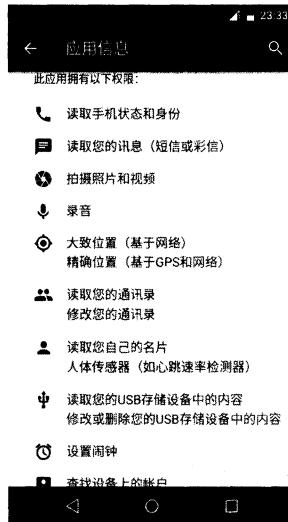


图 7.3 微信的权限列表

这只是微信所申请的一半左右的权限，因为权限太多一屏截不下来。其中有一些权限我并不认可，比如微信为什么要读取我手机的短信和彩信？但是我不认可又能怎样，难道我拒绝安装微信？没错，这种例子比比皆是，当一些软件已经让我们产生依赖的时候就会容易“店大欺客”，反正这个权限我就是要了，你自己看着办吧！

Android 开发团队当然也意识到了这个问题，于是在 6.0 系统中加入了运行时权限功能。也就是说，用户不需要在安装软件的时候一次性授权所有申请的权限，而是在软件的使用过程中再对某一项权限申请进行授权。比如说一款相机应用在运行时申请了地理位置定位权限，就算我拒绝了这个权限，但是我应该仍然可以使用这个应用的其他功能，而不是像之前那样直接无法安装它。

当然，并不是所有权限都需要在运行时申请，对于用户来说，不停地授权也很烦琐。Android 现在将所有的权限归成了两类，一类是普通权限，一类是危险权限。普通权限指的是那些不会直接威胁到用户的安全和隐私的权限，对于这部分权限申请，系统会自动帮我们进行授权，而不需要用户再去手动操作了，比如在 BroadcastTest 项目中申请的两个权限就是普通权限。危险权限则表示那些可能会触及用户隐私，或者对设备安全性造成影响的权限，如获取设备联系人信息、定位设备的地理位置等，对于这部分权限申请，必须要由用户手动点击授权才可以，否则程序就无法使用相应的功能。

但是 Android 中有一共有上百种权限，我们怎么从中区分哪些是普通权限，哪些是危险权限

呢？其实并没有那么难，因为危险权限总共就那么几个，除了危险权限之外，剩余的就都是普通权限了。下表列出了Android中所有的危险权限，一共是9组24个权限。

权限组名	权限名
CALENDAR	READ_CALENDAR
	WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS
	WRITE_CONTACTS
	GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION
	ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE
	CALL_PHONE
	READ_CALL_LOG
	WRITE_CALL_LOG
	ADD_VOICEMAIL
	USE_SIP
SENSORS	PROCESS_OUTGOING_CALLS
	BODY_SENSORS
SMS	SEND_SMS
	RECEIVE_SMS
	READ_SMS
	RECEIVE_WAP_PUSH
	RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE
	WRITE_EXTERNAL_STORAGE

这张表格你看起来可能并不会那么轻松，因为里面的权限全都是你没使用过的。不过没关系，你并不需要了解表格中每个权限的作用，只要把它当成一个参照表来查看就行了。每当要使用一个权限时，可以先到这张表中来查一下，如果是属于这张表中的权限，那么就需要进行运行时权限处理，如果不在这张表中，那么只需要在AndroidManifest.xml文件中添加一下权限声明就可以了。

另外注意一下，表格中每个危险权限都属于一个权限组，我们在进行运行时权限处理时使用的是权限名，但是用户一旦同意授权了，那么该权限所对应的权限组中所有的其他权限也会同时被授权。

访问<http://developer.android.com/reference/android/Manifest.permission.html>可以查看Android系统中完整的权限列表。

好了，关于 Android 权限机制的内容就讲这么多，理论知识你已经了解得非常充足了。接下来我们就学习一下到底如何在程序运行的时候申请权限。

7.2.2 在程序运行时申请权限

首先新建一个 RuntimePermissionTest 项目，我们就在这个项目的基础上来学习运行时权限的使用方法。在开始动手之前还需要考虑一下到底要申请什么权限，其实刚才表中列出的所有权限都是可以申请的，这里简单起见我们就使用 CALL_PHONE 这个权限来作为本小节中的示例吧。

CALL_PHONE 这个权限是编写拨打电话功能的时候需要声明的，因为拨打电话会涉及用户手机的资费问题，因而被列为了危险权限。在 Android 6.0 系统出现之前，拨打电话功能的实现其实非常简单，修改 activity_main.xml 布局文件，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/make_call"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Make Call" />

</LinearLayout>
```

我们在布局文件中只是定义了一个按钮，当点击按钮时就去触发拨打电话的逻辑。接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button makeCall = (Button) findViewById(R.id.make_call);
        makeCall.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                try {
                    Intent intent = new Intent(Intent.ACTION_CALL);
                    intent.setData(Uri.parse("tel:10086"));
                    startActivity(intent);
                } catch (SecurityException e) {
                    e.printStackTrace();
                }
            }
        });
    }
}
```

可以看到，在按钮的点击事件中，我们构建了一个隐式 Intent，Intent 的 action 指定为 Intent.ACTION_CALL，这是一个系统内置的打电话的动作，然后在 data 部分指定了协议是 tel，号码是 10086。其实这部分代码我们在 2.3.3 小节中就已经见过了，只不过当时指定的 action 是 Intent.ACTION_DIAL，表示打开拨号界面，这个是不需要声明权限的，而 Intent.ACTION_CALL 则可以直接拨打电话，因此必须声明权限。另外为了防止程序崩溃，我们将所有操作都放在了异常捕获代码块当中。

那么接下来修改 AndroidManifest.xml 文件，在其中声明如下权限：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.runtimepermissiontest">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
    </application>

</manifest>
```

这样我们就将拨打电话的功能成功实现了，并且在低于 Android 6.0 系统的手机上都是可以正常运行的，但是如果我们在 6.0 或者更高版本系统的手机上运行，点击 Make Call 按钮就没有任何效果，这时观察 logcat 中的打印日志，你会看到如图 7.4 所示的错误信息。

```
java.lang.SecurityException: Permission Denial: starting Intent { act=android.intent.action.CALL
    at android.os.Parcel.readException(Parcel.java:1599)
    at android.os.Parcel.readException(Parcel.java:1520)
    at android.app.ActivityManagerProxy.startActivity(ActivityManagerNative.java:2658)
    at android.app.Instrumentation.execStartActivity(Instrumentation.java:1507)
    at android.app.Activity.startActivityForResult(Activity.java:3917)
    at android.app.Activity.startActivityForResult(Activity.java:3870)
    at android.support.v4.app.FragmentActivity.startActivityForResult(FragmentActivity.java:343)
    at android.app.Activity.startActivity(Activity.java:4200)
    at android.app.Activity.startActivity(Activity.java:4168)
    at com.example.runtimepermissiontest.MainActivity$1.onClick(MainActivity.java:29)
```

图 7.4 错误日志信息

错误信息中提醒我们“Permission Denial”，可以看出，是由于权限被禁止所导致的，因为 6.0 及以上系统在使用危险权限时都必须进行运行时权限处理。

那么下面我们就来尝试修复这个问题，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    @Override
```

```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Button makeCall = (Button) findViewById(R.id.make_call);
    makeCall.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.
                permission.CALL_PHONE) != PackageManager.PERMISSION_GRANTED) {
                ActivityCompat.requestPermissions(MainActivity.this, new
                    String[]{Manifest.permission.CALL_PHONE}, 1);
            } else {
                call();
            }
        }
    });
}

private void call() {
    try {
        Intent intent = new Intent(Intent.ACTION_CALL);
        intent.setData(Uri.parse("tel:10086"));
        startActivity(intent);
    } catch (SecurityException e) {
        e.printStackTrace();
    }
}

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
    int[] grantResults) {
    switch (requestCode) {
        case 1:
            if (grantResults.length > 0 && grantResults[0] == PackageManager.
                PERMISSION_GRANTED) {
                call();
            } else {
                Toast.makeText(this, "You denied the permission", Toast.LENGTH_
                    SHORT).show();
            }
            break;
        default:
    }
}
}

```

上面的代码将运行时权限的完整流程都覆盖了，下面我们来具体解析一下。说白了，运行时权限的核心就是在程序运行过程中由用户授权我们去执行某些危险操作，程序是不可以擅自做主去执行这些危险操作的。因此，第一步就是要先判断用户是不是已经给过我们授权了，借助的是 ContextCompat.checkSelfPermission() 方法。checkSelfPermission() 方法接收两个参数，第一个参数是 Context，这个没什么好说的，第二个参数是具体的权限名，比如打电话的权限名

就是 `Manifest.permission.CALL_PHONE`, 然后我们使用方法的返回值和 `PackageManager.PERMISSION_GRANTED` 做比较, 相等就说明用户已经授权, 不等就表示用户没有授权。

如果已经授权的话就简单了, 直接去执行拨打电话的逻辑操作就可以了, 这里我们把拨打电话的逻辑封装到了 `call()` 方法当中。如果没有授权的话, 则需要调用 `ActivityCompat.requestPermissions()` 方法来向用户申请授权, `requestPermissions()` 方法接收 3 个参数, 第一个参数要求是 `Activity` 的实例, 第二个参数是一个 `String` 数组, 我们把要申请的权限名放在数组中即可, 第三个参数是请求码, 只要是唯一值就可以了, 这里传入了 1。

调用完了 `requestPermissions()` 方法之后, 系统会弹出一个权限申请的对话框, 然后用户可以选择同意或拒绝我们的权限申请, 不论是哪种结果, 最终都会回调到 `onRequestPermissionsResult()` 方法中, 而授权的结果则会封装在 `grantResults` 参数当中。这里我们只需要判断一下最后的授权结果, 如果用户同意的话就调用 `call()` 方法来拨打电话, 如果用户拒绝的话我们只能放弃操作, 并且弹出一条失败提示。

现在重新运行一下程序, 并点击 `Make Call` 按钮, 效果如图 7.5 所示。

由于用户还没有授权过我们拨打电话权限, 因此第一次运行会弹出这样一个权限申请的对话框, 用户可以选择同意或者拒绝, 比如说这里点击了 `DENY`, 结果如图 7.6 所示。

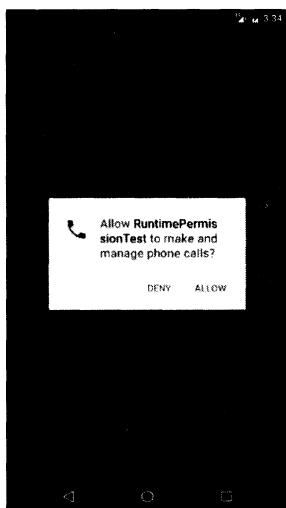


图 7.5 申请电话权限对话框

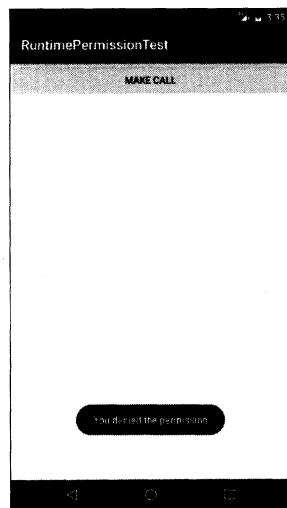


图 7.6 用户拒绝了权限申请

由于用户没有同意授权, 我们只能弹出一个操作失败的提示。下面我们再次点击 `Make Call` 按钮, 仍然会弹出权限申请的对话框, 这次点击 `ALLOW`, 结果如图 7.7 所示。

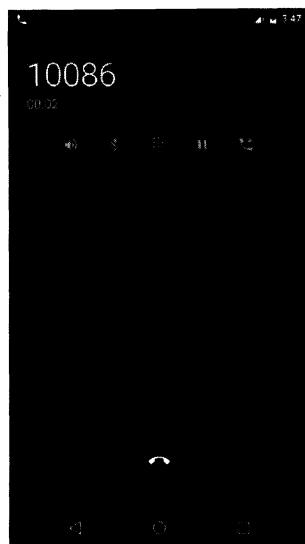


图 7.7 拨打电话界面

可以看到，这次我们就成功进入到拨打电话界面了，并且由于用户已经完成了授权操作，之后再点击 Make Call 按钮就不会再弹出权限申请对话框了，而是可以直接拨打电话。那可能你会担心，万一以后我又后悔了怎么办？没有关系，用户随时都可以将授予程序的危险权限进行关闭，进入 Settings → Apps → RuntimePermissionTest → Permissions，界面如图 7.8 所示。

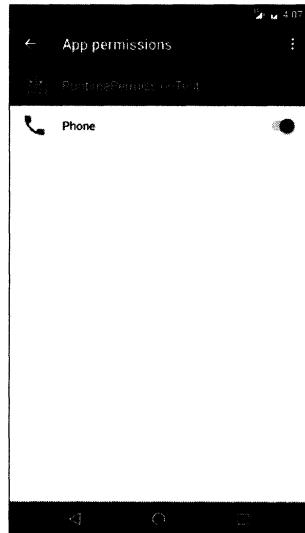


图 7.8 应用程序权限管理界面

在这里我们就可以对任何授予过的危险权限进行关闭了。

好了，关于运行时权限的内容就讲到这里，现在你已经有能力处理 Android 上各种关于权限的问题了，下面我们就来进入本章的正题——内容提供器。

7.3 访问其他程序中的数据

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。

如果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，那么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。

7.3.1 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 ContentResolver 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：authority 和 path。authority 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 authority 就可以命名为 `com.example.app.provider`。path 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 authority 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 path 分别命名为 `/table1` 和 `/table2`，然后把 authority 和 path 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，因为如果使用表名的话，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。是不是感觉一看就懂？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。

7.3.2 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 7.9 所示。

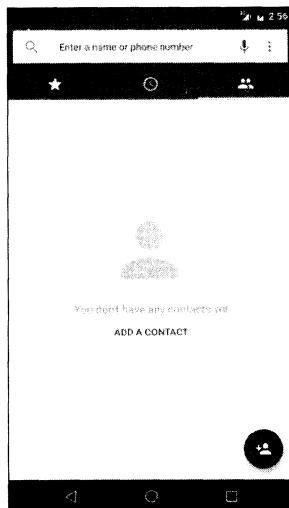


图 7.9 电话簿程序主界面

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 ADD A 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 7.10 所示。

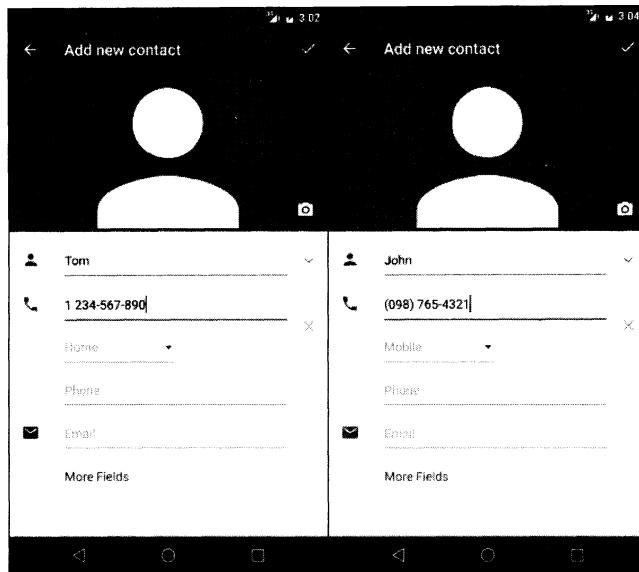


图 7.10 添加两个联系人

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:orientation="vertical"
    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。这里使用 ListView 而不是 RecyclerView，是因为我们要将关注的重点放在读取系统联系人上面，如果使用 RecyclerView 的话，代码偏多，会容易让我们找不着重点。

接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    ArrayAdapter<String> adapter;

    List<String> contactsList = new ArrayList<>();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ListView contactsView = (ListView) findViewById(R.id.contacts_view);
        adapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_
            item_1, contactsList);
        contactsView.setAdapter(adapter);
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_
            CONTACTS) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(this, new String[]{ Manifest.
                permission.READ_CONTACTS }, 1);
        } else {
            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);
            if (cursor != 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);
                }
                adapter.notifyDataSetChanged();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (cursor != null) {
                cursor.close();
            }
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions,
        int[] grantResults) {
        switch (requestCode) {
```

```

        case 1:
            if (grantResults.length > 0 && grantResults[0] == PackageManager.
                PERMISSION_GRANTED) {
                readContacts();
            } else {
                Toast.makeText(this, "You denied the permission", Toast.LENGTH_
                SHORT).show();
            }
            break;
        default:
    }
}

}

```

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，然后开始调用运行时权限的处理逻辑，因为 `READ_CONTACTS` 权限是属于危险权限的。关于运行时权限的处理流程相信你已经熟练掌握了，这里我们在用户授权之后调用 `readContacts()` 方法来读取系统联系人信息。

下面重点看一下 `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，并且在中间加上换行符，然后将拼接后的数据添加到 `ListView` 的数据源里，并通知刷新一下 `ListView`。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，读取系统联系人的权限千万不能忘记声明。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">

    <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了，让我们来运行一下程序吧，效果如图 7.11 所示。

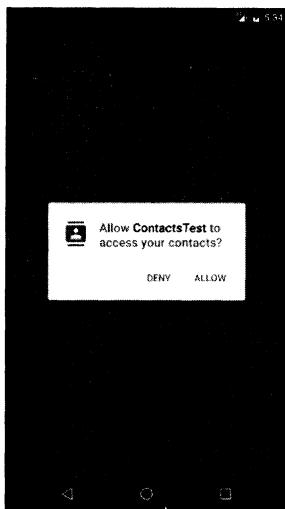


图 7.11 申请访问联系人权限对话框

首先弹出了申请访问联系人权限的对话框，我们点击 ALLOW，然后结果如图 7.12 所示。

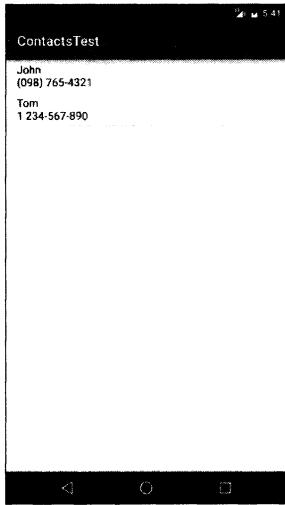


图 7.12 展示系统联系人信息

刚刚添加的两个联系人的数据都成功读取出来了！这说明跨程序访问数据的功能确实是实现了。

7.4 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——解开。

7.4.1 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 6 个抽象方法，我们在使用子类继承它的时候，需要将这 6 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 6 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。

1. 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。

2. 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。

3. 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。

4. 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。

5. 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。

6. 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。

- *：表示匹配任意长度的任意字符。
- #：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 3 个参数，可以分别把 authority、path 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_DIR);
        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 4 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 3 部分组成，Android 对这 3 个部分做了如下格式规定。

- 必须以 vnd 开头。
- 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/。
- 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。

7.4.2 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`。然后创建一个内容提供器，右击 `com.example.broadcasttest` 包 → New → Other → Content Provider，会弹出如图 7.13 所示的窗口。

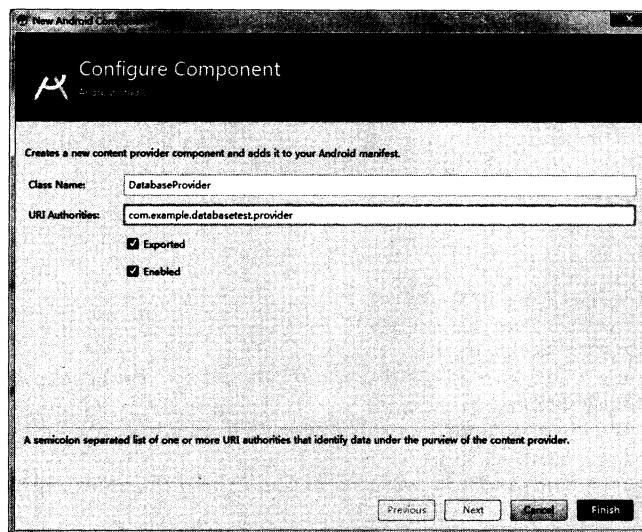


图 7.13 创建内容提供器的窗口

可以看到,这里我们将内容提供器命名为 DatabaseProvider, authority 指定为 com.example.databasetest.provider, Exported 属性表示是否允许外部程序访问我们的内容提供器, Enabled 属性表示是否启用这个内容提供器。将两个属性都勾中, 点击 Finish 完成创建。

接着我们修改 DatabaseProvider 中的代码, 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;
    }
    return uriReturn;
}
```

```
        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 4 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。

另外还有一点需要注意，内容提供器一定要在 `AndroidManifest.xml` 文件中注册才可以使用。不过幸运的是，由于我们是使用 `Android Studio` 的快捷方式创建的内容提供器，因此注册这一步已经被自动完成了。打开 `AndroidManifest.xml` 文件瞧一瞧，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.databasetest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <provider
            android:name=".DatabaseProvider"
            android:authorities="com.example.databasetest.provider"
            android:enabled="true"
            android:exported="true">
        </provider>
    </application>
</manifest>
```

可以看到，`<application>` 标签内出现了一个新的标签 `<provider>`，我们使用它来对 `DatabaseProvider` 这个内容提供器进行注册。`android:name` 属性指定了 `DatabaseProvider` 的类名，

`android:authorities` 属性指定了 `DatabaseProvider` 的 authority，而 `enabled` 和 `exported` 属性则是根据我们刚才勾选的状态自动生成的，这里表示允许 `DatabaseProvider` 被其他应用程序进行访问。

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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <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 4 个按钮，分别用于添加、查询、修改和删除数据。然后修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    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 View.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 View.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 View.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 View.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 4 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 表中的其他行受到影响，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 7.14 所示的界面。

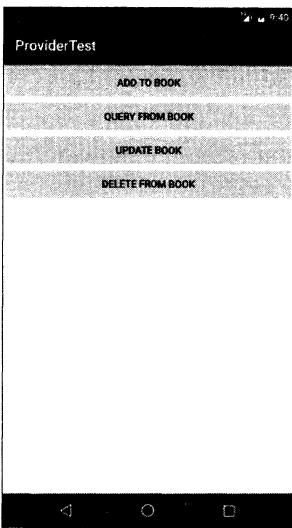


图 7.14 ProviderTest 主界面

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 7.15 所示。

```
Verbose ↻ ⚖ q
com.example.providertest D/MainActivity: book name is A Clash of Kings
com.example.providertest D/MainActivity: book author is George Martin
com.example.providertest D/MainActivity: book pages is 1040
com.example.providertest D/MainActivity: book price is 22.85
```

图 7.15 查询添加的数据

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 7.16 所示。

```
Verbose ↻ ⚖ q
com.example.providertest D/MainActivity: book name is A Storm of Swords
com.example.providertest D/MainActivity: book author is George Martin
com.example.providertest D/MainActivity: book pages is 1216
com.example.providertest D/MainActivity: book price is 24.05
```

图 7.16 查询更新后的数据

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，学习更多关于 Git 的用法。

7.5 Git 时间——版本控制工具进阶

在上一次的 Git 时间里，我们学习了关于 Git 最基本的用法，包括安装 Git、创建代码仓库，以及提交本地代码。本节中我们将要学习更多的使用技巧，不过在开始之前先要把准备工作做好。

所谓的准备工作就是要给一个项目创建代码仓库，这里就选择在 ProviderTest 项目中创建吧，打开 Git Bash，进入到这个项目的根目录下面，然后执行 `git init` 命令，如图 7.17 所示。



图 7.17 创建代码仓库

这样准备工作就已经完成了，让我们继续开始 Git 之旅吧。

7.5.1 忽略文件

代码仓库现在已经创建好了，接下来我们应该去提交 ProviderTest 项目中的代码。不过在提交之前你也许应该思考一下，是不是所有的文件都需要加入到版本控制当中呢？

在第 1 章介绍 Android 项目结构的时候有提到过，`build` 目录下的文件都是编译项目时自动生成的，我们不应该将这部分文件添加到版本控制当中，那么如何才能实现这样的效果呢？

Git 提供了一种可配性很强的机制来允许用户将指定的文件或目录排除在版本控制之外，它会检查代码仓库的目录下是否存在一个名为 `.gitignore` 的文件，如果存在的话，就去一行行读取这个文件中的内容，并把每一行指定的文件或目录排除在版本控制之外。注意 `.gitignore` 中指定的文件或目录是可以使用 “`*`” 通配符的。

神奇的是，我们并不需要自己去创建 `.gitignore` 文件，Android Studio 在创建项目的时候会自动帮我们创建出两个 `.gitignore` 文件，一个在根目录下面，一个在 `app` 模块下面。首先看一下根目录下面的 `.gitignore` 文件，如图 7.18 所示。

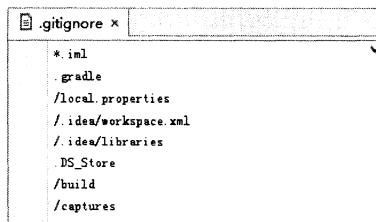


图 7.18 根目录下面的 `.gitignore` 文件

这是 Android Studio 自动生成的一些默认配置，通常情况下，这部分内容都是不用添加到版

本控制当中的。我们来简单阅读一下这个文件，除了*.iml表示指定任意以.iml结尾的文件，其他都是指定的具体的文件名或者目录名，上面配置中的所有内容都不会被添加到版本控制当中，因为基本都是一些由IDE自动生成的配置。

再来看一下app模块下面的.gitignore文件，这个就简单多了，如图7.19所示。

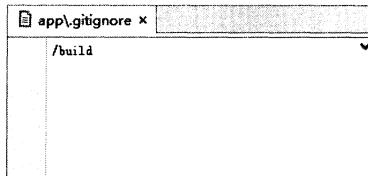


图7.19 app模块下面的.gitignore文件

由于app模块下面基本都是我们编写的代码，因此默认情况下只有其中的build目录不会被添加到版本控制当中。

当然，我们完全可以对以上两个文件进行任意地修改，来满足特定的需求。比如说，app模块下面的所有测试文件都只是给我自己使用的，我并不想把它们添加到版本控制中，那么就可以这样修改app/.gitignore文件中的内容：

```
/build
/src/test
/src/androidTest
```

没错，只需添加这样两行配置，因为所有的测试文件都是放在这两个目录下的。现在我们可以提交代码了，先使用add命令将所有的文件进行添加，如下所示：

```
git add .
```

然后执行commit命令完成提交，如下所示：

```
git commit -m "First commit."
```

7.5.2 查看修改内容

在进行了第一次代码提交之后，我们后面还可能会对项目不断地进行维护或添加新功能等。比较理想的情况是每当完成了一小块功能，就执行一次提交。但是如果某个功能牵扯到的代码比较多，有可能写到后面的时候我们就已经忘记前面修改了什么东西了。遇到这种情况时不用担心，Git全都帮你记着呢！下面我们就来学习一下如何使用Git来查看自上次提交后文件修改的内容。

查看文件修改情况的方法非常简单，只需要使用status命令就可以了，在项目的根目录下输入如下命令：

```
git status
```

然后 Git 会提示目前项目中没有任何可提交的文件，因为我们刚刚才提交过嘛。现在对 ProviderTest 项目中的代码稍做一下改动，修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        addData.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                ...
                values.put("price", 55.55);
                ...
            }
        });
        ...
    }
}
```

这里仅仅是在添加数据的时候，将书的价格由 22.85 改成了 55.55。然后重新输入 git status 命令，这次结果如图 7.20 所示。

图 7.20 查看文件变动情况

可以看到，Git 提醒我们 MainActivity.java 这个文件已经发生了更改，那么如何才能看到更改的内容呢？这就需要借助 diff 命令了，用法如下所示：

```
git diff
```

这样可以查看到所有文件的更改内容，如果你只想查看 MainActivity.java 这个文件的更改内容，可以使用如下命令：

```
git diff app/src/main/java/com/example/providertest/MainActivity.java
```

命令的执行结果如图 7.21 所示。

图 7.21 查看修改的具体内容

其中，减号代表删除的部分，加号代表添加的部分。从图中我们就可以明显地看出，书的价格由 22.85 被修改成了 55.55。

7.5.3 撤销未提交的修改

有时候我们的代码可能会写得过于草率，以至于原本正常的功能，结果反倒被我们改出了问题。遇到这种情况时也不用着急，因为只要代码还未提交，所有修改的内容都是可以撤销的。

比如在上一小节中我们修改了 MainActivity 里一本书的价格，现在如果想要撤销这个修改就可以使用 `checkout` 命令，用法如下所示：

```
git checkout app/src/main/java/com/example/providertest/MainActivity.java
```

执行了这个命令之后，我们对 MainActivity.java 这个文件所做的一切修改就应该都被撤销了。重新运行 `git status` 命令检查一下，结果如图 7.22 所示。

```
git status
# On branch master
# Changes to be committed:
#   (use "git add ..." to include in what will be committed)
#
#       modified:   MainActivity.java
```

图 7.22 重新查看文件变动情况

可以看到，当前项目中没有任何可提交的文件，说明撤销操作确实是成功了。

不过这种撤销方式只适用于那些还没有执行过 `add` 命令的文件，如果某个文件已经被添加过了，这种方式就无法撤销其更改的内容，我们来做个试验瞧一瞧。

首先仍然是将 MainActivity 中那本书的价格改成 55.55，然后输入如下命令：

```
git add .
```

这样就把所有修改的文件都进行了添加，可以输入 `git status` 来检查一下，结果如图 7.23 所示。

```
git status
# On branch master
# Changes to be committed:
#   (use "git add ..." to include in what will be committed)
#
#       modified:   MainActivity.java
```

图 7.23 再次查看文件变动情况

现在我们再执行一遍 `checkout` 命令，你会发现 MainActivity 仍然是处于已添加状态，所修改的内容无法撤销掉。

这种情况应该怎么办？难道我们还没法后悔了？当然不是，只不过对于已添加的文件我们应该先对其取消添加，然后才可以撤回提交。取消添加使用的是 `reset` 命令，用法如下所示：

```
git reset HEAD app/src/main/java/com/example/providertest/MainActivity.java
```

然后再运行一遍 `git status` 命令，你就会发现 `MainActivity.java` 这个文件重新变回了未添加状态，此时就可以使用 `checkout` 命令来将修改的内容进行撤销了。

7.5.4 查看提交记录

当 `ProviderTest` 这个项目开发了几个月之后，我们可能已经执行过上百次的提交操作了，这个时候估计你早就已经忘记每次提交都修改了哪些内容。不过没关系，忠实的 Git 一直都帮我们清清楚楚地记录着呢！可以使用 `log` 命令查看历史提交信息，用法如下所示：

```
git log
```

由于目前我们只执行过一次提交，所以能看到的信息很少，如图 7.24 所示。



图 7.24 查看提交记录

可以看到，每次提交记录都会包含提交 id、提交人、提交日期以及提交描述这 4 个信息。那么我们再次将书价修改成 55.55，然后执行一次提交操作，如下所示：

```
git add .
git commit -m "Change price."
```

现在重新执行 `git log` 命令，结果如图 7.25 所示。



图 7.25 重新查看提交记录

当提交记录非常多的时候，如果我们只想查看其中一条记录，可以在命令中指定该记录的 id，并加上 `-1` 参数表示我们只想看到一行记录，如下所示：

```
git log 1fa380b502a00b82bfc8d84c5ab5e15b8fbf7dac -1
```

而如果想要查看这条提交记录具体修改了什么内容，可以在命令中加入 `-p` 参数，命令如下：

```
git log 1fa380b502a00b82bfc8d84c5ab5e15b8fbf7dac -1 -p
```

查询出的结果如图 7.26 所示，其中减号代表删除的部分，加号代表添加的部分。

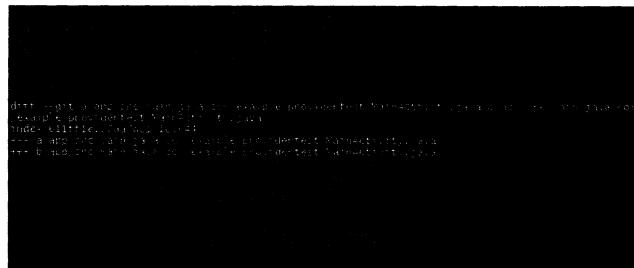


图 7.26 查看提交记录的具体修改内容

好了，本次的 Git 时间就到这里，下面我们来对本章中所学的知识做个回顾吧。

7.6 小结与点评

本章的内容不算多，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，我们一开始先了解了 Android 的权限机制，并且学会了如何在 6.0 以上的系统中使用运行时权限，然后又重点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口味，学习一下 Android 多媒体方面的知识吧。

第 8 章

丰富你的程序——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，手机在我们的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，可以戴着耳机听音乐。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 在这方面也做得非常出色。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。

前面的 7 章内容，我们一直都是使用模拟器来运行程序的，不过本章涉及的一些功能必须要在真正的 Android 手机上运行才看得到效果。因此，首先我们就来学习一下，如何使用 Android 手机来运行程序。

8.1 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，如果你还没有的话，赶紧去购买吧。

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 8.1 所示。

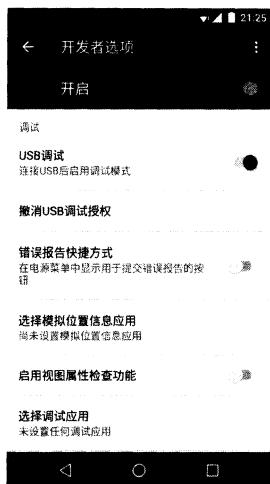


图 8.1 启用 USB 调试

注意从 Android 4.2 系统开始，开发者选项默认是隐藏的，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续点击，就会让开发者选项显示出来。

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 360 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 8.2 所示。



图 8.2 手机成功连接上电脑

现在观察 Android Monitor，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 8.3 所示。



图 8.3 在线设备列表

然后运行一下当前项目，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 8.4 所示。

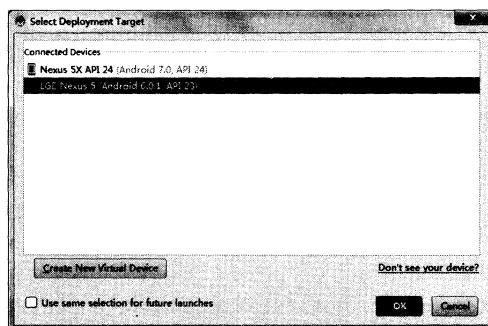


图 8.4 选择运行设备对话框

选中下面的 LGE Nexus 5 后点击 OK，就会将程序运行到手机上了。

8.2 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。

8.2.1 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);
```

接下来需要使用一个 `Builder` 构造器来创建 `Notification` 对象，但问题在于，几乎 Android 系统的每一个版本都会对通知这部分功能进行或多或少的修改，API 不稳定性问题在通知上面突显得尤其严重。那么该如何解决这个问题呢？其实解决方案我们之前已经见过好几回了，就是使用 support 库中提供的兼容 API。`support-v4` 库中提供了一个 `NotificationCompat` 类，使用这个类的构造器来创建 `Notification` 对象，就可以保证我们的程序在所有 Android 系统版本上都能正常工作了，代码如下所示：

```
Notification notification = new NotificationCompat.Builder(context).build();
```

当然，上述代码只是创建了一个空的 `Notification` 对象，并没有什么实际作用，我们可以在最终的 `build()` 方法之前连缀任意多的设置方法来创建一个丰富的 `Notification` 对象，先来看一些最基本的设置：

```
Notification notification = new NotificationCompat.Builder(context)
    .setContentTitle("This is content title")
    .setContentText("This is content text")
    .setWhen(System.currentTimeMillis())
    .setSmallIcon(R.drawable.small_icon)
    .setLargeIcon(BitmapFactory.decodeResource(getResources(),
        R.drawable.large_icon))
    .build();
```

上述代码中一共调用了 5 个设置方法，下面我们来一一解析一下。`setContentTitle()` 方法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。`setContentText()` 方法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。`setWhen()` 方法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。`setSmallIcon()` 方法用于设置通知的小图标，注意只能使用纯 alpha 图层的图片进行设置，小图标会显示在系统状态栏上。`setLargeIcon()` 方法用于设置通知的大图标，当下拉系统状态栏时，就可以看到设置的大图标了。

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"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/send_notice"
        android:layout_width="wrap_content"
        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 AppCompatActivity implements View.OnClickListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button 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 NotificationCompat.Builder(this)
                    .setContentTitle("This is content title")
                    .setContentText("This is content text")
                    .setWhen(System.currentTimeMillis())
                    .setSmallIcon(R.mipmap.ic_launcher)
                    .setLargeIcon(BitmapFactory.decodeResource(getResources(),
                        R.mipmap.ic_launcher))
                    .build();
                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。不过这里简单起见，我将通知栏的大小图都直接设置成了 ic_launcher 这张图，

这样就不用再去专门准备图标了，而在实际项目中千万不要这样偷懒。

现在可以来运行一下程序了，点击 Send notice 按钮，你会在系统状态栏的最左边看到一个小图标，如图 8.5 所示。

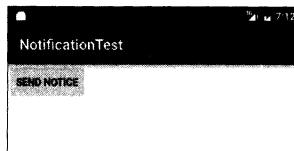


图 8.5 通知的小图标

下拉系统状态栏可以看到该通知的详细信息，如图 8.6 所示。

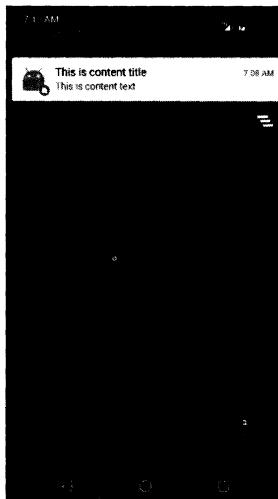


图 8.6 通知的详细信息

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 4 种值可选，每种值的具体含义你可以查看文档，通常情况下这个参数传入 0 就可以了。

对 PendingIntent 有了一定的了解后，我们再回过头来看一下 NotificationCompat.Builder。这个构造器还可以再连缀一个 setContentIntent() 方法，接收的参数正是一个 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，右击 com.example.notificationtest 包 → New → Activity → Empty Activity，新建 NotificationActivity，布局起名为 notification_layout。然后修改 notification_layout.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" >

    <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>
```

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 AppCompatActivity implements View.OnClickListener {

    ...

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.send_notice:
                Intent intent = new Intent(this, NotificationActivity.class);
                PendingIntent pi = PendingIntent.getActivity(this, 0, intent, 0);
                NotificationManager manager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
                Notification notification = new NotificationCompat.Builder(this)
                    .setContentTitle("This is content title")
```

```
        .setContentText("This is content text")
        .setWhen(System.currentTimeMillis())
        .setSmallIcon(R.mipmap.ic_launcher)
        .setLargeIcon(BitmapFactory.decodeResource(getResources(),
            R.mipmap.ic_launcher))
        .setContentIntent(pi)
        .build();
    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，接着在 NotificationCompat.Builder 中调用 setContentIntent()方法，把它作为参数传入即可。

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 8.7 所示。

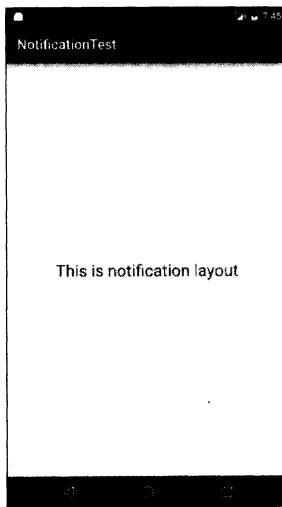


图 8.7 点击通知后打开 NotificationActivity 界面

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。解决的方法有两种，一种是在 NotificationCompat.Builder 中再连缀一个 setAutoCancel()方法，一种是显式地调用 NotificationManager 的 cancel()方法将它取消，两种方法我们都学习一下。

第一种方法写法如下：

```
Notification notification = new NotificationCompat.Builder(this)
    ...
    .setAutoCancel(true)
    .build();
```

可以看到，`setAutoCancel()`方法传入 `true`，就表示当点击了这个通知的时候，通知会自动取消掉。

第二种方法写法如下：

```
public class NotificationActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notification_layout);
        NotificationManager manager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
        manager.cancel(1);
    }
}
```

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，如果你想取消哪条通知，在 `cancel()`方法中传入该通知的 id 就行了。

8.2.2 通知的进阶技巧

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，下面我们就来探究一下通知的更多技巧。

上一小节中创建的通知属于最基本的通知，实际上，`NotificationCompat.Builder` 中提供了非常丰富的 API 来让我们创建出更加多样的通知效果。当然，每一个 API 都详细地讲一遍不太可能，我们只能从中选一些比较常用的 API 来进行学习。先来看看 `setSound()` 方法吧，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。`setSound()` 方法接收一个 `Uri` 参数，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，每个手机的 `/system/media/audio/ringtones` 目录下都有很多的音频文件，我们可以从中随便选一个音频文件，那么在代码中就可以这样指定：

```
Notification notification = new NotificationCompat.Builder(this)
    ...
    .setSound(Uri.fromFile(new File("/system/media/audio/ringtones/Luna.ogg")))
    .build();
```

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：

```
Notification notification = new NotificationCompat.Builder(this)
    ...
    .setVibrate(new long[] {0, 1000, 1000, 1000 })
    .build();
```

不过，想要控制手机振动还需要声明权限。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 `setLights()` 方法来实现这种效果，`setLights()` 方法接收 3 个参数，第一个参数用于指定 LED 灯的颜色，第二个参数用于指定 LED 灯亮起的时长，以毫秒为单位，第三个参数用于指定 LED 灯暗去的时长，也是以毫秒为单位。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 notification = new NotificationCompat.Builder(this)
    ...
    .setLights(Color.GREEN, 1000, 1000)
    .build();
```

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 notification = new NotificationCompat.Builder(this)
    ...
    .setDefaults(NotificationCompat.DEFAULT_ALL)
    .build();
```

注意，以上所涉及的这些进阶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。

8.2.3 通知的高级功能

继续观察 `NotificationCompat.Builder` 这个类，你会发现里面还有很多 API 是我们没有使用过的。那么下面我们就来学习一些更加强大的 API 的用法，从而构建出更加丰富的通知效果。

先来看看 `setStyle()` 方法，这个方法允许我们构建出富文本的通知内容。也就是说通知中不光可以有文字和图标，还可以包含更多的东西。`setStyle()` 方法接收一个 `NotificationCompat.Style` 参数，这个参数就是用来构建具体的富文本信息的，如长文字、图片等。

在开始使用 `setStyle()` 方法之前，我们先来做一个试验吧，之前的通知内容都比较短，如果设置成很长的文字会是什么效果呢？比如这样写：

```
Notification notification = new NotificationCompat.Builder(this)
    ...
    .setContentText("Learn how to build notifications, send and sync data, and use
                    voice actions. Get the official Android IDE and developer tools to build
                    apps for Android.")
    ...
    .build();
```

现在重新运行程序并触发通知，效果如图 8.8 所示。

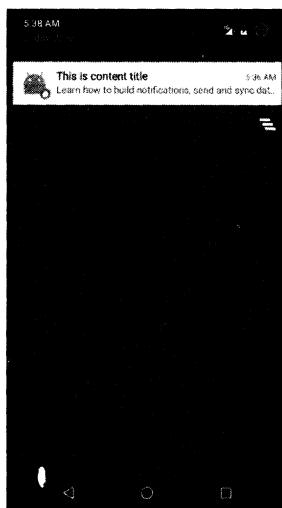


图 8.8 通知内容文字过长的效果

可以看到，通知内容是无法显示完整的，多余的部分会用省略号来代替。其实这也很正常，因为通知的内容本来就应该言简意赅，详细内容放到点击后打开的活动当中会更加合适。

但是如果你真的非常需要在通知当中显示一段长文字，Android 也是支持的，通过 `setStyle()` 方法就可以做到，具体写法如下：

```
Notification notification = new NotificationCompat.Builder(this)
```

```

...
.setStyle(new NotificationCompat.BigTextStyle().bigText("Learn how to build
notifications, send and sync data, and use voice actions. Get the official
Android IDE and developer tools to build apps for Android."))
.build();

```

我们在 `setStyle()` 方法中创建了一个 `NotificationCompat.BigTextStyle` 对象，这个对象就是用于封装长文字信息的，我们调用它的 `bigText()` 方法并将文字内容传入就可以了。

再次重新运行程序并触发通知，效果如图 8.9 所示。

除了显示长文字之外，通知里还可以显示一张大图片，具体用法也是基本相似的：

```

Notification notification = new NotificationCompat.Builder(this)
...
.setStyle(new NotificationCompat.BigPictureStyle().bigPicture
(BitmapFactory.decodeResource(getResources(), R.drawable.big_image)))
.build();

```

可以看到，这里仍然是调用的 `setStyle()` 方法，这次我们在参数中创建了一个 `NotificationCompat.BigPictureStyle` 对象，这个对象就是用于设置大图片的，然后调用它的 `bigPicture()` 方法并将图片传入。这里我事先准备好了一张图片，通过 `BitmapFactory` 的 `decodeResource()` 方法将图片解析成 `Bitmap` 对象，再传入到 `bigPicture()` 方法中就可以了。

现在重新运行一下程序并触发通知，效果如图 8.10 所示。

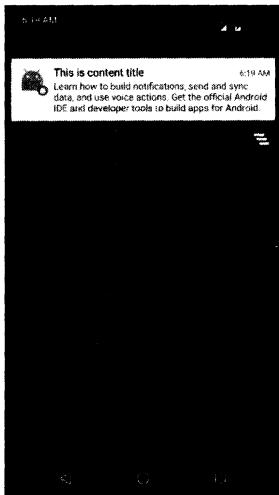


图 8.9 通知中显示长文字的效果

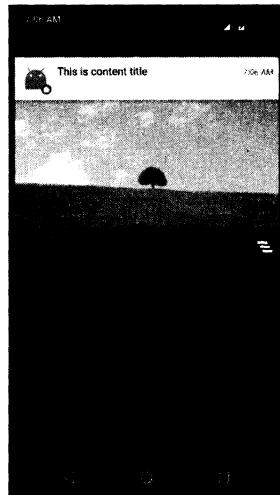


图 8.10 通知中显示大图片的效果

这样我们就把 `setStyle()` 方法中的重要内容基本都掌握了。

接下来再学习一下 `setPriority()` 方法，它可用于设置通知的重要程度。`setPriority()`

方法接收一个整型参数用于设置这条通知的重要程度，一共有 5 个常量值可选：`PRIORITY_DEFAULT` 表示默认的重要程度，和不设置效果是一样的；`PRIORITY_MIN` 表示最低的重要程度，系统可能只会在特定的场景才显示这条通知，比如用户下拉状态栏的时候；`PRIORITY_LOW` 表示较低的重要程度，系统可能会将这类通知缩小，或改变其显示的顺序，将其排在更重要的通知之后；`PRIORITY_HIGH` 表示较高的重要程度，系统可能会将这类通知放大，或改变其显示的顺序，将其排在比较靠前的位置；`PRIORITY_MAX` 表示最高的重要程度，这类通知消息必须要让用户立刻看到，甚至需要用户做出响应操作。具体写法如下：

```
Notification notification = new NotificationCompat.Builder(this)
    ...
    .setPriority(NotificationCompat.PRIORITY_MAX)
    .build();
```

这里我们将通知的重要程度设置成了最高，表示这是一条非常重要的通知，要求用户必须立刻看到。现在重新运行一下程序，并点击 Send notice 按钮，效果如图 8.11 所示。

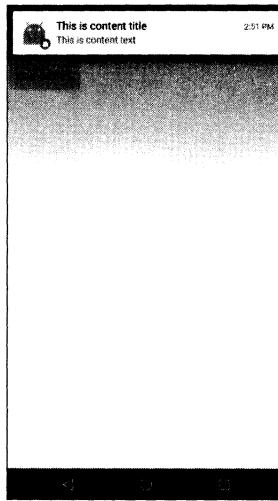


图 8.11 触发一条重要通知

可以看到，这次的通知不是在系统状态栏显示一个小图标了，而是弹出了一个横幅，并附带了通知的详细内容，表示这是一条非常重要的通知。不管用户现在是在玩游戏还是看电影，这条通知都会显示在最上方，以此引起用户的注意。当然，使用这类通知时一定要小心，确保你的通知内容的确是至关重要的，不然如果让用户产生反感的话，很可能会导致我们的应用程序被卸载。

8.3 调用摄像头和相册

我们平时在使用 QQ 或微信的时候经常要和别人分享图片，这些图片可以是用手机摄像头拍

的，也可以是从相册中选取的。类似这样的功能实在是太常见了，几乎在每一个应用程序中都会有，那么本节我们就学习一下调用摄像头和相册方面的知识。

8.3.1 调用摄像头拍照

先来看看摄像头方面的知识，现在很多的应用都会要求用户上传一张图片来作为头像，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 CameraAlbumTest 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <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 AppCompatActivity {

    public static final int TAKE_PHOTO = 1;

    private ImageView picture;

    private Uri imageUri;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button takePhoto = (Button) findViewById(R.id.take_photo);
        picture = (ImageView) findViewById(R.id.picture);
        takePhoto.setOnClickListener(new View.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(getExternalCacheDir(),
        "output_image.jpg");
    try {
        if (outputImage.exists()) {
            outputImage.delete();
        }
        outputImage.createNewFile();
    } catch (IOException e) {
        e.printStackTrace();
    }
    if (Build.VERSION.SDK_INT >= 24) {
        imageUri = FileProvider.getUriForFile(MainActivity.this,
            "com.example.cameraalbumtest.fileprovider", outputImage);
    } else {
        imageUri = Uri.fromFile(outputImage);
    }
    // 启动相机程序
    Intent intent = new Intent("android.media.action.IMAGE_CAPTURE");
    intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
    startActivityForResult(intent, TAKE_PHOTO);
}
});

}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        case TAKE_PHOTO:
            if (resultCode == RESULT_OK) {
                try {
                    // 将拍摄的照片显示出来
                    Bitmap bitmap = BitmapFactory.decodeStream(getContent-
                        Resolver().openInputStream(imageUri));
                    picture.setImageBitmap(bitmap);
                } 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，我们重点看一下这部分代码。

首先这里创建了一个 `File` 对象，用于存放摄像头拍下的图片，这里我们把图片命名为 `output_image.jpg`，并将它存放在手机 SD 卡的应用关联缓存目录下。什么叫作应用关联缓存目录呢？就是指 SD 卡中专门用于存放当前应用缓存数据的位置，调用 `getExternalCacheDir()` 方法可以得到这个目录，具体的路径是 `/sdcard/Android/data/<package name>/cache`。那么为什么要使用应用关联缓存目录来存放图片呢？因为从 Android 6.0 系统开始，读写 SD 卡被列为了危险权限，如果将图片存放在 SD 卡的任何其他目录，都要进行运行时权限处理才行，而使用应用关联目录则可以跳过这一步。

接着会进行一个判断，如果运行设备的系统版本低于 Android 7.0，就调用 `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` 这张图片的本地真实路径。否则，就调用 `FileProvider` 的 `getUriForFile()` 方法将 `File` 对象转换成一个封装过的 `Uri` 对象。`getUriForFile()` 方法接收 3 个参数，第一个参数要求传入 `Context` 对象，第二个参数可以是任意唯一的字符串，第三个参数则是我们刚刚创建的 `File` 对象。之所以要进行这样一层转换，是因为从 Android 7.0 系统开始，直接使用本地真实路径的 `Uri` 被认为是不安全的，会抛出一个 `FileUriExposedException` 异常。而 `FileProvider` 则是一种特殊的内容提供器，它使用了和内容提供器类似的机制来对数据进行保护，可以选择性地将封装过的 `Uri` 共享给外部，从而提高了应用的安全性。

接下来构建出了一个 `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個隱式 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，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。

不过现在还没结束，刚才提到了内容提供器，那么我们自然要在 `AndroidManifest.xml` 中对内容提供器进行注册了，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.cameraalbumtest">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <provider
            android:name="android.support.v4.content.FileProvider"
```

```

        android:authorities="com.example.cameraalbumtest.fileprovider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/file_paths" />
    </provider>
</application>
</manifest>

```

其中，`android:name` 属性的值是固定的，`android:authorities` 属性的值必须要和刚才 `FileProvider.getUriForFile()` 方法中的第二个参数一致。另外，这里还在`<provider>`标签的内部使用`<meta-data>`来指定 Uri 的共享路径，并引用了一个`@xml/file_paths` 资源。当然，这个资源现在还是不存在的，下面我们就来创建它。

右击 `res` 目录→New→Directory，创建一个 `xml` 目录，接着右击 `xml` 目录→New→File，创建一个 `file_paths.xml` 文件。然后修改 `file_paths.xml` 文件中的内容，如下所示：

```

<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-path name="my_images" path="" />
</paths>

```

其中，`external-path` 就是用来指定 Uri 共享的，`name` 属性的值可以随便填，`path` 属性的值表示共享的具体路径。这里设置空值就表示将整个 SD 卡进行共享，当然你也可以仅共享我们存放 `output_image.jpg` 这张图片的路径。

另外还有一点要注意，在 Android 4.4 系统之前，访问 SD 卡的应用关联目录也是要声明权限的，从 4.4 系统开始不再需要权限声明。那么我们为了能够兼容老版本系统的手机，还需要在 `AndroidManifest.xml` 中声明一下访问 SD 卡的权限：

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.cameraalbumtest">
    <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 8.12 所示。拍照完成后，点击中间按钮就会回到我们程序的界面。同时，拍摄的照片也会显示出来了，如图 8.13 所示。



图 8.12 打开摄像头拍照



图 8.13 拍照的最终效果

8.3.2 从相册中选择照片

虽然调用摄像头拍照既方便又快捷，但我们并不是每次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 CameraAlbumTest 项目的基础上进行修改，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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <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 AppCompatActivity {

    ...

    public static final int CHOOSE_PHOTO = 2;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button takePhoto = (Button) findViewById(R.id.take_photo);
        Button chooseFromAlbum = (Button) findViewById(R.id.choose_from_album);
        ...
        chooseFromAlbum.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if (ContextCompat.checkSelfPermission(MainActivity.this,
                        Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.
                        PERMISSION_GRANTED) {
                    ActivityCompat.requestPermissions(MainActivity.this, new
                        String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, 1);
                } else {
                    openAlbum();
                }
            }
        });
    }

    private void openAlbum() {
        Intent intent = new Intent("android.intent.action.GET_CONTENT");
        intent.setType("image/*");
        startActivityForResult(intent, CHOOSE_PHOTO); // 打开相册
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions,
            int[] grantResults) {
        switch (requestCode) {
            case 1:
                if (grantResults.length > 0 && grantResults[0] == PackageManager.
                        PERMISSION_GRANTED) {
                    openAlbum();
                } else {
                    Toast.makeText(this, "You denied the permission",
                            Toast.LENGTH_SHORT).show();
                }
                break;
        }
    }
}
```

```

        default:
    }
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (requestCode) {
        ...
        case CHOOSE_PHOTO:
            if (resultCode == RESULT_OK) {
                // 判断手机系统版本号
                if (Build.VERSION.SDK_INT >= 19) {
                    // 4.4 及以上系统使用这个方法处理图片
                    handleImageOnKitKat(data);
                } else {
                    // 4.4 以下系统使用这个方法处理图片
                    handleImageBeforeKitKat(data);
                }
            }
            break;
        default:
            break;
    }
}

@TargetApi(19)
private void handleImageOnKitKat(Intent data) {
    String imagePath = null;
    Uri uri = data.getData();
    if (DocumentsContract.isDocumentUri(this, uri)) {
        // 如果是 document 类型的 Uri, 则通过 document id 处理
        String docId = DocumentsContract.getDocumentId(uri);
        if("com.android.providers.media.documents".equals(uri.getAuthority())) {
            String id = docId.split(":")[1]; // 解析出数字格式的 id
            String selection = MediaStore.Images.Media._ID + "=" + id;
            imagePath = getImagePath(MediaStore.Images.Media.EXTERNAL_
                CONTENT_URI, selection);
        } else if ("com.android.providers.downloads.documents".equals(uri.
            getAuthority())) {
            Uri contentUri = ContentUris.withAppendedId(Uri.parse("content:
                //downloads/public_downloads"), Long.valueOf(docId));
            imagePath = getImagePath(contentUri, null);
        }
    } else if ("content".equalsIgnoreCase(uri.getScheme())) {
        // 如果是 content 类型的 Uri, 则使用普通方式处理
        imagePath = getImagePath(uri, null);
    } else if ("file".equalsIgnoreCase(uri.getScheme())) {
        // 如果是 file 类型的 Uri, 直接获取图片路径即可
        imagePath = uri.getPath();
    }
    displayImage(imagePath); // 根据图片路径显示图片
}

private void handleImageBeforeKitKat(Intent data) {
    Uri uri = data.getData();
}

```

```

        String imagePath = getImagePath(uri, null);
        displayImage(imagePath);
    }

    private String getImagePath(Uri uri, String selection) {
        String path = null;
        // 通过 Uri 和 selection 来获取真实的图片路径
        Cursor cursor = getContentResolver().query(uri, null, selection, null, null);
        if (cursor != null) {
            if (cursor.moveToFirst()) {
                path = cursor.getString(cursor.getColumnIndex(MediaStore.
                    Images.Media.DATA));
            }
            cursor.close();
        }
        return path;
    }

    private void displayImage(String imagePath) {
        if (imagePath != null) {
            Bitmap bitmap = BitmapFactory.decodeFile(imagePath);
            picture.setImageBitmap(bitmap);
        } else {
            Toast.makeText(this, "failed to get image", Toast.LENGTH_SHORT).show();
        }
    }
}

```

可以看到，在 Choose From Album 按钮的点击事件里我们先是进行了一个运行时权限处理，动态申请 WRITE_EXTERNAL_STORAGE 这个危险权限。为什么需要申请这个权限呢？因为相册中的照片都是存储在 SD 卡上的，我们要从 SD 卡中读取照片就需要申请这个权限。WRITE_EXTERNAL_STORAGE 表示同时授予程序对 SD 卡读写的能力。

当用户授权了权限申请之后会调用 openAlbum()方法，这里我们先是构建出了一个 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，然后调用 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变成了 CHOOSE_PHOTO，这样当从相册选择完图片回到 onActivityResult()方法时，就会进入 CHOOSE_PHOTO 的 case 来处理图片。接下来的逻辑就比较复杂了，首先为了兼容新老版本的手机，我们做了一个判断，如果是 4.4 及以上系统的手机就调用 handleImageOnKitKat()方法来处理图片，否则就调用 handleImageBeforeKitKat()方法来处理图片。之所以要这样做，是因为 Android 系统从 4.4 版本开始，选取相册中的图片不再返回图片真实的 Uri 了，而是一个封装过的 Uri，因此如果是 4.4 版本以上的手机就需要对这个 Uri 进行解析才行。

那么 handleImageOnKitKat()方法中的逻辑就基本是如何解析这个封装过的 Uri 了。这里有好几种判断情况，如果返回的 Uri 是 document 类型的话，那就取出 document id 进行处理，

如果不是的话,那就使用普通的方式处理。另外,如果 Uri 的 authority 是 media 格式的话,document id 还需要再进行一次解析,要通过字符串分割的方式取出后半部分才能得到真正的数字 id。取出的 id 用于构建新的 Uri 和条件语句,然后把这些值作为参数传入到 `getImagePath()` 方法当中,就可以获取到图片的真实路径了。拿到图片的路径之后,再调用 `displayImage()` 方法将图片显示到界面上。

相比于 `handleImageOnKitKat()` 方法, `handleImageBeforeKitKat()` 方法中的逻辑就要简单得多了,因为它的 Uri 是没有封装过的,不需要任何解析,直接将 Uri 传入到 `getImagePath()` 方法当中就能获取到图片的真实路径了,最后同样是调用 `displayImage()` 方法来让图片显示到界面上。

现在将程序重新运行到手机上,然后点击一下 Choose From Album 按钮,首先会弹出权限申请框,如图 8.14 所示。

点击允许之后就会打开手机相册,如图 8.15 所示。

然后随意选择一张照片,回到我们程序的界面,选中的照片应该就会显示出来了,如图 8.16 所示。

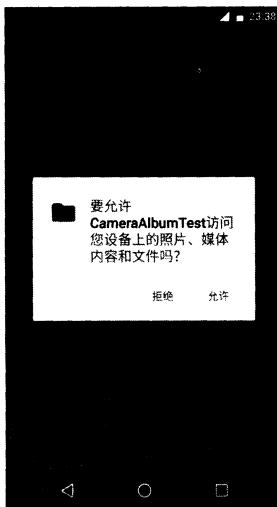


图 8.14 申请访问 SD 卡权限

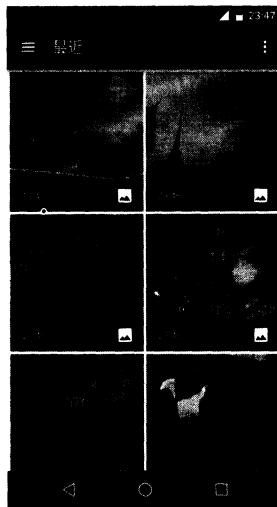


图 8.15 打开手机相册

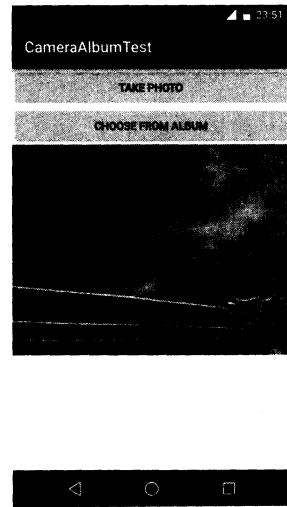


图 8.16 选择照片的最终效果

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的。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了。

8.4 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，越来越多的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。

8.4.1 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
<code>setDataSource()</code>	设置要播放的音频文件的位置
<code>prepare()</code>	在开始播放之前调用这个方法完成准备工作
<code>start()</code>	开始或继续播放音频
<code>pause()</code>	暂停播放音频
<code>reset()</code>	将 <code>MediaPlayer</code> 对象重置到刚刚创建的状态
<code>seekTo()</code>	从指定的位置开始播放音频
<code>stop()</code>	停止播放音频。调用这个方法后的 <code>MediaPlayer</code> 对象无法再播放音频
<code>release()</code>	释放掉与 <code>MediaPlayer</code> 对象相关的资源
<code>isPlaying()</code>	判断当前 <code>MediaPlayer</code> 是否正在播放音频
<code>getDuration()</code>	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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

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

```

```

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

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

</LinearLayout>

```

布局文件中放置了 3 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 AppCompatActivity implements View.OnClickListener{

    private MediaPlayer mediaPlayer = new MediaPlayer();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button play = (Button) findViewById(R.id.play);
        Button pause = (Button) findViewById(R.id.pause);
        Button stop = (Button) findViewById(R.id.stop);
        play.setOnClickListener(this);
        pause.setOnClickListener(this);
        stop.setOnClickListener(this);
        if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(MainActivity.this, new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, 1);
        } else {
            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 onRequestPermissionsResult(int requestCode, String[] permissions,
        int[] grantResults) {

```

```

switch (requestCode) {
    case 1:
        if (grantResults.length > 0 && grantResults[0] == PackageManager.
            PERMISSION_GRANTED) {
            initMediaPlayer();
        } else {
            Toast.makeText(this, "拒绝权限将无法使用程序",
                Toast.LENGTH_SHORT).show();
            finish();
        }
        break;
    default:
}
}

@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先创建了一个 MediaPlayer 的实例，然后在 onCreate() 方法中进行了运行时权限处理，动态申请 WRITE_EXTERNAL_STORAGE 权限。这是由于待会我们会在 SD 卡中放置一个音频文件，程序为了播放这个音频文件必须拥有访问 SD 卡的权限才行。

注意，在 `onRequestPermissionsResult()` 方法中，如果用户拒绝了权限申请，那么就调用 `finish()` 方法将程序直接关掉，因为如果没有 SD 卡的访问权限，我们这个程序将什么都干不了。

用户同意授权之后就会调用 `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()` 方法和 `release()` 方法，将与 `MediaPlayer` 相关的资源释放掉。

另外，千万不要忘记在 `AndroidManifest.xml` 文件中声明用到的权限，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.playaudiotest">
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

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会先弹出权限申请框，如图 8.17 所示。



图 8.17 音乐播放器主界面

同意授权之后就可以开始播放音乐了，点击一下 Play 按钮，优美的音乐就会响起，然后点击 Pause 按钮，音乐就会停住，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，音乐也会停住，但是当再次点击 Play 按钮时，音乐就会从头开始播放了。

8.4.2 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <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" />
    

```

```

        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Replay" />

    </LinearLayout>

    <VideoView
        android:id="@+id/video_view"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</LinearLayout>

```

在这个布局文件中，首先放置了3个按钮，分别用于控制视频的播放、暂停和重新播放。然后在按钮下面又放置了一个VideoView，稍后的视频就将在这里显示。

接下来修改 MainActivity 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity implements View.OnClickListener{

    private VideoView videoView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        videoView = (VideoView) findViewById(R.id.video_view);
        Button play = (Button) findViewById(R.id.play);
        Button pause = (Button) findViewById(R.id.pause);
        Button replay = (Button) findViewById(R.id.replay);
        play.setOnClickListener(this);
        pause.setOnClickListener(this);
        replay.setOnClickListener(this);
        if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.
            permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(MainActivity.this, new String[]{
                Manifest.permission.WRITE_EXTERNAL_STORAGE }, 1);
        } else {
            initVideoPath(); // 初始化 MediaPlayer
        }
    }

    private void initVideoPath() {
        File file = new File(Environment.getExternalStorageDirectory(), "movie.mp4");
        videoView.setVideoPath(file.getPath()); // 指定视频文件的路径
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, String[] permissions,
        int[] grantResults) {
        switch (requestCode) {
            case 1:
                if (grantResults.length > 0 && grantResults[0] == PackageManager.
                    PERMISSION_GRANTED) {

```

```

        initVideoPath();
    } else {
        Toast.makeText(this, "拒绝权限将无法使用程序", Toast.LENGTH_SHORT).
            show();
        finish();
    }
    break;
default:
}
}

@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(); // 暂停播放
            }
            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()` 方法中同样进行了一个运行时权限处理，因为视频文件将会放在 SD 卡上。当用户同意授权了之后就会调用 `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.mp4` 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视频，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()` 方法暂停播放。当点击 Replay 按钮时会判断，如果当前视频正在播放，则调用 `resume()` 方法从头播放视频。

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。

另外，仍然始终要记得在 `AndroidManifest.xml` 文件中声明用到的权限，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.playvideotest">
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    ...
</manifest>
```

现在将程序运行到手机上，会先弹出一个权限申请对话框，同意授权之后点击一下 `Play` 按钮，就可以看到视频已经开始播放了，如图 8.18 所示。



图 8.18 `VideoView` 播放视频的效果

点击 `Pause` 按钮可以暂停视频的播放，点击 `Replay` 按钮可以从头播放视频。

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。

8.5 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。

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，因为后面的学习当中还会用到它。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网络交互的部分，所以下一章中我们将会学习一下 Android 网络编程方面的内容。

第 9 章

看看精彩的世界——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地枯燥乏味。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，在可预见的未来，手表、眼镜、汽车等设备也会逐个加入到这个行列，21 世纪的确是互联网的时代。

当然，Android 手机肯定也是可以上网的，所以作为开发者，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、微信等常见的应用都会大量使用网络技术。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，下面就让我们一起来学习一下吧。

9.1 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 AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        WebView webView = (WebView) findViewById(R.id.web_view);  
        webView.getSettings().setJavaScriptEnabled(true);  
        webView.setWebViewClient(new WebViewClient());  
        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，并传入了一个 `WebViewClient` 的实例。这段代码的作用是，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长什么样吧。

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">  
  
    <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 9.1 所示。

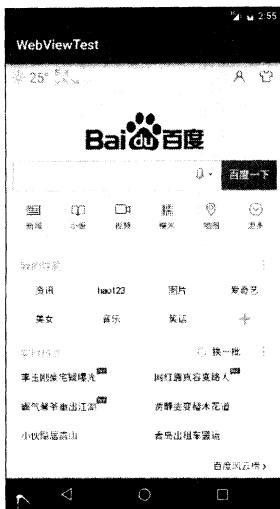


图 9.1 WebView 加载网页

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。

9.2 使用 HTTP 协议访问网络

如果说真的要去深入分析 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。

9.2.1 使用 HttpURLConnection

在过去，Android 上发送 HTTP 请求一般有两种方式：HttpURLConnection 和 HttpClient。不过由于 HttpClient 存在 API 数量过多、扩展困难等缺点，Android 团队越来越不建议我们使用这种方式。终于在 Android 6.0 系统中，HttpClient 的功能被完全移除了，标志着此功能被正式弃用，因此本小节我们就学习一下现在官方建议使用的 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();
```

在得到了 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <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_text"
        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，我们就可以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 AppCompatActivity implements View.OnClickListener {

    TextView responseText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button 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;
                BufferedReader reader = 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
        reader = new BufferedReader(new InputStreamReader(in));
        StringBuilder response = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        showResponse(response.toString());
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (connection != null) {
            connection.disconnect();
        }
    }
}
}).start();
}

private void showResponse(final String response) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            // 在这里进行 UI 操作，将结果显示到界面上
            responseText.setText(response);
        }
    });
}
}

```

可以看到，我们在 Send Request 按钮的点击事件里调用了 `sendRequestWithHttpURLConnection()` 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，并将结果传入到了 `showResponse()` 方法中。而在 `showResponse()` 方法里则是调用了一个 `runOnUiThread()` 方法，然后在这个方法的匿名类参数中进行操作，将返回的数据显示到界面上。那么这里为什么要用这个 `runOnUiThread()` 方法呢？这是因为 Android 是不允许在子线程中进行 UI 操作的，我们需要通过这个方法将线程切换到主线程，然后再更新 UI 元素。关于这部分内容，我们将会在下一章中进行详细讲解，现在你只需要记得必须这么写就可以了。

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">

    <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 9.2 所示。



图 9.2 服务器响应的数据

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。

9.2.2 使用 OkHttp

当然我们并不是只能使用 HttpURLConnection，完全没有任何其他选择，事实上在开源盛行的今天，有许多出色的网络通信库都可以替代原生的 HttpURLConnection，而其中 OkHttp 无疑是做得最出色的一个。

OkHttp 是由鼎鼎大名的 Square 公司开发的，这个公司在开源事业上面贡献良多，除了 OkHttp 之外，还开发了像 Picasso、Retrofit 等著名的开源项目。OkHttp 不仅在接口封装上面做得简单易用，就连在底层实现上也是自成一派，比起原生的 HttpURLConnection，可以说是有过之而无不及，现在已经成了广大 Android 开发者首选的网络通信库。那么本小节我们就来学习一下 OkHttp 的用法，OkHttp 的项目主页地址是：<https://github.com/square/okhttp>。

在使用 OkHttp 之前，我们需要先在项目中添加 OkHttp 库的依赖。编辑 app/build.gradle 文件，在 dependencies 闭包中添加如下内容：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
    compile 'com.squareup.okhttp3:okhttp:3.4.1'
}
```

添加上述依赖会自动下载两个库，一个是 OkHttp 库，一个是 Okio 库，后者是前者的通信基础。其中 3.4.1 是我写本书时 OkHttp 的最新版本，你可以访问 OkHttp 的项目主页来查看当前最新的版本是多少。

下面我们来看一下 OkHttp 的具体用法，首先需要创建一个 OkHttpClient 的实例，如下所示：

```
OkHttpClient client = new OkHttpClient();
```

接下来如果想要发起一条 HTTP 请求，就需要创建一个 Request 对象：

```
Request request = new Request.Builder().build();
```

当然，上述代码只是创建了一个空的 Request 对象，并没有什么实际作用，我们可以在最终的 build() 方法之前连缀很多其他方法来丰富这个 Request 对象。比如可以通过 url() 方法来设置目标的网络地址，如下所示：

```
Request request = new Request.Builder()
    .url("http://www.baidu.com")
    .build();
```

之后调用 OkHttpClient 的 newCall() 方法来创建一个 Call 对象，并调用它的 execute() 方法来发送请求并获取服务器返回的数据，写法如下：

```
Response response = client.newCall(request).execute();
```

其中 `Response` 对象就是服务器返回的数据了,我们可以使用如下写法来得到返回的具体内容:

```
String responseData = response.body().string();
```

如果是发起一条 POST 请求会比 GET 请求稍微复杂一点,我们需要先构建出一个 `RequestBody` 对象来存放待提交的参数,如下所示:

```
RequestBody requestBody = new FormBody.Builder()
    .add("username", "admin")
    .add("password", "123456")
    .build();
```

然后在 `Request.Builder` 中调用一下 `post()` 方法,并将 `RequestBody` 对象传入:

```
Request request = new Request.Builder()
    .url("http://www.baidu.com")
    .post(requestBody)
    .build();
```

接下来的操作就和 GET 请求一样了,调用 `execute()` 方法来发送请求并获取服务器返回的数据即可。

好了,OkHttp 的基本用法就先学到这里,本书中后面所有网络相关的功能我们都将会使用 OkHttp 来实现,到时候再进行进一步的学习。那么现在我们先把 NetworkTest 这个项目改用 OkHttp 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 AppCompatActivity implements View.OnClickListener {

    ...

    @Override
    public void onClick(View v) {
        if (v.getId() == R.id.send_request) {
            sendRequestWithOkHttp();
        }
    }

    private void sendRequestWithOkHttp() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    OkHttpClient client = new OkHttpClient();
                    Request request = new Request.Builder()
                        .url("http://www.baidu.com")
                        .build();
                    Response response = client.newCall(request).execute();
                    String responseData = response.body().string();
                    showResponse(responseData);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

```

        }
    }).start();
}

...
}

```

这里我们并没有做太多的改动，只是添加了一个 `sendRequestWithOkHttp()` 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 `OkHttp` 发出一条 HTTP 请求，请求的目标地址还是百度的首页，`OkHttp` 的用法也正如前面所介绍的一样。最后仍然还是调用了 `showResponse()` 方法来将服务器返回的数据显示到界面上。

仅仅是改了这么多代码，现在我们就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 `OkHttp` 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 `OkHttp` 的基本用法都掌握得差不多了。

9.3 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，下面我们就来一个一个地进行学习，本节首先学习一下如何解析 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，官方下载地址是：<http://httpd.apache.org/download.cgi>。如果你在这个网址中找不到 Windows 版的安装包，也可以直接在百度上搜索“Apache 服务器下载”，将会找到很多下载链接。

下载完成后双击就可以进行安装了，如图 9.3 所示。

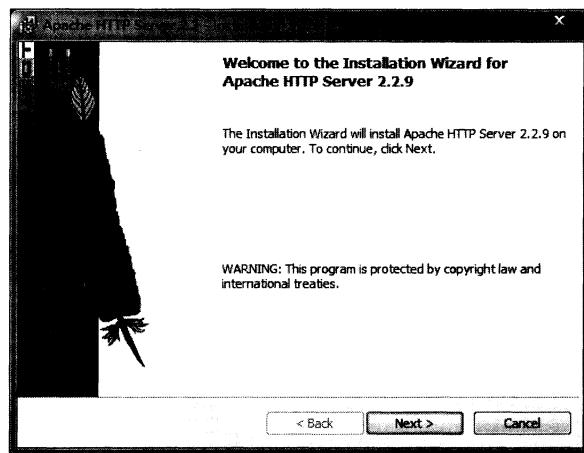


图 9.3 Apache 服务器安装界面

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 9.4 所示。

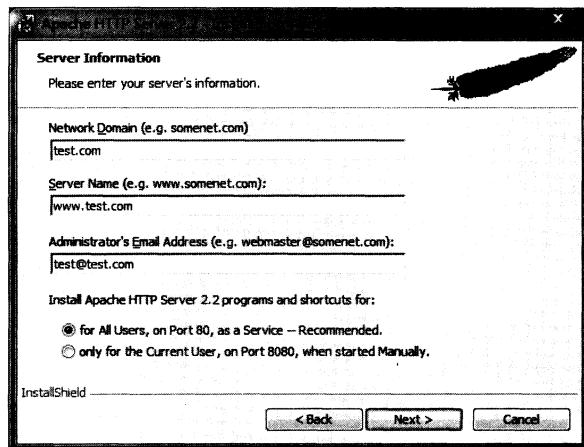


图 9.4 填入域名和服务器信息

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 9.5 所示的界面，就说明服务器已经启动成功了。



图 9.5 Apache 服务器的默认主页

接下来进入到 C:\Apache\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 9.6 所示的内容。

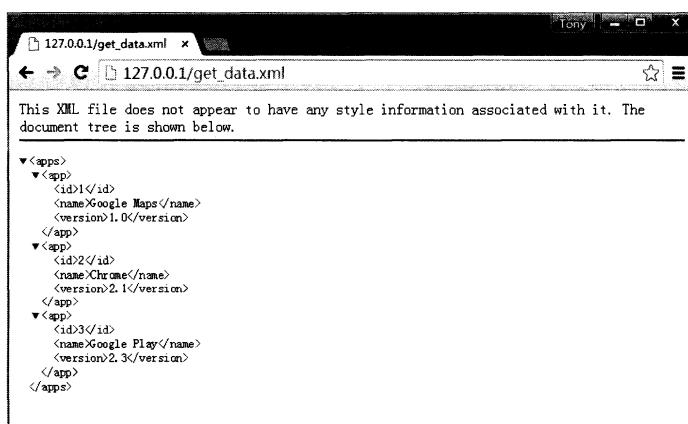


图 9.6 在浏览器验证 XML 数据

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。

9.3.1 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 AppCompatActivity implements View.OnClickListener {  
    ...  
  
    private void sendRequestWithOkHttp() {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                try {  
                    OkHttpClient client = new OkHttpClient();  
                    Request request = new Request.Builder()  
                        // 指定访问的服务器地址是电脑本机  
                        .url("http://10.0.2.2/get_data.xml")  
                        .build();  
                    Response response = client.newCall(request).execute();  
                    String responseData = response.body().string();  
                    parseXMLWithPull(responseData);  
                } 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) {  
                    // 开始解析某个节点  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

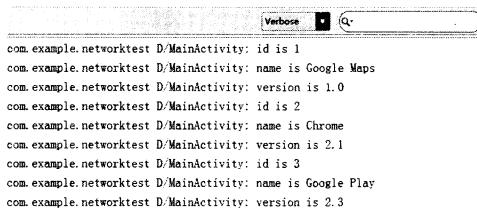
```
        case XmlPullParser.START_TAG: {
            if ("id".equals(nodeName)) {
                id = xmlPullParser.nextText();
            } else if ("name".equals(nodeName)) {
                name = xmlPullParser.nextText();
            } else if ("version".equals(nodeName)) {
                version = xmlPullParser.nextText();
            }
            break;
        }
        // 完成解析某个节点
        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://10.0.2.2/get_data.xml`，`10.0.2.2` 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，我们并不再直接将其展示，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()方法得到当前节点的名字，如果发现节点名等于 id、name 或 version，就调用 nextText()方法来获取节点内具体的内容，每当解析完一个 app 节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 9.7 所示。



```

Verbose
com.example.networktest D/MainActivity: id is 1
com.example.networktest D/MainActivity: name is Google Maps
com.example.networktest D/MainActivity: version is 1.0
com.example.networktest D/MainActivity: id is 2
com.example.networktest D/MainActivity: name is Chrome
com.example.networktest D/MainActivity: version is 2.1
com.example.networktest D/MainActivity: id is 3
com.example.networktest D/MainActivity: name is Google Play
com.example.networktest D/MainActivity: version is 2.3

```

图 9.7 打印从 XML 中解析出的数据

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。

9.3.2 SAX 解析方式

Pull 解析方式虽然非常好用，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 5 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 5 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节点的时候调用，`characters()`方法会在获取节点中内容的时候调用，`endElement()`方法会在完成解析某个节点的时候调用，`endDocument()`方法会在完成整个 XML 解析的时候调用。其中，`startElement()`、`characters()`和 `endElement()`

这 3 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，在获取节点中的内容时，`characters()`方法可能被调用多次，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 5 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 {  
        // 记录当前节点名  
        nodeName = localName;  
    }  
  
    @Override  
    public void characters(char[] ch, int start, int length) throws SAXException {  
        // 根据当前的节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 {
    super.endDocument();
}
}

```

可以看到，我们首先给 `id`、`name` 和 `version` 节点分别定义了一个 `StringBuilder` 对象，并在 `startDocument()` 方法里对它们进行了初始化。每当开始解析某个节点的时候，`startElement()` 方法就会得到调用，其中 `localName` 参数记录着当前节点的名字，这里我们把它记录下来。接着在解析节点中具体内容的时候就会调用 `characters()` 方法，我们会根据当前的节点名进行判断，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` 节点已经解析完成，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 AppCompatActivity implements View.OnClickListener {

    ...
    private void sendRequestWithOkHttp() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    OkHttpClient client = new OkHttpClient();
                    Request request = new Request.Builder()
                        // 指定访问的服务器地址是电脑本机
                        .url("http://10.0.2.2/get_data.xml")
                        .build();
                    Response response = client.newCall(request).execute();
                    String responseData = response.body().string();
                    parseXMLWithSAX(responseData);
                } 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 9.7 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。

9.4 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\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": "Clash of Clans"},  
 {"id": "6", "version": "7.0", "name": "Boom Beach"},  
 {"id": "7", "version": "3.5", "name": "Clash Royale"}]
```

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 9.8 所示的内容。

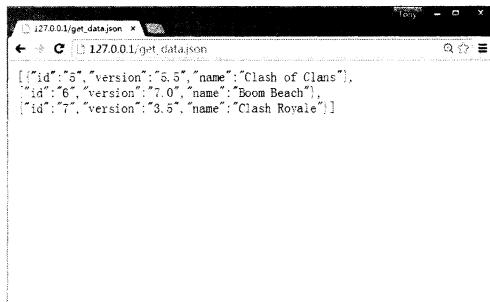


图 9.8 在浏览器验证 JSON 数据

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。

9.4.1 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 AppCompatActivity implements View.OnClickListener {
    ...
    private void sendRequestWithOkHttp() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    OkHttpClient client = new OkHttpClient();
                    Request request = new Request.Builder()
                        // 指定访问的服务器地址是电脑本机
                        .url("http://10.0.2.2/get_data.json")
                        .build();
                    Response response = client.newCall(request).execute();
                    String responseData = response.body().string();
                    parseJSONWithJSONObject(responseData);
                } 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://10.0.2.2/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 9.9 所示。

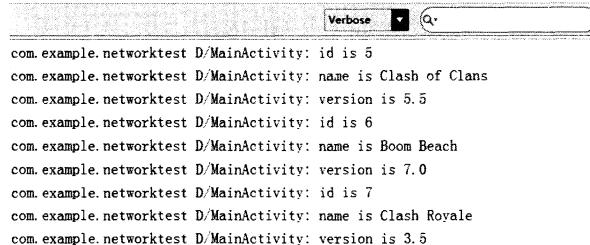


图 9.9 打印从 JSON 中解析出的数据

9.4.2 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，就必

须要在项目中添加 Gson 库的依赖。编辑 app/build.gradle 文件，在 dependencies 闭包中添加如下内容：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:24.2.1'
    compile 'com.squareup.okhttp3:okhttp:3.4.1'
    compile 'com.google.code.gson:gson:2.7'
}
```

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 3 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 AppCompatActivity implements View.OnClickListener {

    ...

    private void sendRequestWithOkHttp() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    OkHttpClient client = new OkHttpClient();
                    Request request = new Request.Builder()
                        // 指定访问的服务器地址是电脑本机
                        .url("http://10.0.2.2/get_data.json")
                        .build();
                    Response response = client.newCall(request).execute();
                    String responseData = response.body().string();
                    parseJSONWithGSON(responseData);
                } 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 9.9 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。

9.5 网络编程的最佳实践

目前你已经掌握了 HttpURLConnection 和 OkHttp 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？其实没有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()` 方法会在服务器还没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。

那么遇到这种情况时应该怎么办呢？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，比如将它命名为 `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));
                    String line;
                    while ((line = reader.readLine()) != null) {
                        listener.onFinish(line);
                    }
                } catch (IOException e) {
                    listener.onError(e);
                }
            }
        }).start();
    }
}
```

```

        (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。

不过你会发现，上述使用 `HttpURLConnection` 的写法总体来说还是比较复杂的，那么使用

OkHttp 会变得简单吗？答案是肯定的，而且要简单得多，下面我们就具体看一下。在 HttpUtil 中加入一个 sendOkHttpRequest()方法，如下所示：

```
public class HttpUtil {
    ...
    public static void sendOkHttpRequest(String address, okhttp3.Callback callback) {
        OkHttpClient client = new OkHttpClient();
        Request request = new Request.Builder()
            .url(address)
            .build();
        client.newCall(request).enqueue(callback);
    }
}
```

可以看到，sendOkHttpRequest()方法中有一个 okhttp3.Callback 参数，这个是 OkHttp 库中自带的一个回调接口，类似于我们刚才自己编写的 HttpCallbackListener。然后在 client.newCall()之后没有像之前那样一直调用 execute()方法，而是调用了一个 enqueue()方法，并把 okhttp3.Callback 参数传入。相信聪明的你已经猜到了，OkHttp 在 enqueue()方法的内部已经帮我们开好子线程了，然后会在子线程中去执行 HTTP 请求，并将最终的请求结果回调到 okhttp3.Callback 当中。

那么我们在调用 sendOkHttpRequest()方法的时候就可以这样写：

```
HttpUtil.sendOkHttpRequest("http://www.baidu.com", new okhttp3.Callback() {
    @Override
    public void onResponse(Call call, Response response) throws IOException {
        // 得到服务器返回的具体内容
        String responseData = response.body().string();
    }

    @Override
    public void onFailure(Call call, IOException e) {
        // 在这里对异常情况进行处理
    }
});
```

由此可以看出，OkHttp 的接口设计得确实非常人性化，它将一些常用的功能进行了很好的封装，使得我们只需编写少量的代码就能完成较为复杂的网络操作。当然这并不是 OkHttp 的全部，后面我们还会继续学习它的其他相关知识。

另外需要注意的是，不管是使用 HttpURLConnection 还是 OkHttp，最终的回调接口都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，除非借助 runOnUiThread()方法来进行线程转换。至于具体的原因，我们很快就会在下一章中学习到了。

9.6 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 OkHttp，相信这两种方式你都已经很好地掌握了。

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。

在进行了一章多媒体和一章网络的相关知识学习后，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 10 章

后台默默的劳动者——探究服务

记得在我上大学的时候，iPhone 是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 早已风光不再，Android 和 iOS 占据了大部分的智能市场份额，Windows Phone 也占据了一小部分，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 和 Windows Phone 一开始都是不支持后台的，后来逐渐意识到这个功能的重要性，才加入了后台功能。而 Android 则是沿用了 Symbian 的老习惯，从一开始就支持后台功能，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。不管怎么说，后台功能属于四大组件之一，其重要程度不言而喻，那么我们自然要好好学习一下它的用法了。

10.1 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。

10.2 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。

10.2.1 线程的基本用法

Android 多线程编程其实并不比 Java 多线程编程特殊，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。

10.2.2 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 AppCompatActivity implements View.OnClickListener {  
  
    private TextView text;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        text = (TextView) findViewById(R.id.text);  
        Button 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 10.1 所示。



图 10.1 在子线程中更新 UI 导致崩溃

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 10.2 所示。

```
android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original
thread that created a view hierarchy can touch its views.
```

图 10.2 崩溃的详细信息

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 AppCompatActivity implements View.OnClickListener {

    public static final int UPDATE_TEXT = 1;

    private TextView text;

    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
            }
        }
    }

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

    @Override
    public void onClick(View v) {
        handler.sendMessage(handler.obtainMessage(UPDATE_TEXT));
    }
}
```

```

        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，显示的内容就被替换成 `Nice to meet you`，如图 10.3 所示。

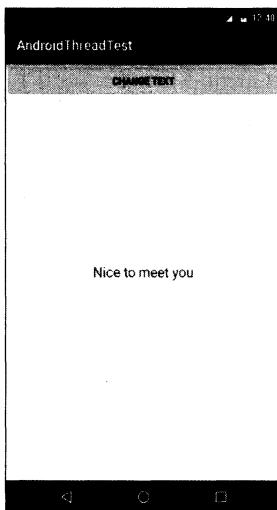


图 10.3 成功替换显示的文字

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。

10.2.3 解析异步消息处理机制

Android 中的异步消息处理主要由 4 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 4 个部分进行一下简要的介绍。

1. 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。

2. 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。

3. 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。

4. 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，我们再来把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 10.4 所示。

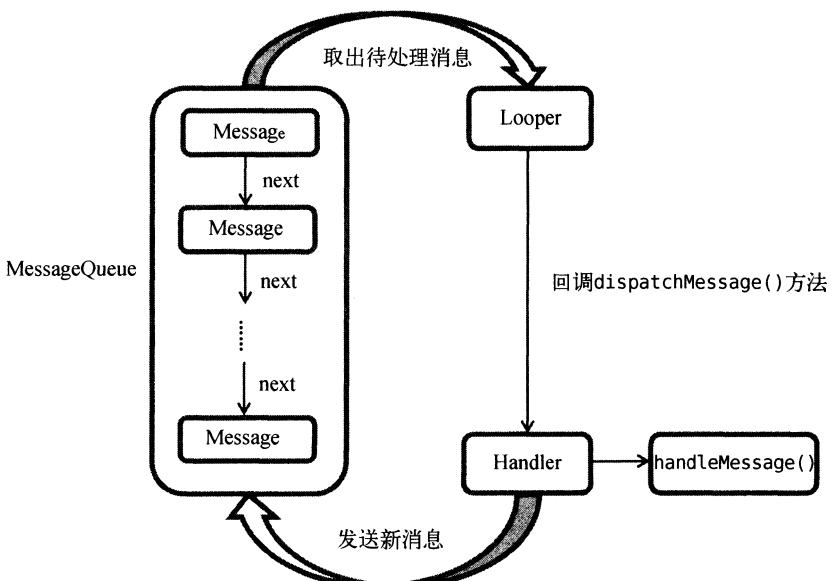


图 10.4 异步消息处理机制流程示意图

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。

而我们在 9.2.1 小节中使用到的 `runOnUiThread()` 方法其实就是一个异步消息处理机制的接口封装，它虽然表面上看起来用法更为简单，但其实背后的实现原理和图 10.4 中的描述是一模一样的。

10.2.4 使用 AsyncTask

不过为了更加方便我们在子线程中对 UI 进行操作，Android 还提供了另外一些好用的工具，比如 `AsyncTask`。借助 `AsyncTask`，即使你对异步消息处理机制完全不了解，也可以十分简单地从子线程切换到主线程。当然，`AsyncTask` 背后的实现原理也是基于异步消息处理机制的，只是 Android 帮我们做了很好的封装而已。

首先来看一下 `AsyncTask` 的基本用法，由于 `AsyncTask` 是一个抽象类，所以如果我们想使用它，就必须要创建一个子类去继承它。在继承时我们可以为 `AsyncTask` 类指定 3 个泛型参数，这 3 个参数的用途如下。

- `Params`。在执行 `AsyncTask` 时需要传入的参数，可用于在后台任务中使用。
- `Progress`。后台任务执行时，如果需要在界面上显示当前的进度，则使用这里指定的泛型作为进度单位。
- `Result`。当任务执行完毕后，如果需要对结果进行返回，则使用这里指定的泛型作为返回值类型。

因此，一个最简单的自定义 `AsyncTask` 就可以写成如下方式：

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {  
    ...  
}
```

这里我们把 `AsyncTask` 的第一个泛型参数指定为 `Void`，表示在执行 `AsyncTask` 的时候不需要传入参数给后台任务。第二个泛型参数指定为 `Integer`，表示使用整型数据来作为进度显示单位。第三个泛型参数指定为 `Boolean`，则表示使用布尔型数据来反馈执行结果。

当然，目前我们自定义的 `DownloadTask` 还是一个空任务，并不能进行任何实际的操作，我们还需要去重写 `AsyncTask` 中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下 4 个。

1. `onPreExecute()`

这个方法会在后台任务开始执行之前调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

2. `doInBackground(Params...)`

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。任务一旦完成就可以通过 `return` 语句来将任务的执行结果返回，如果 `AsyncTask` 的第三个泛型参数指定的是 `Void`，就可以不返回任务执行结果。注意，在这个方法中是不可以进行 UI 操作的，如果需要更新 UI 元素，比如说反馈当前任务的执行进度，可以调用 `publishProgress(Progress...)` 方法来完成。

3. onProgressUpdate(Progress...)

当在后台任务中调用了 `publishProgress(Progress...)` 方法后，`onProgressUpdate(Progress...)` 方法就会很快被调用，该方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对 UI 进行操作，利用参数中的数值就可以对界面元素进行相应的更新。

4. onPostExecute(Result)

当后台任务执行完毕并通过 `return` 语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些 UI 操作，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

因此，一个比较完整的自定义 `AsyncTask` 就可以写成如下方式：

```
class DownloadTask extends AsyncTask<Void, Integer, Boolean> {

    @Override
    protected void onPreExecute() {
        progressDialog.show(); // 显示进度对话框
    }

    @Override
    protected Boolean doInBackground(Void... params) {
        try {
            while (true) {
                int downloadPercent = doDownload(); // 这是一个虚构的方法
                publishProgress(downloadPercent);
                if (downloadPercent >= 100) {
                    break;
                }
            }
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(Integer... values) {
        // 在这里更新下载进度
        progressDialog.setMessage("Downloaded " + values[0] + "%");
    }

    @Override
    protected void onPostExecute(Boolean result) {
        progressDialog.dismiss(); // 关闭进度对话框
        // 在这里提示下载结果
        if (result) {
            Toast.makeText(context, "Download succeeded", Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(context, "Download failed", Toast.LENGTH_SHORT).show();
        }
    }
}
```

```
    }  
}
```

在这个 DownloadTask 中，我们在 `doInBackground()` 方法里去执行具体的下载任务。这个方法里的代码都是在子线程中运行的，因而不会影响到主线程的运行。注意这里虚构了一个 `doDownload()` 方法，这个方法用于计算当前的下载进度并返回，我们假设这个方法已经存在了。在得到了当前的下载进度后，下面就该考虑如何把它显示到界面上了，由于 `doInBackground()` 方法是在子线程中运行的，在这里肯定不能进行 UI 操作，所以我们可以调用 `publishProgress()` 方法并将当前的下载进度传进来，这样 `onProgressUpdate()` 方法就会很快被调用，在这里就可以进行 UI 操作了。

当下载完成后，`doInBackground()` 方法会返回一个布尔型变量，这样 `onPostExecute()` 方法就会很快被调用，这个方法也是在主线程中运行的。然后在这里我们会根据下载的结果来弹出相应的 Toast 提示，从而完成整个 DownloadTask 任务。

简单来说，使用 `AsyncTask` 的诀窍就是，在 `doInBackground()` 方法中执行具体的耗时任务，在 `onProgressUpdate()` 方法中进行 UI 操作，在 `onPostExecute()` 方法中执行一些任务的收尾工作。

如果想要启动这个任务，只需编写以下代码即可：

```
new DownloadTask().execute();
```

以上就是 `AsyncTask` 的基本用法，怎么样，是不是感觉简单方便了许多？我们并不需要去考虑什么异步消息处理机制，也不需要专门使用一个 `Handler` 来发送和接收消息，只需要调用一下 `publishProgress()` 方法，就可以轻松地从子线程切换到 UI 线程了。

在本章的最佳实践环节，我们会对下载这个功能进行完整的实现。

10.3 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。

10.3.1 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，然后右击 `com.example.servicetest` → `New` → `Service` → `Service`，会弹出如图 10.5 所示的窗口。

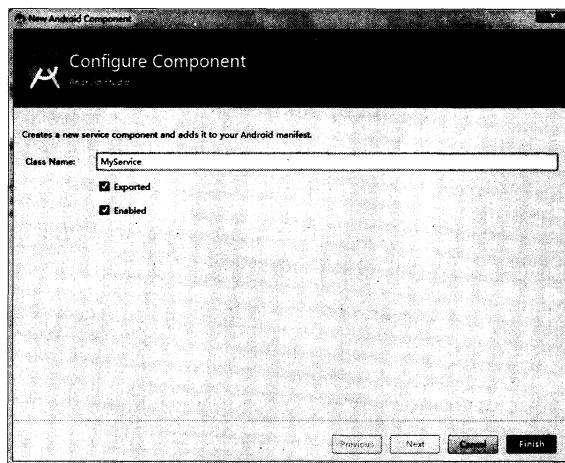


图 10.5 创建服务的窗口

可以看到，这里我们将服务命名为 MyService，**Exported** 属性表示是否允许除了当前程序之外的其他程序访问这个服务，**Enabled** 属性表示是否启用这个服务。将两个属性都勾中，点击 Finish 完成创建。

现在观察 MyService 中的代码，如下所示：

```
public class MyService extends Service {

    public MyService() {
    }

    @Override
    public IBinder onBind(Intent intent) {
        throw new UnsupportedOperationException("Not yet implemented");
    }
}
```

可以看到，MyService 是继承自 Service 类的，说明这是一个服务。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 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 3 个方法，它们是每个服务中最常用到的 3 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。不过相信你已经猜到了，智能的 Android Studio 早已自动帮我们将这一步完成了。打开 `AndroidManifest.xml` 文件瞧一瞧，代码如下所示：

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicetest">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <service
            android:name=".MyService"
            android:enabled="true"
            android:exported="true">
        </service>
    </application>

</manifest>

```

这样的话，就已经将一个服务完全定义好了。

10.3.2 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <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 AppCompatActivity implements View.OnClickListener {

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

```

        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 Service 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 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 10.6 所示。

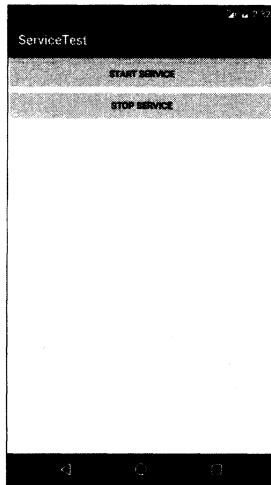


图 10.6 ServiceTest 的主界面

点击一下 Start Service 按钮，观察 logcat 中的打印日志，如图 10.7 所示。

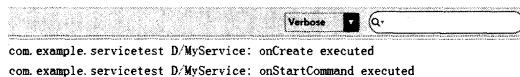


图 10.7 启动服务时的打印日志

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 `Settings→Developer options→Running services` 中找到它，如图 10.8 所示。

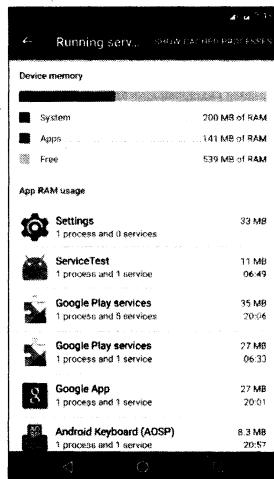


图 10.8 正在运行的服务列表

然后再点击一下 Stop Service 按钮，观察 logcat 中的打印日志，如图 10.9 所示。

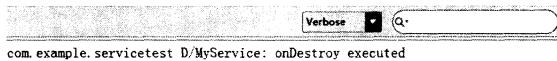


图 10.9 停止服务时的打印日志

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()` 方法到底有什么区别呢？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。

10.3.3 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做了什么事情，以及完成得如何。

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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    ...

    <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 AppCompatActivity implements View.OnClickListener {

    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);
    ...

    Button bindService = (Button) findViewById(R.id.bind_service);
    Button 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 3 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 10.10 所示。

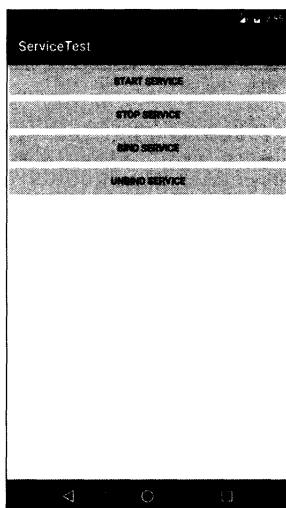


图 10.10 ServiceTest 新的主界面

点击一下 Bind Service 按钮，然后观察 logcat 中的打印日志，如图 10.11 所示。

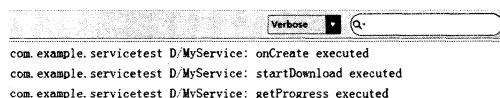


图 10.11 绑定服务时的打印日志

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。

10.4 服务的生命周期

之前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。

10.5 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，关于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。

10.5.1 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，比如说彩云天气这款天气预报应用，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 10.12 所示。

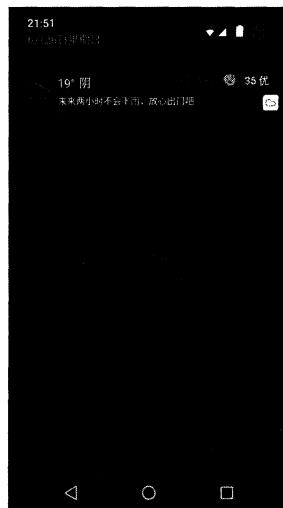


图 10.12 彩云天气的前台服务效果

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();  
        Log.d("MyService", "onCreate executed");  
        Intent intent = new Intent(this, MainActivity.class);  
        PendingIntent pi = PendingIntent.getActivity(this, 0, intent, 0);  
        Notification notification = new NotificationCompat.Builder(this)  
            .setContentTitle("This is content title")  
            .setContentText("This is content text")  
            .setWhen(System.currentTimeMillis())  
            .setSmallIcon(R.mipmap.ic_launcher)  
            .setLargeIcon(BitmapFactory.decodeResource(getResources(),  
                R.mipmap.ic_launcher))  
            .setContentIntent(pi)  
            .build();  
        startForeground(1, notification);  
    }  
}
```

```
    }  
    ...  
}
```

可以看到，这里只是修改了 `onCreate()` 方法中的代码，相信这部分代码你会非常眼熟。没错！这就是我们在第 8 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 10.13 所示。

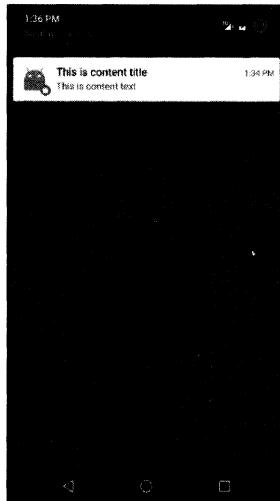


图 10.13 前台服务的状态栏效果

前台服务的用法就这么简单，只要你在第 8 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。

10.5.2 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 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 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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    ...

    <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 AppCompatActivity implements View.OnClickListener {

    ...

```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ...
    Button 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">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        ...
        <service android:name=".MyIntentService" />

    </application>

</manifest>
```

当然你也可以使用 Android Studio 提供的快捷方式来创建 IntentService，不过由于这样会自动生成一些我们用不到的代码，因此这里我采用了手动创建的方式。

现在重新运行一下程序，界面如图 10.14 所示。

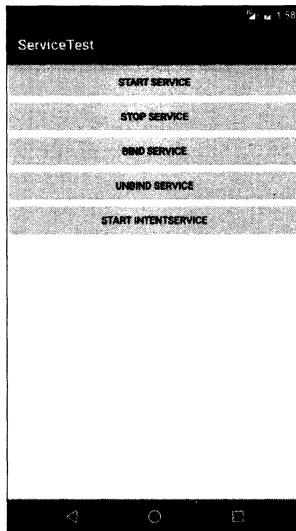


图 10.14 ServiceTest 更新后的主界面

点击 Start IntentService 按钮后，观察 logcat 中的打印日志，如图 10.15 所示。

```
Verbose
/com.example.servicetest D/MainActivity: Thread id is 1
/com.example.servicetest D/MyIntentService: Thread id is 154
/com.example.servicetest D/MyIntentService: onDestroy executed
```

图 10.15 启动 IntentService 时的打印日志

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。

10.6 服务的最佳实践——完整版的下载示例

本章中你已经掌握了很多关于服务的使用技巧，但是当在真正的项目里需要用到服务的时候，可能还会有一些棘手的问题让你不知所措。因此，下面我们就来综合运用一下，尝试实现一个在服务中经常会使用到的功能——下载。

本节中我们将要编写一个完整版的下载示例，其中会涉及第 7 章、第 8 章、第 9 章和第 10 章的部分内容，算是目前为止综合程度最高的一个例子了。准备好了吗？创建一个 ServiceBestPractice 项目，然后开始本节的学习之旅吧。

首先我们需要将项目中会使用到的依赖库添加好，编辑 app/build.gradle 文件，在 dependencies 闭包中添加如下内容：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
    compile 'com.squareup.okhttp3:okhttp:3.4.1'
}
```

这里只需添加一个 OkHttp 的依赖就行了，待会儿在编写网络相关的功能时，我们将使用 OkHttp 来进行实现。

接下来需要定义一个回调接口，用于对下载过程中的各种状态进行监听和回调。新建一个 DownloadListener 接口，代码如下所示：

```
public interface DownloadListener {
    void onProgress(int progress);
    void onSuccess();
    void onFailed();
    void onPause();
    void onCancel();
}
```

可以看到，这里我们一共定义了 5 个回调方法，onProgress() 方法用于通知当前的下载进度，onSuccess() 方法用于通知下载成功事件，onFailed() 方法用于通知下载失败事件，onPaused() 方法用于通知下载暂时事件，onCancelled() 方法用于通知下载取消事件。

回调接口定义好了之后，下面我们就可以开始编写下载功能了。这里我准备使用本章中刚学的 AsyncTask 来进行实现，新建一个 DownloadTask 继承自 AsyncTask，代码如下所示：

```
public class DownloadTask extends AsyncTask<String, Integer, Integer> {
    public static final int TYPE_SUCCESS = 0;
    public static final int TYPE_FAILED = 1;
    public static final int TYPE_PAUSED = 2;
    public static final int TYPE_CANCELED = 3;

    private DownloadListener listener;
```

```
private boolean isCanceled = false;

private boolean isPaused = false;

private int lastProgress;

public DownloadTask(DownloadListener listener) {
    this.listener = listener;
}

@Override
protected Integer doInBackground(String... params) {
    InputStream is = null;
    RandomAccessFile savedFile = null;
    File file = null;
    try {
        long downloadedLength = 0; // 记录已下载的文件长度
        String downloadUrl = params[0];
        String fileName = downloadUrl.substring(downloadUrl.lastIndexOf("/"));
        String directory = Environment.getExternalStoragePublicDirectory
            (Environment.DIRECTORY_DOWNLOADS).getPath();
        file = new File(directory + fileName);
        if (file.exists()) {
            downloadedLength = file.length();
        }
        long contentLength = getContentLength(downloadUrl);
        if (contentLength == 0) {
            return TYPE_FAILED;
        } else if (contentLength == downloadedLength) {
            // 已下载字节和文件总字节相等，说明已经下载完成了
            return TYPE_SUCCESS;
        }
    OkHttpClient client = new OkHttpClient();
    Request request = new Request.Builder()
        // 断点下载，指定从哪个字节开始下载
        .addHeader("RANGE", "bytes=" + downloadedLength + "-")
        .url(downloadUrl)
        .build();
    Response response = client.newCall(request).execute();
    if (response != null) {
        is = response.body().byteStream();
        savedFile = new RandomAccessFile(file, "rw");
        savedFile.seek(downloadedLength); // 跳过已下载的字节
        byte[] b = new byte[1024];
        int total = 0;
        int len;
        while ((len = is.read(b)) != -1) {
            if (isCanceled) {
                return TYPE_CANCELED;
            } else if (isPaused) {
                return TYPE_PAUSED;
            } else {
                total += len;
            }
        }
    }
}
```

```
        savedFile.write(b, 0, len);
        // 计算已下载的百分比
        int progress = (int) ((total + downloadedLength) * 100 /
            contentLength);
        publishProgress(progress);
    }
}
response.body().close();
return TYPE_SUCCESS;
}
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (is != null) {
            is.close();
        }
        if (savedFile != null) {
            savedFile.close();
        }
        if (isCanceled && file != null) {
            file.delete();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
return TYPE_FAILED;
}

@Override
protected void onProgressUpdate(Integer... values) {
    int progress = values[0];
    if (progress > lastProgress) {
        listener.onProgress(progress);
        lastProgress = progress;
    }
}

@Override
protected void onPostExecute(Integer status) {
    switch (status) {
        case TYPE_SUCCESS:
            listener.onSuccess();
            break;
        case TYPE_FAILED:
            listener.onFailed();
            break;
        case TYPE_PAUSED:
            listener.onPause();
            break;
        case TYPE_CANCELED:
            listener.onCanceled();
            break;
        default:
```

```

        break;
    }
}

public void pauseDownload() {
    isPaused = true;
}

public void cancelDownload() {
    isCanceled = true;
}

private long getContentLength(String downloadUrl) throws IOException {
    OkHttpClient client = new OkHttpClient();
    Request request = new Request.Builder()
        .url(downloadUrl)
        .build();
    Response response = client.newCall(request).execute();
    if (response != null && response.isSuccessful()) {
        long contentLength = response.body().contentLength();
        response.close();
        return contentLength;
    }
    return 0;
}
}

```

这段代码就比较长了，我们需要一步步地进行分析。首先看一下 `AsyncTask` 中的 3 个泛型参数：第一个泛型参数指定为 `String`，表示在执行 `AsyncTask` 的时候需要传入一个字符串参数给后台任务；第二个泛型参数指定为 `Integer`，表示使用整型数据来作为进度显示单位；第三个泛型参数指定为 `Integer`，则表示使用整型数据来反馈执行结果。

接下来我们定义了 4 个整型常量用于表示下载的状态，`TYPE_SUCCESS` 表示下载成功，`TYPE_FAILED` 表示下载失败，`TYPE_PAUSED` 表示暂停下载，`TYPE_CANCELED` 表示取消下载。然后在 `DownloadTask` 的构造函数中要求传入一个刚刚定义的 `DownloadListener` 参数，我们待会就会将下载的状态通过这个参数进行回调。

接着就是要重写 `doInBackground()`、`onProgressUpdate()` 和 `onPostExecute()` 这 3 个方法了，我们之前已经学习过这 3 个方法各自的作用，因此在这里它们各自所负责的任务也是明确的：`doInBackground()` 方法用于在后台执行具体的下载逻辑，`onProgressUpdate()` 方法用于在界面上更新当前的下载进度，`onPostExecute()` 用于通知最终的下载结果。

那么先来看一下 `doInBackground()` 方法，首先我们从参数中获取到了下载的 URL 地址，并根据 URL 地址解析出了下载的文件名，然后指定将文件下载到 `Environment.DIRECTORY_DOWNLOADS` 目录下，也就是 SD 卡的 `Download` 目录。我们还要判断一下 `Download` 目录中是

不是已经存在要下载的文件了，如果已经存在的话则读取已下载的字节数，这样就可以在后面启用断点续传的功能。接下来先是调用了 `getContentLength()` 方法来获取待下载文件的总长度，如果文件长度等于 0 则说明文件有问题，直接返回 `TYPE_FAILED`，如果文件长度等于已下载文件长度，那么就说明文件已经下载完了，直接返回 `TYPE_SUCCESS` 即可。紧接着使用 OkHttp 来发送一条网络请求，需要注意的是，这里在请求中添加了一个 header，用于告诉服务器我们想要从哪个字节开始下载，因为已下载过的部分就不需要再重新下载了。接下来读取服务器响应的数据，并使用 Java 的文件流方式，不断从网络上读取数据，不断写入到本地，一直到文件全部下载完成为止。在这个过程中，我们还要判断用户有没有触发暂停或者取消的操作，如果说有的话则返回 `TYPE_PAUSED` 或 `TYPE_CANCELED` 来中断下载，如果没有的话则实时计算当前的下载进度，然后调用 `publishProgress()` 方法进行通知。暂停和取消操作都是使用一个布尔型的变量来进行控制的，调用 `pauseDownload()` 或 `cancelDownload()` 方法即可更改变量的值。

接下来看一下 `onProgressUpdate()` 方法，这个方法就简单得多了，它首先从参数中获取到当前的下载进度，然后和上一次的下载进度进行对比，如果有变化的话则调用 `DownloadListener` 的 `onProgress()` 方法来通知下载进度更新。

最后是 `onPostExecute()` 方法，也非常简单，就是根据参数中传入的下载状态来进行回调。下载成功就调用 `DownloadListener` 的 `onSuccess()` 方法，下载失败就调用 `onFailed()` 方法，暂停下载就调用 `onPaused()` 方法，取消下载就调用 `onCanceled()` 方法。

这样我们就把具体的下载功能完成了，下面为了保证 `DownloadTask` 可以一直在后台运行，我们还需要创建一个下载的服务。右击 `com.example.servicebestpractice` → `New` → `Service` → `Service`，新建 `DownloadService`，然后修改其中的代码，如下所示：

```
public class DownloadService extends Service {
    private DownloadTask downloadTask;
    private String downloadUrl;
    private DownloadListener listener = new DownloadListener() {
        @Override
        public void onProgress(int progress) {
            getNotificationManager().notify(1, getNotification("Downloading...", progress));
        }
    };
    @Override
    public void onSuccess() {
        downloadTask = null;
        // 下载成功时将前台服务通知关闭，并创建一个下载成功的通知
        stopForeground(true);
        getNotificationManager().notify(1, getNotification("Download Success", -1));
        Toast.makeText(DownloadService.this, "Download Success", Toast.LENGTH_SHORT).show();
    }
}
```

```
}

@Override
public void onFailed() {
    downloadTask = null;
    // 下载失败时将前台服务通知关闭，并创建一个下载失败的通知
    stopForeground(true);
    getNotificationManager().notify(1, getNotification("Download Failed",
        -1));
    Toast.makeText(DownloadService.this, "Download Failed",
        Toast.LENGTH_SHORT).show();
}

@Override
public void onPause() {
    downloadTask = null;
    Toast.makeText(DownloadService.this, "Paused", Toast.LENGTH_SHORT).
        show();
}

@Override
public void onCancel() {
    downloadTask = null;
    stopForeground(true);
    Toast.makeText(DownloadService.this, "Cancelled", Toast.LENGTH_SHORT).
        show();
}

private DownloadBinder mBinder = new DownloadBinder();

@Override
public IBinder onBind(Intent intent) {
    return mBinder;
}

class DownloadBinder extends Binder {

    public void startDownload(String url) {
        if (downloadTask == null) {
            downloadUrl = url;
            downloadTask = new DownloadTask(listener);
            downloadTask.execute(downloadUrl);
            startForeground(1, getNotification("Downloading...", 0));
            Toast.makeText(DownloadService.this, "Downloading...", Toast.
                LENGTH_SHORT).show();
        }
    }

    public void pauseDownload() {
        if (downloadTask != null) {
            downloadTask.pauseDownload();
        }
    }
}
```

```

    }

    public void cancelDownload() {
        if (downloadTask != null) {
            downloadTask.cancelDownload();
        } else {
            if (downloadUrl != null) {
                // 取消下载时需将文件删除，并将通知关闭
                String fileName = downloadUrl.substring(downloadUrl.
                    lastIndexOf("/"));
                String directory = Environment.getExternalStoragePublicDirectory
                    (Environment.DIRECTORY_DOWNLOADS).getPath();
                File file = new File(directory + fileName);
                if (file.exists()) {
                    file.delete();
                }
                getNotificationManager().cancel(1);
                stopForeground(true);
                Toast.makeText(DownloadService.this, "Canceled",
                    Toast.LENGTH_SHORT).show();
            }
        }
    }

private NotificationManager getNotificationManager() {
    return (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
}

private Notification getNotification(String title, int progress) {
    Intent intent = new Intent(this, MainActivity.class);
    PendingIntent pi = PendingIntent.getActivity(this, 0, intent, 0);
    NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
    builder.setSmallIcon(R.mipmap.ic_launcher);
    builder.setLargeIcon(BitmapFactory.decodeResource(getResources(),
        R.mipmap.ic_launcher));
    builder.setContentIntent(pi);
    builder.setContentTitle(title);
    if (progress > 0) {
        // 当 progress 大于或等于 0 时才显示下载进度
        builder.setContentText(progress + "%");
        builder.setProgress(100, progress, false);
    }
    return builder.build();
}
}

```

这段代码同样也比较长，我们还是得耐心慢慢看。首先这里创建了一个 DownloadListener 的匿名类实例，并在匿名类中实现了 onProgress()、onSuccess()、onFailed()、onPaused() 和 onCanceled() 这 5 个方法。在 onProgress() 方法中，我们调用 getNotification() 方法构

建了一个用于显示下载进度的通知，然后调用 `NotificationManager` 的 `notify()` 方法去触发这个通知，这样就可以在下拉状态栏中实时看到当前下载的进度了。在 `onSuccess()` 方法中，我们首先是将正在下载的前台通知关闭，然后创建一个新的通知用于告诉用户下载成功了。其他几个方法也都是类似的，分别用于告诉用户下载失败、暂停和取消这几个事件。

接下来为了要让 `DownloadService` 可以和活动进行通信，我们又创建了一个 `DownloadBinder`。`DownloadBinder` 中提供了 `startDownload()`、`pauseDownload()` 和 `cancelDownload()` 这 3 个方法，那么顾名思义，它们分别是用于开始下载、暂停下载和取消下载的。在 `startDownload()` 方法中，我们创建了一个 `DownloadTask` 的实例，把刚才的 `DownloadListener` 作为参数传入，然后调用 `execute()` 方法开启下载，并将下载文件的 URL 地址传入到 `execute()` 方法中。同时，为了让这个下载服务成为一个前台服务，我们还调用了 `startForeground()` 方法，这样就会在系统状态栏中创建一个持续运行的通知了。接着往下看，`pauseDownload()` 方法中的代码就非常简单了，就是简单地调用了一下 `DownloadTask` 中的 `pauseDownload()` 方法。`cancelDownload()` 方法中的逻辑也基本类似，但是要注意，取消下载的时候我们需要将正在下载的文件删除掉，这一点和暂停下载是不同的。

另外，`DownloadService` 类中所有使用到的通知都是调用 `getNotification()` 方法进行构建的，这个方法中的代码我们之前基本都是学过的，只有一个 `setProgress()` 方法没有见过。`setProgress()` 方法接收 3 个参数，第一个参数传入通知的最大进度，第二个参数传入通知的当前进度，第三个参数表示是否使用模糊进度条，这里传入 `false`。设置完 `setProgress()` 方法，通知上就会有进度条显示出来了。

现在下载的服务也已经成功实现，后端的工作基本都完成了，那么接下来我们开始编写前端的部分。修改 `activity_main.xml` 中的代码，如下所示：

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

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

    <Button
        android:id="@+id/pause_download"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Pause Download" />

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

```

    android:layout_height="wrap_content"
    android:text="Cancel Download" />

</LinearLayout>

```

布局文件还是非常简单的，这里在 LinearLayout 中放置了 3 个按钮，分别用于开始下载、暂停下载和取消下载。

然后修改 MainActivity 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private DownloadService.DownloadBinder downloadBinder;

    private ServiceConnection connection = new ServiceConnection() {

        @Override
        public void onServiceDisconnected(ComponentName name) {
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            downloadBinder = (DownloadService.DownloadBinder) service;
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button startDownload = (Button) findViewById(R.id.start_download);
        Button pauseDownload = (Button) findViewById(R.id.pause_download);
        Button cancelDownload = (Button) findViewById(R.id.cancel_download);
        startDownload.setOnClickListener(this);
        pauseDownload.setOnClickListener(this);
        cancelDownload.setOnClickListener(this);
        Intent intent = new Intent(this, DownloadService.class);
        startService(intent); // 启动服务
        bindService(intent, connection, BIND_AUTO_CREATE); // 绑定服务
        if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.
            permission.WRITE_EXTERNAL_STORAGE)!= PackageManager.PERMISSION_GRANTED) {
            ActivityCompat.requestPermissions(MainActivity.this, new
                String[]{ Manifest.permission.WRITE_EXTERNAL_STORAGE }, 1);
        }
    }

    @Override
    public void onClick(View v) {
        if (downloadBinder == null) {
            return;
        }
        switch (v.getId()) {

```

```

        case R.id.start_download:
            String url = "https://raw.githubusercontent.com/guolindev/eclipse/
                         master/eclipse-inst-win64.exe";
            downloadBinder.startDownload(url);
            break;
        case R.id.pause_download:
            downloadBinder.pauseDownload();
            break;
        case R.id.cancel_download:
            downloadBinder.cancelDownload();
            break;
        default:
            break;
    }
}

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
                                       int[] grantResults) {
    switch (requestCode) {
        case 1:
            if (grantResults.length > 0 && grantResults[0] != PackageManager.
                PERMISSION_GRANTED) {
                Toast.makeText(this, "拒绝权限将无法使用程序", Toast.LENGTH_SHORT).
                    show();
                finish();
            }
            break;
        default:
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unbindService(connection);
}
}

```

可以看到，这里我们首先创建了一个 `ServiceConnection` 的匿名类，然后在 `onServiceConnected()`方法中获取到 `DownloadBinder` 的实例，有了这个实例，我们就可以在活动中调用服务提供的各种方法了。

接下来看一下 `onCreate()`方法，在这里我们对各个按钮都进行了初始化操作并设置了点击事件，然后分别调用了 `startService()`和 `bindService()`方法来启动和绑定服务。这一点至关重要，因为启动服务可以保证 `DownloadService`一直在后台运行，绑定服务则可以让 `MainActivity` 和 `DownloadService` 进行通信，因此两个方法调用都必不可少。在 `onCreate()`方法的最后，我们还进行了 `WRITE_EXTERNAL_STORAGE` 的运行时权限申请，因为下载文件是要下载到 SD 卡的 `Download` 目录下的，如果没有这个权限的话，我们整个程序都无法正常工作。

接下来的代码就非常简单了，在 `onClick()` 方法中我们对点击事件进行判断，如果点击了开始按钮就调用 `DownloadBinder` 的 `startDownload()` 方法，如果点击了暂停按钮就调用 `pauseDownload()` 方法，如果点击了取消按钮就调用 `cancelDownload()` 方法。`startDownload()` 方法中你可以传入任意的下载地址，这里我使用了一个 Eclipse 的下载地址，以此向这个 Android 平台上曾经最出色的开发工具致敬。

另外还有一点需要注意，如果活动被销毁了，那么一定要记得对服务进行解绑，不然就有可能会造成内存泄漏。这里我们在 `onDestroy()` 方法中完成了解绑操作。

现在只差最后一步了，我们还需要在 `AndroidManifest.xml` 文件中声明使用到的权限。当然除了权限之外，`MainActivity` 和 `DownloadService` 也是需要声明的，不过 Android Studio 应该早就帮我们将这两个组件声明好了，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.servicebestpractice">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service
            android:name=".DownloadService"
            android:enabled="true"
            android:exported="true" />
    </application>

</manifest>
```

其中，由于我们的程序使用到了网络和访问 SD 卡的功能，因此需要声明 `INTERNET` 和 `WRITE_EXTERNAL_STORAGE` 这两个权限。

这样所有代码就都编写完了，现在终于可以运行一下程序了，如图 10.16 所示。

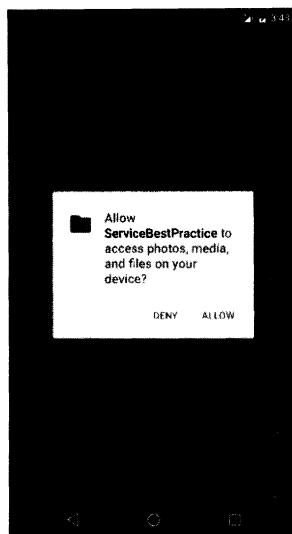


图 10.16 申请访问 SD 卡权限

程序一启动立刻就会申请访问 SD 卡的权限,这里我们点击 ALLOW,然后点击 Start Download 按钮就可以开始下载了。下载过程中可以下拉系统状态栏查看实时的下载进度,如图 10.17 所示。

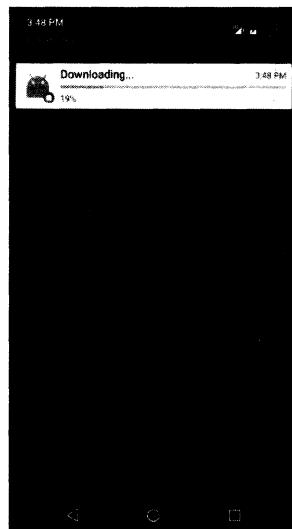


图 10.17 查看实时的下载进度

同时,我们还可以点击 Pause Download 或 Cancel Download,甚至于断网操作来测试这个下载程序的健壮性。最终下载完成后会弹出一个 Download Success 的通知,然后我们可以通过任意一个文件浏览器来查看一下 SD 卡的 Download 目录,如图 10.18 所示。

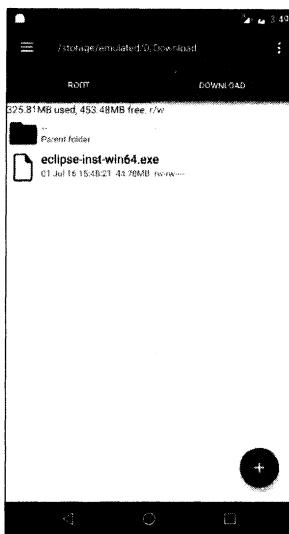


图 10.18 查看 SD 卡的 Download 目录

可以看到，文件已经成功下载下来了。

当然，我们还可以做一些更加丰富的操作，比如说再次点击 Start Download 按钮，你会发现程序会立刻弹出一个 Download Success 的提示，因为它检测到文件已经下载完成了，因而不会再重新去下载一遍。如果我们点击 Cancel Download 按钮先将下载文件删除掉，然后再点击 Start Download 按钮，你就会发现程序又会开始重新下载了。

总体来说，这个下载示例的稳定性还是挺不错的，而且综合性很强，将这个示例完全掌握了之后，你的水平肯定又更进一步了。

好了，最佳实践部分到此结束，下面我们就来回顾一下本章所学的内容吧。

10.7 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你在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下载示例程序，相信以后不管遇到什么样的服务难题，你都能从容解决。

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一大半了。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。

那么后面我们应该再接再厉，争取进一步提升自身的能力，所以现在还不是放松的时候，下一章中我们准备去学习一下 Android 特色开发的相关内容。

第 11 章

Android 特色开发——基于位置的服务

现在你已经学会了非常多的 Android 技能，并且通过这些技能你完全可以编写出相当不错的应用程序了。不过本章中，我们将要学习一些全新的 Android 技术，这些技术有别于传统的 PC 或 Web 领域的应用技术，是只有在移动设备上才能实现的。

说到只有在移动设备上才能实现的技术，很容易就让人联想到基于位置的服务（Location Based Service）。由于移动设备相比于电脑可以随身携带，我们通过地理定位的技术就可以随时得知自己所在的位置，从而围绕这一点开发出很多有意思的应用。本章中我们就将针对这一点进行讨论，学习一下基于位置的服务究竟是如何实现的。

11.1 基于位置的服务简介

基于位置的服务简称 LBS，随着移动互联网的兴起，这个技术在最近的几年里十分火爆。其实它本身并不是什么时髦的技术，主要的工作原理就是利用无线电通讯网络或 GPS 等定位方式来确定出移动设备所在的位置，而这种定位技术早在很多年前就已经出现了。

那为什么 LBS 技术直到最近几年才开始流行呢？这主要是因为，在过去移动设备的功能极其有限，即使定位到了设备所在的位置，也就仅仅只是定位到了而已，我们并不能在位置的基础上进行一些其他的操作。而现在就大大不同了，有了 Android 系统作为载体，我们可以利用定位出的位置进行许多丰富多彩的操作。比如说天气预报程序可以根据用户所在的位置自动选择城市，发微博的时候我们可以向朋友们晒一下自己在哪里，不认识路的时候随时打开地图就可以查询路线，等等。

介绍了这么多，相信你已经按捺不住了吧？我们马上就要开始本章的学习之旅，但在开始之前，还有一些事情是你必须要知道的。

首先你要清楚，基于位置的服务所围绕的核心就是要先确定出用户所在的位置。通常有两种技术方式可以实现：一种是通过 GPS 定位，一种是通过网络定位。GPS 定位的工作原理是基于手机内置的 GPS 硬件直接和卫星交互来获取当前的经纬度信息，这种定位方式精确度非常高，

但缺点是只能在室外使用，室内基本无法接收到卫星的信号。网络定位的工作原理是根据手机当前网络附近的三个基站进行测速，以此计算出手机和每个基站之间的距离，再通过三角定位确定出一个大概的位置，这种定位方式精确度一般，但优点是在室内室外都可以使用。

Android 对这两种定位方式都提供了相应的 API 支持，但是由于一些特殊原因，Google 的网络服务在中国不可访问，从而导致网络定位方式的 API 失效。而 GPS 定位虽然不需要网络，但是必须要在室外才可以使用，因此你在室内开发的时候很有可能会遇到不管使用哪种定位方式都无法成功定位的情况。

基于以上原因，我决定就不在本书中讲解 Android 原生定位 API 的用法了，而是使用一些国内第三方公司的 SDK。目前国内在这一领域做得比较好的一个是百度，一个是高德，本章我们就来学习一下百度在 LBS 方面提供的丰富多彩的功能。

11.2 申请 API Key

要想在自己的应用程序里使用百度的 LBS 功能，首先必须申请一个 API Key。你得拥有一个百度账号才能进行申请，我相信大多数人早就已经拥有了吧？如果你还没有的话，赶快去注册一个吧。

有了百度账号之后，我们就可以申请成为一名百度开发者了，登录你的百度账号，并打开 <http://developer.baidu.com/user/reg> 这个网址，在这里填写一些注册信息即可，如图 11.1 所示。

* 类型：
 个人 公司

* 开发者来源：
 开发者

* 开发者姓名：

* 开发者简介：

* Email 地址：
 修改

* 手机号：
 重新发送 (5 秒)

* 验证码：

开发者官方网站：

品牌 LOGO：
112px*54px, 支持PNG/JPG/GIF格式, 应用提交至PC
Web渠道时进行展示

我已阅读并同意百度开放云平台注册协议

提交

图 11.1 填写开发者信息

只需填写带有“*”号的那部分内容就足够了，接下来点击提交，会显示如图 11.2 所示的界面。



图 11.2 验证邮箱

接着点击“去我的邮箱”，将会进入到我们刚才填写的邮箱当中，这时收件箱中应该会有一封刚刚收到的邮件，这就是百度发送给我们的验证邮件，点击邮件当中的链接就可以完成注册了，如图 11.3 所示。

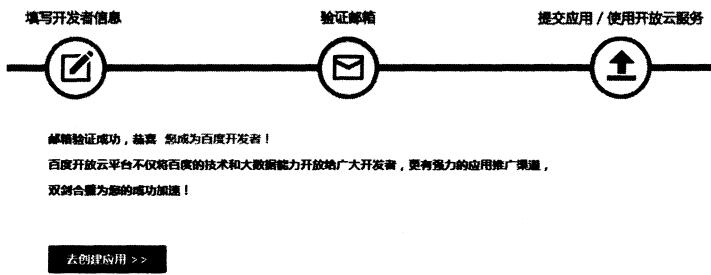


图 11.3 成为百度开发者

到此一切顺利！这样你就已经成为一名百度开发者了。接着访问 <http://lbsyun.baidu.com/apiconsole/key> 这个地址，然后同意百度开发者协议，会看到如图 11.4 所示的界面。



图 11.4 百度 LBS 开放平台主界面

由于这是一个刚刚注册的账号，所以目前的应用列表是空的。接下来点击创建应用就可以去申请 API Key 了，应用名称可以随便填，应用类型选择 Android SDK，启用服务保持默认即可，如图 11.5 所示。

应用名称： LBSTest

应用类型： Android SDK ▼

启用服务：

<input type="checkbox"/> 云检索API	<input checked="" type="checkbox"/> Javascript API	<input checked="" type="checkbox"/> Place API v2
<input checked="" type="checkbox"/> Geocoding API v2	<input checked="" type="checkbox"/> IP定位API	<input checked="" type="checkbox"/> 路线交通API
<input checked="" type="checkbox"/> Android地图SDK	<input checked="" type="checkbox"/> Android导航离线SDK	<input checked="" type="checkbox"/> Android导航SDK
<input checked="" type="checkbox"/> 静态图API	<input checked="" type="checkbox"/> 全景静态图API	<input checked="" type="checkbox"/> 坐标转换API
<input checked="" type="checkbox"/> 网络API	<input checked="" type="checkbox"/> 全景URL API	<input checked="" type="checkbox"/> Android导航 HUD SDK
<input checked="" type="checkbox"/> 云逆地理编码API	<input checked="" type="checkbox"/> Routematrix API	

* 发布版SHA1： 请输入发布版SHA1。

开发版SHA1： 请输入开发版SHA1。

* 包名： 请输入包名。

安全码： 输入sha1和包名后自动生成

Android SDK安全码组成：SHA1+包名。[\(查看详细配置方法\)](#)

新申请的Mobile与Browser类型的ak不再支持云存储接口的访问，如要使用云存储，请申请Server类型ak。

提交

图 11.5 创建应用界面

那么，这个发布版 SHA1 和开发版 SHA1 又是个什么东西呢？这是我们申请 API Key 所必须填写的一个字段，它指的是打包程序时所用签名文件的 SHA1 指纹，可以通过 Android Studio 查看到。打开 Android Studio 中的任意一个项目，点击右侧工具栏的 Gradle→项目名→:app→Tasks→android，如图 11.6 所示。

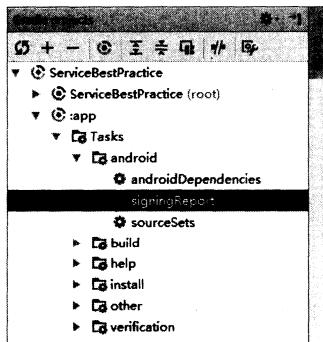


图 11.6 查看内置 Gradle Tasks

这里展示了一个 Android Studio 项目中所有内置的 Gradle Tasks，其中 signingReport 这个 Task 就可以用来查看签名文件信息。双击 signingReport，结果如图 11.7 所示。

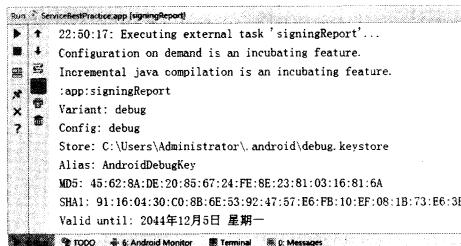


图 11.7 signingReport Task 的执行结果

其中，91:16:04:30:C0:8B:6E:53:92:47:57:E6:FB:10:EF:08:1B:73:E6:3E 就是我们所需的 SHA1 指纹了，当然你的 Android Studio 中显示的指纹和我的肯定是不一样的。另外需要注意，目前我们使用的是 debug.keystore 文件所生成的指纹，这是 Android 自动生成的一个用于测试的签名文件。而当你的应用程序发布时还需要创建一个正式的签名文件，如果要得到它的指纹，可以在 cmd 中输入如下命令：

```
keytool -list -v -keystore <签名文件路径>
```

然后输入正确的密码就可以了。创建签名文件的方法我们将在第 15 章中学习。

那么也就是说，现在得到的这个 SHA1 指纹实际上是一个开发版的 SHA1 指纹，不过因为暂时我们还没有一个发布版的 SHA1 指纹，因此这两个值都填成一样的就可以了。最后还剩下一个包名选项，虽然目前我们的应用程序还不存在，但可以先将包名预定下来，比如就叫 com.example.lbtest，这样所有的内容就都填写完整了，如图 11.8 所示。

应用名称：LBTest

应用类型：Android SDK

启用服务：

- 云检索API
- Javascript API
- Place API v2
- Geocoding API v2
- IP定位API
- 路线交通API
- Android地图SDK
- 全景静态图API
- Android导航SDK
- 静态图API
- 全景URL API
- 坐标转换API
- 唐僧API
- 全景HUD API
- Android导航HUD SDK
- 云逆地理编码API
- Routematrix API

* 发布版SHA1：91:16:04:30:C0:8B:6E:53:92:47:57:E6:FB:10:EF:08:1B:73:E6:3E

开发版SHA1：91:16:04:30:C0:8B:6E:53:92:47:57:E6:FB:10:EF:08:1B:73:E6:3E 输入正确

* 包名：com.example.lbtest

安全码：91:16:04:30:C0:8B:6E:53:92:47:57:E6:FB:10:EF:08:1B:73:E6:3E:com.example.lbtest
91:16:04:30:C0:8B:6E:53:92:47:57:E6:FB:10:EF:08:1B:73:E6:3E:com.example.lbtest

Android SDK安全码组成：SHA1+包名。[\(查看详细配置方法\)](#)

新申请的Mobile与Browser类型的ak不再支持云存储接口的访问，如要使用云存储，请申请Server类型ak。

提交

图 11.8 填写完整所有创建应用的信息

接下来点击提交，应用就应该创建成功了，如图 11.9 所示。



图 11.9 查看已创建的应用

其中，i6VD2fHKM3msMfZtIOXAhFSzDiYGFJwL 就是申请到的 API Key，有了它就可以进行后续的 LBS 开发工作了，那么我们马上开始吧。

11.3 使用百度定位

现在正是趁热打铁的好时机，新建一个 LBSTest 项目，包名应该就会自动被命名为 com.example.lbstest。另外需要注意，本章中所写的代码建议你都在手机上运行，虽然模拟器中也提供了模拟地理位置的功能，但在手机上可以得到真实的位置数据，你的感受会更加深刻。

11.3.1 准备 LBS SDK

在开始编码之前，我们还需要先将百度 LBS 开放平台的 SDK 准备好，下载地址是：<http://lbsyun.baidu.com/sdk/download>。本章中我们会用到基础地图和定位功能这两个 SDK，将它们勾选上，然后点击“开发包”下载按钮即可，如图 11.10 所示。



图 11.10 下载 SDK 界面

下载完成后对该压缩包解压，其中会有一个 libs 目录，这里面的内容就是我们所需要的一切了，如图 11.11 所示。

名称	类型	大小
arm64-v8a	文件夹	
armeabi	文件夹	
armeabi-v7a	文件夹	
x86	文件夹	
x86_64	文件夹	
BaiduLBS_Android.jar	Executable Jar File	1,232 KB

图 11.11 压缩包 libs 目录下的内容

libs 目录下的内容又分为两部分，BaiduLBS_Android.jar 这个文件是 Java 层要使用到的，其他子目录下的 so 文件是 Native 层要用到的。so 文件是用 C/C++语言进行编写，然后再用 NDK 编译出来的。当然这里我们并不需要去编写 C/C++的代码，因为百度都已经做好了封装，但是我们需要将 libs 目录下的每一个文件都放置到正确的位置。

首先观察一下当前的项目结构，你会发现 app 模块下面有一个 libs 目录，这里就是用来存放所有的 Jar 包的，我们将 BaiduLBS_Android.jar 复制到这里，如图 11.12 所示。

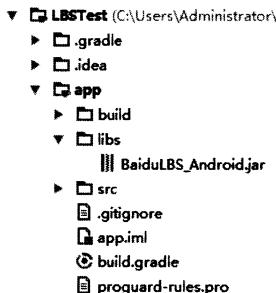


图 11.12 将 Jar 包放置到 libs 目录中

接下来展开 src/main 目录，右击该目录→New→Directory，再创建一个名为 jniLibs 的目录，这里就是专门用来存放 so 文件的，然后把压缩包里的其他所有目录直接复制到这里，如图 11.13 所示。

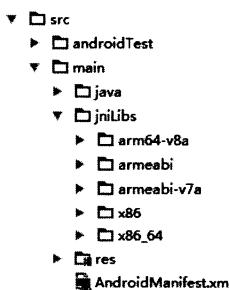


图 11.13 将 so 文件放置到 jniLibs 目录中

另外，虽然所有新创建的项目中，app/build.gradle 文件都会默认配置以下这段声明：

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    ...
}
```

这表示会将 libs 目录下所有以.jar 结尾的文件添加到当前项目的引用中。但是由于我们是直接将 Jar 包复制到 libs 目录下的，并没有修改 gradle 文件，因此不会弹出我们平时熟悉的 Sync Now 提示。这个时候必须手动点击一下 Android Studio 顶部工具栏中的 Sync 按钮（图 11.14 中最左边的按钮），不然项目将无法引用到 Jar 包中提供的任何接口。

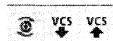


图 11.14 Android Studio 顶部工具栏

点击 Sync 按钮之后，libs 目录下的 jar 文件就会多出一个向右的箭头，这就表示项目已经能引用到这些 Jar 包了，如图 11.15 所示。



图 11.15 Jar 包引用成功

好了，这样我们就把 LBS 的 SDK 都准备好了，接下来开始编码吧。

11.3.2 确定自己位置的经纬度

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:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <TextView
        android:id="@+id/position_text_view"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

布局文件中的内容非常简单，只有一个 TextView 控件，用于稍后显示当前位置的经纬度信息。

然后修改 AndroidManifest.xml 文件中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.lbstest">

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

```

<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>

<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <meta-data
        android:name="com.baidu.lbsapi.API_KEY"
        android:value="i6VD2fHKM3msMfZtIOXAhFSzDiYGFiwl" />

    <activity android:name=".MainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <service android:name="com.baidu.location.f" android:enabled="true"
        android:process=":remote">
    </service>
</application>

</manifest>

```

AndroidManifest.xml 文件改动比较多，我们来仔细阅读一下。可以看到，这里首先添加了很多行权限声明，每一个权限都是百度 LBS SDK 内部要用到的。然后在<application>标签的内部添加了一个<meta-data>标签，这个标签的 android:name 部分是固定的，必须填 com.baidu.lbsapi.API_KEY， android:value 部分则应该填入我们在 11.2 节申请到的 API Key。最后，还需要再注册一个 LBS SDK 中的服务，不用对这个服务的名字感到疑惑，因为百度 LBS SDK 中的代码都是混淆过的。

接下来修改 MainActivity 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity {

    public LocationClient mLocationClient;

    private TextView positionText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {

```

```
super.onCreate(savedInstanceState);
mLocationClient = new LocationClient(getApplicationContext());
mLocationClient.registerLocationListener(new MyLocationListener());
setContentView(R.layout.activity_main);
positionText = (TextView) findViewById(R.id.position_text_view);
List<String> permissionList = new ArrayList<>();
if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
    permissionList.add(Manifest.permission.ACCESS_FINE_LOCATION);
}
if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.permission.READ_PHONE_STATE) != PackageManager.PERMISSION_GRANTED) {
    permissionList.add(Manifest.permission.READ_PHONE_STATE);
}
if (ContextCompat.checkSelfPermission(MainActivity.this, Manifest.permission.WRITE_EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
    permissionList.add(Manifest.permission.WRITE_EXTERNAL_STORAGE);
}
if (!permissionList.isEmpty()) {
    String [] permissions = permissionList.toArray(new String[permissionList.size()]);
    ActivityCompat.requestPermissions(MainActivity.this, permissions, 1);
} else {
    requestLocation();
}
}

private void requestLocation() {
    mLocationClient.start();
}

@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
    int[] grantResults) {
    switch (requestCode) {
        case 1:
            if (grantResults.length > 0) {
                for (int result : grantResults) {
                    if (result != PackageManager.PERMISSION_GRANTED) {
                        Toast.makeText(this, "必须同意所有权限才能使用本程序",
                            Toast.LENGTH_SHORT).show();
                        finish();
                        return;
                    }
                }
                requestLocation();
            } else {
                Toast.makeText(this, "发生未知错误", Toast.LENGTH_SHORT).show();
                finish();
            }
            break;
        default:
    }
}
```

```

public class MyLocationListener implements BDLocationListener {

    @Override
    public void onReceiveLocation(BDLocation location) {
        StringBuilder currentPosition = new StringBuilder();
        currentPosition.append("纬度: ").append(location.getLatitude()).
            append("\n");
        currentPosition.append("经线: ").append(location.getLongitude()).
            append("\n");
        currentPosition.append("定位方式: ");
        if (location.getLocType() == BDLocation.TypeGpsLocation) {
            currentPosition.append("GPS");
        } else if (location.getLocType() == BDLocation.TypeNetworkLocation) {
            currentPosition.append("网络");
        }
        positionText.setText(currentPosition);
    }
}

```

可以看到，在 `onCreate()` 方法中，我们首先创建了一个 `LocationClient` 的实例，`LocationClient` 的构建函数接收一个 `Context` 参数，这里调用 `getApplicationContext()` 方法来获取一个全局的 `Context` 参数并传入。然后调用 `LocationClient` 的 `registerLocationListener()` 方法来注册一个定位监听器，当获取到位置信息的时候，就会回调这个定位监听器。

接下来看一下这里运行时权限的用法，由于我们在 `AndroidManifest.xml` 中声明了很多权限，参考一下 7.2.1 小节中的危险权限表格可以发现，其中 `ACCESS_COARSE_LOCATION`、`ACCESS_FINE_LOCATION`、`READ_PHONE_STATE`、`WRITE_EXTERNAL_STORAGE` 这 4 个权限是需要进行运行时权限处理的，不过由于 `ACCESS_COARSE_LOCATION` 和 `ACCESS_FINE_LOCATION` 都属于同一个权限组，因此两者只要申请其一就可以了。那么怎样才能在运行时一次性申请 3 个权限呢？这里我们使用了一种新的用法，首先创建一个空的 `List` 集合，然后依次判断这 3 个权限有没有被授权，如果没被授权就添加到 `List` 集合中，最后将 `List` 转换成数组，再调用 `ActivityCompat.requestPermissions()` 方法一次性申请。

除此之外，`onRequestPermissionsResult()` 方法中对权限申请结果的逻辑处理也和之前有所不同，这次我们通过一个循环将申请的每个权限都进行了判断，如果有任何一个权限被拒绝，那么就直接调用 `finish()` 方法关闭当前程序，只有当所有权限都被用户同意了，才会调用 `requestLocation()` 方法开始地理位置定位。

`requestLocation()` 方法中的代码比较简单，只是调用了一下 `LocationClient` 的 `start()` 方法就能开始定位了。定位的结果会回调到我们前面注册的监听器当中，也就是 `MyLocationListener`。观察一下 `MyLocationListener` 的 `onReceiveLocation()` 方法中，在这里我们通过 `BDLocation` 的 `getLatitude()` 方法获取当前位置的纬度，通过 `getLongitude()` 方法获

取当前位置的经度，通过 `getLocType()` 方法获取当前的定位方式，最终将结果组装成一个字符串，显示到 `TextView` 上面。

现在我们可以来运行一下程序了，如图 11.16 所示。毫无疑问，打开程序首先就会弹出运行时权限的申请对话框，注意看对话框的底部，提示我们一共有 3 项权限申请，当前是第 1 项，授权了第 1 项后就会显示第 2 项，这里我们全部点击允许，然后就会立刻开始定位了，结果如图 11.17 所示。



图 11.16 运行时权限申请对话框



图 11.17 地理位置定位的结果

可以看到，设备当前的经纬度信息已经成功定位出来了。

不过，在默认情况下，调用 `LocationClient` 的 `start()` 方法只会定位一次，如果我们正在快速移动中，怎样才能实时更新当前的位置呢？为此，百度 LBS SDK 提供了一系列的设置方法，来允许我们更改默认的行为，修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    ...

    private void requestLocation() {
        initLocation();
        mLocationClient.start();
    }

    private void initLocation(){
        LocationClientOption option = new LocationClientOption();
        option.setScanSpan(5000);
        mLocationClient.setLocOption(option);
    }

    @Override
```