

第 4 章

手机平板要兼顾——探究碎片

当今是移动设备发展非常迅速的时代，不仅手机已经成为了生活必需品，就连平板电脑也变得越来越普及。平板电脑和手机最大的区别就在于屏幕的大小，一般手机屏幕的大小会在 3 英寸到 6 英寸之间，而一般平板电脑屏幕的大小会在 7 英寸到 10 英寸之间。屏幕大小差距过大有可能会让同样的界面在视觉效果上有较大的差异，比如一些界面在手机上看起来非常美观，但在平板电脑上看起来就可能会有控件被过分拉长、元素之间空隙过大等情况。

作为一名专业的 Android 开发人员，能够同时兼顾手机和平板的开发是我们必须做到的事情。Android 自 3.0 版本开始引入了碎片的概念，它可以让界面在平板上更好地展示，下面我们就来一起学习一下。

4.1 碎片是什么

碎片（Fragment）是一种可以嵌入在活动当中的 UI 片段，它能让程序更加合理和充分地利用大屏幕的空间，因而在平板上应用得非常广泛。虽然碎片对你来说应该是个全新的概念，但我相信你学习起来应该毫不费力，因为它和活动实在是太像了，同样都能包含布局，同样都有自己的生命周期。你甚至可以将碎片理解成一个迷你型的活动，虽然这个迷你型的活动有可能和普通的活动是一样大的。

那么究竟要如何使用碎片才能充分地利用平板屏幕的空间呢？想象我们正在开发一个新闻应用，其中一个界面使用 RecyclerView 展示了一组新闻的标题，当点击了其中一个标题时，就打开另一个界面显示新闻的详细内容。如果是在手机中设计，我们可以将新闻标题列表放在一个活动中，将新闻的详细内容放在另一个活动中，如图 4.1 所示。

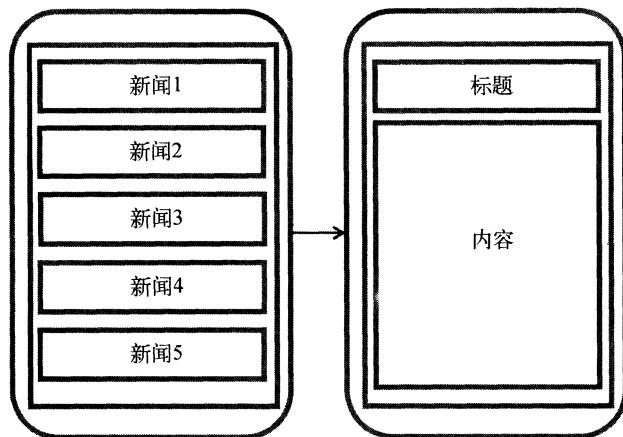


图 4.1 手机的设计方案

可是如果在平板上也这么设计，那么新闻标题列表将会被拉长至填充满整个平板的屏幕，而新闻的标题一般都不会太长，这样将会导致界面上有大量的空白区域，如图 4.2 所示。

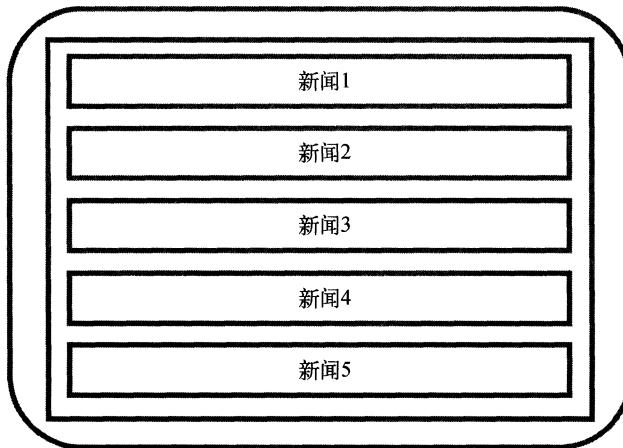


图 4.2 平板的新闻列表

因此，更好的设计方案是将新闻标题列表界面和新闻详细内容界面分别放在两个碎片中，然后在同一个活动里引入这两个碎片，这样就可以将屏幕空间充分地利用起来了，如图 4.3 所示。

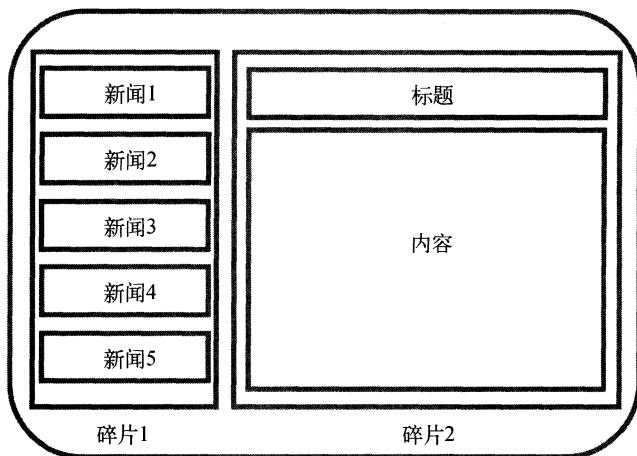


图 4.3 平板的双页设计

4.2 碎片的使用方式

介绍了这么多抽象的东西，也是时候学习一下碎片的具体用法了。你已经知道，碎片通常都是在平板开发中使用的，因此我们首先要做的就是创建一个平板模拟器。创建模拟器的方法我们在第1章已经学过了，创建完成后启动平板模拟器，效果如图4.4所示。

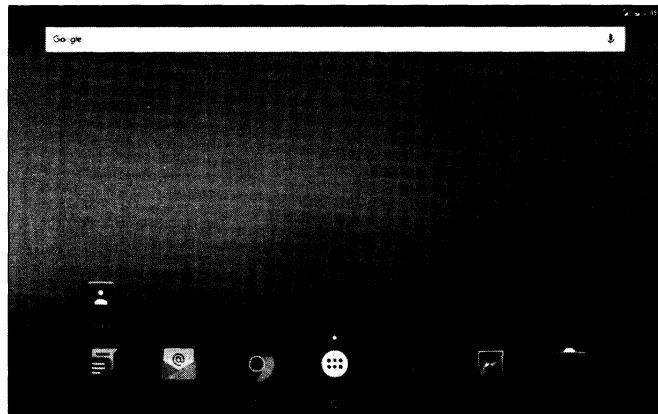


图 4.4 平板模拟器的运行效果

好了，准备工作都完成了，接着新建一个 FragmentTest 项目，然后开始我们的碎片探索之旅吧。

4.2.1 碎片的简单用法

这里我们准备先写一个最简单的碎片示例来练练手，在一个活动中添加两个碎片，并让这

两个碎片平分活动空间。

新建一个左侧碎片布局 left_fragment.xml，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/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:text="Button"
    />

</LinearLayout>
```

这个布局非常简单，只放置了一个按钮，并让它水平居中显示。然后新建右侧碎片布局 right_fragment.xml，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:background="#00ff00"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="20sp"
        android:text="This is right fragment"
    />

</LinearLayout>
```

可以看到，我们将这个布局的背景色设置成了绿色，并放置了一个 TextView 用于显示一段文本。

接着新建一个 `LeftFragment` 类，并让它继承自 `Fragment`。注意，这里可能会有两个不同包下的 `Fragment` 供你选择，一个是系统内置的 `android.app.Fragment`，一个是 `support-v4` 库中的 `android.support.v4.app.Fragment`。这里我强烈建议你使用 `support-v4` 库中的 `Fragment`，因为它可以让碎片在所有 Android 系统版本中保持功能一致性。比如说在 `Fragment` 中嵌套使用 `Fragment`，这个功能是在 Android 4.2 系统中才开始支持的，如果你使用的是系统内置的 `Fragment`，那么很遗憾，4.2 系统之前的设备运行你的程序就会崩溃。而使用 `support-v4` 库中的 `Fragment` 就不会出现这个问题，只要你保证使用的是最新的 `support-v4` 库就可以了。另外，我们并不需要在 `build.gradle` 文件中添加 `support-v4` 库的依赖，因为 `build.gradle` 文件中已经添加了 `appcompat-v7`

库的依赖，而这个库会将 support-v4 库也一起引入进来。

现在编写一下 `LeftFragment` 中的代码，如下所示：

```
public class LeftFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.left_fragment, container, false);
        return view;
    }

}
```

这里仅仅是重写了 `Fragment` 的 `onCreateView()` 方法，然后在这个方法中通过 `LayoutInflater` 的 `inflate()` 方法将刚才定义的 `left_fragment` 布局动态加载进来，整个方法简单明了。接着我们用同样的方法再新建一个 `RightFragment`，代码如下所示：

```
public class RightFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.right_fragment, container, false);
        return view;
    }

}
```

基本上代码都是相同的，相信已经没有必要再做什么解释了。接下来修改 `activity_main.xml` 中的代码，如下所示：

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

    <fragment
        android:id="@+id/left_fragment"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/right_fragment"
        android:name="com.example.fragmenttest.RightFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

</LinearLayout>
```

可以看到，我们使用了`<fragment>`标签在布局中添加碎片，其中指定的大多数属性都是你熟悉的，只不过这里还需要通过`android:name`属性来显式指明要添加的碎片类名，注意一定要将类的包名也加上。

这样最简单的碎片示例就已经写好了，现在运行一下程序，效果如图 4.5 所示。

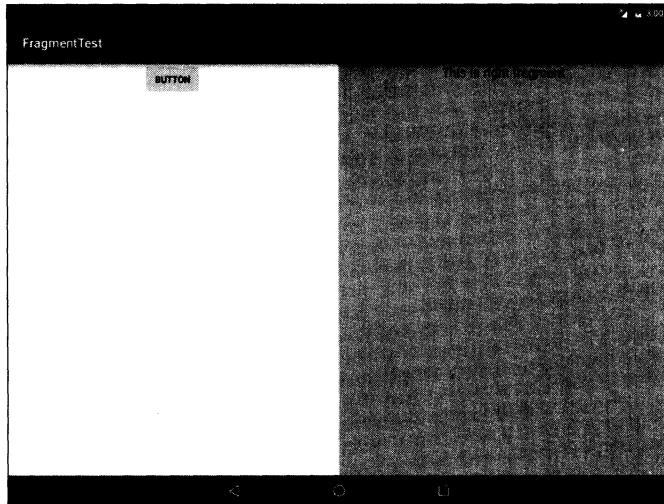


图 4.5 碎片的简单运行效果

正如我们所期待的一样，两个碎片平分了整个活动的布局。不过这个例子实在是太简单了，在真正的项目中很难有什么实际的作用，因此我们马上来看一看，关于碎片更加高级的使用技巧。

4.2.2 动态添加碎片

在上一节当中，你已经学会了在布局文件中添加碎片的方法，不过碎片真正的强大之处在于，它可以在程序运行时动态地添加到活动当中。根据具体情况来动态地添加碎片，你就可以将程序界面定制得更加多样化。

我们还是在上一节代码的基础上继续完善，新建`another_right_fragment.xml`，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:background="#ffff00"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:textSize="20sp"
```

```

        android:text="This is another right fragment"
    />

</LinearLayout>

```

这个布局文件的代码和 right_fragment.xml 中的代码基本相同，只是将背景色改成了黄色，并将显示的文字改了改。然后新建 AnotherRightFragment 作为另一个右侧碎片，代码如下所示：

```

public class AnotherRightFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.another_right_fragment, container,
                                     false);
        return view;
    }

}

```

代码同样非常简单，在 onCreateView() 方法中加载了刚刚创建的 another_right_fragment 布局。这样我们就准备好了另一个碎片，接下来看一下如何将它动态地添加到活动当中。修改 activity_main.xml，代码如下所示：

```

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

    <fragment
        android:id="@+id/left_fragment"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/right_layout"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" >
    </FrameLayout>

</LinearLayout>

```

可以看到，现在将右侧碎片替换成了一个 FrameLayout 中，还记得这个布局吗？在上一章中我们学过，这是 Android 中最简单的一种布局，所有的控件默认都会摆放在布局的左上角。由于这里仅需要在布局里放入一个碎片，不需要任何定位，因此非常适合使用 FrameLayout。

下面我们将向 FrameLayout 里添加内容，从而实现动态添加碎片的功能。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 button = (Button) findViewById(R.id.button);
        button.setOnClickListener(this);
        replaceFragment(new RightFragment());
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.button:
                replaceFragment(new AnotherRightFragment());
                break;
            default:
                break;
        }
    }

    private void replaceFragment(Fragment fragment) {
        FragmentManager fragmentManager = getSupportFragmentManager();
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        transaction.replace(R.id.right_layout, fragment);
        transaction.commit();
    }
}

```

可以看到，首先我们给左侧碎片中的按钮注册了一个点击事件，然后调用 `replaceFragment()` 方法动态添加了 `RightFragment` 这个碎片。当点击左侧碎片中的按钮时，又会调用 `replaceFragment()` 方法将右侧碎片替换成 `AnotherRightFragment`。结合 `replaceFragment()` 方法中的代码可以看出，动态添加碎片主要分为 5 步。

- (1) 创建待添加的碎片实例。
- (2) 获取 `FragmentManager`，在活动中可以直接通过调用 `getSupportFragmentManager()` 方法得到。
- (3) 开启一个事务，通过调用 `beginTransaction()` 方法开启。
- (4) 向容器内添加或替换碎片，一般使用 `replace()` 方法实现，需要传入容器的 id 和待添加的碎片实例。
- (5) 提交事务，调用 `commit()` 方法来完成。

这样就完成了在活动中动态添加碎片的功能，重新运行程序，可以看到和之前相同的界面，然后点击一下按钮，效果如图 4.6 所示。

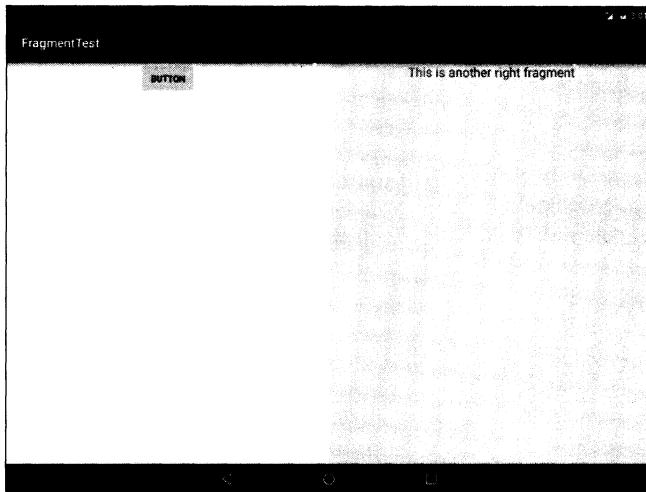


图 4.6 动态添加碎片的效果

4.2.3 在碎片中模拟返回栈

在上一小节中，我们成功实现了向活动中动态添加碎片的功能，不过你尝试一下就会发现，通过点击按钮添加了一个碎片之后，这时按下 Back 键程序就会直接退出。如果这里我们想模仿类似于返回栈的效果，按下 Back 键可以回到上一个碎片，该如何实现呢？

其实很简单，`FragmentTransaction` 中提供了一个 `addToBackStack()` 方法，可以用于将一个事务添加到返回栈中，修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    ...

    private void replaceFragment(Fragment fragment) {
        FragmentManager fragmentManager = getSupportFragmentManager();
        FragmentTransaction transaction = fragmentManager.beginTransaction();
        transaction.replace(R.id.right_layout, fragment);
        transaction.addToBackStack(null);
        transaction.commit();
    }
}
```

这里我们在事务提交之前调用了 `FragmentTransaction` 的 `addToBackStack()` 方法，它可以接收一个名字用于描述返回栈的状态，一般传入 `null` 即可。现在重新运行程序，并点击按钮将 `AnotherRightFragment` 添加到活动中，然后按下 Back 键，你会发现程序并没有退出，而是回到了 `RightFragment` 界面，继续按下 Back 键，`RightFragment` 界面也会消失，再次按下 Back 键，程序才会退出。

4.2.4 碎片和活动之间进行通信

虽然碎片都是嵌入在活动中显示的，可是实际上它们的关系并没有那么亲密。你可以看出，碎片和活动都是各自存在于一个独立的类当中的，它们之间并没有那么明显的方式来直接进行通信。如果想要在活动中调用碎片里的方法，或者在碎片中调用活动里的方法，应该如何实现呢？

为了方便碎片和活动之间进行通信，`FragmentManager` 提供了一个类似于 `findViewById()` 的方法，专门用于从布局文件中获取碎片的实例，代码如下所示：

```
RightFragment rightFragment = (RightFragment) getFragmentManager()
    .findFragmentById(R.id.right_fragment);
```

调用 `FragmentManager` 的 `findFragmentById()` 方法，可以在活动中得到相应碎片的实例，然后就能轻松地调用碎片里的方法了。

掌握了如何在活动中调用碎片里的方法，那在碎片中又该怎样调用活动里的方法呢？其实这就更简单了，在每个碎片中都可以通过调用 `getActivity()` 方法来得到和当前碎片相关联的活动实例，代码如下所示：

```
MainActivity activity = (MainActivity) getActivity();
```

有了活动实例之后，在碎片中调用活动里的方法就变得轻而易举了。另外当碎片中需要使用 `Context` 对象时，也可以使用 `getActivity()` 方法，因为获取到的活动本身就是一个 `Context` 对象。

这时不知道你心中会不会产生一个疑问：既然碎片和活动之间的通信问题已经解决了，那么碎片和碎片之间可不可以进行通信呢？

说实在的，这个问题并没有看上去那么复杂，它的基本思路非常简单，首先在一个碎片中可以得到与它相关联的活动，然后再通过这个活动去获取另外一个碎片的实例，这样也就实现了不同碎片之间的通信功能，因此这里我们的答案是肯定的。

4.3 碎片的生命周期

和活动一样，碎片也有自己的生命周期，并且它和活动的生命周期实在是太像了，我相信你很快就能学会，下面我们马上就来看一下。

4.3.1 碎片的状态和回调

还记得每个活动在其生命周期内可能会有哪几种状态吗？没错，一共有运行状态、暂停状态、停止状态和销毁状态这 4 种。类似地，每个碎片在其生命周期内也可能会经历这几种状态，只不过在一些细小的地方会有部分区别。

1. 运行状态

当一个碎片是可见的，并且它所关联的活动正处于运行状态时，该碎片也处于运行状态。

2. 暂停状态

当一个活动进入暂停状态时（由于另一个未占满屏幕的活动被添加到了栈顶），与它相关联的可见碎片就会进入到暂停状态。

3. 停止状态

当一个活动进入停止状态时，与它相关联的碎片就会进入到停止状态，或者通过调用 FragmentTransaction 的 `remove()`、`replace()` 方法将碎片从活动中移除，但如果在事务提交之前调用 `addToBackStack()` 方法，这时的碎片也会进入到停止状态。总的来说，进入停止状态的碎片对用户来说是完全不可见的，有可能会被系统回收。

4. 销毁状态

碎片总是依附于活动而存在的，因此当活动被销毁时，与它相关联的碎片就会进入到销毁状态。或者通过调用 FragmentTransaction 的 `remove()`、`replace()` 方法将碎片从活动中移除，但在事务提交之前并没有调用 `addToBackStack()` 方法，这时的碎片也会进入到销毁状态。

结合之前的活动状态，相信你理解起来应该毫不费力吧。同样地，Fragment 类中也提供了一系列的回调方法，以覆盖碎片生命周期的每个环节。其中，活动中有的回调方法，碎片中几乎都有，不过碎片还提供了一些附加的回调方法，那我们就重点看一下这几个回调。

- `onAttach()`。当碎片和活动建立关联的时候调用。
- `onCreateView()`。为碎片创建视图（加载布局）时调用。
- `onActivityCreated()`。确保与碎片相关联的活动一定已经创建完毕的时候调用。
- `onDestroyView()`。当与碎片关联的视图被移除的时候调用。
- `onDetach()`。当碎片和活动解除关联的时候调用。

碎片完整的生命周期示意图可参考图 4.7，图片源自 Android 官网。

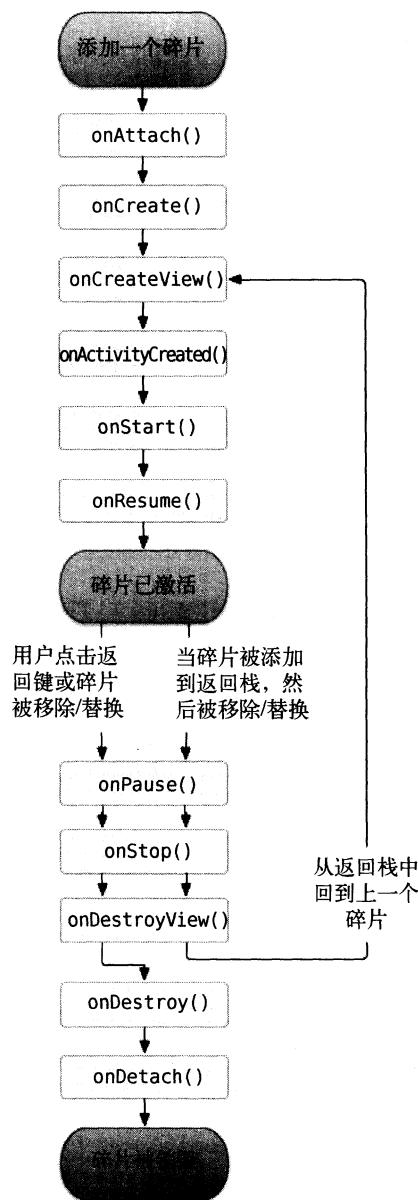


图 4.7 碎片的生命周期

4.3.2 体验碎片的生命周期

为了让你能够更加直观地体验碎片的生命周期，我们还是通过一个例子来实践一下。例子很简单，仍然是在 FragmentTest 项目的基础上改动的。

修改 RightFragment 中的代码，如下所示：

```
public class RightFragment extends Fragment {

    public static final String TAG = "RightFragment";

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        Log.d(TAG, "onAttach");
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        Log.d(TAG, "onCreateView");
        View view = inflater.inflate(R.layout.right_fragment, container, false);
        return view;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        Log.d(TAG, "onActivityCreated");
    }

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

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

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

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

```

    }

    @Override
    public void onDestroyView() {
        super.onDestroyView();
        Log.d(TAG, "onDestroyView");
    }

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

    @Override
    public void onDetach() {
        super.onDetach();
        Log.d(TAG, "onDetach");
    }

}

```

我们在 RightFragment 中的每一个回调方法里都加入了打印日志的代码，然后重新运行程序，这时观察 logcat 中的打印信息，如图 4.8 所示。

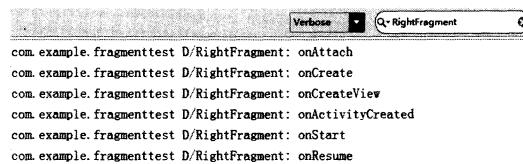


图 4.8 启动程序时的打印日志

可以看到，当 RightFragment 第一次被加载到屏幕上时，会依次执行 `onAttach()`、`onCreate()`、`onCreateView()`、`onActivityCreated()`、`onStart()` 和 `onResume()` 方法。然后点击 LeftFragment 中的按钮，此时打印信息如图 4.9 所示。

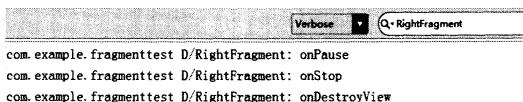


图 4.9 替换成 AnotherRightFragment 时的打印日志

由于 AnotherRightFragment 替换了 RightFragment，此时的 RightFragment 进入了停止状态，因此 `onPause()`、`onStop()` 和 `onDestroyView()` 方法会得到执行。当然如果在替换的时候没有调用 `addToBackStack()` 方法，此时的 RightFragment 就会进入销毁状态，`onDestroy()` 和 `onDetach()` 方法就会得到执行。

接着按下 Back 键，RightFragment 会重新回到屏幕，打印信息如图 4.10 所示。

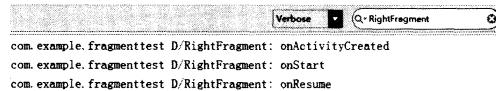


图 4.10 返回 RightFragment 时的打印日志

由于 RightFragment 重新回到了运行状态，因此 `onActivityCreated()`、`onStart()` 和 `onResume()` 方法会得到执行。注意此时 `onCreate()` 和 `onCreateView()` 方法并不会执行，因为我们借助了 `addToBackStack()` 方法使得 RightFragment 和它的视图并没有销毁。

再次按下 Back 键退出程序，打印信息如图 4.11 所示。

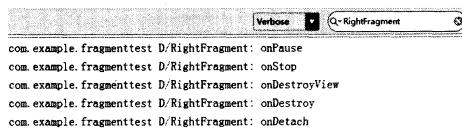


图 4.11 退出程序时的打印日志

依次会执行 `onPause()`、`onStop()`、`onDestroyView()`、`onDestroy()` 和 `onDetach()` 方法，最终将活动和碎片一起销毁。这样碎片完整的生命周期你也体验了一遍，是不是理解得更加深刻了？

另外值得一提的是，在碎片中你也是可以通过 `onSaveInstanceState()` 方法来保存数据的，因为进入停止状态的碎片有可能在系统内存不足的时候被回收。保存下来的数据在 `onCreate()`、`onCreateView()` 和 `onActivityCreated()` 这 3 个方法中你都可以重新得到，它们都含有一个 `Bundle` 类型的 `savedInstanceState` 参数。具体的代码我就不在这里给出了，如果你忘记了该如何编写，可以参考 2.4.5 小节。

4.4 动态加载布局的技巧

虽然动态添加碎片的功能很强大，可以解决很多实际开发中的问题，但是它毕竟只是在一个布局文件中进行一些添加和替换操作。如果程序能够根据设备的分辨率或屏幕大小在运行时来决定加载哪个布局，那我们可发挥的空间就更多了。因此本节我们就来探讨一下 Android 中动态加载布局的技巧。

4.4.1 使用限定符

如果你经常使用平板电脑，应该会发现现在很多的平板应用都采用的是双页模式（程序会在左侧的面板上显示一个包含子项的列表，在右侧的面板上显示内容），因为平板电脑的屏幕足够大，完全可以同时显示下两页的内容，但手机的屏幕一次就只能显示一页的内容，因此两个页面需要分开显示。

那么怎样才能在运行时判断程序应该是使用双页模式还是单页模式呢？这就需要借助限定符（Qualifiers）来实现了。下面我们通过一个例子来学习一下它的用法，修改 FragmentTest 项目中的 activity_main.xml 文件，代码如下所示：

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

    <fragment
        android:id="@+id/left_fragment"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</LinearLayout>
```

这里将多余的代码都删掉，只留下一个左侧碎片，并让它充满整个父布局。接着在 res 目录下新建 layout-large 文件夹，在这个文件夹下新建一个布局，也叫作 activity_main.xml，代码如下所示：

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

    <fragment
        android:id="@+id/left_fragment"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/right_fragment"
        android:name="com.example.fragmenttest.RightFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />

</LinearLayout>
```

可以看到，layout/activity_main 布局只包含了一个碎片，即单页模式，而 layout-large/activity_main 布局包含了两个碎片，即双页模式。其中 large 就是一个限定符，那些屏幕被认为是 large 的设备就会自动加载 layout-large 文件夹下的布局，而小屏幕的设备则还是会加载 layout 文件夹下的布局。

然后将 MainActivity 中 replaceFragment() 方法里的代码注释掉，并在平板模拟器上重新运行程序，效果如图 4.12 所示。

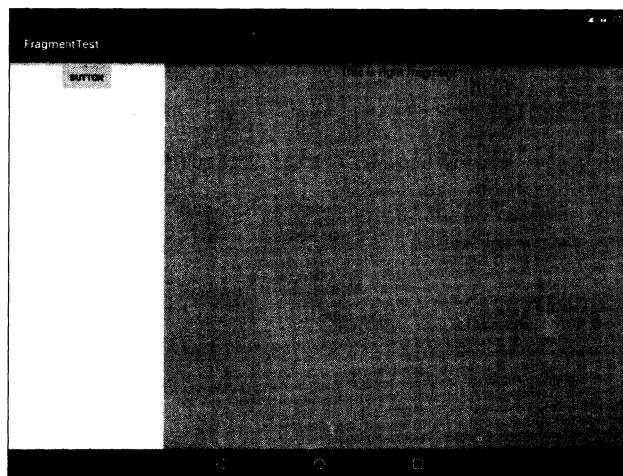


图 4.12 双页模式运行效果

再启动一个手机模拟器，并在这个模拟器上重新运行程序，效果如图 4.13 所示。

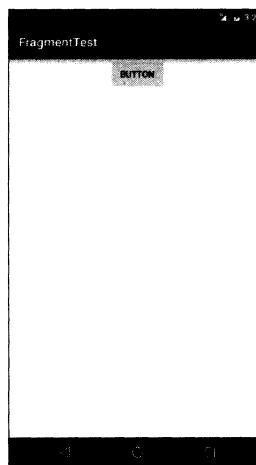


图 4.13 单页模式运行效果

这样我们就实现了在程序运行时动态加载布局的功能。

Android 中一些常见的限定符可以参考下表。

屏幕特征	限定符	描述
大小	small	提供给小屏幕设备的资源
	normal	提供给中等屏幕设备的资源
	large	提供给大屏幕设备的资源
	xlarge	提供给超大屏幕设备的资源

(续)

屏幕特征	限定符	描述
分辨率	ldpi	提供给低分辨率设备的资源（120dpi以下）
	mdpi	提供给中等分辨率设备的资源（120dpi~160dpi）
	hdpi	提供给高分辨率设备的资源（160dpi~240dpi）
	xhdpi	提供给超高分辨率设备的资源（240dpi~320dpi）
	xxhdpi	提供给超超高分辨率设备的资源（320dpi~480dpi）
方向	land	提供给横屏设备的资源
	port	提供给竖屏设备的资源

4.4.2 使用最小宽度限定符

在上一小节中我们使用 `large` 限定符成功解决了单页双页的判断问题，不过很快又有一个新的问题出现了，`large` 到底是指多大呢？有的时候我们希望可以更加灵活地为不同设备加载布局，不管它们是不是被系统认定为 `large`，这时就可以使用最小宽度限定符（Smallest-width Qualifier）了。

最小宽度限定符允许我们对屏幕的宽度指定一个最小值（以 `dp` 为单位），然后以这个最小值为临界点，屏幕宽度大于这个值的设备就加载一个布局，屏幕宽度小于这个值的设备就加载另一个布局。

在 `res` 目录下新建 `layout-sw600dp` 文件夹，然后在这个文件夹下新建 `activity_main.xml` 布局，代码如下所示：

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

    <fragment
        android:id="@+id/left_fragment"
        android:name="com.example.fragmenttest.LeftFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <fragment
        android:id="@+id/right_fragment"
        android:name="com.example.fragmenttest.RightFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" />

</LinearLayout>
```

这就意味着，当程序运行在屏幕宽度大于 `600dp` 的设备上时，会加载 `layout-sw600dp/activity_main` 布局，当程序运行在屏幕宽度小于 `600dp` 的设备上时，则仍然加载默认的 `layout/activity_main` 布局。

4.5 碎片的最佳实践——一个简易版的新闻应用

现在你已经将关于碎片的重要知识点都掌握得差不多了，不过在灵活运用方面可能还有些欠缺，因此下面该进入我们本章的最佳实践环节了。

前面有提到过，碎片很多时候都是在平板开发当中使用的，主要是为了解决屏幕空间不能充分利用的问题。那是不是就表明，我们开发的程序都需要提供一个手机版和一个 Pad 版呢？确实有不少公司都是这么做的，但是这样会浪费很多的人力物力。因为维护两个版本的代码成本很高，每当增加什么新功能时，需要在两份代码里各写一遍，每当发现一个 bug 时，需要在两份代码里各修改一次。因此今天我们最佳实践的内容就是，教你如何编写同时兼容手机和平板的应用程序。

还记得在本章开始的时候提到过的一个新闻应用吗？现在我们就将运用本章中所学的知识来编写一个简易版的新闻应用，并且要求它是可以同时兼容手机和平板的。新建好一个 FragmentBestPractice 项目，然后开始动手吧！

由于待会在编写新闻列表时会使用到 RecyclerView，因此首先需要在 app/build.gradle 当中添加依赖库，如下所示：

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

接下来我们要准备好一个新闻的实体类，新建类 News，代码如下所示：

```
public class News {  
  
    private String title;  
  
    private String content;  
  
    public String getTitle() {  
        return title;  
    }  
  
    public void setTitle(String title) {  
        this.title = title;  
    }  
  
    public String getContent() {  
        return content;  
    }  
  
    public void setContent(String content) {  
        this.content = content;  
    }  
}
```

News 类的代码还是比较简单的，title 字段表示新闻标题，content 字段表示新闻内容。接着新建布局文件 news_content_frag.xml，用于作为新闻内容的布局：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <LinearLayout  
        android:id="@+id/visibility_layout"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:orientation="vertical"  
        android:visibility="invisible" >  
  
        <TextView  
            android:id="@+id/news_title"  
            android:layout_width="match_parent"  
            android:layout_height="wrap_content"  
            android:gravity="center"  
            android:padding="10dp"  
            android:textSize="20sp" />  
  
        <View  
            android:layout_width="match_parent"  
            android:layout_height="1dp"  
            android:background="#000" />  
  
        <TextView  
            android:id="@+id/news_content"  
            android:layout_width="match_parent"  
            android:layout_height="0dp"  
            android:layout_weight="1"  
            android:padding="15dp"  
            android:textSize="18sp" />  
  
    </LinearLayout>  
  
    <View  
        android:layout_width="1dp"  
        android:layout_height="match_parent"  
        android:layout_alignParentLeft="true"  
        android:background="#000" />  
  
</RelativeLayout>
```

新闻内容的布局主要可以分为两个部分，头部部分显示新闻标题，正文部分显示新闻内容，中间使用一条细线分隔开。这里的细线是利用 View 来实现的，将 View 的宽或高设置为 1dp，再通过 background 属性给细线设置一下颜色就可以了。这里我们把细线设置成黑色。

然后再新建一个 NewsContentFragment 类，继承自 Fragment，代码如下所示：

```
public class NewsContentFragment extends Fragment {
```

```

private View view;

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    view = inflater.inflate(R.layout.news_content_frag, container, false);
    return view;
}

public void refresh(String newsTitle, String newsContent) {
    View visibilityLayout = view.findViewById(R.id.visibility_layout);
    visibilityLayout.setVisibility(View.VISIBLE);
    TextView newsTitleText = (TextView) view.findViewById(R.id.news_title);
    TextView newsContentText = (TextView) view.findViewById(R.id.news_content);
    newsTitleText.setText(newsTitle); // 刷新新闻的标题
    newsContentText.setText(newsContent); // 刷新新闻的内容
}
}

```

首先在 `onCreateView()` 方法里加载了我们刚刚创建的 `news_content_frag` 布局，这个没什么好解释的。接下来又提供了一个 `refresh()` 方法，这个方法就是用于将新闻的标题和内容显示在界面上的。可以看到，这里通过 `findViewById()` 方法分别获取到新闻标题和内容的控件，然后将方法传递进来的参数设置进去。

这样我们就把新闻内容的碎片和布局都创建好了，但是它们都是在双页模式中使用的，如果想在单页模式中使用的话，我们还需要再创建一个活动。右击 `com.example.fragmentbestpractice` 包 → New → Activity → Empty Activity，新建一个 `NewsContentActivity`，并将布局名指定成 `news_content`，然后修改 `news_content.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">

    <fragment
        android:id="@+id/news_content_fragment"
        android:name="com.example.fragmentbestpractice.NewsContentFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    />

</LinearLayout>

```

这里我们充分发挥了代码的复用性，直接在布局中引入了 `NewsContentFragment`，这样也就相当于把 `news_content_frag` 布局的内容自动加了进来。

然后修改 `NewsContentActivity` 中的代码，如下所示：

```
public class NewsContentActivity extends AppCompatActivity {
```

```

public static void actionStart(Context context, String newsTitle, String
newsContent) {
    Intent intent = new Intent(context, NewsContentActivity.class);
    intent.putExtra("news_title", newsTitle);
    intent.putExtra("news_content", newsContent);
    context.startActivity(intent);
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.news_content);
    String newsTitle = getIntent().getStringExtra("news_title"); // 获取传入的新
   闻标题
    String newsContent = getIntent().getStringExtra("news_content"); // 获取传入的新闻内
    容
    NewsContentFragment newsContentFragment = (NewsContentFragment)
    getSupportFragmentManager().findFragmentById(R.id.news_content_fragment);
    newsContentFragment.refresh(newsTitle, newsContent); // 刷新 NewsContent-
    Fragment 界面
}

}

```

可以看到，在 `onCreate()` 方法中我们通过 `Intent` 获取到了传入的新闻标题和新闻内容，然后调用 `FragmentManager` 的 `findFragmentById()` 方法得到了 `NewsContentFragment` 的实例，接着调用它的 `refresh()` 方法，并将新闻的标题和内容传入，就可以把这些数据显示出来了。注意这里我们还提供了一个 `actionStart()` 方法，还记得它的作用吗？如果忘记的话就再去阅读一遍 2.6.3 小节吧。

接下来还需要再创建一个用于显示新闻列表的布局，新建 `news_title_frag.xml`，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">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/news_title_recycler_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    />

</LinearLayout>

```

这个布局的代码就非常简单了，里面只有一个用于显示新闻列表的 `RecyclerView`。既然要用到 `RecyclerView`，那么就必定少不了子项的布局。新建 `news_item.xml` 作为 `RecyclerView` 子项的布局，代码如下所示：

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

```

    android:id="@+id/news_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:singleLine="true"
    android:ellipsize="end"
    android:textSize="18sp"
    android:paddingLeft="10dp"
    android:paddingRight="10dp"
    android:paddingTop="15dp"
    android:paddingBottom="15dp" />

```

子项的布局也非常简单，只有一个 TextView。仔细观察 TextView，你会发现其中有几个属性是我们之前没有学过的。`android:padding` 表示给控件的周围加上补白，这样不至于让文本内容会紧靠在边缘上。`android:singleLine` 设置为 `true` 表示让这个 TextView 只能单行显示。`android:ellipsize` 用于设定当文本内容超出控件宽度时，文本的缩略方式，这里指定成 `end` 表示在尾部进行缩略。

既然新闻列表和子项的布局都已经创建好了，那么接下来我们就需要一个用于展示新闻列表的地方。这里新建 NewsTitleFragment 作为展示新闻列表的碎片，代码如下所示：

```

public class NewsTitleFragment extends Fragment {

    private boolean isTwoPane;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.news_title_frag, container, false);
        return view;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if (getActivity().findViewById(R.id.news_content_layout) != null) {
            isTwoPane = true; // 可以找到 news_content_layout 布局时，为双页模式
        } else {
            isTwoPane = false; // 找不到 news_content_layout 布局时，为单页模式
        }
    }
}

```

可以看到，NewsTitleFragment 中并没有多少代码，在 `onCreateView()` 方法中加载了 `news_title_frag` 布局，这个没什么好说的。我们注意看一下 `onActivityCreated()` 方法，这个方法通过在活动中能否找到一个 id 为 `news_content_layout` 的 View 来判断当前是双页模式还是单页模式，因此我们需要让这个 id 为 `news_content_layout` 的 View 只在双页模式中才会出现。

那么怎样才能实现这个功能呢？其实并不复杂，只需要借助我们刚刚学过的限定符就可以

了。首先修改 activity_main.xml 中的代码，如下所示：

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

    <fragment
        android:id="@+id/news_title_fragment"
        android:name="com.example.fragmentbestpractice.NewsTitleFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
    />

</FrameLayout>
```

上述代码表示，在单页模式下，只会加载一个新闻标题的碎片。

然后新建 layout-sw600dp 文件夹，在这个文件夹下再新建一个 activity_main.xml 文件，代码如下所示：

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

    <fragment
        android:id="@+id/news_title_fragment"
        android:name="com.example.fragmentbestpractice.NewsTitleFragment"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1" />

    <FrameLayout
        android:id="@+id/news_content_layout"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="3" >

        <fragment
            android:id="@+id/news_content_fragment"
            android:name="com.example.fragmentbestpractice.NewsContentFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </FrameLayout>

</LinearLayout>
```

可以看出，在双页模式下我们同时引入了两个碎片，并将新闻内容的碎片放在了一个 FrameLayout 布局下，而这个布局的 id 正是 news_content_layout。因此，能够找到这个 id 的时候就是双页模式，否则就是单面模式。

现在我们已经将绝大部分的工作都完成了，但还剩下至关重要的一点，就是在 NewsTitle-

Fragment 中通过 RecyclerView 将新闻列表展示出来。我们在 NewsTitleFragment 中新建一个内部类 NewsAdapter 来作为 RecyclerView 的适配器，如下所示：

```
public class NewsTitleFragment extends Fragment {  
  
    private boolean isTwoPane;  
  
    ...  
  
    class NewsAdapter extends RecyclerView.Adapter<NewsAdapter.ViewHolder> {  
  
        private List<News> mNewsList;  
  
        class ViewHolder extends RecyclerView.ViewHolder {  
  
            TextView newsTitleText;  
  
            public ViewHolder(View view) {  
                super(view);  
                newsTitleText = (TextView) view.findViewById(R.id.news_title);  
            }  
  
        }  
  
        public NewsAdapter(List<News> newsList) {  
            mNewsList = newsList;  
        }  
  
        @Override  
        public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {  
            View view = LayoutInflater.from(parent.getContext())  
                .inflate(R.layout.news_item, parent, false);  
            final ViewHolder holder = new ViewHolder(view);  
            view.setOnClickListener(new View.OnClickListener() {  
                @Override  
                public void onClick(View v) {  
                    News news = mNewsList.get(holder.getAdapterPosition());  
                    if (isTwoPane) {  
                        // 如果是双页模式，则刷新 NewsContentFragment 中的内容  
                        NewsContentFragment newsContentFragment =  
                            (NewsContentFragment) getFragmentManager()  
                                .findFragmentById(R.id.news_content_fragment);  
                        newsContentFragment.refresh(news.getTitle(),  
                            news.getContent());  
                    } else {  
                        // 如果是单页模式，则直接启动 NewsContentActivity  
                        NewsContentActivity.actionStart(getActivity(),  
                            news.getTitle(), news.getContent());  
                    }  
                }  
            });  
            return holder;  
        }  
    }  
}
```

```

@Override
public void onBindViewHolder(ViewHolder holder, int position) {
    News news = mNewsList.get(position);
    holder.newsTitleText.setText(news.getTitle());
}

@Override
public int getItemCount() {
    return mNewsList.size();
}

}

```

RecyclerView 的用法你已经相当熟练了，因此这个适配器的代码对你来说应该没有什么难度吧？需要注意的是，之前我们都是将适配器写成一个独立的类，其实也是可以写成内部类的，这里写成内部类的好处就是可以直接访问 NewsTitleFragment 的变量，比如 isTwoPane。

观察一下 onCreateViewHolder()方法中注册的点击事件，首先获取到了点击项的 News 实例，然后通过 isTwoPane 变量来判断当前是单页还是双页模式，如果是单页模式，就启动一个新的活动去显示新闻内容，如果是双页模式，就更新新闻内容碎片里的数据。

现在还剩最后一步收尾工作，就是向 RecyclerView 中填充数据了。修改 NewsTitleFragment 中的代码，如下所示：

```

public class NewsTitleFragment extends Fragment {

    ...

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                           Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.news_title_frag, container, false);
        RecyclerView newsTitleRecyclerView = (RecyclerView) view.findViewById
            (R.id.news_title_recycler_view);
        LinearLayoutManager layoutManager = new LinearLayoutManager(getActivity());
        newsTitleRecyclerView.setLayoutManager(layoutManager);
        NewsAdapter adapter = new NewsAdapter(getNews());
        newsTitleRecyclerView.setAdapter(adapter);
        return view;
    }

    private List<News> getNews() {
        List<News> newsList = new ArrayList<>();
        for (int i = 1; i <= 50; i++) {
            News news = new News();
            news.setTitle("This is news title " + i);
            news.setContent(getRandomLengthContent("This is news content " + i + "."));
            newsList.add(news);
        }
    }
}

```

```

        return newsList;
    }

    private String getRandomLengthContent(String content) {
        Random random = new Random();
        int length = random.nextInt(20) + 1;
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < length; i++) {
            builder.append(content);
        }
        return builder.toString();
    }

    ...
}

}

```

可以看到，`onCreateView()`方法中添加了`RecyclerView`标准的使用方法，在碎片中使用`RecyclerView`和在活动中使用几乎是一模一样的，相信没有什么需要解释的。另外，这里调用了`getNews()`方法来初始化50条模拟新闻数据，同样使用了一个`getRandomLengthContent()`方法来随机生成新闻内容的长度，以保证每条新闻的内容差距比较大，相信你对这个方法肯定不会陌生了。

这样我们所有的编写工作就已经完成了，赶快来运行一下吧！首先在手机模拟器上运行，效果如图4.14所示。

可以看到许多条新闻的标题，然后点击第一条新闻，会启动一个新的活动来显示新闻的内容，效果如图4.15所示。



图4.14 单页模式的新闻列表界面

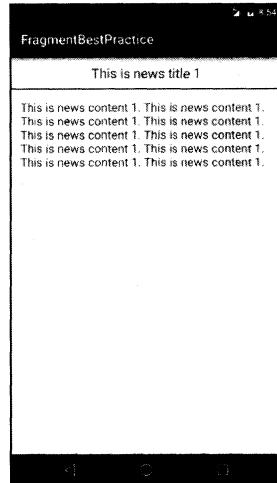


图4.15 单页模式的新闻内容界面

接下来将程序在平板模拟器上运行，同样点击第一条新闻，效果如图4.16所示。

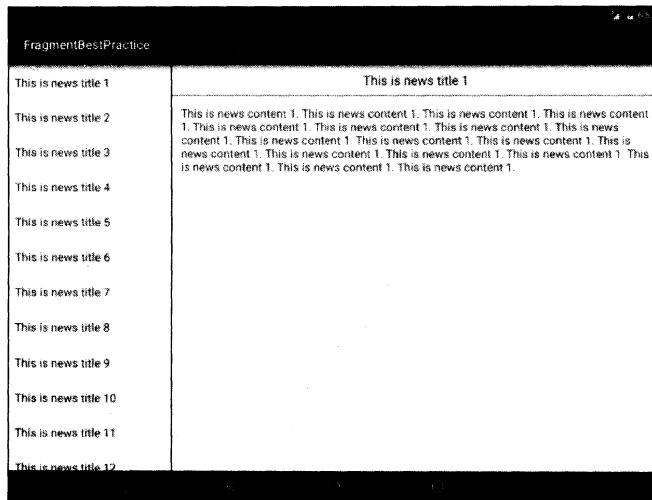


图 4.16 双页模式的新闻标题和内容界面

怎么样？同样的一份代码，在手机和平板上运行却分别是两种完全不同的效果，说明我们程序的兼容性已经相当不错了。通过这个例子，我相信你对碎片的理解一定又加深了很多，现在就让我们一起来总结一下吧。

4.6 小结与点评

你应该可以感觉到，上一节中我们开发的新闻应用，代码复杂度还是有点高的，比起只需要兼容一个终端的应用，我们要考虑的东西多了很多。不过在开发的过程中多付出一些，在以后的代码维护中就可以轻松很多。因此，有时候提前的付出还是很值得的。

我们再来看看本章所学的内容吧，首先你了解了碎片的基本概念以及使用场景，接着通过几个实例掌握了碎片的常见用法，随后又学习了碎片生命周期的相关内容以及动态加载布局的技巧，最后在本章的最佳实践部分将前面所学的内容综合运用了一遍，相信你已经将碎片相关知识点都牢记在心，并可以较为熟练地应用了。

本章其实是具有一个里程碑式的纪念意义的，因为到这里为止，我们已经基本将 Android UI 相关的重要知识点都讲完了。后面在很长一段时间内都不会再系统性地介绍 UI 方面的知识，而是将结合前面所学的 UI 知识来更好地讲解相应章节的内容。那么我们下一章将要学习什么呢？还记得在第 1 章里介绍过的 Android 四大组件吧？目前我们只掌握了活动这一个组件，那么下一章就来学习广播接收器吧。跟上脚步，准备继续前进！

第 5 章

全局大喇叭——详解广播机制

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

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

5.1 广播机制简介

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

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

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

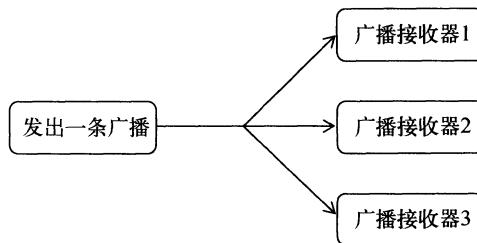


图 5.1 标准广播工作示意图

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

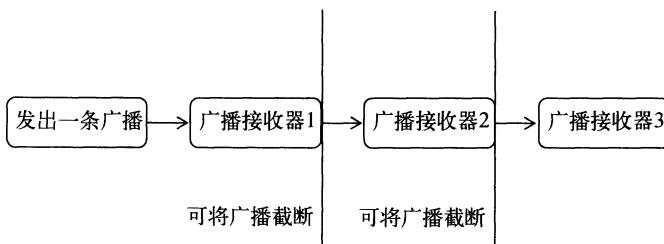


图 5.2 有序广播工作示意图

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

5.2 接收系统广播

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

5.2.1 动态注册监听网络变化

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

那么该如何创建一个广播接收器呢？其实只需要新建一个类，让它继承自 `BroadcastReceiver`，并重写父类的 `onReceive()` 方法就行了。这样当有广播到来时，`onReceive()` 方法

就会得到执行，具体的逻辑就可以在这个方法中处理。

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

```
public class MainActivity extends AppCompatActivity {

    private IntentFilter intentFilter;

    private NetworkChangeReceiver networkChangeReceiver;

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

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

    class NetworkChangeReceiver extends BroadcastReceiver {

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

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

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

监听网络变化的功能。

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

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

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

```
public class MainActivity extends AppCompatActivity {
    ...
    class NetworkChangeReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            ConnectivityManager connectionManager = (ConnectivityManager)
                getSystemService(Context.CONNECTIVITY_SERVICE);
            NetworkInfo networkInfo = connectionManager.getActiveNetworkInfo();
            if (networkInfo != null && networkInfo.isAvailable()) {
                Toast.makeText(context, "network is available",
                    Toast.LENGTH_SHORT).show();
            } else {
                Toast.makeText(context, "network is unavailable",
                    Toast.LENGTH_SHORT).show();
            }
        }
    }
}
```

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

另外，这里有非常重要的一点需要说明，Android 系统为了保护用户设备的安全和隐私，做了严格的规定：如果程序需要进行一些对用户来说比较敏感的操作，就必须在配置文件中声明权限才可以，否则程序将会直接崩溃。比如这里访问系统的网络状态就是需要声明权限的。打开 `AndroidManifest.xml` 文件，在里面加入如下权限就可以访问系统网络状态了：

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

```

package="com.example.broadcasttest"

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

```

这是你第一次遇到权限的问题,其实Android中有许多操作都是需要声明权限才可以进行的,后面我们还会不断使用新的权限。不过目前这个访问系统网络状态的权限还是比较简单的,只需要在AndroidManifest.xml文件中声明一下就可以了,而Android 6.0系统中引入了更加严格的运行时权限,从而能够更好地保证用户设备的安全和隐私,关于这部分内容我们将在第7章中学习。

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

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

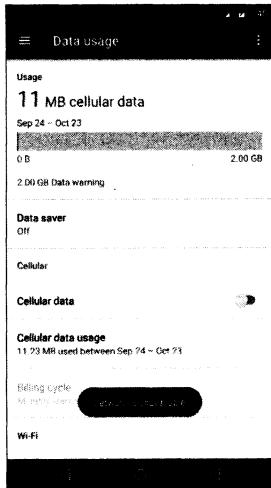


图5.3 禁用系统网络

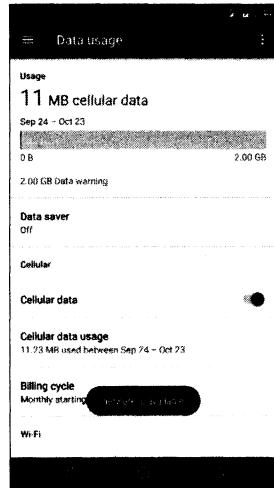


图5.4 启用系统网络

5.2.2 静态注册实现开机启动

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

这里我们准备让程序接收一条开机广播,当收到这条广播时就可以在onReceive()方法里执行相应的逻辑,从而实现开机启动的功能。可以使用Android Studio提供的快捷方式来创建一个广播接收器,右击com.example.broadcasttest包→New→Other→Broadcast Receiver,会弹出如

图 5.5 所示的窗口。

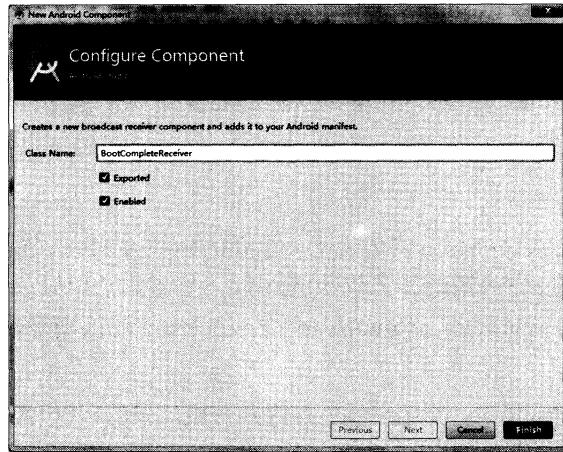


图 5.5 创建广播接收器的窗口

可以看到，这里我们将广播接收器命名为 BootCompleteReceiver， Exported 属性表示是否允许这个广播接收器接收本程序以外的广播， Enabled 属性表示是否启用这个广播接收器。勾选这两个属性，点击 Finish 完成创建。

然后修改 BootCompleteReceiver 中的代码，如下所示：

```
public class BootCompleteReceiver extends BroadcastReceiver {

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

代码非常简单，我们只是在 onReceive() 方法中使用 Toast 弹出一段提示信息。

另外，静态的广播接收器一定要在 AndroidManifest.xml 文件中注册才可以使用，不过由于我们是使用 Android Studio 的快捷方式创建的广播接收器，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.broadcasttest">

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
```

```

    android:theme="@style/AppTheme">
    ...
<receiver
    android:name=".BootCompleteReceiver"
    android:enabled="true"
    android:exported="true">
</receiver>
</application>

</manifest>

```

可以看到，`<application>`标签内出现了一个新的标签`<receiver>`，所有静态的广播接收器都是在这里进行注册的。它的用法其实和`<activity>`标签非常相似，也是通过`android:name`来指定具体注册哪一个广播接收器，而`enabled`和`exported`属性则是根据我们刚才勾选的状态自动生成的。

不过目前`BootCompleteReceiver`还是不能接收到开机广播的，我们还需要对`AndroidManifest.xml`文件进行修改才行，如下所示：

```

<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" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <receiver
            android:name=".BootCompleteReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
        </receiver>
    </application>

</manifest>

```

由于Android系统启动完成后会发出一条值为`android.intent.action.BOOT_COMPLETED`的广播，因此我们在`<intent-filter>`标签里添加了相应的action。另外，监听系统开机广播也是需要声明权限的，可以看到，我们使用`<uses-permission>`标签又加入了一条`android.permission.RECEIVE_BOOT_COMPLETED`权限。

现在重新运行程序后，我们的程序就已经可以接收开机广播了。将模拟器关闭并重新启动，在启动完成之后就会收到开机广播，如图5.6所示。

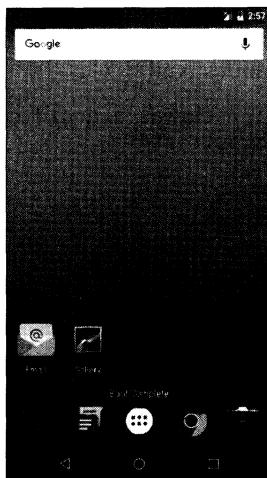


图 5.6 接收系统开机广播

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

5.3 发送自定义广播

现在你已经学会了通过广播接收器来接收系统广播，接下来我们就要学习一下如何在应用程序中发送自定义的广播。前面已经介绍过了，广播主要分为两种类型：标准广播和有序广播，在本节中我们就将通过实践的方式来看一下这两种广播具体的区别。

5.3.1 发送标准广播

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

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

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

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest">
    ...
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <receiver
            android:name=".MyBroadcastReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter>
                <action android:name="com.example.broadcasttest.MY_BROADCAST"/>
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

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

接下来修改 activity_main.xml 中的代码，如下所示：

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

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

</LinearLayout>
```

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

```
public class MainActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new
        Intent("com.example.broadcasttest.MY_BROADCAST");
        sendBroadcast(intent);
    }
});
...
}

}
```

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

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

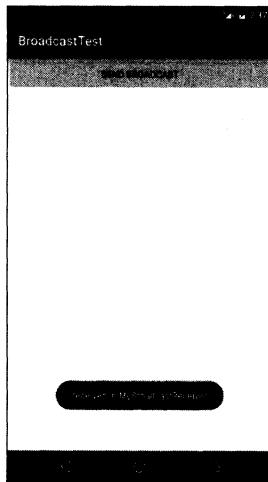


图 5.7 接收到自定义广播

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

5.3.2 发送有序广播

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

需要再新建一个 BroadcastTest2 项目，点击 Android Studio 导航栏→File→New→New Project 进行创建。

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

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

这里仍然是在广播接收器的 onReceive() 方法中弹出了一段文本信息。然后在 AndroidManifest.xml 中对这个广播接收器进行修改，代码如下所示：

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

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

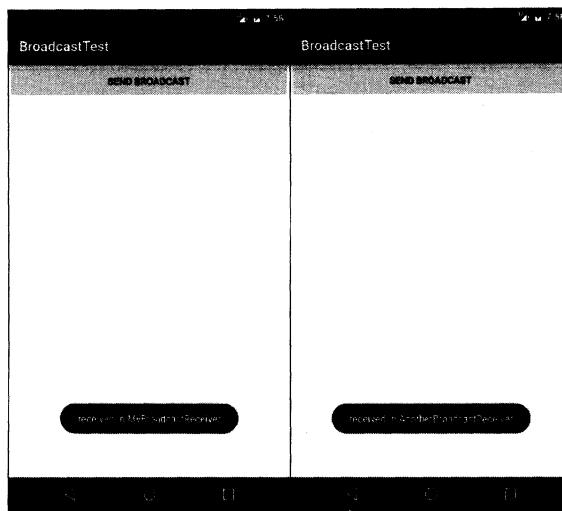


图 5.8 两个程序中都接收到自定义广播

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

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

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

可以看到，发送有序广播只需要改动一行代码，即将 `sendBroadcast()`方法改成 `sendOrderedBroadcast()`方法。`sendOrderedBroadcast()`方法接收两个参数，第一个参数仍然是

`Intent`, 第二个参数是一个与权限相关的字符串, 这里传入 `null` 就行了。现在重新运行程序, 并点击 Send Broadcast 按钮, 你会发现, 两个应用程序仍然都可以接收到这条广播。

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

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

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcasttest2">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
        <receiver
            android:name=".AnotherBroadcastReceiver"
            android:enabled="true"
            android:exported="true">
            <intent-filter android:priority="100">
                <action android:name="com.example.broadcasttest.MY_BROADCAST" />
            </intent-filter>
        </receiver>
    </application>

</manifest>
```

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

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

```
public class MyBroadcastReceiver extends BroadcastReceiver {

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

如果在 `onReceive()` 方法中调用了 `abortBroadcast()` 方法, 就表示将这条广播截断, 后面

的广播接收器将无法再接收到这条广播。现在重新运行程序，并点击一下 Send Broadcast 按钮，你会发现，只有 MyBroadcastReceiver 中的 Toast 信息能够弹出，说明这条广播经过 MyBroadcastReceiver 之后确实是终止传递了。

5.4 使用本地广播

前面我们发送和接收的广播全部属于系统全局广播，即发出的广播可以被其他任何应用程序接收到，并且我们也可以接收来自于其他任何应用程序的广播。这样就很容易引起安全性的问题，比如说我们发送的一些携带关键性数据的广播有可能被其他的应用程序截获，或者其他的应用程序不停地向我们的广播接收器里发送各种垃圾广播。

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

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

```
public class MainActivity extends AppCompatActivity {

    private IntentFilter intentFilter;
    private LocalReceiver localReceiver;
    private LocalBroadcastManager localBroadcastManager;

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

```

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

class LocalReceiver extends BroadcastReceiver {

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

```

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

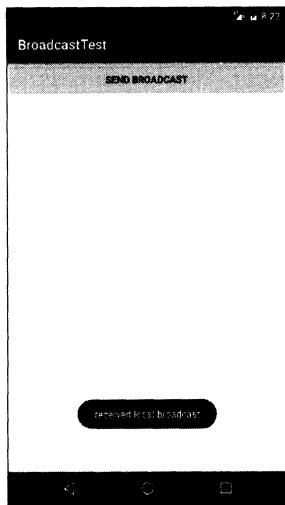


图 5.9 接收到本地广播

可以看到，LocalReceiver 成功接收到了这条本地广播，并通过 Toast 提示了出来。如果你还有兴趣进行实验，可以尝试在 BroadcastTest2 中也去接收 com.example.broadcasttest.

`LOCAL_BROADCAST` 这条广播。答案是显而易见的，肯定无法收到，因为这条广播只会在 `BroadcastTest` 程序内传播。

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

最后我们再来盘点一下使用本地广播的几点优势吧。

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

5.5 广播的最佳实践——实现强制下线功能

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

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

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

```
public class ActivityCollector {  
  
    public static List<Activity> activities = new ArrayList<>();  
  
    public static void addActivity(Activity activity) {  
        activities.add(activity);  
    }  
  
    public static void removeActivity(Activity activity) {  
        activities.remove(activity);  
    }  
  
    public static void finishAll() {  
        for (Activity activity : activities) {  
            if (!activity.isFinishing()) {  
                activity.finish();  
            }  
        }  
    }  
}
```

```
}
```

然后创建 `BaseActivity` 类作为所有活动的父类，代码如下所示：

```
public class BaseActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        ActivityCollector.addActivity(this);  
    }  
  
    @Override  
    protected void onDestroy() {  
        super.onDestroy();  
        ActivityCollector.removeActivity(this);  
    }  
}
```

以上代码都是直接拿之前写好的内容，非常开心。不过从这里开始，就要靠我们自己去动手实现了。首先需要创建一个登录界面的活动，新建 LoginActivity，并让 Android Studio 帮我们自动生成相应的布局文件。然后编辑布局文件 activity_login.xml，代码如下所示：

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

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="60dp">
        <TextView
            android:layout_width="90dp"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:textSize="18sp"
            android:text="Account:" />

        <EditText
            android:id="@+id/account"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_gravity="center_vertical" />
    </LinearLayout>

    <LinearLayout
        android:orientation="horizontal"
```

```

    android:layout_width="match_parent"
    android:layout_height="60dp">
        <TextView
            android:layout_width="90dp"
            android:layout_height="wrap_content"
            android:layout_gravity="center_vertical"
            android:textSize="18sp"
            android:text="Password:" />

        <EditText
            android:id="@+id/password"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:layout_gravity="center_vertical"
            android:inputType="textPassword" />
    </LinearLayout>

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

```

这里我们使用 `LinearLayout` 编写出了一个登录布局，最外层是一个纵向的 `LinearLayout`，里面包含了 3 行直接子元素。第一行是一个横向 `LinearLayout`，用于输入账号信息；第二行也是一个横向的 `LinearLayout`，用于输入密码信息；第三行是一个登录按钮。这个布局文件里面用到的全部都是我们之前学过的内容，相信你理解起来应该不会费劲。

接下来修改 `LoginActivity` 中的代码，如下所示：

```

public class LoginActivity extends BaseActivity {

    private EditText accountEdit;
    private EditText passwordEdit;
    private Button login;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
        accountEdit = (EditText) findViewById(R.id.account);
        passwordEdit = (EditText) findViewById(R.id.password);
        login = (Button) findViewById(R.id.login);
        login.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String account = accountEdit.getText().toString();

```

```
        String password = passwordEdit.getText().toString();
        // 如果账号是 admin 且密码是 123456, 就认为登录成功
        if (account.equals("admin") && password.equals("123456")) {
            Intent intent = new Intent(LoginActivity.this, MainActivity.class);
            startActivity(intent);
            finish();
        } else {
            Toast.makeText(LoginActivity.this, "account or password is invalid", Toast.LENGTH_SHORT).show();
        }
    });
}
```

这里我们模拟了一个非常简单的登录功能。首先要将 LoginActivity 的继承结构改成继承自 BaseActivity，然后调用 `findViewById()` 方法分别获取到账号输入框、密码输入框以及登录按钮的实例。接着在登录按钮的点击事件里面对输入的账号和密码进行判断，如果账号是 admin 并且密码是 123456，就认为登录成功并跳转到 MainActivity，否则就提示用户账号或密码错误。

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

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

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

</LinearLayout>
```

非常简单，只有一个按钮而已，用于触发强制下线功能。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends BaseActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        Button forceOffline = (Button) findViewById(R.id.force_offline);  
        forceOffline.setOnClickListener(new View.OnClickListener() {  
            @Override
```

```

        public void onClick(View v) {
            Intent intent = new Intent("com.example.broadcastbestpractice.
                FORCE_OFFLINE");
            sendBroadcast(intent);
        }
    });
}

```

同样非常简单，不过这里有个重点，我们在按钮的点击事件里面发送了一条广播，广播的值为 `com.example.broadcastbestpractice.FORCE_OFFLINE`，这条广播就是用于通知程序强制用户下线的。也就是说强制用户下线的逻辑并不是写在 `MainActivity` 里的，而是应该写在接收这条广播的广播接收器里面，这样强制下线的功能就不会依附于任何的界面，不管是在程序的任何地方，只需要发出这样一条广播，就可以完成强制下线的操作了。

那么毫无疑问，接下来我们就需要创建一个广播接收器来接收这条强制下线广播，唯一的问题就是，应该在哪里创建呢？由于广播接收器里面需要弹出一个对话框来阻塞用户的正常操作，但如果创建的是一个静态注册的广播接收器，是没有办法在 `onReceive()` 方法里弹出对话框这样的 UI 控件的，而我们显然也不可能在每个活动中都去注册一个动态的广播接收器。

那么到底应该怎么办呢？答案其实很明显，只需要在 `BaseActivity` 中动态注册一个广播接收器就可以了，因为所有的活动都是继承自 `BaseActivity` 的。

修改 `BaseActivity` 中的代码，如下所示：

```

public class BaseActivity extends AppCompatActivity {

    private ForceOfflineReceiver receiver;

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

    @Override
    protected void onResume() {
        super.onResume();
        IntentFilter intentFilter = new IntentFilter();
        intentFilter.addAction("com.example.broadcastbestpractice.FORCE_OFFLINE");
        receiver = new ForceOfflineReceiver();
        registerReceiver(receiver, intentFilter);
    }

    @Override
    protected void onPause() {
        super.onPause();
        if (receiver != null) {
            unregisterReceiver(receiver);
        }
    }
}

```

```

        receiver = null;
    }
}

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

class ForceOfflineReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(final Context context, Intent intent) {
        AlertDialog.Builder builder = new AlertDialog.Builder(context);
        builder.setTitle("Warning");
        builder.setMessage("You are forced to be offline. Please try to login again.");
        builder.setCancelable(false);
        builder.setPositiveButton("OK", new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                ActivityCollector.finishAll(); // 销毁所有活动
                Intent intent = new Intent(context, LoginActivity.class);
                context.startActivity(intent); // 重新启动 LoginActivity
            }
        });
        builder.show();
    }
}
}

```

先来看一下 ForceOfflineReceiver 中的代码，这次 `onReceive()`方法里可不再是仅仅弹出一个 `Toast` 了，而是加入了较多的代码，那我们就来仔细地看看吧。首先肯定是使用 `AlertDialog.Builder` 来构建一个对话框，注意这里一定要调用 `setCancelable()`方法将对话框设为不可取消，否则用户按一下 `Back` 键就可以关闭对话框继续使用程序了。然后使用 `setPositiveButton()`方法来给对话框注册确定按钮，当用户点击了确定按钮时，就调用 `ActivityCollector` 的 `finishAll()`方法来销毁掉所有活动，并重新启动 `LoginActivity` 这个活动。

再来看一下我们是怎么注册 `ForceOfflineReceiver` 这个广播接收器的，可以看到，这里重写了 `onResume()` 和 `onPause()` 这两个生命周期函数，然后分别在这两个方法里注册和取消注册了 `ForceOfflineReceiver`。

那么为什么要这样写呢？之前不都是在 `onCreate()` 和 `onDestroy()` 方法里来注册和取消注册广播接收器的么？这是因为我们始终需要保证只有处于栈顶的活动才能接收到这条强制下线广播，非栈顶的活动不应该也没有必要去接收这条广播，所以写在 `onResume()` 和 `onPause()` 方法里就可以很好地解决这个问题，当一个活动失去栈顶位置时就会自动取消广播接收器的注册。

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

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

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

</manifest>
```

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

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

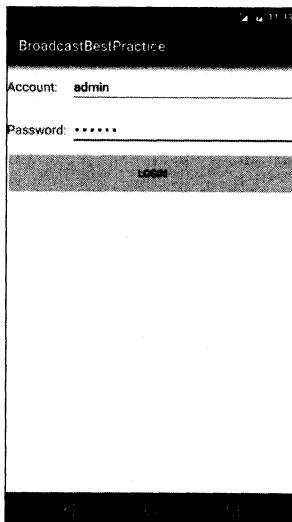


图 5.10 登录界面

如果输入的账号是 admin，密码是 123456，点击登录按钮就会进入到程序的主界面，如图 5.11 所示。这时点击一下发送广播的按钮，就会发出一条强制下线的广播，ForceOfflineReceiver 里收到这条广播后会弹出一个对话框提示用户已被强制下线，如图 5.12 所示。

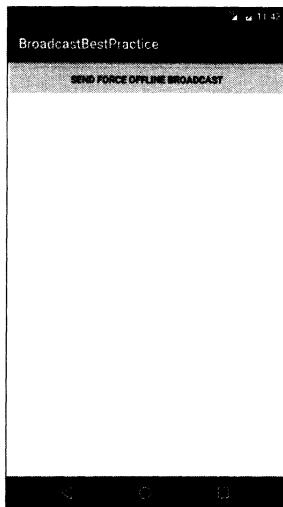


图 5.11 主界面



图 5.12 强制下线提示

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

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

5.6 Git 时间——初识版本控制工具

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

5.6.1 安装 Git

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

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

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

按下回车后输入密码，即可完成 Git 的安装。

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



图 5.13 git for windows 主页

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

5.6.2 创建代码仓库

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

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

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

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

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



图 5.14 查看 git 用户名和邮箱

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

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



图 5.15 切换到 BroadcastBestPractice 项目目录下

然后在这个目录下面输入如下命令：

```
git init
```

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



图 5.16 创建代码仓库

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

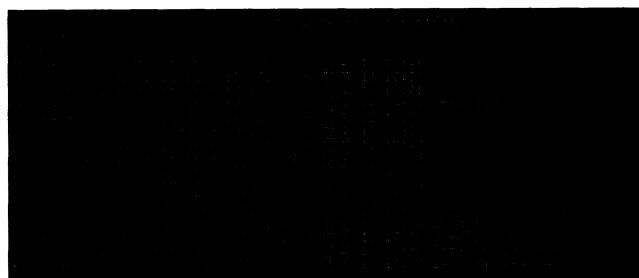


图 5.17 查看.git 文件

如果你想要删除本地仓库，只需要删除这个文件夹就行了。

5.6.3 提交本地代码

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

```
git add build.gradle
```

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

```
git add app
```

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

```
git add .
```

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

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

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

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

5.7 小结与点评

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

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

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

第 6 章

数据存储全方案——详解持久化技术

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

而我们前面章节所编写的众多例子中也有用到各种各样的数据，例如第 3 章最佳实践部分在聊天界面编写的聊天内容，第 5 章最佳实践部分在登录界面输入的账号和密码。这些数据都有一个共同点，即它们都属于瞬时数据。那么什么是瞬时数据呢？就是指那些存储在内存当中，有可能会因为程序关闭或其他原因导致内存被回收而丢失的数据。这对于一些关键性的数据信息来说是绝对不能容忍的，谁都不希望自己刚发出去的一条微博，刷新一下就没了吧。那么怎样才能保证一些关键性的数据不会丢失呢？这就需要用到数据持久化技术了。

6.1 持久化技术简介

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

持久化技术被广泛应用于各种程序设计的领域当中，而本书中要探讨的自然是 Android 中的数据持久化技术。Android 系统中主要提供了 3 种方式用于简单地实现数据持久化功能，即文件存储、SharedPreference 存储以及数据库存储。当然，除了这 3 种方式之外，你还可以将数据保存在手机的 SD 卡中，不过使用文件、SharedPreference 或数据库来保存数据会相对更简单一些，而且比起将数据保存在 SD 卡中会更加地安全。

那么下面我就将对这 3 种数据持久化的方式一一进行详细的讲解。

6.2 文件存储

文件存储是 Android 中最基本的一种数据存储方式，它不对存储的内容进行任何的格式化处理，所有数据都是原封不动地保存到文件当中的，因而它比较适合用于存储一些简单的文本数据或二进制数据。如果你想使用文件存储的方式来保存一些较为复杂的文本数据，就需要定义一套自己的格式规范，这样可以方便之后将数据从文件中重新解析出来。

那么首先我们就来看一看，Android 中是如何通过文件来保存数据的。

6.2.1 将数据存储到文件中

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

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

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

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

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

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

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

</LinearLayout>
```

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

```
public class MainActivity extends AppCompatActivity {

    private EditText edit;

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

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

    public void save(String inputText) {
        FileOutputStream out = null;
        BufferedWriter writer = null;
        try {
```

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

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

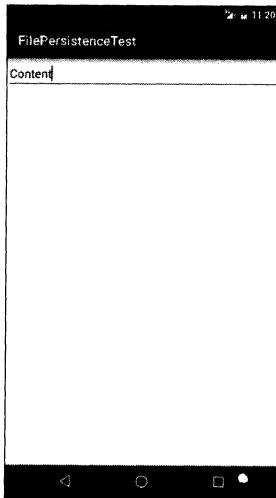


图 6.1 在 `EditText` 中随意输入点内容

然后按下 Back 键关闭程序，这时我们输入的内容就已经保存到文件中了。那么如何才能证实数据确实已经保存成功了呢？我们可以借助 `Android Device Monitor` 工具来查看一下。点击 `Android Studio` 导航栏中的 `Tools→Android`，会看到如图 6.2 所示的工具列表。

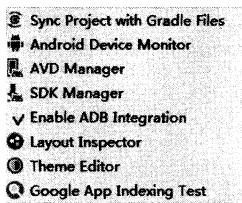


图 6.2 Android 工具列表

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

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

图 6.3 生成的 data 文件

然后点击图 6.4 中左边的按钮可以将这个文件导出到电脑上。



图 6.4 导入导出按钮

使用记事本打开这个文件，里面的内容如图 6.5 所示。

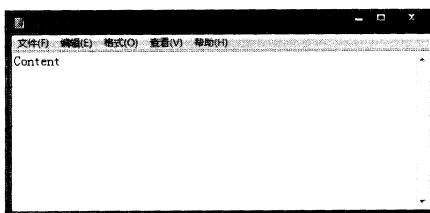


图 6.5 data 文件中的内容

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

不过只是成功将数据保存下来还不够，我们还需要想办法在下次启动程序的时候让这些数据

能够还原到 EditText 中，因此接下来我们就要学习一下如何从文件中读取数据。

6.2.2 从文件中读取数据

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

以下是一段简单的代码示例，展示了如何从文件中读取文本数据：

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

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

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

```
public class MainActivity extends AppCompatActivity {

    private EditText edit;
```

```

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

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

```

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

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

现在重新运行一下程序，刚才保存的 Content 字符串肯定会被填充到 EditText 中，然后编写一点其他的内容，比如在 EditText 中输入 Hello，接着按下 Back 键退出程序，再重新启动程序，这时刚才输入的内容并不会丢失，而是还原到了 EditText 中，如图 6.6 所示。

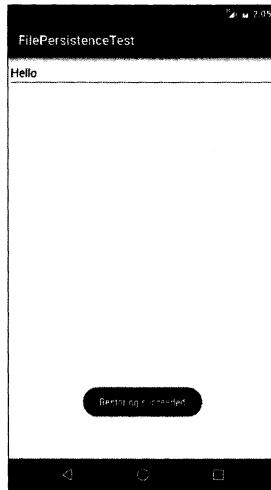


图 6.6 成功还原保存的内容

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

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

6.3 SharedPreferences 存储

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

这样你应该就能明显地感觉到，使用 `SharedPreferences` 来进行数据持久化要比使用文件方便很多，下面我们就来看一下它的具体用法吧。

6.3.1 将数据存储到 `SharedPreferences` 中

要想使用 `SharedPreferences` 来存储数据，首先需要获取到 `SharedPreferences` 对象。Android

中主要提供了3种方法用于得到 SharedPreferences 对象。

1. Context 类中的 getSharedPreferences()方法

此方法接收两个参数，第一个参数用于指定 SharedPreferences 文件的名称，如果指定的文件不存在则会创建一个，SharedPreferences 文件都是存放在 /data/data/<package name>/shared_prefs/ 目录下的。第二个参数用于指定操作模式，目前只有 MODE_PRIVATE 这一种模式可选，它是默认的操作模式，和直接传入 0 效果是相同的，表示只有当前的应用程序才可以对这个 SharedPreferences 文件进行读写。其他几种操作模式均已被废弃，MODE_WORLD_READABLE 和 MODE_WORLD_WRITEABLE 这两种模式是在 Android 4.2 版本中被废弃的，MODE_MULTI_PROCESS 模式是在 Android 6.0 版本中被废弃的。

2. Activity 类中的 getPreferences()方法

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

3. PreferenceManager 类中的 getDefaultSharedPreferences()方法

这是一个静态方法，它接收一个 Context 参数，并自动使用当前应用程序的包名作为前缀来命名 SharedPreferences 文件。得到了 SharedPreferences 对象之后，就可以开始向 SharedPreferences 文件中存储数据了，主要可以分为3步实现。

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

(2) 向 SharedPreferences.Editor 对象中添加数据，比如添加一个布尔型数据就使用 putBoolean() 方法，添加一个字符串则使用 putString() 方法，以此类推。

(3) 调用 apply() 方法将添加的数据提交，从而完成数据存储操作。

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

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

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

</LinearLayout>
```

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

```
public class MainActivity extends AppCompatActivity {

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

可以看到，这里首先给按钮注册了一个点击事件，然后在点击事件中通过 `getSharedPreferences()` 方法指定 SharedPreferences 的文件名为 `data`，并得到了 `SharedPreferences.Editor` 对象。接着向这个对象中添加了 3 条不同类型的数据，最后调用 `apply()` 方法进行提交，从而完成了数据存储的操作。

很简单吧？现在就可以运行一下程序了，进入程序的主界面后，点击一下 `Save data` 按钮。这时的数据应该已经保存成功了，不过为了证实一下，我们还是要借助 File Explorer 来进行查看。打开 Android Device Monitor，并点击 File Explorer 标签页，然后进入到`/data/data/com.example.sharedpreferencestest/shared_prefs/` 目录下，可以看到生成了一个 `data.xml` 文件，如图 6.7 所示。

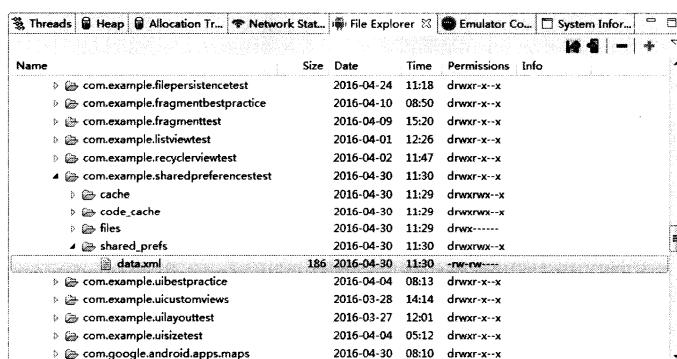


图 6.7 生成的 `data.xml` 文件

接下来，同样是点击导出按钮将这个文件导出到电脑上，并用记事本进行查看，里面的内容如图 6.8 所示。

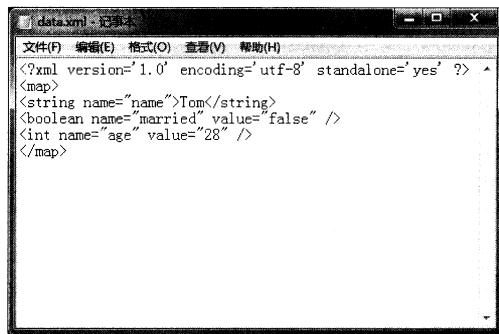


图 6.8 data.xml 文件中的内容

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

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

6.3.2 从 SharedPreferences 中读取数据

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

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

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

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

```

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

```

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

```

public class MainActivity extends AppCompatActivity {

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

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

```

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

现在重新运行一下程序，并点击界面上的 Restore data 按钮，然后查看 logcat 中的打印信息，如图 6.9 所示。

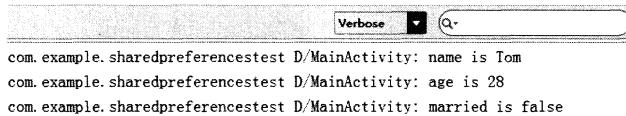


图 6.9 打印 data.xml 中存储的内容

所有之前存储的数据都成功读取出来了！通过这个例子，我们就把 SharedPreferences 存储

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

6.3.3 实现记住密码功能

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

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

    ...

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

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

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

然后修改 LoginActivity 中的代码，如下所示：

```
public class LoginActivity extends BaseActivity {

    private SharedPreferences pref;
    private SharedPreferences.Editor editor;
```

```
private EditText accountEdit;
private EditText passwordEdit;
private Button login;
private CheckBox rememberPass;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    pref = PreferenceManager.getDefaultSharedPreferences(this);
    accountEdit = (EditText) findViewById(R.id.account);
    passwordEdit = (EditText) findViewById(R.id.password);
    rememberPass = (CheckBox) findViewById(R.id.remember_pass);
    login = (Button) findViewById(R.id.login);
    boolean isRemember = pref.getBoolean("remember_password", false);
    if (isRemember) {
        // 将账号和密码都设置到文本框中
        String account = pref.getString("account", "");
        String password = pref.getString("password", "");
        accountEdit.setText(account);
        passwordEdit.setText(password);
        rememberPass.setChecked(true);
    }
    login.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            String account = accountEdit.getText().toString();
            String password = passwordEdit.getText().toString();
            // 如果账号是 admin 且密码是 123456, 就认为登录成功
            if (account.equals("admin") && password.equals("123456")) {
                editor = pref.edit();
                if (rememberPass.isChecked()) { // 检查复选框是否被选中
                    editor.putBoolean("remember_password", true);
                    editor.putString("account", account);
                    editor.putString("password", password);
                } else {
                    editor.clear();
                }
                editor.apply();
                Intent intent = new Intent(LoginActivity.this, MainActivity.class);
                startActivity(intent);
                finish();
            } else {
                Toast.makeText(LoginActivity.this, "account or password is invalid",
                        Toast.LENGTH_SHORT).show();
            }
        }
    });
});
```

```

    }
}

```

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

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

现在重新运行一下程序，可以看到界面上多出了一个记住密码复选框，如图 6.10 所示。

然后账号输入 `admin`，密码输入 `123456`，并选中记住密码复选框，点击登录，就会跳转到 `MainActivity`。接着在 `MainActivity` 中发出一条强制下线广播，会让程序重新回到登录界面，此时你会发现，账号密码都已经自动填充到界面上了，如图 6.11 所示。

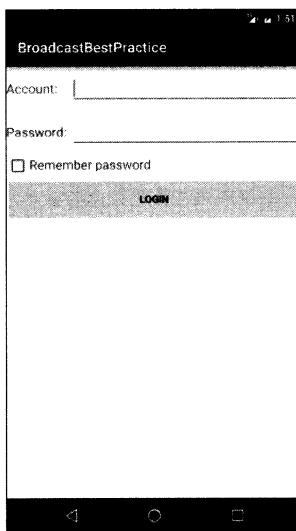


图 6.10 带记住密码复选框的登录界面

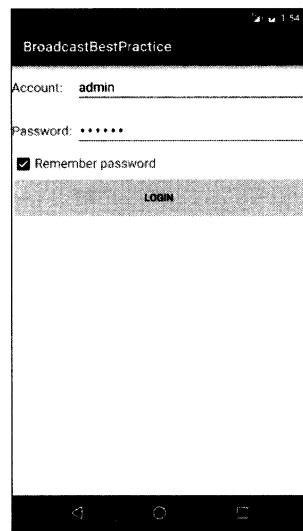


图 6.11 实现记住账号密码功能

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

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

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

6.4 SQLite 数据库存储

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

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

6.4.1 创建数据库

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

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

`SQLiteOpenHelper` 中还有两个非常重要的实例方法：`getReadableDatabase()` 和 `getWritableDatabase()`。这两个方法都可以创建或打开一个现有的数据库（如果数据库已存在则直接打开，否则创建一个新的数据库），并返回一个可对数据库进行读写操作的对象。不同的是，当数据库不可写入的时候（如磁盘空间已满），`getReadableDatabase()` 方法返回的对象将以只

读的方式去打开数据库，而 `getWritableDatabase()` 方法则将出现异常。

`SQLiteOpenHelper` 中有两个构造方法可供重写，一般使用参数少一点的那个构造方法即可。这个构造方法中接收 4 个参数，第一个参数是 `Context`，这个没什么好说的，必须要有它才能对数据库进行操作。第二个参数是数据库名，创建数据库时使用的就是这里指定的名称。第三个参数允许我们在查询数据的时候返回一个自定义的 `Cursor`，一般都是传入 `null`。第四个参数表示当前数据库的版本号，可用于对数据库进行升级操作。构建出 `SQLiteOpenHelper` 的实例之后，再调用它的 `getReadableDatabase()` 或 `getWritableDatabase()` 方法就能够创建数据库了，数据库文件会存放在 `/data/data/<package name>/databases/` 目录下。此时，重写的 `onCreate()` 方法也会得到执行，所以通常会在这里去处理一些创建表的逻辑。

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

这里我们希望创建一个名为 `BookStore.db` 的数据库，然后在这个数据库中新建一张 `Book` 表，表中有 `id`（主键）、作者、价格、页数和书名等列。创建数据库表当然还是需要用建表语句的，这里也是要考验一下你的 SQL 基本功了，`Book` 表的建表语句如下所示：

```
create table Book (
    id integer primary key autoincrement,
    author text,
    price real,
    pages integer,
    name text)
```

只要你对 SQL 方面的知识稍微有一些了解，上面的建表语句对你来说应该都不难吧。SQLite 不像其他的数据库拥有众多繁杂的数据类型，它的数据类型很简单，`integer` 表示整型，`real` 表示浮点型，`text` 表示文本类型，`blob` 表示二进制类型。另外，上述建表语句中我们还使用了 `primary key` 将 `id` 列设为主键，并用 `autoincrement` 关键字表示 `id` 列是自增长的。

然后需要在代码中去执行这条 SQL 语句，才能完成创建表的操作。新建 `MyDatabaseHelper` 类继承自 `SQLiteOpenHelper`，代码如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {

    public static final String CREATE_BOOK = "create table Book (" +
        "id integer primary key autoincrement, " +
        "author text, " +
        "price real, " +
        "pages integer, " +
        "name text);"

    private Context mContext;

    public MyDatabaseHelper(Context context, String name,
                           SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }
}
```

```

        mContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK);
        Toast.makeText(mContext, "Create succeeded", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }

}

```

可以看到，我们把建表语句定义成了一个字符串常量，然后在 `onCreate()` 方法中又调用了 `SQLiteDatabase` 的 `execSQL()` 方法去执行这条建表语句，并弹出一个 `Toast` 提示创建成功，这样就可以保证在数据库创建完成的同时还能成功创建 Book 表。

现在修改 `activity_main.xml` 中的代码，如下所示：

```

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

    <Button
        android:id="@+id/create_database"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Create database"
        />

</LinearLayout>

```

布局文件很简单，就是加入了一个按钮，用于创建数据库。最后修改 `MainActivity` 中的代码，如下所示：

```

public class MainActivity extends AppCompatActivity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 1);
        Button createDatabase = (Button) findViewById(R.id.create_database);
        createDatabase.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                dbHelper.getWritableDatabase();
            }
        });
    }
}

```

```

    });
}
}

```

这里我们在 `onCreate()` 方法中构建了一个 `MyDatabaseHelper` 对象，并且通过构造函数的参数将数据库名指定为 `BookStore.db`，版本号指定为 1，然后在 `Create database` 按钮的点击事件里调用了 `getWritableDatabase()` 方法。这样当第一次点击 `Create database` 按钮时，就会检测到当前程序中并没有 `BookStore.db` 这个数据库，于是会创建该数据库并调用 `MyDatabaseHelper` 中的 `onCreate()` 方法，这样 `Book` 表也就得到了创建，然后会弹出一个 `Toast` 提示创建成功。再次点击 `Create database` 按钮时，会发现此时已经存在 `BookStore.db` 数据库了，因此不会再创建一次。

现在就可以运行一下代码了，在程序主界面点击 `Create database` 按钮，结果如图 6.12 所示。

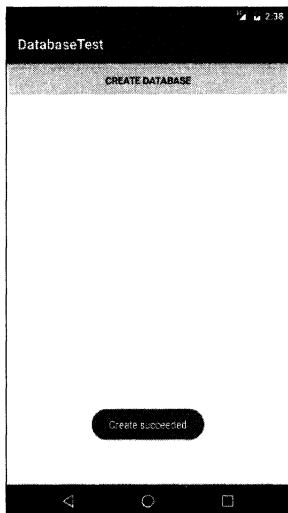


图 6.12 创建数据库成功

此时 `BookStore.db` 数据库和 `Book` 表应该都已经创建成功了，因为当你再次点击 `Create database` 按钮时，不会再有 `Toast` 弹出。可是又回到了之前的那个老问题，怎样才能证实它们的确创建成功了？如果还是使用 `File Explorer`，那么最多你只能看到 `databases` 目录下出现了一个 `BookStore.db` 文件，`Book` 表是无法通过 `File Explorer` 看到的。因此这次我们准备换一种查看方式，使用 `adb shell` 来对数据库和表的创建情况进行检查。

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

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

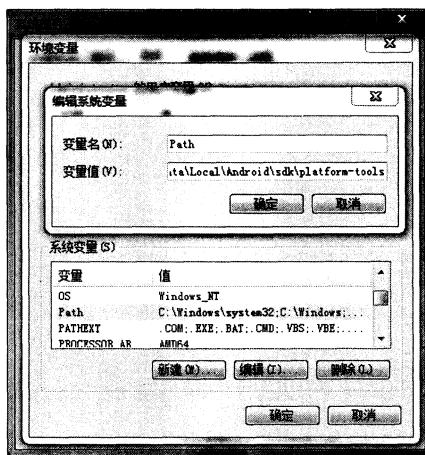


图 6.13 Windows 下配置环境变量

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

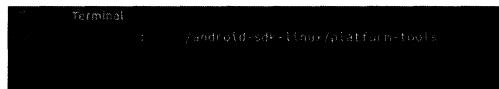


图 6.14 Linux 或 Mac 下配置环境变量

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



图 6.15 进入设备的控制台

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



图 6.16 查看数据库文件

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

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

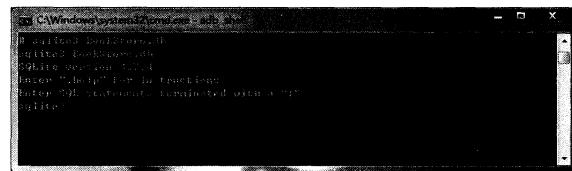


图 6.17 打开 BookStore.db 数据库

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



图 6.18 查看表

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

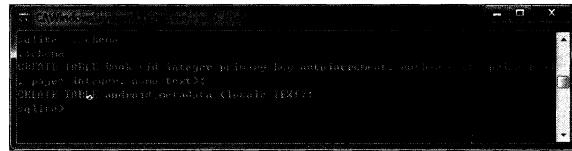


图 6.19 查看建表语句

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

6.4.2 升级数据库

如果你足够细心，一定会发现 `MyDatabaseHelper` 中还有一个空方法呢！没错，`onUpgrade()` 方法是用于对数据库进行升级的，它在整个数据库的管理工作当中起着非常重要的作用，可千万

不能忽视它哟。

目前 DatabaseTest 项目中已经有一张 Book 表用于存放书的各种详细数据，如果我们想再添加一张 Category 表用于记录图书的分类，该怎么做呢？

比如 Category 表中有 id（主键）、分类名和分类代码这几个列，那么建表语句就可以写成：

```
create table Category (
    id integer primary key autoincrement,
    category_name text,
    category_code integer)
```

接下来我们将这条建表语句添加到 MyDatabaseHelper 中，代码如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {

    public static final String CREATE_BOOK = "create table Book (" +
        + "id integer primary key autoincrement, " +
        + "author text, " +
        + "price real, " +
        + "pages integer, " +
        + "name text)";

    public static final String CREATE_CATEGORY = "create table Category (" +
        + "id integer primary key autoincrement, " +
        + "category_name text, " +
        + "category_code integer)";

    private Context mContext;

    public MyDatabaseHelper(Context context, String name,
                           SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
        mContext = context;
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_BOOK);
        db.execSQL(CREATE_CATEGORY);
        Toast.makeText(mContext, "Create succeeded", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    }
}
```

看上去好像都挺对的吧？现在我们重新运行一下程序，并点击 Create database 按钮，咦？竟然没有弹出创建成功的提示。当然，你也可以通过 adb 工具到数据库中再去检查一下，这样你会更加地确认 Category 表没有创建成功！

其实没有创建成功的原因不难思考，因为此时 BookStore.db 数据库已经存在了，之后不管我们怎样点击 Create database 按钮，MyDatabaseHelper 中的 `onCreate()` 方法都不会再次执行，因此新添加的表也就无法得到创建了。

解决这个问题的办法也相当简单，只需要先将程序卸载掉，然后重新运行，这时 BookStore.db 数据库已经不存在了，如果再点击 Create database 按钮，MyDatabaseHelper 中的 `onCreate()` 方法就会执行，这时 Category 表就可以创建成功了。

不过，通过卸载程序的方式来新增一张表毫无疑问是很极端的做法，其实我们只需要巧妙地运用 SQLiteOpenHelper 的升级功能就可以很轻松地解决这个问题。修改 MyDatabaseHelper 中的代码，如下所示：

```
public class MyDatabaseHelper extends SQLiteOpenHelper {
    ...
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("drop table if exists Book");
        db.execSQL("drop table if exists Category");
        onCreate(db);
    }
}
```

可以看到，我们在 `onUpgrade()` 方法中执行了两条 `DROP` 语句，如果发现数据库中已经存在 Book 表或 Category 表了，就将这两张表删除掉，然后再调用 `onCreate()` 方法重新创建。这里先将已经存在的表删除掉，因为如果在创建表时发现这张表已经存在了，就会直接报错。

接下来的问题就是如何让 `onUpgrade()` 方法能够执行了，还记得 SQLiteOpenHelper 的构造方法里接收的第四个参数吗？它表示当前数据库的版本号，之前我们传入的是 1，现在只要传入一个比 1 大的数，就可以让 `onUpgrade()` 方法得到执行了。修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {
    private MyDatabaseHelper dbHelper;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        Button createDatabase = (Button) findViewById(R.id.create_database);
        createDatabase.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                dbHelper.getWritableDatabase();
            }
        });
    }
}
```

```

    }
}

```

这里将数据库版本号指定为 2，表示我们对数据库进行升级了。现在重新运行程序，并点击 `Create database` 按钮，这时就会再次弹出创建成功的提示。为了验证一下 `Category` 表是不是已经创建成功了，我们在 `adb shell` 中打开 `BookStore.db` 数据库，然后键入 `.table` 命令，结果如图 6.20 所示。



图 6.20 查看新增表

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



图 6.21 查看新增建表语句

由此可以看出，`Category` 表已经创建成功了，同时也说明我们的升级功能的确起到了作用。

6.4.3 添加数据

现在你已经掌握了创建和升级数据库的方法，接下来就该学习一下如何对表中的数据进行操作了。其实我们可以对数据进行的操作无非有 4 种，即 CRUD。其中 C 代表添加（Create），R 代表查询（Retrieve），U 代表更新（Update），D 代表删除（Delete）。每一种操作又各自对应了一种 SQL 命令，如果你比较熟悉 SQL 语言的话，一定会知道添加数据时使用 `insert`，查询数据时使用 `select`，更新数据时使用 `update`，删除数据时使用 `delete`。但是开发者的水平总会是参差不齐的，未必每一个人都能非常熟悉地使用 SQL 语言，因此 Android 也提供了一系列的辅助性方法，使得在 Android 中即使不去编写 SQL 语句，也能轻松完成所有的 CRUD 操作。

前面我们已经知道，调用 `SQLiteOpenHelper` 的 `getReadableDatabase()` 或 `getWritableDatabase()` 方法是可以用于创建和升级数据库的，不仅如此，这两个方法还都会返回一个 `SQLiteDatabase` 对象，借助这个对象就可以对数据进行 CRUD 操作了。

那么下面我们首先学习一下如何向数据库的表中添加数据吧。`SQLiteDatabase` 中提供了一个 `insert()` 方法，这个方法就是专门用于添加数据的。它接收 3 个参数，第一个参数是表名，

我们希望向哪张表里添加数据，这里就传入该表的名字。第二个参数用于在未指定添加数据的情况下给某些可为空的列自动赋值 NULL，一般我们用不到这个功能，直接传入 null 即可。第三个参数是一个 ContentValues 对象，它提供了一系列的 put()方法重载，用于向 ContentValues 中添加数据，只需要将表中的每个列名以及相应的待添加数据传入即可。

介绍完了基本用法，接下来还是让我们通过例子的方式来亲身体验一下如何添加数据吧。修改 activity_main.xml 中的代码，如下所示：

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

    ...
    <Button
        android:id="@+id/add_data"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add data"
    />
</LinearLayout>
```

可以看到，我们在布局文件中又新增了一个按钮，稍后就会在这个按钮的点击事件里编写添加数据的逻辑。接着修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ...
        Button addData = (Button) findViewById(R.id.add_data);
        addData.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                ContentValues values = new ContentValues();
                // 开始组装第一条数据
                values.put("name", "The Da Vinci Code");
                values.put("author", "Dan Brown");
                values.put("pages", 454);
                values.put("price", 16.96);
                db.insert("Book", null, values); // 插入第一条数据
                values.clear();
                // 开始组装第二条数据
                values.put("name", "The Lost Symbol");
                values.put("author", "Dan Brown");
            }
        });
    }
}
```

```
        values.put("pages", 510);
        values.put("price", 19.95);
        db.insert("Book", null, values); // 插入第二条数据
    }
}
}
```

在添加数据按钮的点击事件里面，我们先获取到了 `SQLiteDatabase` 对象，然后使用 `ContentValues` 来对要添加的数据进行组装。如果你比较细心的话应该会发现，这里只对 `Book` 表里其中四列的数据进行了组装，`id` 那一列没并没给它赋值。这是因为在前面创建表的时候，我们就将 `id` 列设置为自增长了，它的值会在入库的时候自动生成，所以不需要手动给它赋值了。接下来调用了 `insert()` 方法将数据添加到表当中，注意这里我们实际上添加了两条数据，上述代码中使用 `ContentValues` 分别组装了两次不同的内容，并调用了两次 `insert()` 方法。

好了，现在可以重新运行一下程序了，界面如图 6.22 所示。

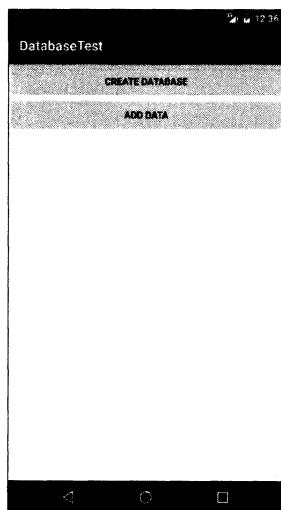


图 6.22 加入添加数据按钮

点击一下 Add data 按钮，此时两条数据应该都已经添加成功了，不过为了证实一下，我们还是打开 BookStore.db 数据库瞧一瞧。输入 SQL 查询语句 `select * from Book`，结果如图 6.23 所示。



图 6.23 查看添加的数据

由此可以看出，我们刚刚组装的两条数据都已经准确无误地添加到 Book 表中了。

6.4.4 更新数据

学习完了如何向表中添加数据，接下来我们看看怎样才能修改表中已有的数据。SQLite-Database 中也提供了一个非常好用的 `update()`方法，用于对数据进行更新，这个方法接收 4 个参数，第一个参数和 `insert()`方法一样，也是表名，在这里指定去更新哪张表里的数据。第二个参数是 `ContentValues` 对象，要把更新数据在这里组装进去。第三、第四个参数用于约束更新某一行或某几行中的数据，不指定的话默认就是更新所有行。

那么接下来我们仍然是在 DatabaseTest 项目的基础上修改，看一下更新数据的具体用法。比如说刚才添加到数据库里的第一本书，由于过了畅销季，卖得不是很火了，现在需要通过降低价格的方式来吸引更多的顾客，我们应该怎么操作呢？首先修改 `activity_main.xml` 中的代码，如下所示：

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

    ...

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

布局文件中的代码已经非常简单了，就是添加了一个用于更新数据的按钮。然后修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ...
        Button updateData = (Button) findViewById(R.id.update_data);
        updateData.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                ContentValues values = new ContentValues();
                values.put("price", 10.99);
```

```
        db.update("Book", values, "name = ?", new String[] { "The Da Vinci
Code" });
    }
});
}

}
```

这里在更新数据按钮的点击事件里面构建了一个 `ContentValues` 对象，并且只给它指定了
一组数据，说明我们只是想把价格这一列的数据更新成 10.99。然后调用了 `SQLiteDatabase` 的
`update()` 方法去执行具体的更新操作，可以看到，这里使用了第三、第四个参数来指定具体更
新哪几行。第三个参数对应的是 SQL 语句的 `where` 部分，表示更新所有 `name` 等于?的行，而?
是一个占位符，可以通过第四个参数提供的一个字符串数组为第三个参数中的每个占位符指定相
应的内容。因此上述代码想表达的意图是将名字是 `The Da Vinci Code` 的这本书的价格改成 10.99。

现在重新运行一下程序，界面如图 6.24 所示。

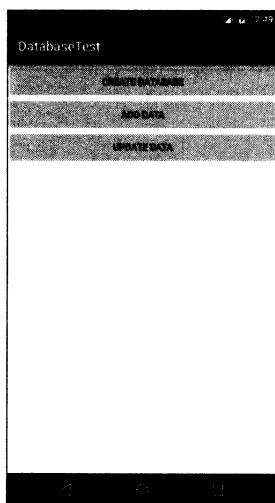


图 6.24 加入更新数据按钮

点击一下 `Update data` 按钮后，再次输入查询语句查看表中的数据情况，结果如图 6.25 所示。



图 6.25 查看更新后的数据

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

6.4.5 删除数据

怎么样？添加和更新数据的功能都还挺简单的吧，代码也不多，理解起来又容易，那么我们要马不停蹄地开始学习下一种操作了，即从表中删除数据。

删除数据对你来说应该就更简单了，因为它所需要用到的知识点你全部已经学过了。SQLiteDatabase 中提供了一个 delete()方法，专门用于删除数据，这个方法接收 3 个参数，第一个参数仍然是表名，这个已经没什么好说的了，第二、第三个参数又是用于约束删除某一行或某几行的数据，不指定的话默认就是删除所有行。

是不是理解起来很轻松了？那我们就继续动手实践吧，修改 activity_main.xml 中的代码，如下所示：

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

    ...

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

仍然是在布局文件中添加了一个按钮，用于删除数据。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ...
        Button deleteButton = (Button) findViewById(R.id.delete_data);
        deleteButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                db.delete("Book", "pages > ?", new String[] { "500" });
            }
        });
    }
}
```

```

    }
}

```

可以看到，我们在删除按钮的点击事件里指明去删除 Book 表中的数据，并且通过第二、第三个参数来指定仅删除那些页数超过 500 页的书。当然这个需求很奇怪，这里也仅仅是为了做个测试。你可以先查看一下当前 Book 表里的数据，其中 The Lost Symbol 这本书的页数超过了 500 页，也就是说当我们点击删除按钮时，这条记录应该会被删除掉。

现在重新运行一下程序，界面如图 6.26 所示。

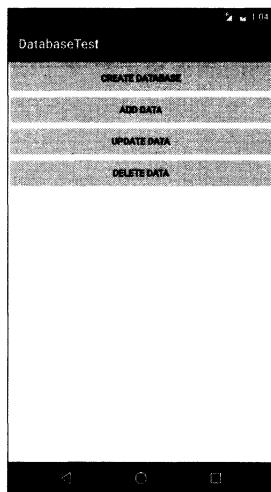


图 6.26 加入删除数据按钮

点击一下 Delete data 按钮后，再次输入查询语句查看表中的数据情况，结果如图 6.27 所示。



图 6.27 查看删除后的数据

6.4.6 查询数据

终于到了最后一种操作了，掌握了查询数据的方法之后，你就将数据库的 CRUD 操作全部学完了。不过千万不要因此而放松，因为查询数据是 CRUD 中最复杂的一种操作。

我们都知道 SQL 的全称是 Structured Query Language，翻译成中文就是结构化查询语言。它的大部分功能都体现在“查”这个字上的，而“增删改”只是其中的一小部分功能。由于 SQL 查

询涉及的内容实在是太多了，因此在这里我不准备对它展开来讲解，而是只会介绍 Android 上的查询功能。如果你对 SQL 语言非常感兴趣，可以找一本专门介绍 SQL 的书进行学习。

相信你已经猜到了，`SQLiteDatabase` 中还提供了一个 `query()` 方法用于对数据进行查询。这个方法的参数非常复杂，最短的一个方法重载也需要传入 7 个参数。那我们就先来看一下这 7 个参数各自的含义吧。第一个参数不用说，当然还是表名，表示我们希望从哪张表中查询数据。第二个参数用于指定去查询哪几列，如果不指定则默认查询所有列。第三、第四个参数用于约束查询某一行或某几行的数据，不指定则默认查询所有行的数据。第五个参数用于指定需要去 `group by` 的列，不指定则表示不对查询结果进行 `group by` 操作。第六个参数用于对 `group by` 之后的数据进行进一步的过滤，不指定则表示不进行过滤。第七个参数用于指定查询结果的排序方式，不指定则表示使用默认的排序方式。更多详细的内容可以参考下表。其他几个 `query()` 方法的重载其实也大同小异，你可以自己去研究一下，这里就不再进行介绍了。

query()方法参数	对应SQL部分	描述
table	<code>from table_name</code>	指定查询的表名
columns	<code>select column1, column2</code>	指定查询的列名
selection	<code>where column = value</code>	指定where的约束条件
selectionArgs	-	为where中的占位符提供具体的值
groupBy	<code>group by column</code>	指定需要group by的列
having	<code>having column = value</code>	对group by后的结果进一步约束
orderBy	<code>order by column1, column2</code>	指定查询结果的排序方式

虽然 `query()` 方法的参数非常多，但是不要对它产生畏惧，因为我们不必为每条查询语句都指定所有的参数，多数情况下只需要传入少数几个参数就可以完成查询操作了。调用 `query()` 方法后会返回一个 `Cursor` 对象，查询到的所有数据都将从这个对象中取出。

下面还是让我们通过例子的方式来体验一下查询数据的具体用法，修改 `activity_main.xml` 中的代码，如下所示：

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

    ...

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

这个已经没什么好说的了，添加了一个按钮用于查询数据。然后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private MyDatabaseHelper dbHelper;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dbHelper = new MyDatabaseHelper(this, "BookStore.db", null, 2);
        ...

        Button queryButton = (Button) findViewById(R.id.query_data);
        queryButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                SQLiteDatabase db = dbHelper.getWritableDatabase();
                // 查询 Book 表中所有的数据
                Cursor cursor = db.query("Book", null, null, null, null, null, null);
                if (cursor.moveToFirst()) {
                    do {
                        // 遍历 Cursor 对象，取出数据并打印
                        String name = cursor.getString(cursor.getColumnIndex
                            ("name"));
                        String author = cursor.getString(cursor.getColumnIndex
                            ("author"));
                        int pages = cursor.getInt(cursor.getColumnIndex("pages"));
                        double price = cursor.getDouble(cursor.getColumnIndex
                            ("price"));
                        Log.d("MainActivity", "book name is " + name);
                        Log.d("MainActivity", "book author is " + author);
                        Log.d("MainActivity", "book pages is " + pages);
                        Log.d("MainActivity", "book price is " + price);
                    } while (cursor.moveToNext());
                }
                cursor.close();
            }
        });
    }
}
```

可以看到，我们首先在查询按钮的点击事件里面调用了 SQLiteDatabase 的 query() 方法去查询数据。这里的 query() 方法非常简单，只是使用了第一个参数指明去查询 Book 表，后面的参数全部为 null。这就表示希望查询这张表中的所有数据，虽然这张表中目前只剩下一条数据了。查询完之后就得到了一个 Cursor 对象，接着我们调用它的 moveToFirst() 方法将数据的指针移动到第一行的位置，然后进入了一个循环当中，去遍历查询到的每一行数据。在这个循环中可以通过 Cursor 的 getColumnIndex() 方法获取到某一列在表中对应的位置索引，然后将这个索引传入到相应的取值方法中，就可以得到从数据库中读取到的数据了。接着我们使用 Log 的方式将

取出的数据打印出来，借此来检查一下读取工作有没有成功完成。最后别忘了调用 `close()` 方法来关闭 Cursor。

好了，现在再次重新运行程序，界面如图 6.28 所示。

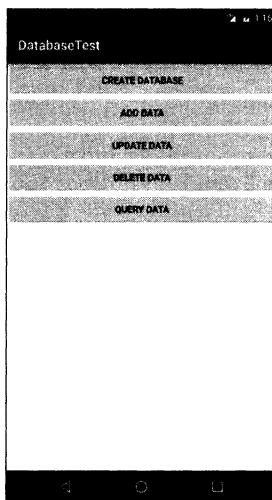


图 6.28 加入查询数据按钮

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

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

图 6.29 打印查询到的数据

可以看到，这里已经将 Book 表中唯一的一条数据成功地读取出来了。

当然这个例子只是对查询数据的用法进行了最简单的示范，在真正的项目中你可能会遇到比这要复杂得多的查询功能，更多高级的用法还需要你自己去慢慢摸索，毕竟 `query()` 方法中还有那么多的参数我们都还没用到呢。

6.4.7 使用 SQL 操作数据库

虽然 Android 已经给我们提供了很多非常方便的 API 用于操作数据库，不过总会有一些人不习惯去使用这些辅助性的方法，而是更加青睐于直接使用 SQL 来操作数据库。这种人一般都属于 SQL 大牛，如果你也是其中之一的话，那么恭喜，Android 充分考虑到了你们的编程习惯，同样提供了一系列的方法，使得可以直接通过 SQL 来操作数据库。

下面我就来简略演示一下，如何直接使用 SQL 来完成前面几小节中学过的 CRUD 操作。

□ 添加数据的方法如下：

```
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",  
        new String[] { "The Da Vinci Code", "Dan Brown", "454", "16.96" });  
db.execSQL("insert into Book (name, author, pages, price) values(?, ?, ?, ?)",  
        new String[] { "The Lost Symbol", "Dan Brown", "510", "19.95" });
```

□ 更新数据的方法如下：

```
db.execSQL("update Book set price = ? where name = ?", new String[] { "10.99",  
        "The Da Vinci Code" });
```

□ 删除数据的方法如下：

```
db.execSQL("delete from Book where pages > ?", new String[] { "500" });
```

□ 查询数据的方法如下：

```
db.rawQuery("select * from Book", null);
```

可以看到，除了查询数据的时候调用的是 SQLiteDatabase 的 rawQuery() 方法，其他的操作都是调用的 execSQL() 方法。以上演示的几种方式，执行结果会和前面几小节中我们学习的 CRUD 操作的结果完全相同，选择使用哪一种方式就看你个人的喜好了。

6.5 使用 LitePal 操作数据库

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

6.5.1 LitePal 简介

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

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

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

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

6.5.2 配置 LitePal

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

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

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

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

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

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

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

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

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

最后还需要再配置一下 LitePalApplication，修改 AndroidManifest.xml 中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.litepaltest">
    <application
        android:name="org.litepal.LitePalApplication"
        android:allowBackup="true"
```

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

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

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

6.5.3 创建和升级数据库

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

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

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

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

```

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

public String getAuthor() {
    return author;
}

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

public double getPrice() {
    return price;
}

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

public int getPages() {
    return pages;
}

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

public String getName() {
    return name;
}

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

```

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

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

```

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

```

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

```

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

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

```

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

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

```

public class MainActivity extends AppCompatActivity {

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

```

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



图 6.30 查看数据库文件

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

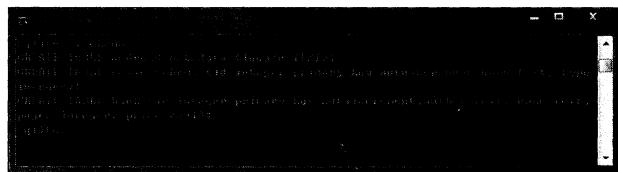


图 6.31 查看建表语句

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

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

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

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

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

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

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

```

private String categoryName;

private int categoryCode;

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

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

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

}

```

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

```

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

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

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

```

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

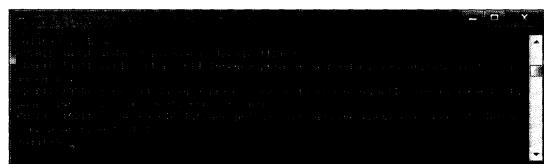


图 6.32 升级数据库后的建表语句

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

6.5.4 使用 LitePal 添加数据

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

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

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

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

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

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

```
public class MainActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        ...  
        Button addData = (Button) findViewById(R.id.add_data);  
        addData.setOnClickListener(new View.OnClickListener() {  
            @Override  
            public void onClick(View v) {  
                Book book = new Book();  
                book.setName("The Da Vinci Code");  
                book.setAuthor("Dan Brown");  
                book.setPages(454);  
                book.setPrice(16.96);  
                book.setPress("Unknow");  
                book.save();  
            }  
        });  
    }  
}
```

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

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

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



图 6.33 查看添加的数据

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

6.5.5 使用 LitePal 更新数据

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

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

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

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

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

```
public class MainActivity extends AppCompatActivity {

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

```

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

```

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

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



图 6.34 查看更新后的数据

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

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

```

public class MainActivity extends AppCompatActivity {

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

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

```

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

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

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



图 6.35 再次查看更新后的数据

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

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

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

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

6.5.6 使用 LitePal 删除数据

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

修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

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

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

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



图 6.36 查看删除后的数据

可以看到，价格低于 15 的那本书已经被删除掉了。

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

6.5.7 使用 LitePal 查询数据

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

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

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

首先分析一下上述代码，`query()` 方法中使用了第一个参数指明去查询 Book 表，后面的参数全部为 `null`，这就表示希望查询这张表中的所有数据。那么使用 LitePal 如何完成同样的功能呢？非常简单，只需要这样写：

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

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

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

```
public class MainActivity extends AppCompatActivity {

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

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

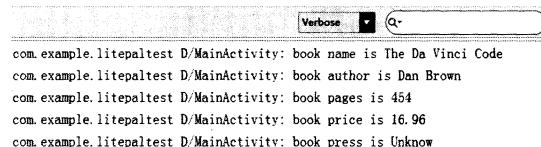


图 6.37 打印查询到的数据

Book 表中只剩下一条数据，由此可见，我们已经将这条数据成功查询出来了。

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

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

查询 Book 表中的最后一条数据就可以这样写：

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

我们还可以通过连缀查询来定制更多的查询功能。

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

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

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

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

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

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

其中 `desc` 表示降序排列，`asc` 或者不写表示升序排列。

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

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

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

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

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

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

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

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

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

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

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

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

6.6 小结与点评

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

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