

## 第 2 章

---

# 先从看得到的入手——探究活动

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

## 2.1 活动是什么

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

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

## 2.2 活动的基本用法

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

由于 Android Studio 在一个工作区间内只允许打开一个项目，因此首先你需要将当前的项目关闭，点击导航栏 `File`→`Close Project`。然后再新建一个 Android 项目，项目名可以叫作 `ActivityTest`，包名我们就使用默认值 `com.example.activitytest`。新建项目的步骤你已经在上一章学习过了，不过图 1.12 中的那一步需要稍做修改，我们不再选择 `Empty Activity` 这个选项，而是选择 `Add No`

Activity，因为这次我们准备手动创建活动，如图 2.1 所示。

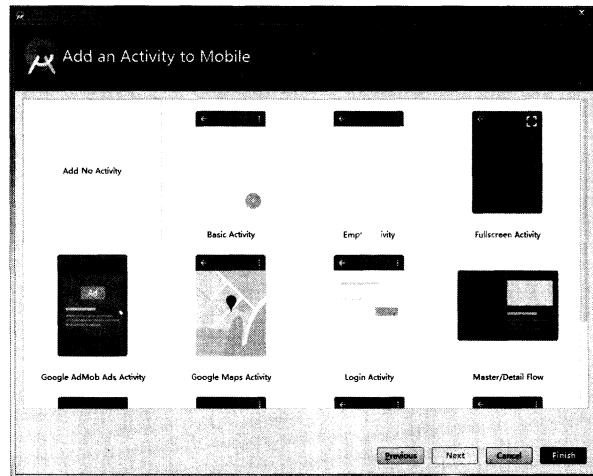


图 2.1 选择不添加活动

点击 Finish，等待 Gradle 构建完成后，项目就创建成功了。

### 2.2.1 手动创建活动

项目创建成功后，仍然会默认使用 Android 模式的项目结构，这里我们手动改成 Project 模式，本书中后面的所有项目都要这样修改，以后就不再赘述了。目前 ActivityTest 项目中虽然还是会自动生成很多文件，但是 app/src/main/java/com.example.activitytest 目录应该是空的了，如图 2.2 所示。

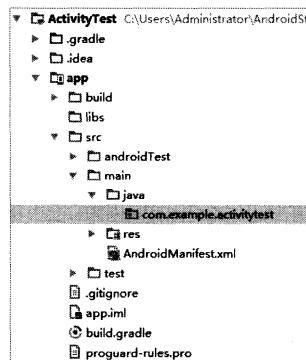


图 2.2 初始项目结构

现在右击 com.example.activitytest 包 → New → Activity → Empty Activity，会弹出一个创建活动的对话框，我们将活动命名为 FirstActivity，并且不要勾选 Generate Layout File 和 Launcher Activity 这两个选项，如图 2.3 所示。

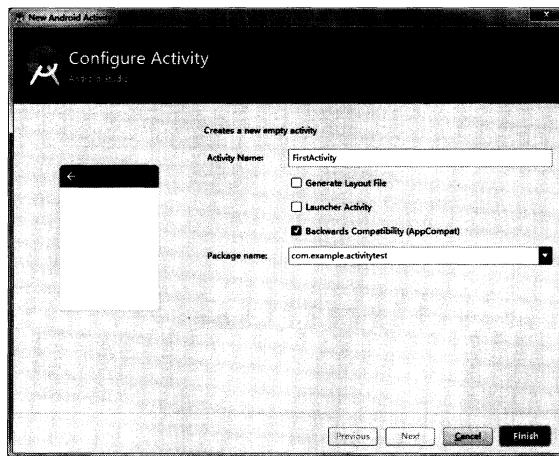


图 2.3 新建活动对话框

勾选 Generate Layout File 表示会自动为 FirstActivity 创建一个对应的布局文件，勾选 Launcher Activity 表示会自动将 FirstActivity 设置为当前项目的主活动，这里由于你是第一次手动创建活动，这些自动生成的东西暂时都不要勾选，下面我们将一个个手动来完成。勾选 Backwards Compatibility 表示会为项目启用向下兼容的模式，这个选项要勾上。点击 Finish 完成创建。

你需要知道，项目中的任何活动都应该重写 Activity 的 `onCreate()` 方法，而目前我们的 FirstActivity 中已经重写了这个方法，这是由 Android Studio 自动帮我们完成的，代码如下所示：

```
public class FirstActivity extends AppCompatActivity {

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

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

## 2.2.2 创建和加载布局

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

右击 `app/src/main/res` 目录 → New → Directory，会弹出一个新建目录的窗口，这里先创建一个名为 `layout` 的目录。然后对着 `layout` 目录右键 → Layout resource file，又会弹出一个新建布局资源文件的窗口，我们将这个布局文件命名为 `first_layout`，根元素就默认选择为 `LinearLayout`，如图 2.4 所示。

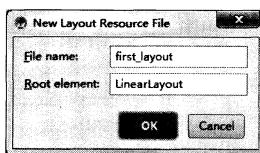


图 2.4 新建布局资源文件

点击 OK 完成布局的创建，这时候你会看到如图 2.5 所示的布局编辑器。

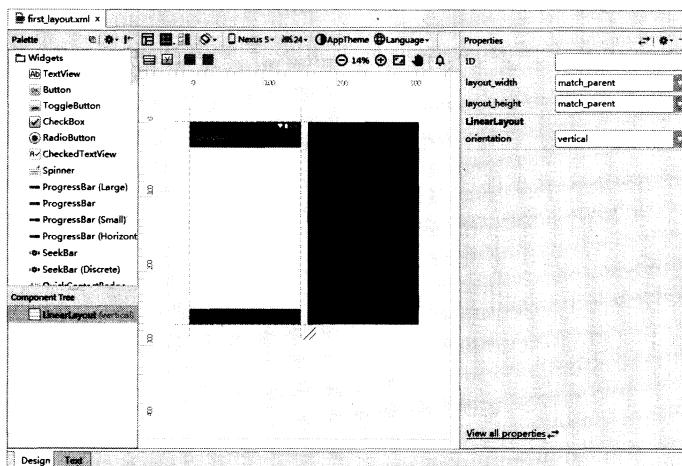


图 2.5 布局编辑器

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

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

</LinearLayout>
```

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

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

<Button
    android:id="@+id/button_1"
```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button 1"
/>
</LinearLayout>
```

这里添加了一个 Button 元素，并在 Button 元素的内部增加了几个属性。`android:id` 是给当前的元素定义一个唯一标识符，之后可以在代码中对这个元素进行操作。你可能会对`@+id/button_1` 这种语法感到陌生，但如果把加号去掉，变成`@id/button_1`，这样你就会觉得有些熟悉了吧，这不就是在 XML 中引用资源的语法吗？只不过是把 `string` 替换成了 `id`。是的，如果你需要在 XML 中引用一个 `id`，就使用`@id/id_name` 这种语法，而如果你需要在 XML 中定义一个 `id`，则要使用`@+id/id_name` 这种语法。随后 `android:layout_width` 指定了当前元素的宽度，这里使用 `match_parent` 表示让当前元素和父元素一样宽。`android:layout_height` 指定了当前元素的高度，这里使用 `wrap_content` 表示当前元素的高度只要能刚好包含里面的内容就行。`android:text` 指定了元素中显示的文字内容。如果你还不能完全看明白，没有关系，关于编写布局的详细内容我会在下一章中重点讲解，本章只是先简单涉及一些。现在按钮已经添加完了，你可以通过右侧工具栏的 Preview 来预览一下当前布局，如图 2.6 所示。

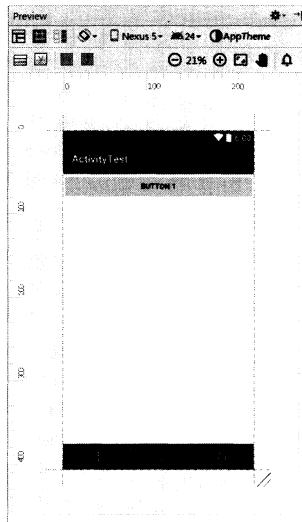


图 2.6 预览当前布局

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

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

```
public class FirstActivity extends AppCompatActivity {
    @Override
```

```

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

```

可以看到，这里调用了 `setContentView()` 方法来给当前的活动加载一个布局，而在 `setContentView()` 方法中，我们一般都会传入一个布局文件的 id。在第 1 章介绍项目资源的时候我曾提到过，项目中添加的任何资源都会在 R 文件中生成一个相应的资源 id，因此我们刚才创建的 `first_layout.xml` 布局的 id 现在应该是已经添加到 R 文件中了。在代码中去引用布局文件的方法你也已经学过了，只需要调用 `R.layout.first_layout` 就可以得到 `first_layout.xml` 布局的 id，然后将这个值传入 `setContentView()` 方法即可。

### 2.2.3 在 AndroidManifest 文件中注册

别忘了在上一章我们学过，所有的活动都要在 `AndroidManifest.xml` 中进行注册才能生效，而实际上 `FirstActivity` 已经在 `AndroidManifest.xml` 中注册过了，我们打开 `app/src/main/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.activitytest">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".FirstActivity"></activity>
    </application>
</manifest>

```

可以看到，活动的注册声明要放在 `<application>` 标签内，这里是通过 `<activity>` 标签来对活动进行注册的。那么又是谁帮我们自动完成了对 `FirstActivity` 的注册呢？当然是 Android Studio 了，之前在使用 Eclipse 创建活动或其他系统组件时，很多人都会忘记要去 `Android Manifest.xml` 中注册一下，从而导致程序运行崩溃，很显然 Android Studio 在这方面做得更加人性化。

在 `<activity>` 标签中我们使用了 `android:name` 来指定具体注册哪一个活动，那么这里填入的 `.FirstActivity` 是什么意思呢？其实这不过就是 `com.example.activitytest.FirstActivity` 的缩写而已。由于在最外层的 `<manifest>` 标签中已经通过 `package` 属性指定了程序的包名是 `com.example.activitytest`，因此在注册活动时这一部分就可以省略了，直接使用 `.FirstActivity` 就足够了。

不过，仅仅是这样注册了活动，我们的程序仍然是不能运行的，因为还没有为程序配置主活动，也就是说，当程序运行起来的时候，不知道要首先启动哪个活动。配置主活动的方法其实在第 1 章中已经介绍过了，就是在 `<activity>` 标签的内部加入 `<intent-filter>` 标签，并在这个

标签里添加<action android:name="android.intent.action.MAIN"/>和<category android:name="android.intent.category.LAUNCHER" />这两句声明即可。

除此之外，我们还可以使用 android:label 指定活动中标题栏的内容，标题栏是显示在活动最顶部的，待会儿运行的时候你就会看到。需要注意的是，给主活动指定的 label 不仅会成为标题栏中的内容，还会成为启动器（Launcher）中应用程序显示的名称。

修改后的 AndroidManifest.xml 文件，代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.activitytest">
    <application
        ...
        <activity android:name=".FirstActivity"
            android:label="This is FirstActivity"
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

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

好了，现在一切都已准备就绪，让我们来运行一下程序吧，结果如图 2.7 所示。

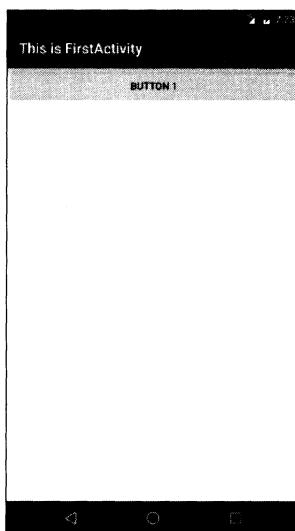


图 2.7 首次运行结果

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

### 2.2.4 在活动中使用 Toast

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

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

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.first_layout);
    Button button1 = (Button) findViewById(R.id.button_1);
    button1.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(FirstActivity.this, "You clicked Button 1",
                Toast.LENGTH_SHORT).show();
        }
    });
}
```

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

Toast 的用法非常简单，通过静态方法 `makeText()` 创建出一个 `Toast` 对象，然后调用 `show()` 将 `Toast` 显示出来就可以了。这里需要注意的是，`makeText()` 方法需要传入 3 个参数。第一个参数是 `Context`，也就是 `Toast` 要求的上下文，由于活动本身就是一个 `Context` 对象，因此这里直接传入 `FirstActivity.this` 即可。第二个参数是 `Toast` 显示的文本内容，第三个参数是 `Toast` 显示的时长，有两个内置常量可以选择 `Toast.LENGTH_SHORT` 和 `Toast.LENGTH_LONG`。

现在重新运行程序，并点击一下按钮，效果如图 2.8 所示。

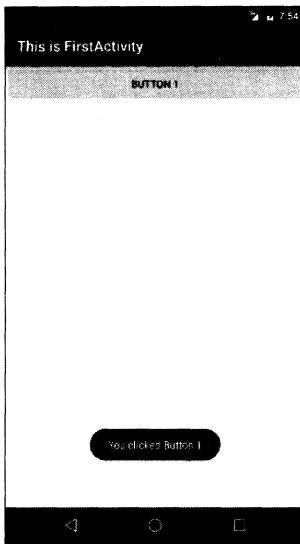


图 2.8 Toast 运行效果

### 2.2.5 在活动中使用 Menu

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

首先在 res 目录下新建一个 menu 文件夹，右击 res 目录→New→Directory，输入文件夹名 menu，点击 OK。接着在这个文件夹下再新建一个名叫 main 的菜单文件，右击 menu 文件夹→New→Menu resource file，如图 2.9 所示。

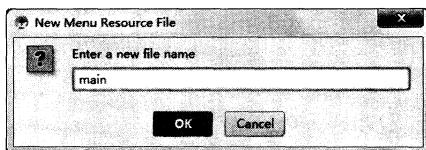


图 2.9 新建 Menu 资源文件

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

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/add_item"
        android:title="Add"/>
    <item
        android:id="@+id/remove_item"
```

```
        android:title="Remove"/>
</menu>
```

这里我们创建了两个菜单项，其中`<item>`标签就是用来创建具体的某一个菜单项，然后通过`android:id`给这个菜单项指定一个唯一的标识符，通过`android:title`给这个菜单项指定一个名称。

接着重新回到`FirstActivity`中来重写`onCreateOptionsMenu()`方法，重写方法可以使用`Ctrl + O`快捷键（Mac系统是`control + O`），如图2.10所示。

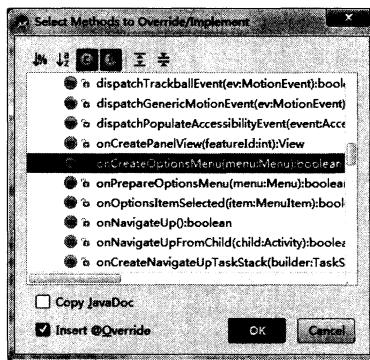


图2.10 重写`onCreateOptionsMenu()`方法

然后在`onCreateOptionsMenu()`方法中编写如下代码：

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

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

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

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

```

        break;
    default:
    }
    return true;
}

```

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

重新运行程序，你会发现在标题栏的右侧多了一个三点的符号，这个就是菜单按钮了，如图 2.11 所示。

可以看到，菜单里的菜单项默认是不会显示出来的，只有点击一下菜单按钮才会弹出里面具体的内容，因此它不会占用任何活动的空间，如图 2.12 所示。

然后如果你点击了 Add 菜单项就会弹出 You clicked Add 提示（如图 2.13 所示），如果点击了 Remove 菜单项就会弹出 You clicked Remove 提示。

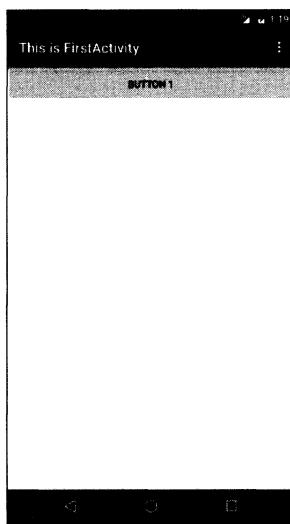


图 2.11 带菜单按钮的活动

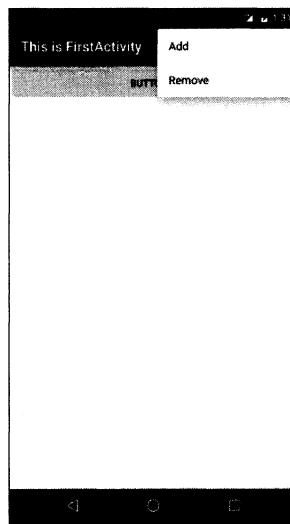


图 2.12 弹出菜单项的界面

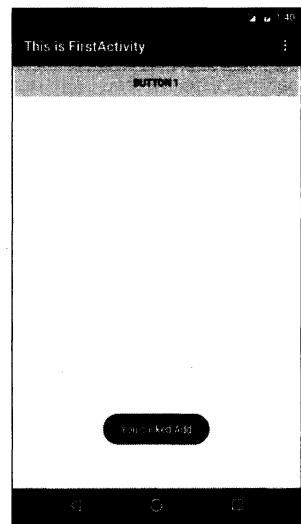


图 2.13 点击了 Add 菜单项

## 2.2.6 销毁一个活动

通过上一节的学习，你已经掌握了手动创建活动的方法，并学会了如何在活动中创建 `Toast` 和创建菜单。或许你现在心中会有个疑惑，如何销毁一个活动呢？

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

修改按钮监听器中的代码，如下所示：

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

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

## 2.3 使用 Intent 在活动之间穿梭

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

### 2.3.1 使用显式 Intent

你应该已经对创建活动的流程比较熟悉了，那我们现在快速地在 ActivityTest 项目中再创建一个活动。

仍然还是右击 com.example.activitytest 包→New→Activity→Empty Activity，会弹出一个创建活动的对话框，我们这次将活动命名为 SecondActivity，并勾选 Generate Layout File，给布局文件起名为 second\_layout，但不要勾选 Launcher Activity 选项，如图 2.14 所示。

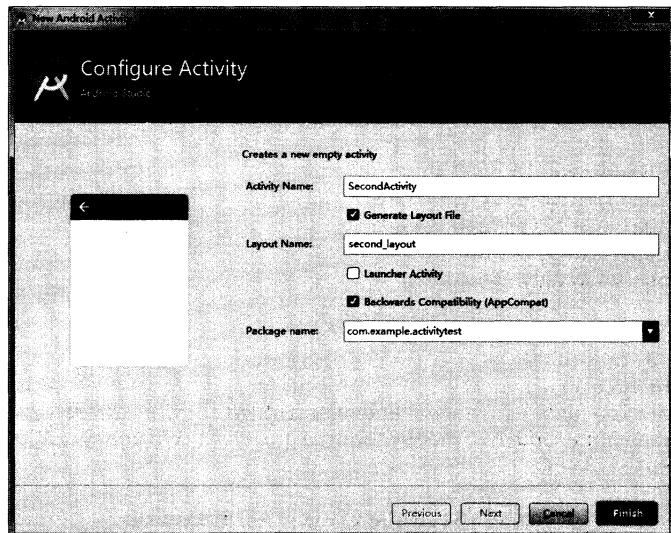


图 2.14 创建 SecondActivity

点击 Finish 完成创建, Android Studio 会为我们自动生成 SecondActivity.java 和 second\_layout.xml 这两个文件。不过自动生成的布局代码目前对你来说可能有些复杂, 这里我们仍然还是使用最熟悉的 LinearLayout, 编辑 second\_layout.xml, 将里面的代码替换成如下内容:

```
<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_2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 2"
    />

</LinearLayout>
```

我们还是定义了一个按钮, 按钮上显示 Button 2。

然后 SecondActivity 中的代码已经自动生成了一部分, 我们保持默认不变就好, 如下所示:

```
public class SecondActivity extends AppCompatActivity {

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

另外不要忘记, 任何一个活动都是需要在 AndroidManifest.xml 中注册的, 不过幸运的是, Android Studio 已经帮我们自动完成了, 你可以打开 AndroidManifest.xml 瞧一瞧:

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

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

Intent 是 Android 程序中各组件之间进行交互的一种重要方式，它不仅可以指明当前组件想要执行的动作，还可以在不同组件之间传递数据。Intent 一般可被用于启动活动、启动服务以及发送广播等场景，由于服务、广播等概念你暂时还未涉及，那么本章我们的目光无疑就锁定在了启动活动上面。

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

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

修改 FirstActivity 中按钮的点击事件，代码如下所示：

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

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

重新运行程序，在 FirstActivity 的界面点击一下按钮，结果如图 2.15 所示。

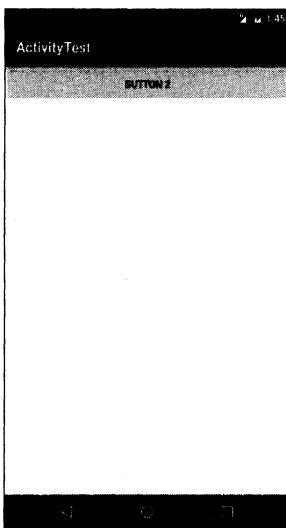


图 2.15 SecondActivity 界面

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

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

### 2.3.2 使用隐式 Intent

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

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

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

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

在<action>标签中我们指明了当前活动可以响应 com.example.activitytest.ACTION\_START 这个 action，而<category>标签则包含了一些附加信息，更精确地指明了当前的活动能

够响应的 Intent 中还可能带有的 category。只有<action>和<category>中的内容同时能够匹配上 Intent 中指定的 action 和 category 时，这个活动才能响应该 Intent。

修改 FirstActivity 中按钮的点击事件，代码如下所示：

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

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

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

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

修改 FirstActivity 中按钮的点击事件，代码如下所示：

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

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

现在重新运行程序，在 FirstActivity 的界面点击一下按钮，你会发现，程序崩溃了！这是你第一次遇到程序崩溃，可能会有些束手无策。别紧张，其实大多数的崩溃问题都是很好解决的，只要你善于分析。在 logcat 界面查看错误日志，你会看到如图 2.16 所示的错误信息。

```
Process: com.example.activitytest, PID: 24027
android.content.ActivityNotFoundException: No Activity found to handle Intent {
    act=com.example.activitytest.ACTION_START cat=[com.example.activitytest.MY_CATEGORY] }
```

图 2.16 错误信息

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

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

再次重新运行程序，你就会发现一切都正常了。

### 2.3.3 更多隐式 Intent 的用法

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

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

修改 FirstActivity 中按钮点击事件的代码，如下所示：

```
button1.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("http://www.baidu.com"));
        startActivity(intent);
    }
});
```

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

重新运行程序，在 FirstActivity 界面点击按钮就可以看到打开了系统浏览器，如图 2.17 所示。

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

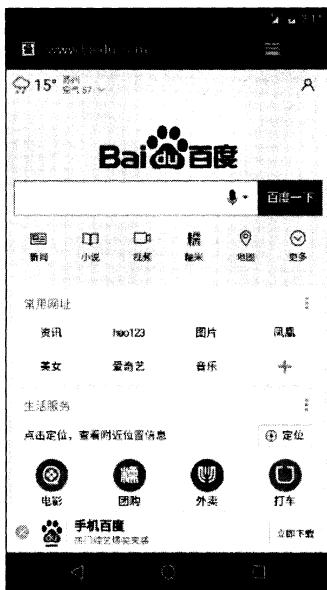


图 2.17 系统浏览器界面

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

- **android:scheme**。用于指定数据的协议部分，如上例中的 http 部分。
- **android:host**。用于指定数据的主机名部分，如上例中的 www.baidu.com 部分。
- **android:port**。用于指定数据的端口部分，一般紧随在主机名之后。
- **android:path**。用于指定主机名和端口之后的部分，如一段网址中跟在域名之后的内容。
- **android:mimeType**。用于指定可以处理的数据类型，允许使用通配符的方式进行指定。

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

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

右击 com.example.activitytest 包 → New → Activity → Empty Activity，新建 ThirdActivity，并勾选 Generate Layout File，给布局文件起名为 third\_layout，点击 Finish 完成创建。然后编辑 third\_layout.xml，将里面的代码替换成如下内容：

```
<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_3"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Button 3"
/>

```

```
</LinearLayout>
```

ThirdActivity 中的代码保持不变就可以了，最后在 AndroidManifest.xml 中修改 ThirdActivity 的注册信息：

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

我们在 ThirdActivity 的<intent-filter>中配置了当前活动能够响应的 action 是 Intent.ACTION\_VIEW 的常量值，而 category 则毫无疑问指定了默认的 category 值，另外在<data>标签中我们通过 android:scheme 指定了数据的协议必须是 http 协议，这样 ThirdActivity 应该就和浏览器一样，能够响应一个打开网页的 Intent 了。让我们运行一下程序试试吧，在 FirstActivity 的界面点击一下按钮，结果如图 2.18 所示。

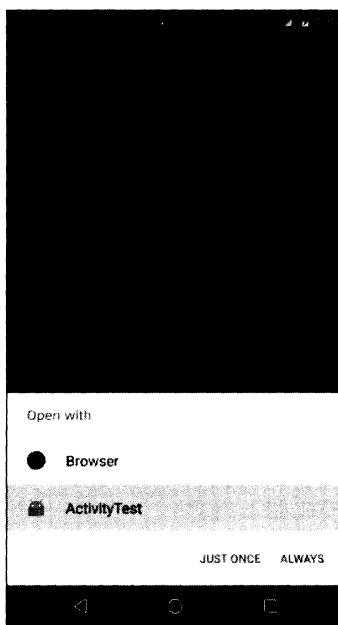


图 2.18 选择响应 Intent 的程序

可以看到，系统自动弹出了一个列表，显示了目前能够响应这个 Intent 的所有程序。选择 Browser 还会像之前一样打开浏览器，并显示百度的主页，而如果选择了 ActivityTest，则会启动 ThirdActivity。JUST ONCE 表示只是这次使用选择的程序打开，ALWAYS 则表示以后一直都使用这次选择的程序打开。需要注意的是，虽然我们声明了 ThirdActivity 是可以响应打开网页的 Intent 的，但实际上这个活动并没有加载并显示网页的功能，所以在真正的项目中尽量不要出现这种有可能误导用户的行为，不然会让用户对我们的应用产生负面的印象。

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

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

首先指定了 Intent 的 action 是 Intent.ACTION\_DIAL，这又是一个 Android 系统的内置动作。然后在 data 部分指定了协议是 tel，号码是 10086。重新运行一下程序，在 FirstActivity 的界面上点击一下按钮，结果如图 2.19 所示。

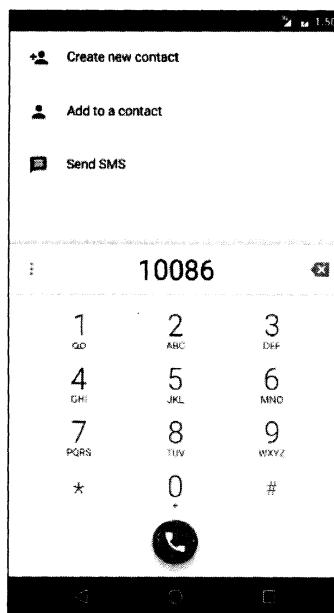


图 2.19 系统拨号界面

### 2.3.4 向下一个活动传递数据

经过前面几节的学习，你已经对 Intent 有了一定的了解。不过到目前为止，我们都只是简单地使用 Intent 来启动一个活动，其实 Intent 还可以在启动活动的时候传递数据，下面我们就一起来看一下。

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

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

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

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

```
public class SecondActivity extends AppCompatActivity {

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

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

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

打印信息，如图 2.20 所示。

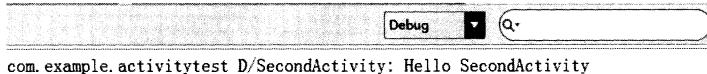


图 2.20 SecondActivity 中的打印信息

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

### 2.3.5 返回数据给上一个活动

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

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

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

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

```
public class SecondActivity extends AppCompatActivity {

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

```

        }
    });
}
}

```

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

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

```

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

```

`onActivityResult()` 方法带有三个参数，第一个参数 `requestCode`，即我们在启动活动时传入的请求码。第二个参数 `resultCode`，即我们在返回数据时传入的处理结果。第三个参数 `data`，即携带着返回数据的 Intent。由于在一个活动中有可能调用  `startActivityForResult()` 方法去启动很多不同的活动，每一个活动返回的数据都会回调到 `onActivityResult()` 这个方法中，因此我们首先要做的就是通过检查 `requestCode` 的值来判断数据来源。确定数据是从 `SecondActivity` 返回的之后，我们再通过 `resultCode` 的值来判断处理结果是否成功。最后从 `data` 中取值并打印出来，这样就完成了向上一个活动返回数据的工作。

重新运行程序，在 `FirstActivity` 的界面点击按钮会打开 `SecondActivity`，然后在 `SecondActivity` 界面点击 `Button 2` 按钮会回到 `FirstActivity`，这时查看 `logcat` 的打印信息，如图 2.21 所示。

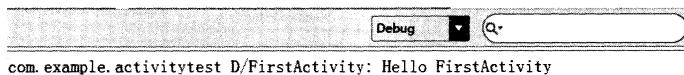


图 2.21 FirstActivity 中的打印信息

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

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

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

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

## 2.4 活动的生命周期

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

### 2.4.1 返回栈

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

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

示意图 2.22 展示了返回栈是如何管理活动入栈出栈操作的。

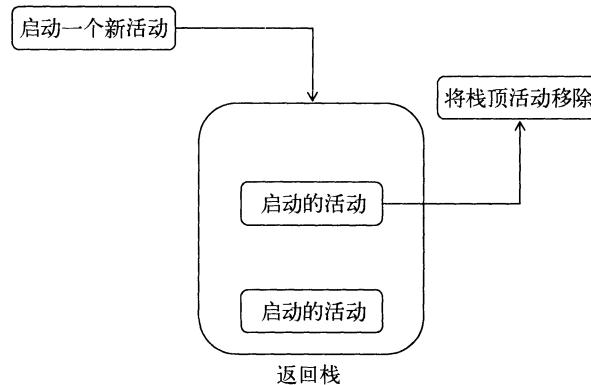


图 2.22 返回栈工作示意图

### 2.4.2 活动状态

每个活动在其生命周期中最多可能会有 4 种状态。

#### 1. 运行状态

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

#### 2. 暂停状态

当一个活动不再处于栈顶位置，但仍然可见时，这时活动就进入了暂停状态。你可能会觉得既然活动已经不在栈顶了，还怎么会可见呢？这是因为并不是每一个活动都会占满整个屏幕的，比如对话框形式的活动只会占用屏幕中间的部分区域，你很快就会在后面看到这种活动。处于暂停状态的活动仍然是完全存活的，系统也不愿意去回收这种活动（因为它还是可见的，回收可见的东西都会在用户体验方面有不好的影响），只有在内存极低的情况下，系统才会去考虑回收这种活动。

#### 3. 停止状态

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

#### 4. 销毁状态

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

### 2.4.3 活动的生存期

Activity 类中定义了 7 个回调方法，覆盖了活动生命周期的每一个环节，下面就来一一介绍这 7 个方法。

- ❑ **onCreate()**。这个方法你已经看到过很多次了，每个活动中我们都重写了这个方法，它会在活动第一次被创建的时候调用。你应该在这个方法中完成活动的初始化操作，比如说加载布局、绑定事件等。
- ❑ **onStart()**。这个方法在活动由不可见变为可见的时候调用。
- ❑ **onResume()**。这个方法在活动准备好和用户进行交互的时候调用。此时的活动一定位于返回栈的栈顶，并且处于运行状态。
- ❑ **onPause()**。这个方法在系统准备去启动或者恢复另一个活动的时候调用。我们通常会在这个方法中将一些消耗 CPU 的资源释放掉，以及保存一些关键数据，但这个方法的执行速度一定要快，不然会影响到新的栈顶活动的使用。
- ❑ **onStop()**。这个方法在活动完全不可见的时候调用。它和 **onPause()**方法的主要区别在于，如果启动的新活动是一个对话框式的活动，那么 **onPause()**方法会得到执行，而 **onStop()**方法并不会执行。
- ❑ **onDestroy()**。这个方法在活动被销毁之前调用，之后活动的状态将变为销毁状态。
- ❑ **onRestart()**。这个方法在活动由停止状态变为运行状态之前调用，也就是活动被重新启动了。

以上 7 个方法中除了 **onRestart()**方法，其他都是两两相对的，从而又可以将活动分为 3 种生存期。

- ❑ **完整生存期**。活动在 **onCreate()**方法和 **onDestroy()**方法之间所经历的，就是完整生存期。一般情况下，一个活动会在 **onCreate()**方法中完成各种初始化操作，而在 **onDestroy()**方法中完成释放内存的操作。
- ❑ **可见生存期**。活动在 **onStart()**方法和 **onStop()**方法之间所经历的，就是可见生存期。在可见生存期内，活动对于用户总是可见的，即便有可能无法和用户进行交互。我们可以通过这两个方法，合理地管理那些对用户可见的资源。比如在 **onStart()**方法中对资源进行加载，而在 **onStop()**方法中对资源进行释放，从而保证处于停止状态的活动不会占用过多内存。
- ❑ **前台生存期**。活动在 **onResume()**方法和 **onPause()**方法之间所经历的就是前台生存期。在前台生存期内，活动总是处于运行状态的，此时的活动是可以和用户进行交互的，我们平时看到和接触最多的也就是这个状态下的活动。

为了帮助你能够更好地理解，Android 官方提供了一张活动生命周期的示意图，如图 2.23 所示。

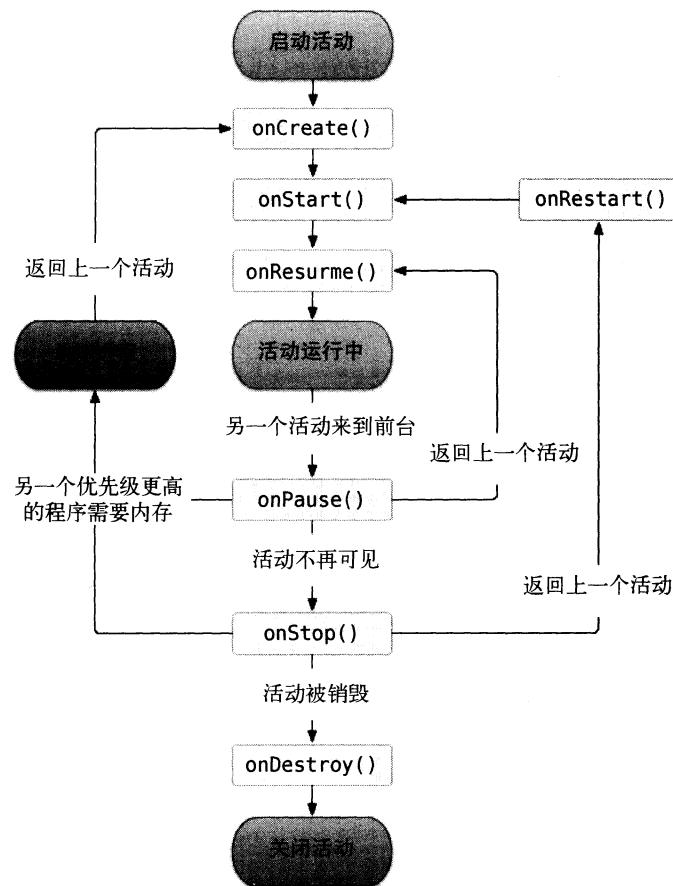


图 2.23 活动的生命周期

#### 2.4.4 体验活动的生命周期

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

这次我们不准备在 ActivityTest 这个项目的基础上修改了，而是新建一个项目。因此，首先关闭 ActivityTest 项目，点击导航栏 File→Close Project。然后再新建一个 ActivityLifeCycleTest 项目，新建项目的过程你应该已经非常清楚了，不需要我再进行赘述，这次我们允许 Android Studio 帮我们自动创建活动和布局，并且勾选 Launcher Activity 来将创建的活动设置为主活动，这样可以省去不少工作，创建的活动名和布局名都使用默认值。

这样主活动就创建完成了，我们还需要分别再创建两个子活动——NormalActivity 和 DialogActivity，下面一步步来实现。

右击 com.example.activitylifecycletest 包→New→Activity→Empty Activity，新建 NormalActivity，布局起名为 normal\_layout。然后使用同样的方式创建 DialogActivity，布局起名为 dialog\_layout。

现在编辑 normal\_layout.xml 文件，将里面的代码替换成如下内容：

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

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

</LinearLayout>
```

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

然后再编辑 dialog\_layout.xml 文件，将里面的代码替换成如下内容：

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

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

</LinearLayout>
```

两个布局文件的代码几乎没有区别，只是显示的文字不同而已。

NormalActivity 和 DialogActivity 中的代码我们保持默认就好，不需要改动。

其实从名字上你就可以看出，这两个活动一个是普通的活动，一个是对话框式的活动。可是我们并没有修改活动的任何代码，两个活动的代码应该几乎是一模一样的，在哪里有体现出将活动设成对话框式的呢？别着急，下面我们马上开始设置。修改 AndroidManifest.xml 的<activity>标签的配置，如下所示：

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

这里是两个活动的注册代码，但是 DialogActivity 的代码有些不同，我们给它使用了一个 android:theme 属性，这是用于给当前活动指定主题的，Android 系统内置有很多主题可以选择，当然我们也可以定制自己的主题，而这里@android:style/Theme.Dialog 则毫无疑问是让 DialogActivity 使用对话框式的主题。

接下来我们修改 activity\_main.xml，重新定制主活动的布局，将里面的代码替换成如下内容：

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

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

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

</LinearLayout>
```

可以看到，我们在 LinearLayout 中加入了两个按钮，一个用于启动 NormalActivity，一个用于启动 DialogActivity。

最后修改 MainActivity 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    public static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button startNormalActivity = (Button) findViewById(R.id.start_normal_activity);
        Button startDialogActivity = (Button) findViewById(R.id.start_dialog_activity);
        startNormalActivity.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(MainActivity.this, NormalActivity.class);
                startActivity(intent);
            }
        });
        startDialogActivity.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
```

```
        Intent intent = new Intent(MainActivity.this, DialogActivity.class);
        startActivity(intent);
    }
});

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

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

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

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

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

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

}
```

在 `onCreate()` 方法中，我们分别为两个按钮注册了点击事件，点击第一个按钮会启动 `NormalActivity`，点击第二个按钮会启动 `DialogActivity`。然后在 `Activity` 的 7 个回调方法中分别打印了一句话，这样就可以通过观察日志的方式来更直观地理解活动的生命周期。

现在运行程序，效果如图 2.24 所示。

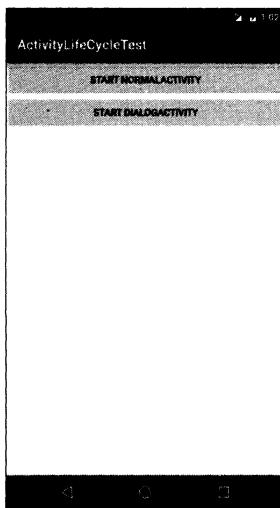


图 2.24 MainActivity 界面

这时观察 logcat 中的打印日志，如图 2.25 所示。

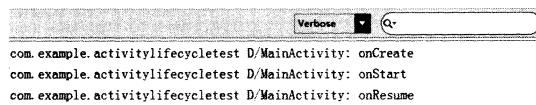


图 2.25 启动程序时的打印日志

可以看到，当 MainActivity 第一次被创建时会依次执行 `onCreate()`、`onStart()` 和 `onResume()` 方法。然后点击第一个按钮，启动 NormalActivity，如图 2.26 所示。

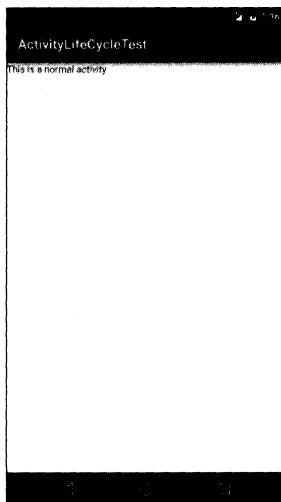


图 2.26 NormalActivity 界面

此时的打印信息如图 2.27 所示。

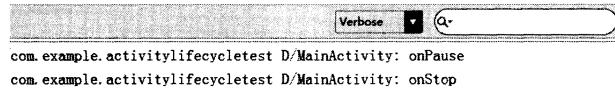


图 2.27 打开 NormalActivity 时的打印日志

由于 NormalActivity 已经把 MainActivity 完全遮挡住，因此 onPause() 和 onStop() 方法都会得到执行。然后按下 Back 键返回 MainActivity，打印信息如图 2.28 所示。

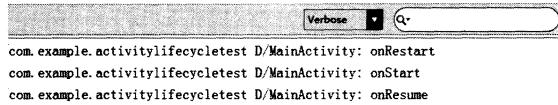


图 2.28 返回 MainActivity 的打印日志

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

然后再点击第二个按钮，启动 DialogActivity，如图 2.29 所示。



图 2.29 DialogActivity 界面

此时观察打印信息，如图 2.30 所示。

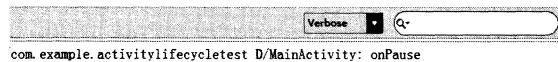


图 2.30 打开 DialogActivity 时的打印日志

可以看到，只有 `onPause()` 方法得到了执行，`onStop()` 方法并没有执行，这是因为 `DialogActivity` 并没有完全遮挡住 `MainActivity`，此时 `MainActivity` 只是进入了暂停状态，并没有进入停止状态。相应地，按下 Back 键返回 `MainActivity` 也应该只有 `onResume()` 方法会得到执行，如图 2.31 所示。

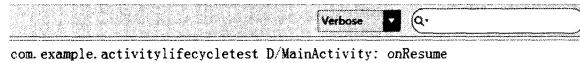


图 2.31 再次返回 `MainActivity` 的打印日志

最后在 `MainActivity` 按下 Back 键退出程序，打印信息如图 2.32 所示。

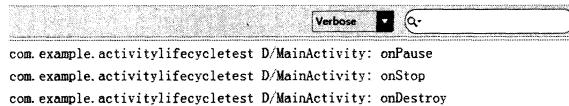


图 2.32 退出程序时的打印日志

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

这样活动完整的生命周期你已经体验了一遍，是不是理解得更加深刻了？

## 2.4.5 活动被回收了怎么办

前面我们已经说过，当一个活动进入到了停止状态，是有可能被系统回收的。那么想象以下场景：应用中有一个活动 A，用户在活动 A 的基础上启动了活动 B，活动 A 就进入了停止状态，这个时候由于系统内存不足，将活动 A 回收掉了，然后用户按下 Back 键返回活动 A，会出现什么情况呢？其实还是会正常显示活动 A 的，只不过这时并不会执行 `onRestart()` 方法，而是会执行活动 A 的 `onCreate()` 方法，因为活动 A 在这种情况下会被重新创建一次。

这样看上去好像一切正常，可是别忽略了一个重要问题，活动 A 中是可能存在临时数据和状态的。打个比方，`MainActivity` 中有一个文本输入框，现在你输入了一段文字，然后启动 `NormalActivity`，这时 `MainActivity` 由于系统内存不足被回收掉，过了一会你又点击了 Back 键回到 `MainActivity`，你会发现刚刚输入的文字全部都没了，因为 `MainActivity` 被重新创建了。

如果我们的应用出现了这种情况，是会严重影响用户体验的，所以必须要想想办法解决这个问题。查阅文档可以看出，`Activity` 中还提供了一个 `onSaveInstanceState()` 回调方法，这个方法可以保证在活动被回收之前一定会被调用，因此我们可以通过这个方法来解决活动被回收时临时数据得不到保存的问题。

`onSaveInstanceState()` 方法会携带一个 `Bundle` 类型的参数，`Bundle` 提供了一系列的方法用于保存数据，比如可以使用 `putString()` 方法保存字符串，使用 `.putInt()` 方法保存整型数据，以此类推。每个保存方法需要传入两个参数，第一个参数是键，用于后面从 `Bundle` 中取值，

第二个参数是真正要保存的内容。

在 MainActivity 中添加如下代码就可以将临时数据进行保存：

```
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    String tempData = "Something you just typed";
    outState.putString("data_key", tempData);
}
```

数据是已经保存下来了，那么我们应该在哪里进行恢复呢？细心的你也许早就发现，我们一直使用的 `onCreate()` 方法其实也有一个 `Bundle` 类型的参数。这个参数在一般情况下都是 `null`，但是如果在活动被系统回收之前有通过 `onSaveInstanceState()` 方法来保存数据的话，这个参数就会带有之前所保存的全部数据，我们只需要再通过相应的取值方法将数据取出即可。

修改 MainActivity 的 `onCreate()` 方法，如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate");
    setContentView(R.layout.activity_main);
    if (savedInstanceState != null) {
        String tempData = savedInstanceState.getString("data_key");
        Log.d(TAG, tempData);
    }
    ...
}
```

取出值之后再做相应的恢复操作就可以了，比如说将文本内容重新赋值到文本输入框上，这里我们只是简单地打印一下。

不知道你有没有察觉，使用 `Bundle` 来保存和取出数据是不是有些似曾相识呢？没错！我们在使用 `Intent` 传递数据时也是用的类似的方法。这里跟你提醒一点，`Intent` 还可以结合 `Bundle` 一起用于传递数据，首先可以把需要传递的数据都保存在 `Bundle` 对象中，然后再将 `Bundle` 对象存放在 `Intent` 里。到了目标活动之后先从 `Intent` 中取出 `Bundle`，再从 `Bundle` 中一一取出数据。具体的代码我就不写了，要学会举一反三哦。

## 2.5 活动的启动模式

活动的启动模式对你来说应该是个全新的概念，在实际项目中我们应该根据特定的需求为每个活动指定恰当的启动模式。启动模式一共有 4 种，分别是 `standard`、`singleTop`、`singleTask` 和 `singleInstance`，可以在 `AndroidManifest.xml` 中通过给 `<activity>` 标签指定 `android:launchMode` 属性来选择启动模式。下面我们就逐个进行学习。

### 2.5.1 standard

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

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

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

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

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

现在重新运行程序，然后在 FirstActivity 界面连续点击两次按钮，可以看到 logcat 中打印信息如图 2.33 所示。

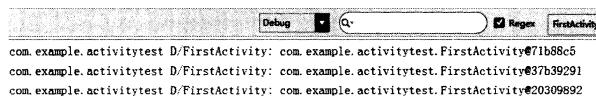


图 2.33 standard 模式下的打印日志

从打印信息中我们就可以看出，每点击一次按钮就会创建出一个新的 FirstActivity 实例。此时返回栈中也会存在 3 个 FirstActivity 的实例，因此你需要按压 3 次 Back 键才能退出程序。

standard 模式的原理示意图，如图 2.34 所示。

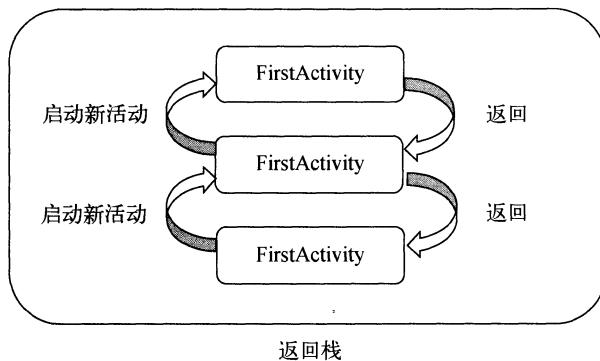


图 2.34 standard 模式示意图

### 2.5.2 singleTop

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

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

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

然后重新运行程序，查看 logcat 会看到已经创建了一个 FirstActivity 的实例，如图 2.35 所示。

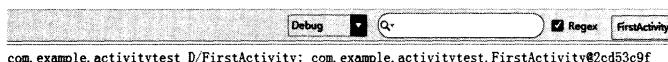


图 2.35 singleTop 模式下的打印日志

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

不过当 FirstActivity 并未处于栈顶位置时，这时再启动 FirstActivity，还是会创建新的实例的。

下面我们来实验一下，修改 FirstActivity 中 onCreate() 方法的代码，如下所示：

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

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

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

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

查看 logcat 中的打印信息，如图 2.36 所示。

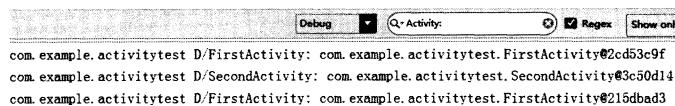


图 2.36 singleTop 模式下的打印日志

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

Back 键才会退出程序。

singleTop 模式的原理示意图，如图 2.37 所示。

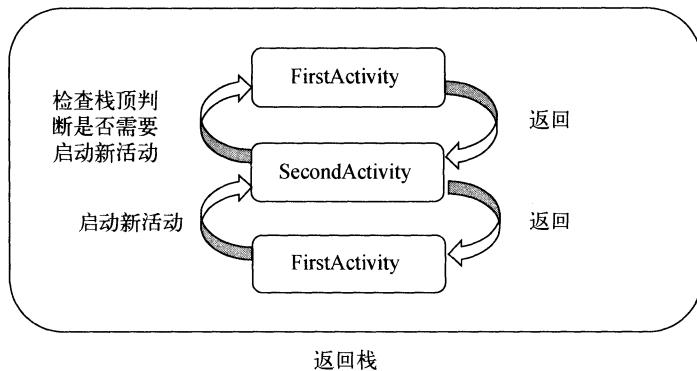


图 2.37 singleTop 模式示意图

### 2.5.3 singleTask

使用 singleTop 模式可以很好地解决重复创建栈顶活动的问题，但是正如你在上一节所看到的，如果该活动并没有处于栈顶的位置，还是可能会创建多个活动实例的。那么有没有什么办法可以让某个活动在整个应用程序的上下文中只存在一个实例呢？这就要借助 singleTask 模式来实现了。当活动的启动模式指定为 singleTask，每次启动该活动时系统首先会在返回栈中检查是否存在该活动的实例，如果发现已经存在则直接使用该实例，并把在这个活动之上的所有活动统统出栈，如果没有发现就会创建一个新的活动实例。

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

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

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

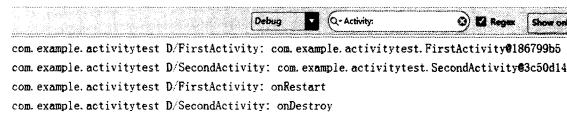
```
@Override
protected void onRestart() {
    super.onRestart();
    Log.d("FirstActivity", "onRestart");
}
```

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

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

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

查看 logcat 中的打印信息，如图 2.38 所示。



```
com.example.activitytest D/FirstActivity: com.example.activitytest.FirstActivity@186799b5
com.example.activitytest D/SecondActivity: com.example.activitytest.SecondActivity@3c50d14
com.example.activitytest D/FirstActivity: onRestart
com.example.activitytest D/SecondActivity: onDestroy
```

图 2.38 singleTask 模式下的打印日志

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

singleTask 模式的原理示意图，如图 2.39 所示。

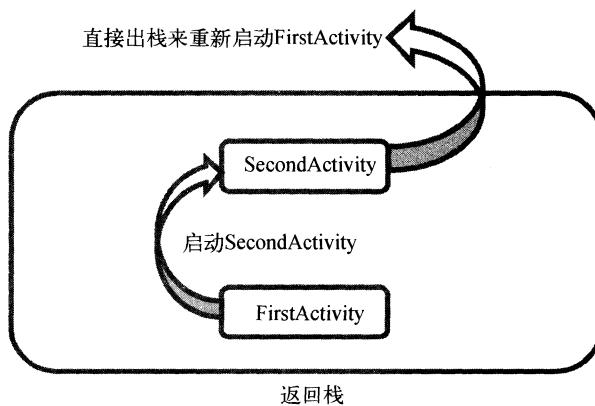


图 2.39 singleTask 模式示意图

#### 2.5.4 singleInstance

singleInstance 模式应该算是 4 种启动模式中最特殊也最复杂的一个了，你也需要多花点功夫来理解这个模式。不同于以上 3 种启动模式，指定为 singleInstance 模式的活动会启用一个新的返

回栈来管理这个活动（其实如果 singleTask 模式指定了不同的 taskAffinity，也会启动一个新的返回栈）。那么这样做有什么意义呢？想象以下场景，假设我们的程序中有一个活动是允许其他程序调用的，如果我们想实现其他程序和我们的程序可以共享这个活动的实例，应该如何实现呢？使用前面 3 种启动模式肯定是做不到的，因为每个应用程序都会有自己的返回栈，同一个活动在不同的返回栈中入栈时必然是创建了新的实例。而使用 singleInstance 模式就可以解决这个问题，在这种模式下会有一个单独的返回栈来管理这个活动，不管是哪个应用程序来访问这个活动，都共用的同一个返回栈，也就解决了共享活动实例的问题。

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

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

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

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

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

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("SecondActivity", "Task id is " + getTaskId());
    setContentView(R.layout.second_layout);
    Button button2 = (Button) findViewById(R.id.button_2);
    button2.setOnClickListener(new View.OnClickListener() {
        @Override
```

```

        public void onClick(View v) {
            Intent intent = new Intent(SecondActivity.this, ThirdActivity.class);
            startActivity(intent);
        }
    });
}

```

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

```

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

```

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

查看 logcat 中的打印信息，如图 2.40 所示。

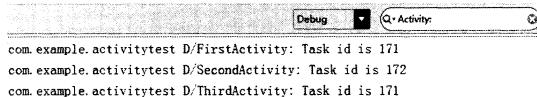


图 2.40 singleInstance 模式下的打印日志

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

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

`singleInstance` 模式的原理示意图，如图 2.41 所示。

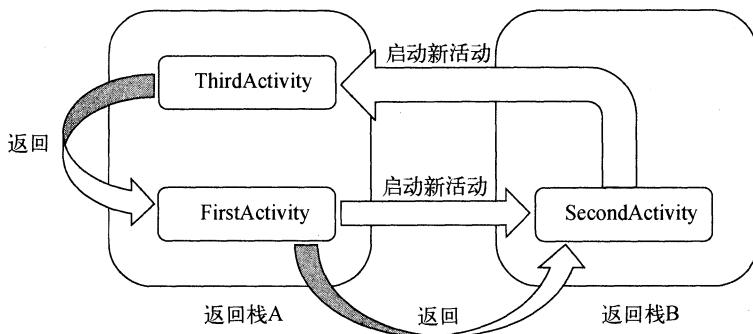


图 2.41 singleInstance 模式示意图

## 2.6 活动的最佳实践

你已经掌握了关于活动非常多的知识，不过恐怕离能够完全灵活运用还有一段距离。虽然知识点只有这么多，但运用的技巧却是多种多样。所以，在这里我准备教你几种关于活动的最佳实践技巧，这些技巧在你以后的开发工作当中将会非常受用。

### 2.6.1 知晓当前是在哪一个活动

这个技巧将教会你如何根据程序当前的界面就能判断出这是哪一个活动。可能你会觉得挺纳闷的，我自己写的代码怎么会不知道这是哪一个活动呢？很不幸的是，在你真正进入到企业之后，更有可能的是接手一份别人写的代码，因为你刚进公司就正好有一个新项目启动的概率并不高。阅读别人的代码时有一个很头疼的问题，就是当你需要在某个界面上修改一些非常简单的东西时，却半天找不到这个界面对应的活动是哪一个。学会了本节的技巧之后，这对你来说就再也不是难题了。

我们还是在 ActivityTest 项目的基础上修改，首先需要新建一个  `BaseActivity` 类。右击 com.example.activitytest 包 → New → Java Class，在弹出的窗口中输入  `BaseActivity`，如图 2.42 所示。

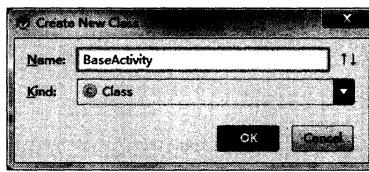


图 2.42 创建 BaseActivity 类

注意这里  `BaseActivity` 和普通活动的创建方式不一样，因为我们不需要让  `BaseActivity` 在 `AndroidManifest.xml` 中注册，所以选择创建一个普通的 Java 类就可以了。然后让  `BaseActivity` 继承自  `AppCompatActivity`，并重写 `onCreate()` 方法，如下所示：

```

public class BaseActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("BaseActivity", getClass().getSimpleName());
    }
}

```

我们在 `onCreate()` 方法中获取了当前实例的类名，并通过 `Log` 打印了出来。

接下来我们需要让 `BaseActivity` 成为 `ActivityTest` 项目中所有活动的父类。修改 `FirstActivity`、`SecondActivity` 和 `ThirdActivity` 的继承结构，让它们不再继承自 `AppCompatActivity`，而是继承自 `BaseActivity`。而由于 `BaseActivity` 又是继承自 `AppCompatActivity` 的，所以项目中所有活动的现有功能并不受影响，它们仍然完全继承了 `Activity` 中的所有特性。

现在重新运行程序，然后通过点击按钮分别进入到 `FirstActivity`、`SecondActivity` 和 `ThirdActivity` 的界面，这时观察 `logcat` 中的打印信息，如图 2.43 所示。

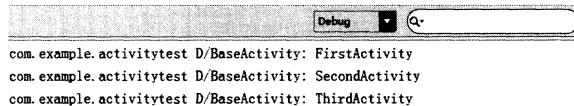


图 2.43 `BaseActivity` 中的打印日志

现在每当我们进入到一个活动的界面，该活动的类名就会被打印出来，这样我们就可以时刻知晓当前界面对应的是哪一个活动了。

## 2.6.2 随时随地退出程序

如果目前你手机的界面还停留在 `ThirdActivity`，你会发现当前想退出程序是非常不方便的，需要按压 3 次 `Back` 键才行。按 `Home` 键只是把程序挂起，并没有退出程序。其实这个问题就足以引起你的思考，如果我们的程序需要一个注销或者退出的功能该怎么办呢？必须要有一个随时随地都能退出程序的方案才行。

其实解决思路也很简单，只需要用一个专门的集合类对所有的活动进行管理就可以了，下面我们就来实现一下。

新建一个 `ActivityCollector` 类作为活动管理器，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();
        }
    }
}
}

```

在活动管理器中，我们通过一个 List 来暂存活动，然后提供了一个 `addActivity()` 方法用于向 List 中添加一个活动，提供了一个 `removeActivity()` 方法用于从 List 中移除活动，最后提供了一个 `finishAll()` 方法用于将 List 中存储的活动全部销毁掉。

接下来修改 `BaseActivity` 中的代码，如下所示：

```

public class BaseActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("BaseActivity", getClass().getSimpleName());
        ActivityCollector.addActivity(this);
    }

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

```

在 `BaseActivity` 的 `onCreate()` 方法中调用了 `ActivityCollector` 的 `addActivity()` 方法，表明将目前正在创建的活动添加到活动管理器里。然后在 `BaseActivity` 中重写 `onDestroy()` 方法，并调用了 `ActivityCollector` 的 `removeActivity()` 方法，表明将一个马上要销毁的活动从活动管理器里移除。

从此以后，不管你想在什么地方退出程序，只需要调用 `ActivityCollector.finishAll()` 方法就可以了。例如在 `ThirdActivity` 界面想通过点击按钮直接退出程序，只需将代码改成如下所示：

```

public class ThirdActivity extends BaseActivity {

    @Override

```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d("ThirdActivity", "Task id is " + getTaskId());
    setContentView(R.layout.third_layout);
    Button button3 = (Button) findViewById(R.id.button_3);
    button3.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            ActivityCollector.finishAll();
        }
    });
}
```

当然你还可以在销毁所有活动的代码后面再加上杀掉当前进程的代码，以保证程序完全退出，杀掉进程的代码如下所示：

```
    android.os.Process.killProcess(android.os.Process.myPid());
```

其中，`killProcess()`方法用于杀掉一个进程，它接收一个进程 `id` 参数，我们可以通过 `myPid()` 方法来获得当前程序的进程 `id`。需要注意的是，`killProcess()` 方法只能用于杀掉当前程序的进程，我们不能使用这个方法去杀掉其他程序。

### 2.6.3 启动活动的最佳写法

启动活动的方法相信你已经非常熟悉了，首先通过 Intent 构建出当前的“意图”，然后调用 `startActivity()` 或 `startActivityForResult()` 方法将活动启动起来，如果有数据需要从一个活动传递到另一个活动，也可以借助 Intent 来完成。

假设 SecondActivity 中需要用到两个非常重要的字符串参数，在启动 SecondActivity 的时候必须要传递过来，那么我们很容易会写出如下代码：

```
Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
intent.putExtra("param1", "data1");
intent.putExtra("param2", "data2");
startActivity(intent);
```

这样写是完全正确的，不管是从语法上还是规范上，只是在真正的项目开发中经常会有对接的问题出现。比如 SecondActivity 并不是由你开发的，但现在你负责的部分需要有启动 SecondActivity 这个功能，而你却不清楚启动这个活动需要传递哪些数据。这时无非就有两种办法，一个是你自己去阅读 SecondActivity 中的代码，二是询问负责编写 SecondActivity 的同事。你会不会觉得很麻烦呢？其实只需要换一种写法，就可以轻松解决掉上面的窘境。

修改 SecondActivity 中的代码，如下所示：

```
public class SecondActivity extends BaseActivity {
```

```

public static void actionStart(Context context, String data1, String data2) {
    Intent intent = new Intent(context, SecondActivity.class);
    intent.putExtra("param1", data1);
    intent.putExtra("param2", data2);
    context.startActivity(intent);
}
...
}

```

我们在 SecondActivity 中添加了一个 `actionStart()` 方法，在这个方法中完成了 Intent 的构建，另外所有 SecondActivity 中需要的数据都是通过 `actionStart()` 方法的参数传递过来的，然后把它们存储到 Intent 中，最后调用 `startActivity()` 方法启动 SecondActivity。

这样写的好处在哪里呢？最重要的一点就是一目了然，SecondActivity 所需要的数据在方法参数中全部体现出来了，这样即使不用阅读 SecondActivity 中的代码，不去询问负责编写 SecondActivity 的同事，你也可以非常清晰地知道启动 SecondActivity 需要传递哪些数据。另外，这样写还简化了启动活动的代码，现在只需要一行代码就可以启动 SecondActivity，如下所示：

```

button1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        SecondActivity.actionStart(FirstActivity.this, "data1", "data2");
    }
});

```

养成一个良好的习惯，给你编写的每个活动都添加类似的启动方法，这样不仅可以让启动活动变得非常简单，还可以节省不少有同事过来询问你的时间。

## 2.7 小结与点评

真是好疲惫啊！没错，学习了这么多的东西不疲惫才怪呢。但是，你内心那种掌握了知识的喜悦感相信也是无法掩盖的。本章的收获非常多啊，不管是理论型还是实践型的东西都涉及了，从活动的基本用法，到启动活动和传递数据的方式，再到活动的生命周期，以及活动的启动模式，你几乎已经学会了关于活动所有重要的知识点。另外在本章的最后，还学习了几种可以应用在活动中的最佳实践技巧，毫不夸张地说，你在 Android 活动方面已经算是一个小高手了。

不过你的 Android 旅途才刚刚开始呢，后面需要学习的东西还很多，也许会比现在还累，一定要做好心理准备哦。总体来说，我给你现在的状态打满分，毕竟你已经学会了那么多的东西，也是时候放松一下了。自己适当控制一下休息的时间，然后我们继续前进吧！