

第 13 章 继续进阶，你还应该掌握的高级技巧

本书的内容虽然已经接近尾声了，但是千万不要因此而放松，现在正是你继续进阶的时机。相信基础性的 Android 知识已经没有什么能够难倒你的了，那么本章中我们就来学习一些你还应该掌握的高级技巧吧。

13.1 全局获取 Context 的技巧

回想这么久以来我们所学的内容，你会发现有很多地方都需要用到 Context，弹出 Toast 的时候需要、启动活动的时候需要、发送广播的时候需要、操作数据库的时候需要、使用通知的时候需要等等等等。

或许目前你还没有为得不到 Context 而发愁过，因为我们很多的操作都是在活动中进行的，而活动本身就是一个 Context 对象。但是，当应用程序的架构逐渐开始复杂起来的时候，很多的逻辑代码都将脱离 Activity 类，但此时你又恰恰需要使用 Context，也许这个时候你就会感到有些伤脑筋了。

举个例子来说吧，在第 10 章的最佳实践环节，我们编写了一个 HttpUtil 类，在这里将一些通用的网络操作封装了起来，代码如下所示：

```
public class HttpUtil {

    public static void sendHttpRequest(final String address, final
    HttpCallbackListener listener) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                HttpURLConnection connection = null;
                try {
                    URL url = new URL(address);
                    connection = (HttpURLConnection) url.openConnection();
                    connection.setRequestMethod("GET");
```

```

        connection.setConnectTimeout(8000);
        connection.setReadTimeout(8000);
        connection.setDoInput(true);
        connection.setDoOutput(true);
        InputStream in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new
InputStreamReader(in));
        StringBuilder response = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        if (listener != null) {
            listener.onFinish(response.toString());
        }
    } catch (Exception e) {
        if (listener != null) {
            listener.onError(e);
        }
    } finally {
        if (connection != null) {
            connection.disconnect();
        }
    }
}

}).start();
}

}

```

这里使用 `sendHttpRequest()` 方法来发送 HTTP 请求显然是没有问题的，并且我们还可以在回调方法中处理服务器返回的数据。但现在我们想对 `sendHttpRequest()` 方法进行一些优化，当检测到网络不存在的时候就给用户一个 Toast 提示，并且不再执行后面的代码。看似一个挺简单的功能，可是却存在一个让人头疼的问题，弹出 Toast 提示需要一个 Context 参数，而我们在 `HttpUtil` 类中显然是获取不到 Context 对象的，这该怎么办呢？

其实要想快速解决这个问题也很简单，大不了在 `sendHttpRequest()` 方法中添加一个 Context 参数就行了嘛，于是可以将 `HttpUtil` 中的代码进行如下修改：

```

public class HttpUtil {

```

```

    public static void sendHttpRequest(final Context context,
        final String address, final HttpCallbackListener listener) {
        if (!isNetworkAvailable()) {
            Toast.makeText(context, "network is unavailable",
                Toast.LENGTH_SHORT).show();
            return;
        }
        new Thread(new Runnable() {
            @Override
            public void run() {
                .....
            }
        }).start();
    }

    private static boolean isNetworkAvailable() {
        .....
    }
}

```

可以看到，这里在方法中添加了一个 Context 参数，并且假设有一个 isNetworkAvailable() 方法用于判断当前网络是否可用，如果网络不可用的话就弹出 Toast 提示，并将方法 return 掉。

虽说这也确实是一种解决方案，但是却有点推卸责任的嫌疑，因为我们将获取 Context 的任务转移给了 sendHttpRequest() 方法的调用方，至于调用方能不能得到 Context 对象，那就不是我们需要考虑的问题了。

由此可以看出，在某些情况下，获取 Context 并非是那么容易的一件事，有时候还是挺伤脑筋的。不过别担心，下面我们就来学习一种技巧，让你在项目的任何地方都能够轻松获取到 Context。

Android 提供了一个 Application 类，每当应用程序启动的时候，系统就会自动将这个类进行初始化。而我们可以定制一个自己的 Application 类，以便于管理程序内一些全局的状态信息，比如说全局 Context。

定制一个自己 Application 其实并不复杂，首先我们需要创建一个 MyApplication 类继承自 Application，代码如下所示：

```

public class MyApplication extends Application {

    private static Context context;
}

```

```

@Override
public void onCreate() {
    context = getApplicationContext();
}

public static Context getContext() {
    return context;
}
}

```

可以看到，MyApplication 中的代码非常简单。这里我们重写了父类的 onCreate() 方法，并通过调用 getApplicationContext() 方法得到了一个应用程序级别的 Context，然后又提供了一个静态的 getContext() 方法，在这里将刚才获取到的 Context 进行返回。

接下来我们需要告知系统，当程序启动的时候应该初始化 MyApplication 类，而不是默认的 Application 类。这一步也很简单，在 AndroidManifest.xml 文件的 <application> 标签下进行指定就可以了，代码如下所示：

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.networktest"
    android:versionCode="1"
    android:versionName="1.0" >
    .....
    <application
        android:name="com.example.networktest.MyApplication"
        ..... >
        .....
    </application>
</manifest>

```

注意这里在指定 MyApplication 的时候一定要加上完整的包名，不然系统将无法找到这个类。

这样我们就已经实现了一种全局获取 Context 的机制，之后不管你想在项目的任何地方使用 Context，只需要调用一下 MyApplication.getContext() 就可以了。

那么接下来我们再对 sendHttpRequest() 方法进行优化，代码如下所示：

```

public static void sendHttpRequest(final String address, final
HttpCallbackListener listener) {
    if (!isNetworkAvailable()) {
        Toast.makeText(MyApplication.getContext(), "network is unavailable",
            Toast.LENGTH_SHORT).show();
    }
}

```

```

        return;
    }
    .....
}

```

可以看到，`sendHttpRequest()`方法不需要再通过传参的方式来得到 `Context` 对象，而只需调用一下 `MyApplication.getContext()`方法就可以了。有了这个技巧，你再也不用为得不到 `Context` 对象而发愁了。

13.2 使用 Intent 传递对象

`Intent` 的用法相信你 already 比较熟悉了，我们可以借助它来启动活动、发送广播、启动服务等。在进行上述操作的时候，我们还可以在 `Intent` 中添加一些附加数据，以达到传值的效果，比如在 `FirstActivity` 中添加如下代码：

```

Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
intent.putExtra("string_data", "hello");
intent.putExtra("int_data", 100);
startActivity(intent);

```

这里调用了 `Intent` 的 `putExtra()`方法来添加要传递的数据，之后在 `SecondActivity` 中就可以得到这些值了，代码如下所示：

```

getIntent().getStringExtra("string_data");
getIntent().getIntExtra("int_data", 0);

```

但是不知道你有没有发现，`putExtra()`方法中所支持的数据类型是有限的，虽然常用的一些数据类型它都会支持，但是当你想去传递一些自定义对象的时候就会发现无从下手。不用担心，下面我们就学习一下使用 `Intent` 来传递对象的技巧。

13.2.1 Serializable 方式

使用 `Intent` 来传递对象通常有两种实现方式，`Serializable` 和 `Parcelable`，本小节中我们先来学习一下第一种实现方式。

`Serializable` 是序列化的意思，表示将一个对象转换成可存储或可传输的状态。序列化后的对象可以在网络上进行传输，也可以存储到本地。至于序列化的方法也很简单，只需要让一个类去实现 `Serializable` 这个接口就可以了。

比如说有一个 `Person` 类，其中包含了 `name` 和 `age` 这两个字段，想要将它序列化就可以这样写：

```

public class Person implements Serializable{

```

```

    private String name;

    private int age;

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

其中 get、set 方法都是用于赋值和读取字段数据的，最重要的部分是在第一行。这里让 Person 类去实现了 Serializable 接口，这样所有的 Person 对象就都是可序列化的了。

接下来在 FirstActivity 中的写法非常简单：

```

Person person = new Person();
person.setName("Tom");
person.setAge(20);
Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
intent.putExtra("person_data", person);
startActivity(intent);

```

可以看到，这里我们创建了一个 Person 的实例，然后就直接将它传入到 putExtra() 方法中了。由于 Person 类实现了 Serializable 接口，所以才可以这样写。

接下来在 SecondActivity 中获取这个对象也很简单，写法如下：

```

Person person = (Person) getIntent().getSerializableExtra("person_data");

```

这里调用了 getSerializableExtra() 方法来获取通过参数传递过来的序列化对象，接着再将它向下转型成 Person 对象，这样我们就成功实现了使用 Intent 来传递对象的功能了。

13.2.2 Parcelable 方式

除了 Serializable 之外，使用 Parcelable 也可以实现相同的效果，不过不同于将对象进行序列化，Parcelable 方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是 Intent 所支持的数据类型，这样也就实现传递对象的功能了。

下面我们来看一下 Parcelable 的实现方式，修改 Person 中的代码，如下所示：

```
public class Person implements Parcelable {

    private String name;

    private int age;
    .....

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name); // 写出name
        dest.writeInt(age); // 写出age
    }

    public static final Parcelable.Creator<Person> CREATOR = new Parcelable.
    Creator<Person>() {

        @Override
        public Person createFromParcel(Parcel source) {
            Person person = new Person();
            person.name = source.readString(); // 读取name
            person.age = source.readInt(); // 读取age
            return person;
        }

        @Override
        public Person[] newArray(int size) {
            return new Person[size];
        }
    };
}
```

Parcelable 的实现方式要稍微复杂一些。可以看到，首先我们让 Person 类去实现了 Parcelable 接口，这样就必须重写 describeContents() 和 writeToParcel() 这两个方法。其中 describeContents() 方法直接返回 0 就可以了，而 writeToParcel() 方法中我们需要调用 Parcel 的 writeXxx() 方法将 Person 类中的字段一一写出。注意字符串型数据就调用 writeString() 方法，整型数据就调用 writeInt() 方法，以此类推。

除此之外，我们还必须在 Person 类中提供一个名为 CREATOR 的常量，这里创建了 Parcelable.Creator 接口的一个实现，并将泛型指定为 Person。接着需要重写 createFromParcel() 和 newArray() 这两个方法，在 createFromParcel() 方法中我们要去读取刚才写出的 name 和 age 字段，并创建一个 Person 对象进行返回，其中 name 和 age 都是调用 Parcel 的 readXxx() 方法读取到的，注意这里读取的顺序一定要和刚才写出的顺序完全相同。而 newArray() 方法中的实现就简单多了，只需要 new 出一个 Person 数组，并使用方法中传入的 size 作为数组大小就可以了。

接下来在 FirstActivity 中我们仍然可以使用相同的代码来传递 Person 对象，只不过在 SecondActivity 中获取对象的时候需要稍加改动，如下所示：

```
Person person = (Person) getIntent().getParcelableExtra("person_data");
```

注意这里不再是调用 getSerializableExtra() 方法，而是调用 getParcelableExtra() 方法来获取传递过来的对象了，其他的地方都完全相同。

这样我们就把使用 Intent 来传递对象的两种实现方式都学习完了，对比一下，Serializable 的方式较为简单，但由于会把整个对象进行序列化，因此效率方面会比 Parcelable 方式低一些，所以在通常情况下还是更加推荐使用 Parcelable 的方式来实现 Intent 传递对象的功能。

13.3 定制自己的日志工具

早在第 1 章的 1.4 节中我们就已经学过了 Android 日志工具的用法，并且日志工具也确实贯穿了我们整本书的学习，基本上每一章都有用到过。虽然 Android 中自带的日志工具功能非常强大，但也不能说是完全没有缺点，例如在打印日志的控制方面就做得不够好。

打个比方，你正在编写一个比较庞大的项目，期间为了方便调试，在代码的很多地方都打印了大量的日志。最近项目已经基本完成了，但是却有一个非常让人头疼的问题，之前用于调试的那些日志，在项目正式上线之后仍然会照常打印，这样不仅会降低程序的运行效率，还有可能将一些机密性的数据泄露出去。

那该怎么办呢，难道要一行一行把所有打印日志的代码都删掉？显然这不是什么好点子，不仅费时费力，而且以后你继续维护这个项目的时候可能还会需要这些日志。因此，最理想的情况是能够自由地控制日志的打印，当程序处于开发阶段就让日志打印出来，当程序上线了之后就把日志屏蔽掉。

看起来好像是挺高级的一个功能，其实并不复杂，我们只需要定制一个自己的日志工具就可以轻松完成了。比如新建一个 LogUtil 类，代码如下所示：

```
public class LogUtil {

    public static final int VERBOSE = 1;

    public static final int DEBUG = 2;

    public static final int INFO = 3;

    public static final int WARN = 4;

    public static final int ERROR = 5;

    public static final int NOTHING = 6;

    public static final int LEVEL = VERBOSE;

    public static void v(String tag, String msg) {
        if (LEVEL <= VERBOSE) {
            Log.v(tag, msg);
        }
    }

    public static void d(String tag, String msg) {
        if (LEVEL <= DEBUG) {
            Log.d(tag, msg);
        }
    }

    public static void i(String tag, String msg) {
        if (LEVEL <= INFO) {
            Log.i(tag, msg);
        }
    }

    public static void w(String tag, String msg) {
        if (LEVEL <= WARN) {
            Log.w(tag, msg);
        }
    }
}
```

```

    }
}

public static void e(String tag, String msg) {
    if (LEVEL <= ERROR) {
        Log.e(tag, msg);
    }
}
}

```

可以看到，我们在 LogUtil 中先是定义了 VERBOSE、DEBUG、INFO、WARN、ERROR、NOTHING 这六个整型常量，并且它们对应的值都是递增的。然后又定义了一个 LEVEL 常量，可以将它的值指定为上面六个常量中的任意一个。

接下来我们提供了 v()、d()、i()、w()、e() 这五个自定义的日志方法，在其内部分别调用了 Log.v()、Log.d()、Log.i()、Log.w()、Log.e() 这五个方法来打印日志，只不过在这些自定义的方法中都加入了一个 if 判断，只有当 LEVEL 常量的值小于或等于对应日志级别值的时候，才会将日志打印出来。

这样就把一个自定义的日志工具创建好了，之后在项目里我们可以像使用普通的日志工具一样使用 LogUtil，比如打印一行 DEBUG 级别的日志就可以这样写：

```
LogUtil.d("TAG", "debug log");
```

打印一行 WARN 级别的日志就可以这样写：

```
LogUtil.w("TAG", "warn log");
```

然后我们只需要修改 LEVEL 常量的值，就可以自由地控制日志的打印行为了。比如让 LEVEL 等于 VERBOSE 就可以把所有的日志都打印出来，让 LEVEL 等于 WARN 就可以只打印警告以上级别的日志，让 LEVEL 等于 NOTHING 就可以把所有日志都屏蔽掉。

使用了这种方法之后，刚才所说的那个问题就不复存在了，你只需要在开发阶段将 LEVEL 指定成 VERBOSE，当项目正式上线的时候将 LEVEL 指定成 NOTHING 就可以了。

13.4 调试 Android 程序

当开发过程中遇到一些奇怪的 bug，但又迟迟定位不出来原因是什么的时候，最好的办法就是进行调试。相信使用 Eclipse 来调试 Java 程序的功能你一定早就会用了，但如何使用它来调试 Android 程序呢？这也就是本小节的重点了。

还记得在第 5 章的最佳实践环节中编写的那个强制下线程序吗？就让我们通过这个例

子来学习下 Android 程序的调试方法吧。这个程序中有一个登录功能，比如说现在登录出现了问题，我们就可以通过调试来定位问题的原因。

不用多说，调试工作的第一步肯定是添加断点，这里由于我们要调试登录部分的问题，所以断点可以加在登录按钮的点击事件里面。添加断点的方法也很简单，只需要在相应代码行的左边双击一下就可以了，如图 13.1 所示。如果想要取消这个断点，对着它再次双击就可以了。

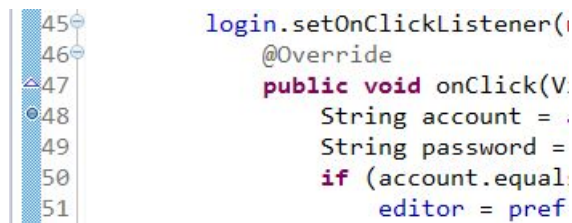


图 13.1

添加好了断点，接下来就可以对程序进行调试了，右击 BroadcastBestPractice 项目 → Debug As → Android Application，注意这里没有选择 Run As，而是 Debug As，表示我们要以调试模式来启动程序。等到程序运行起来的时候首先会看到一个提示框，如图 13.2 所示。

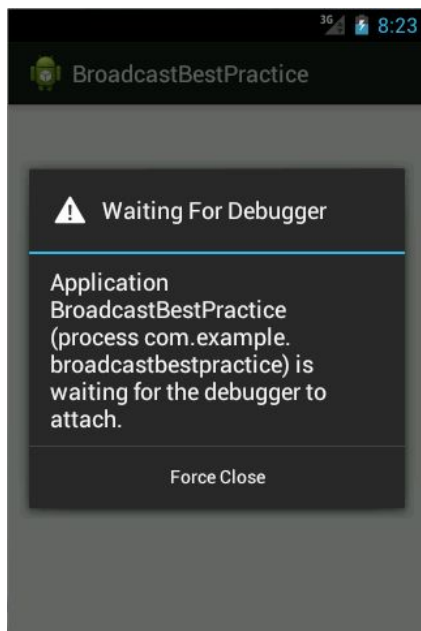


图 13.2

这个框很快就会自动消失，然后在输入框里输入账号和密码，并点击 Login 按钮，这时 Eclipse 就会自动跳转到 Debug 视图，如图 13.3 所示。

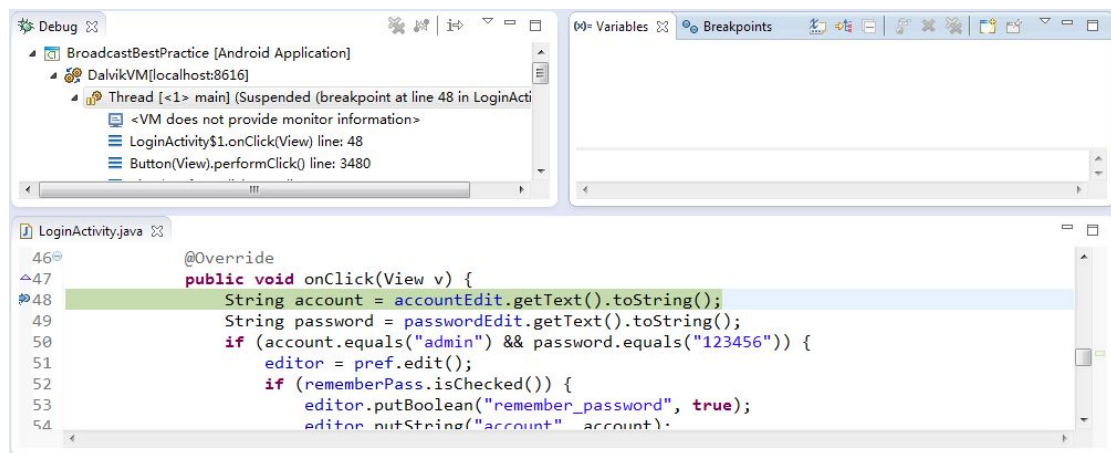


图 13.3

接下来每按一次 F6 键，代码就会向下执行一行，并且通过 Variables 视图还可以看到内存中的数据，如图 13.4 所示。

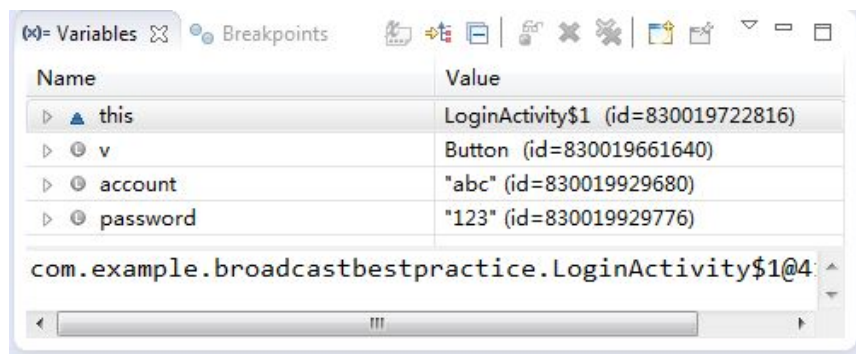


图 13.4

可以看到，我们从输入框里获取到的账号密码分别是 abc 和 123，而程序里要求正确的账号密码是 admin 和 123456，所以登录才会出现问题。这样我们就通过调试的方式轻松地把问题定位出来了，调试结束之后点击图 13.5 中的最右边的按钮来断开调试连接即可。



图 13.5

这种调试方式虽然完全可以正常工作，但在调试模式下程序的运行效率将会大大地降低，如果你的断点加在一个比较靠后的位置，需要执行很多的操作才能运行到这个断点，那么前面这些操作就都会有一些卡顿的感觉。没关系，Android 还提供了另外一种调试的方式，可以让程序随时进入到调试模式，下面我们就来尝试一下。

这次不需要选择 Debug As 来运行程序了，就使用 Run As→Android Application 来正常地启动程序，由于现在不是在调试模式下，程序的运行速度比较快，可以先把账号和密码输入好。然后进入到 DDMS 视图，在 Devices 窗口中们可以看到所有正在运行的进程，其中最后一个就是我们这个程序的进程，如图 13.6 所示。

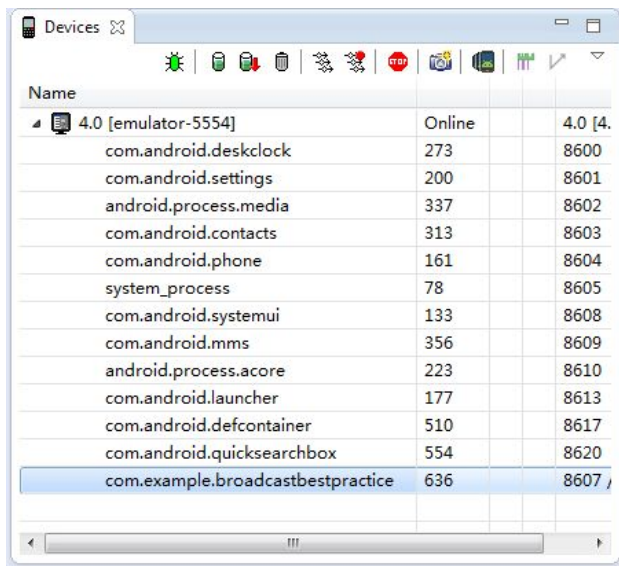


图 13.6

选中这个进程之后，点击最上面一行的第一个按钮，就会让这个进程进入到调试模式了。进入调试模式的进程名前会有一个虫子样式图标，如图 13.7 所示。


com.android.defcontainer	510
com.android.quicksearchbox	554
 com.example.broadcastbestpractice	636

图 13.7

接下来在模拟器中点击 Login 按钮，Eclipse 同样也会跳转到 Debug 视图，之后的流程都是相同的了。相比起来，第二种调试方式会比第一种更加灵活，也更加常用。

13.5 编写测试用例

测试是软件工程中一个非常重要的环节，而测试用例又可以显著地提高测试的效率和准确性。测试用例其实就是一段普通的程序代码，通常是带有期望的运行结果的，测试者可以根据最终的运行结果来判断程序是否能正常工作。

我相信大多数的程序员都是不喜欢编写测试用例的，因为这是一件很繁琐的事情。明明运行一下程序，观察运行结果就能知道对与错了，为什么还要通过代码来进行判断呢？确实，如果只是普通的一个小程序，编写测试用例是有些多此一举，但是当你正在维护一个非常庞大的工程时，你就会发现编写测试用例是非常有必要的。

举个例子吧，比如你确实正在维护一个很庞大的工程，里面有许许多多数也数不清的功能。某天，你的领导要求你对其中一个功能进行修改，难度也不高，你很快就解决了，并且测试通过。但是几天之后，突然有人发现其他功能出现了问题，最终定位出来的原因竟然就是你之前修改的那个功能所导致的！这下你可冤死了。不过千万别以为这是天方夜谭，在大型的项目中，这种情况还是很常见的。由于项目里的很多代码都是公用的，你为了完成一个功能而去修改某行代码，完全有可能因此而导致另一个功能无法正常工作。

所以，当项目比较庞大的时候，一般都应该去编写测试用例的。如果我们给项目的每一项功能都编写了测试用例，每当修改或新增任何功能之后，就将所有的测试用例都跑一遍，只要有任何测试用例没有通过，就说明修改或新增的这个功能影响到现有功能了，这样就可以及早地发现问题，避免事故的发生。

13.5.1 创建测试工程

介绍了这么多，也是时候该动手尝试一下了，下面我们就来创建一个测试工程。在创建之前你需要知道，测试工程通常都不是独立存在的，而是依赖于某个现有工程的，一般比较常见的做法是在现有工程下新建一个 `tests` 文件夹，测试工程就存放在这里。

那么我们就给 `BroadcastBestPractice` 这个项目创建一个测试工程吧。在 Eclipse 的导航栏中点击 `File`→`New`→`Other`，会打开一个对话框，展开 `Android` 目录，在里面选中 `Android Test Project`，如图 13.8 所示。

点击 `Next` 后会弹出创建 `Android` 测试工程的对话框，在这里我们可以输入测试工程的名字，并选择测试工程的路径。按照惯例，我们将路径选择为 `BroadcastBestPractice` 项目的 `tests` 文件夹下，如图 13.9 所示。

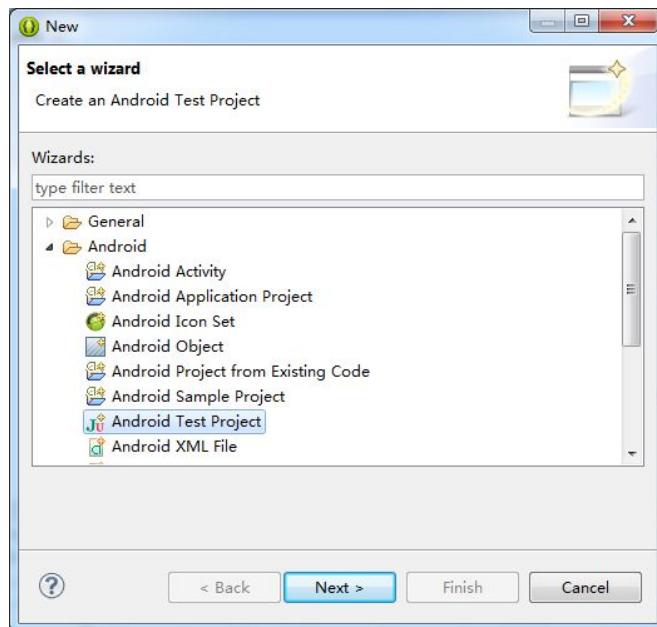


图 13.8

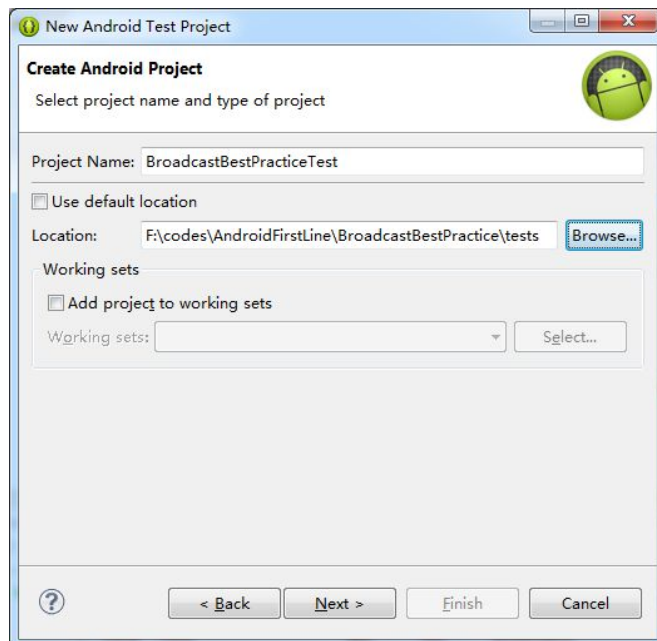


图 13.9

继续点击 Next，这时会让我们选择为哪一个项目创建测试功能，这里当然选择 BroadcastBestPractice 了，如图 13.10 所示。

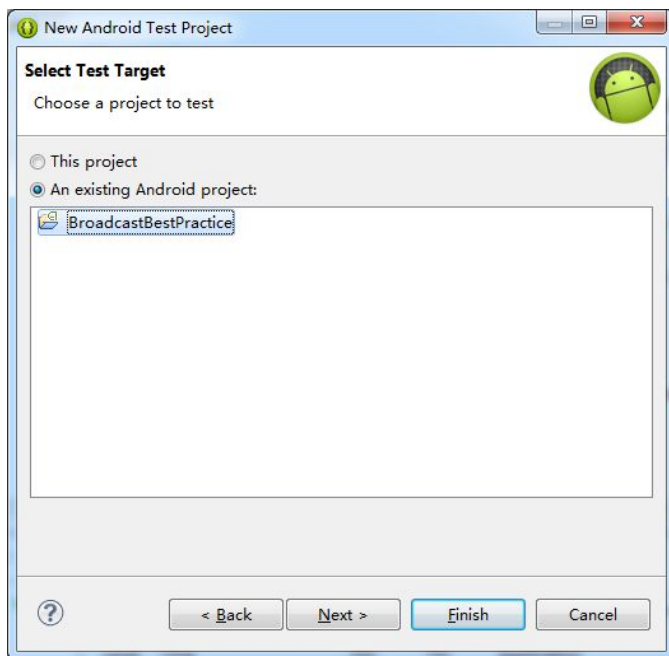


图 13.10

现在点击 Finish 就可以完成测试工程的创建了。观察测试工程中 AndroidManifest.xml 文件的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.broadcastbestpractice.test"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="14" />

    <instrumentation
        android:name="android.test.InstrumentationTestRunner"
        android:targetPackage="com.example.broadcastbestpractice" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
```



```
<uses-library android:name="android.test.runner" />
</application>
```

```
</manifest>
```

其中<instrumentation>和<uses-library>标签是自动生成的，表示这是一个测试工程，在<instrumentation>标签中还通过 android:targetPackage 属性指定了测试目标的包名。

13.5.2 进行单元测试

创建好了测试工程，下面我们来对 BroadcastBestPractice 这个项目进行单元测试。单元测试是指对软件中最小的功能模块进行测试，如果软件中的每一个单元都能通过测试，说明代码的健壮性就已经非常好了。

BroadcastBestPractice 项目中有一个 ActivityCollector 类，主要是用于对所有的 Activity 进行管理的，那么我们就来测试这个类吧。首先在 BroadcastBestPracticeTest 项目中新建一个 ActivityCollectorTest 类，并让它继承自 AndroidTestCase，然后重写 setUp() 和 tearDown() 方法，如下所示。

```
public class ActivityCollectorTest extends AndroidTestCase {

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

}
```

其中 setUp() 方法会在所有的测试用例执行之前调用，可以在这里进行一些初始化操作。tearDown() 方法会在所有的测试用例执行之后调用，可以在这里进行一些资源释放的操作。

那么该如何编写测试用例呢？其实也很简单，只需要定义一个以 test 开头的方法，测试框架就会自动调用这个方法了。然后我们在方法中可以通过断言（assert）的形式来期望一个运行结果，再和实际的运行结果进行对比，这样一条测试用例就完成了。测试用例覆盖的功能越广泛，程序出现 bug 的概率就会越小。

比如说 ActivityCollector 中的 addActivity() 方法是用于向集合里添加活动的，那么我们就可以给这个方法编写一些测试用例，代码如下所示：

```
public class ActivityCollectorTest extends AndroidTestCase {

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testAddActivity() {
        assertEquals(0, ActivityCollector.activities.size());
        LoginActivity loginActivity = new LoginActivity();
        ActivityCollector.addActivity(loginActivity);
        assertEquals(1, ActivityCollector.activities.size());
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

}
```

可以看到，这里我们添加了一个 `testAddActivity()` 方法，在这个方法的一开始就调用了 `assertEquals()` 方法来进行断言，认为目前 `ActivityCollector` 中的活动个数是 0。接下来 `new` 出了一个 `LoginActivity` 的实例，并调用 `addActivity()` 方法将这个活动添加到 `ActivityCollector` 中，然后再次调用 `assertEquals()` 方法进行断言，认为目前 `ActivityCollector` 中的活动个数是 1。

现在可以右击测试工程→Run As→Android JUnit Test 来运行这个测试用例，结果如图 13.11 所示。

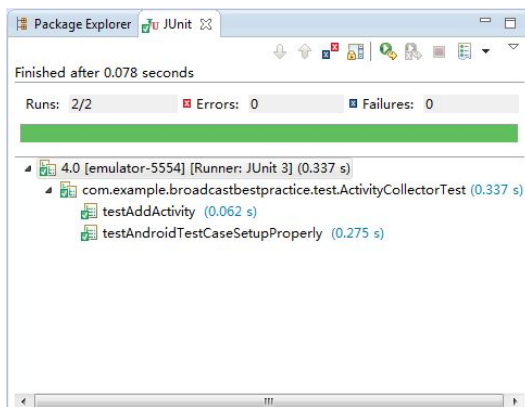


图 13.11

可以看到，我们刚刚编写的测试用例已经成功跑通了。

不过，现在这个测试用例其实只是覆盖了很多的情况而已，我们应该再编写一些特殊情况下的断言，看看程序是不是仍然能够正常工作。修改 `ActivityCollectorTest` 中的代码，如下所示。

```
public class ActivityCollectorTest extends AndroidTestCase {
    .....

    public void testAddActivity() {
        assertEquals(0, ActivityCollector.activities.size());
        LoginActivity loginActivity = new LoginActivity();
        ActivityCollector.addActivity(loginActivity);
        assertEquals(1, ActivityCollector.activities.size());
        ActivityCollector.addActivity(loginActivity);
        assertEquals(1, ActivityCollector.activities.size());
    }
    .....
}
```

可以看到，这里我们又调用了一次 `addActivity()` 方法来添加活动，并且添加的仍然还是 `LoginActivity`。连续添加两次相同活动的实例，这应该算是一种比较特殊的情况了。这时我们觉得 `ActivityCollector` 有能力去过滤掉重复的数据，因此在断言的时候认为目前 `ActivityCollector` 中的活动个数仍然是 1。重新运行一遍测试用例，结果如图 13.12 所示。

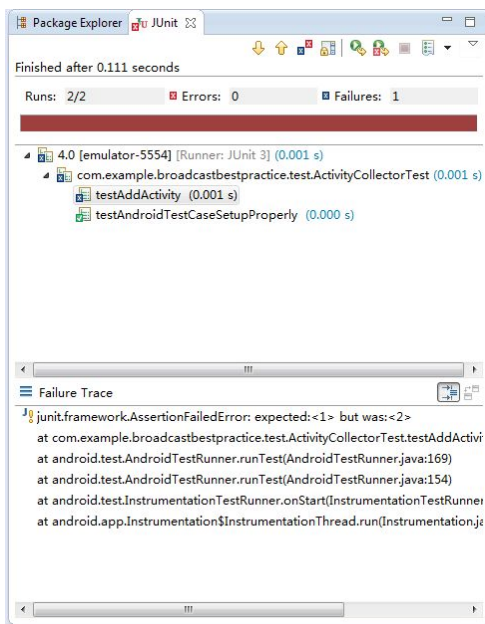


图 13.12

很遗憾，测试用例没有通过，提示我们期望结果是 1，但实际结果是 2。从这个测试用例中我们发现，`addActivity()`方法中的代码原来是不够健壮的，这个时候就应该对代码进行优化了。修改 `ActivityCollector` 中的代码，如下所示：

```
public class ActivityCollector {

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

    public static void addActivity(Activity activity) {
        if (!activities.contains(activity)) {
            activities.add(activity);
        }
    }

    .....
}
```

这里我们在 `addActivity()`方法中加入了一个 `if` 判断，只有当集合中不包含传入的 `Activity` 实例的时候才会将它添加到集合中，这样就可以解决掉活动重复的 `bug` 了。现在重新运行一遍测试用例，你就会发现测试又能成功通过了。

之后你可以不断地补充新的测试用例，让程序永远都可以跑通所有的测试用例，这样的程序才会更加健壮，出现 `bug` 的概率也会更小。

13.6 总结

整整十三章的内容你已经全部学完了！本书的所有知识点也到此结束，是不是感觉有些激动呢？下面就让我们来回顾和总结一下这么久以来学过的所有东西吧。

十三章的内容不算很多，但却已经把 `Android` 中绝大部分比较重要的知识点都覆盖到了。我们从搭建开发环境开始学起，后面逐步学习了四大组件、`UI`、碎片、数据存储、多媒体、网络、定位服务、传感器等内容，本章中又学习了如全局获取 `Context`、使用 `Intent` 传递对象、定制日志工具、调试程序、编写测试用例等高级技巧，相信你已经从一名初学者蜕变成一位 `Android` 开发好手了。

不过，虽然你已经储备了足够多的知识，并掌握了很多的最佳实践技巧，但是你还从来没有真正开发过一个完整的项目，也许在将所有学到的知识混合到一起使用的时候，你会感到有些手足无措。因此，前进的脚步仍然不能停下，下一章中我们会结合前面章节所学的内容，一起开发一个天气预报程序。锻炼的机会可千万不能错过，赶快进入到下一章吧。

经验值：+500000 升级！（由鹰升级至巨鹰） 目前经验值：864905

级别：巨鹰

赢得宝物：战胜神殿五圣。神殿五圣是神界五位最高阶法师，他们的日常工作是维系神界的大气、土壤、山川江海的生态平衡。如果有不乖的大山和河流因顽皮和淘气而导致生灵的投诉，他们会对这些犯了错的大山和河流进行教育和引导，从而让山河们意识到自己的错误，不断修行，以趋于臻美。战胜神殿五圣，其实只是接住了他们每人发出的三招五成功力。这是 Android 开发之旅中我必经的考核，我通过了考验。作为奖赏，五圣合力对我进行了加持，一瞬间，我的小鸡翅增大百倍。我猛然感到了双翅的分量，但随即也感到我的体能在迅速增强，很快抵消了双翅的重量。加持完毕，我轻松展开双翅，翼展达 6 米，雪白并发出柔和的光芒，我稍用力扇动两下，已然腾空数米。哈哈，很好用，是我身体的一部分。大恩不言谢，我拜别五圣。拍击双翅，腾空而起，一飞冲天，向着前方飞去。