

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。

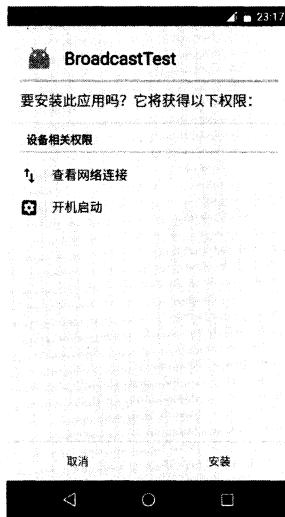


图 7.1 安装界面的权限提醒

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。

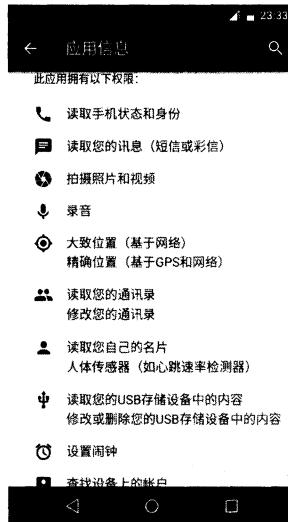


图 7.3 微信的权限列表

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。

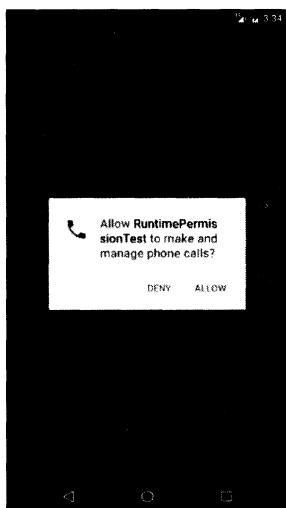


图 7.5 申请电话权限对话框

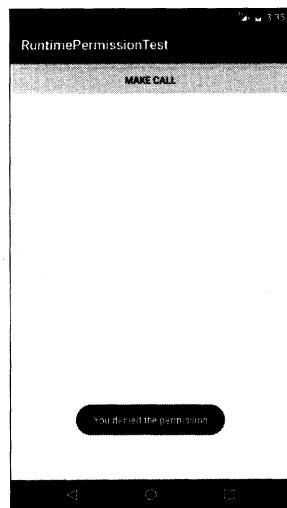


图 7.6 用户拒绝了权限申请

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。

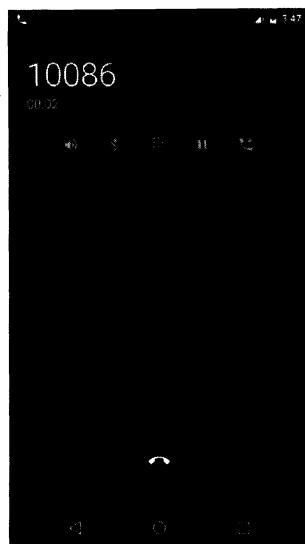


图 7.7 拨打电话界面

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。

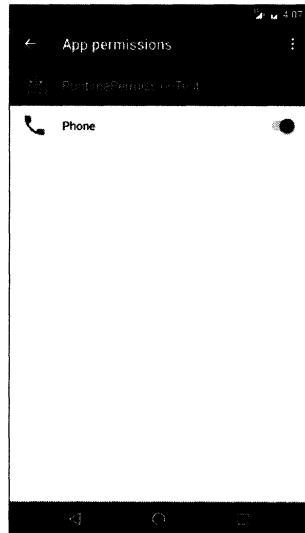


图 7.8 应用程序权限管理界面

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。

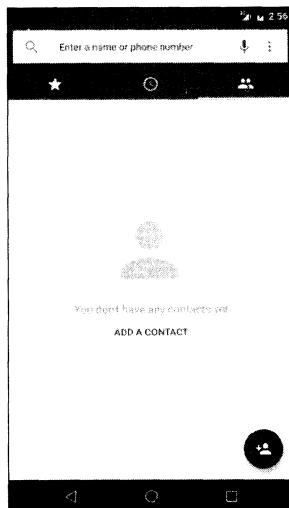


图 7.9 电话簿程序主界面

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。

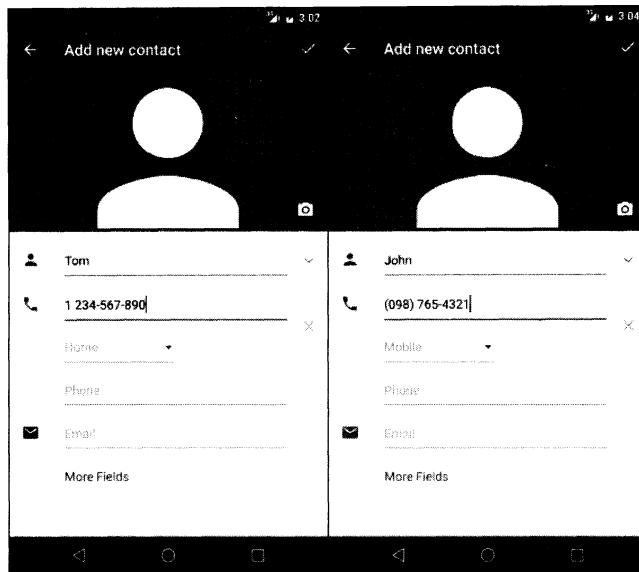


图 7.10 添加两个联系人

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。

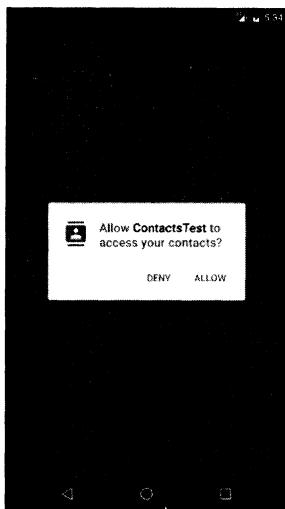


图 7.11 申请访问联系人权限对话框

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。

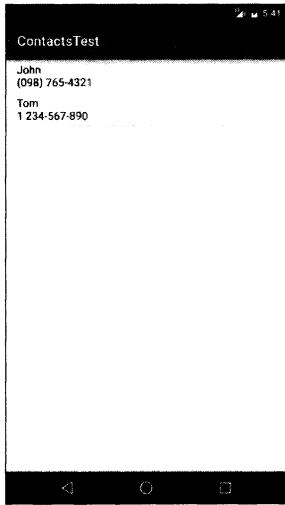


图 7.12 展示系统联系人信息

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。

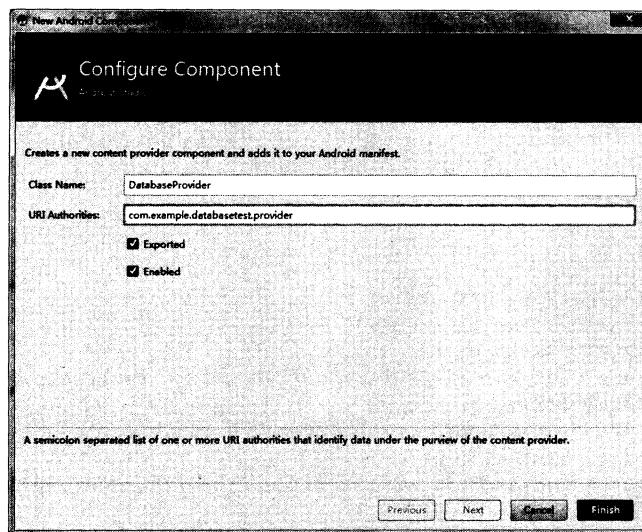


图 7.13 创建内容提供器的窗口

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。

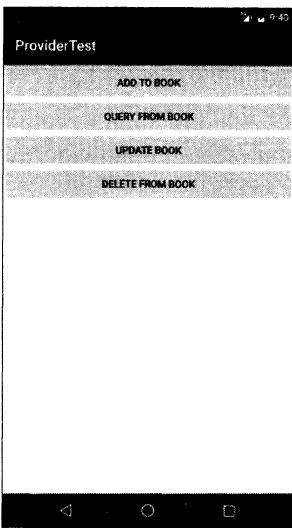


图 7.14 ProviderTest 主界面

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。

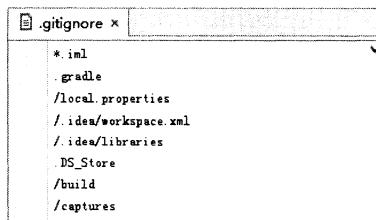


图 7.18 根目录下面的 `.gitignore` 文件

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。

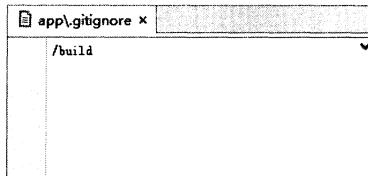


图7.19 app模块下面的.gitignore文件

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。

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。

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。

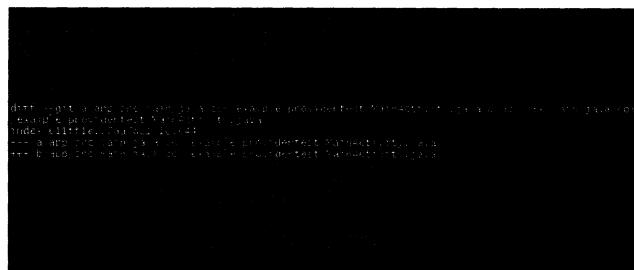


图 7.26 查看提交记录的具体修改内容

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。

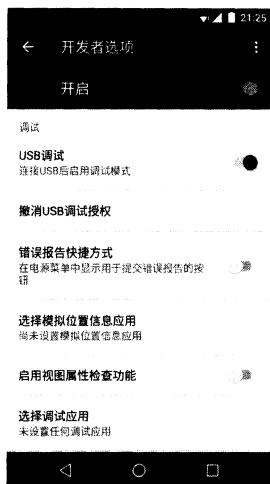


图 8.1 启用 USB 调试

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。

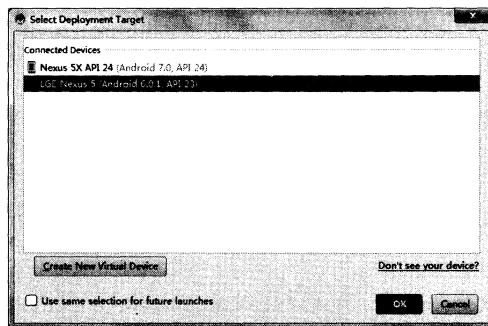


图 8.4 选择运行设备对话框

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。

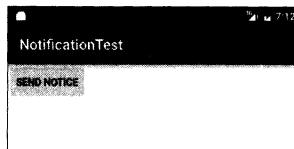


图 8.5 通知的小图标

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。

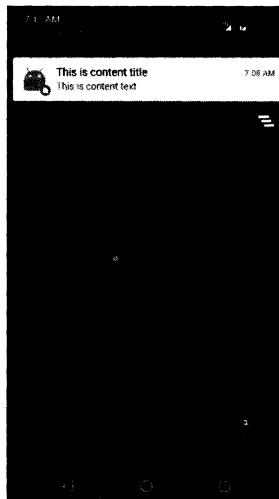


图 8.6 通知的详细信息

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。

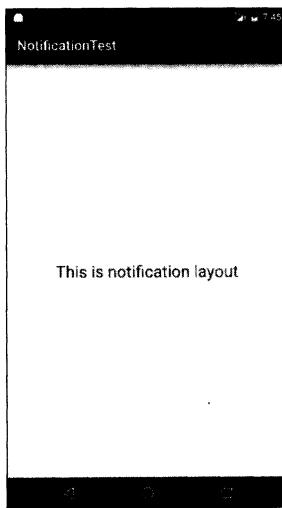


图 8.7 点击通知后打开 NotificationActivity 界面

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。

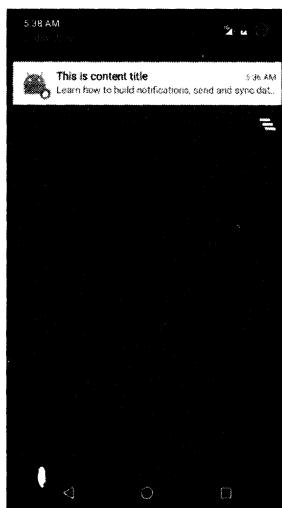


图 8.8 通知内容文字过长的效果

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。

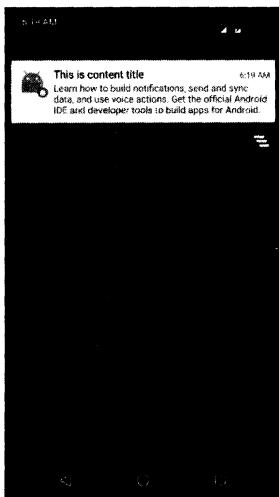


图 8.9 通知中显示长文字的效果

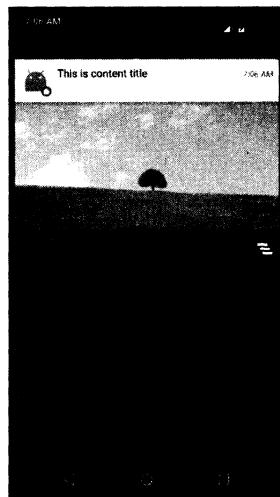


图 8.10 通知中显示大图片的效果

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。

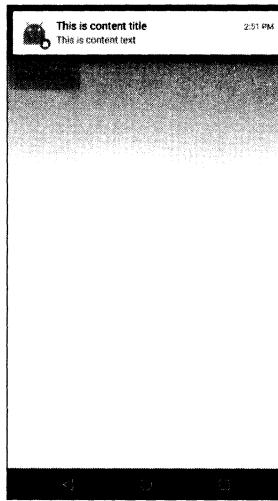


图 8.11 触发一条重要通知

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。

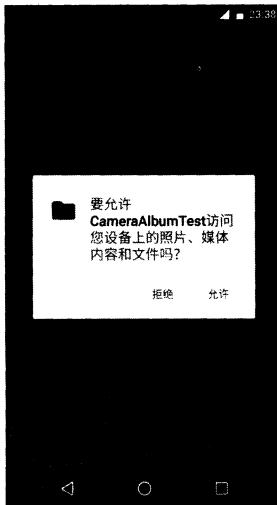


图 8.14 申请访问 SD 卡权限

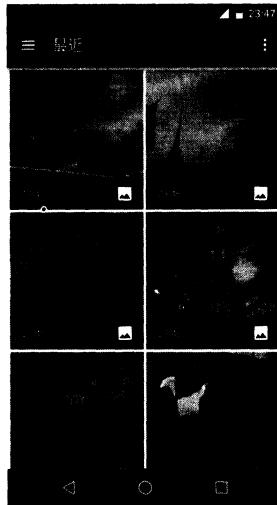


图 8.15 打开手机相册

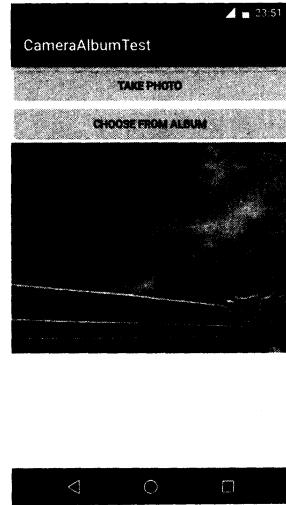


图 8.16 选择照片的最终效果

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。

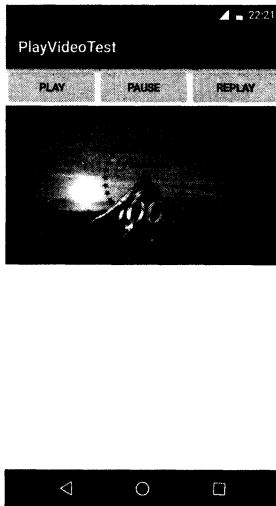


图 8.18 `VideoView` 播放视频的效果

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。

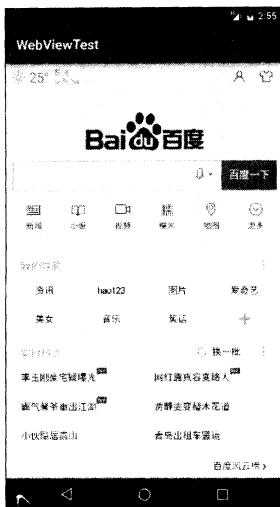


图 9.1 WebView 加载网页

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();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}

```

```

        // 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。

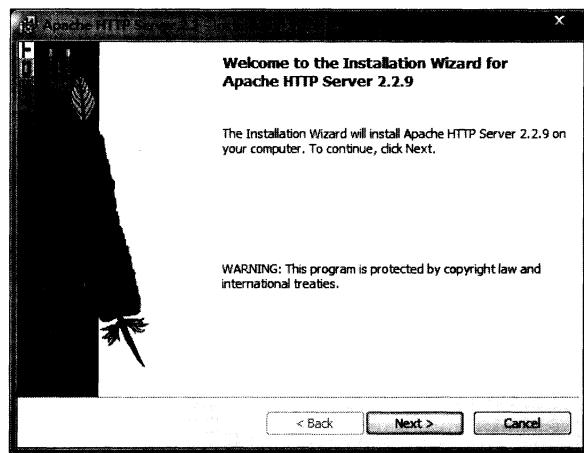


图 9.3 Apache 服务器安装界面

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。

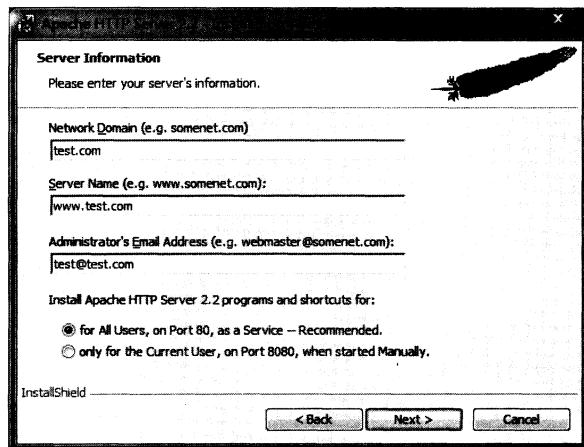


图 9.4 填入域名和服务器信息

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。

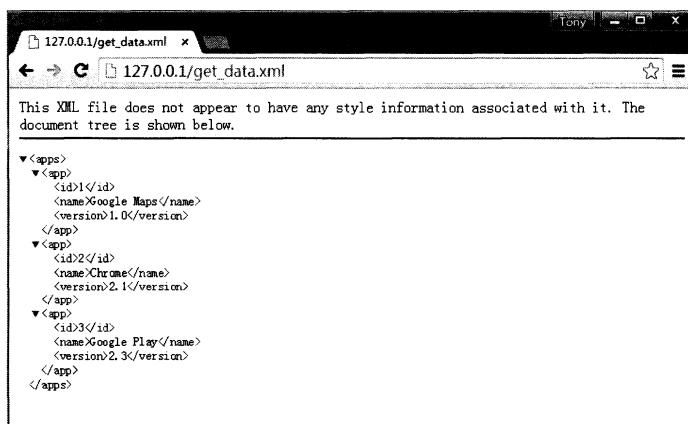


图 9.6 在浏览器验证 XML 数据

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();  
        }  
    }  
}
```

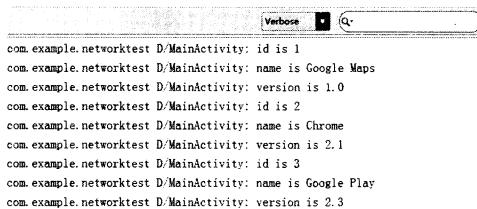
```
        case XmlPullParser.START_TAG: {
            if ("id".equals(nodeName)) {
                id = xmlPullParser.nextText();
            } else if ("name".equals(nodeName)) {
                name = xmlPullParser.nextText();
            } else if ("version".equals(nodeName)) {
                version = 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。

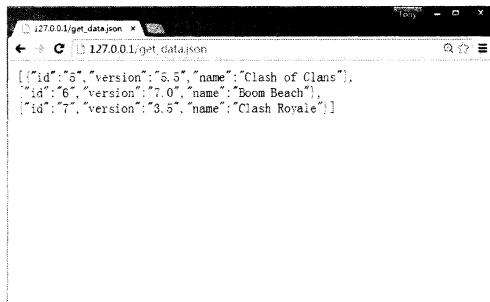


图 9.8 在浏览器验证 JSON 数据

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。

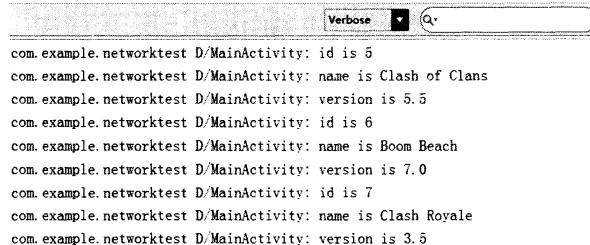


图 9.9 打印从 JSON 中解析出的数据

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。