

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 知识已经没有太多能够难倒你的了，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 9 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");  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
                if (connection != null) {  
                    connection.connect();  
                    byte[] buffer = new byte[1024];  
                    int len = 0; // 读取的数据长度  
                    // 从连接中读取数据  
                    while ((len = connection.getInputStream().read(buffer)) != -1) {  
                        // 将读取的数据写入到缓冲区  
                        System.out.write(buffer, 0, len);  
                    }  
                }  
            }  
        }).start();  
    }  
}
```

```
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) {
    // 回调 onFinish()方法
    listener.onFinish(response.toString());
}
} catch (Exception e) {
    if (listener != null) {
        // 回调 onError()方法
        listener.onError(e);
    }
} finally {
    if (connection != null) {
        connection.disconnect();
    }
}
}).start();
}
```

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`。

定制一个自己的 `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，而是调用一下 `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。

然后我们再回顾一下 6.5.2 小节学过的内容，当时为了让 `LitePal` 可以正常工作，要求必须在 `AndroidManifest.xml` 中配置如下内容：

```
<application
    android:name="org.litepal.LitePalApplication"
    ...>
    ...
</application>
```

其实道理也是一样的，因为经过这样的配置之后，`LitePal` 就能在内部自动获取到 `Context` 了。

不过这里你可能又会产生疑问，如果我们已经配置过了自己的 `Application` 怎么办？这样岂不是和 `LitePalApplication` 冲突了？没错，任何一个项目都只能配置一个 `Application`，

对于这种情况，LitePal 提供了很简单的解决方案，那就是在我们自己的 Application 中去调用 LitePal 的初始化方法就可以了，如下所示：

```
public class MyApplication extends Application {

    private static Context context;

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

    public static Context getContext() {
        return context;
    }

}
```

使用这种写法，就相当于我们把全局的 Context 对象通过参数传递给了 LitePal，效果和在 AndroidManifest.xml 中配置 LitePalApplication 是一模一样的。

13.2 使用 Intent 传递对象

Intent 的用法相信你已经比较熟悉了，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？难道要一行一行地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时就让日志打印出来，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 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 6 个整型常量，并且它们对应的值都是递增的。然后又定义了一个静态变量 `level`，可以将它的值指定为上面 6 个常量中的任意一个。

接下来我们提供了 `v()`、`d()`、`i()`、`w()`、`e()` 这 5 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 5 个方法来打印日志，只不过在这些自定义的方法中我们都加入了一个 `if` 判断，只有当 `level` 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` 变量的值，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，最好的解决办法就是调试了。调试允许我们逐行地执行代码，并可以实时观察内存中的数据，从而能够比较轻易地查出问题的原因。那么本节中我们就来学习一下使用 Android Studio 来调试 Android 程序的技巧。

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一下 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点击一下就可以了，如图 13.1 所示。

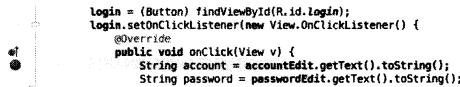


图 13.1 添加断点

如果想要取消这个断点，对着它再次点击就可以了。

添加好了断点，接下来就可以对程序进行调试了，点击 Android Studio 顶部工具栏中的 Debug 按钮（图 13.2 中最右边的按钮），就会使用调试模式来启动程序。



图 13.2 调试按钮

等到程序运行起来的时候，首先会看到一个提示框，如图 13.3 所示。

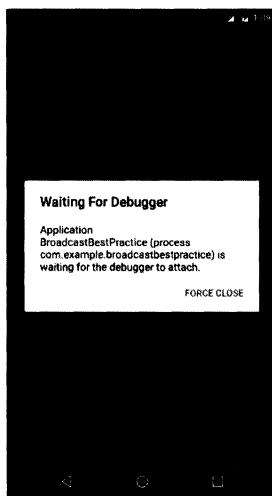


图 13.3 等待调试器提示框

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 Android Studio 就会自动打开 Debug 窗口，如图 13.4 所示。

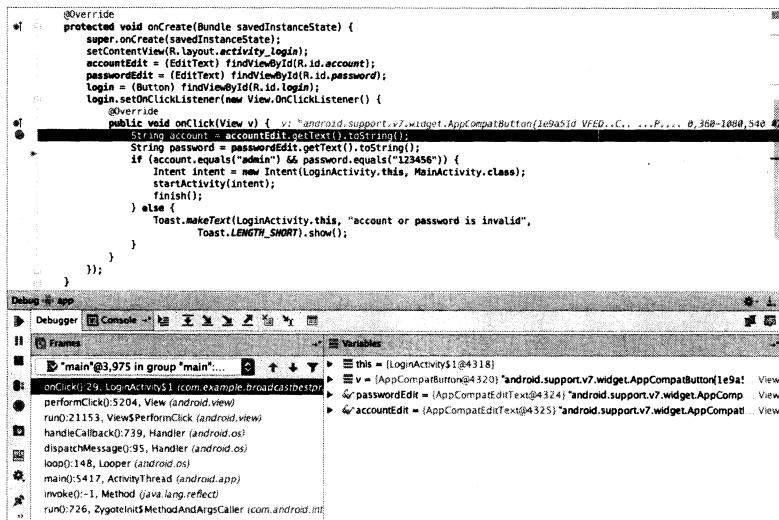


图 13.4 Debug 窗口

接下来每按一次 F8 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 13.5 所示。

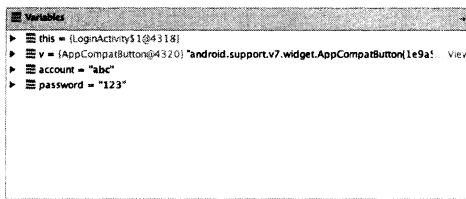


图 13.5 Variables 视图

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，调试完成之后点击 Debug 窗口中的 Stop 按钮（图 13.6 中最下边的按钮）来结束调试即可。



图 13.6 结束调试按钮

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。

这次不需要选择调试模式来启动程序了，就使用正常的方式来启动程序。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。然后点击 Android Studio 顶部工具栏的 Attach debugger to Android process 按钮（图 13.7 中最左边的按钮）。

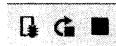


图 13.7 动态调试按钮

此时会弹出一个进程选择提示框，如图 13.8 所示。

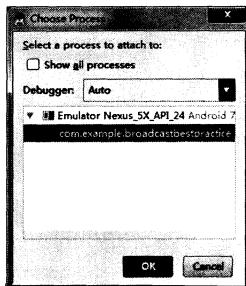


图 13.8 进程选择提示框

这里目前只列出了一个进程，也就是我们当前程序的进程。选中这个进程，然后点击 OK 按钮，就会让这个进程进入到调试模式了。

接下来在程序中点击 Login 按钮，Android Studio 同样也会自动打开 Debug 窗口，之后的流程就都是相同的了。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 创建定时任务

Android 中的定时任务一般有两种实现方式，一种是使用 Java API 里提供的 Timer 类，一种是使用 Android 的 Alarm 机制。这两种方式在多数情况下都能实现类似的效果，但 Timer 有一个明显的短板，它并不太适用于那些需要长期在后台运行的定时任务。我们都应该，为了能让电池更加耐用，每种手机都会有自己的休眠策略，Android 手机就会在长时间不操作的情况下自动让 CPU 进入到睡眠状态，这就有可能导致 Timer 中的定时任务无法正常运行。而 Alarm 则具有唤醒 CPU 的功能，它可以保证在大多数情况下需要执行定时任务的时候 CPU 都能正常工作。需要注意，这里唤醒 CPU 和唤醒屏幕完全不是一个概念，千万不要产生混淆。

13.5.1 Alarm 机制

那么首先我们来看一下 Alarm 机制的用法吧，其实并不复杂，主要就是借助了 `AlarmManager` 类来实现的。这个类和 `NotificationManager` 有点类似，都是通过调用 `Context` 的 `getSystemService()` 方法来获取实例的，只是这里需要传入的参数是 `Context.ALARM_SERVICE`。因此，获取一个 `AlarmManager` 的实例就可以写成：

```
AlarmManager manager = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
```

接下来调用 `AlarmManager` 的 `set()` 方法就可以设置一个定时任务了，比如说想要设定一个任务在 10 秒钟后执行，就可以写成：

```
long triggerAtTime = SystemClock.elapsedRealtime() + 10 * 1000;
manager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggerAtTime, pendingIntent);
```

上面的两行代码你不一定能看得明白，因为 `set()` 方法中需要传入的 3 个参数稍微有点复杂，下面我们就来仔细地分析一下。第一个参数是一个整型参数，用于指定 `AlarmManager` 的工作类型，有 4 种值可选，分别是 `ELAPSED_REALTIME`、`ELAPSED_REALTIME_WAKEUP`、`RTC` 和 `RTC_WAKEUP`。其中 `ELAPSED_REALTIME` 表示让定时任务的触发时间从系统开机开始算起，但不会唤醒 CPU。`ELAPSED_REALTIME_WAKEUP` 同样表示让定时任务的触发时间从系统开机开始算起，但会唤醒 CPU。`RTC` 表示让定时任务的触发时间从 1970 年 1 月 1 日 0 点开始算起，但不会唤醒 CPU。`RTC_WAKEUP` 同样表示让定时任务的触发时间从 1970 年 1 月 1 日 0 点开始算起，但会唤醒 CPU。使用 `SystemClock.elapsedRealtime()` 方法可以获取到系统开机至今所经历时间的毫秒数，使用 `System.currentTimeMillis()` 方法可以获取到 1970 年 1 月 1 日 0 点至今所经历时间的毫秒数。

然后看一下第二个参数，这个参数就好理解多了，就是定时任务触发的时间，以毫秒为单位。如果第一个参数使用的是 `ELAPSED_REALTIME` 或 `ELAPSED_REALTIME_WAKEUP`，则这里传入开机至今的时间再加上延迟执行的时间。如果第一个参数使用的是 `RTC` 或 `RTC_WAKEUP`，则这里传入 1970 年 1 月 1 日 0 点至今的时间再加上延迟执行的时间。

第三个参数是一个 `PendingIntent`，对于它你应该已经不会陌生了吧。这里我们一般会调用 `getService()` 方法或者 `getBroadcast()` 方法来获取一个能够执行服务或广播的 `PendingIntent`。这样当定时任务被触发的时候，服务的 `onStartCommand()` 方法或广播接收器的 `onReceive()` 方法就可以得到执行。

了解了 `set()` 方法的每个参数之后，你应该能想到，设定一个任务在 10 秒钟后执行也可以写成：

```
long triggerAtTime = System.currentTimeMillis() + 10 * 1000;
manager.set(AlarmManager.RTC_WAKEUP, triggerAtTime, pendingIntent);
```

那么，如果我们要实现一个长时间在后台定时运行的服务该怎么做呢？其实很简单，首先新建一个普通的服务，比如把它起名叫 `LongRunningService`，然后将触发定时任务的代码写到 `onStartCommand()` 方法中，如下所示：

```
public class LongRunningService extends Service {

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

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 在这里执行具体的逻辑操作
            }
        }).start();
        AlarmManager manager = (AlarmManager) getSystemService(ALARM_SERVICE);
        int anHour = 60 * 60 * 1000; // 这是一小时的毫秒数
        long triggerAtTime = SystemClock.elapsedRealtime() + anHour;
        Intent i = new Intent(this, LongRunningService.class);
        PendingIntent pi = PendingIntent.getService(this, 0, i, 0);
        manager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggerAtTime, pi);
        return super.onStartCommand(intent, flags, startId);
    }
}
```

可以看到，我们先是在 `onStartCommand()` 方法中开启了一个子线程，这样就可以在这里执行具体的逻辑操作了。之所以要在子线程里执行逻辑操作，是因为逻辑操作也是需要耗时的，如果放在主线程里执行可能会对定时任务的准确性造成轻微的影响。

创建线程之后的代码就是我们刚刚讲解的 Alarm 机制的用法了，先是获取到了 `AlarmManager` 的实例，然后定义任务的触发时间为一小时后，再使用 `PendingIntent` 指定处理定时任务的服务为 `LongRunningService`，最后调用 `set()` 方法完成设定。

这样我们就将一个长时间在后台定时运行的服务成功实现了。因为一旦启动了 `LongRunningService`，就会在 `onStartCommand()` 方法里设定一个定时任务，这样一小时后将会再次启动 `LongRunningService`，从而也就形成了一个永久的循环，保证 `LongRunningService` 的 `onStartCommand()` 方法可以每隔一小时就执行一次。

最后，只需要在你想要启动定时服务的时候调用如下代码即可：

```
Intent intent = new Intent(context, LongRunningService.class);
context.startService(intent);
```

另外需要注意的是，从 Android 4.4 系统开始，Alarm 任务的触发时间将会变得不准确，有可能会延迟一段时间后任务才能得到执行。这并不是个 bug，而是系统在耗电性方面进行的优化。系统会自动检测目前有多少 Alarm 任务存在，然后将触发时间相近的几个任务放在一起执行，这就可以大幅度地减少 CPU 被唤醒的次数，从而有效延长电池的使用时间。

当然，如果你要求 Alarm 任务的执行时间必须准确无误，Android 仍然提供了解决方案。使用 `AlarmManager` 的 `setExact()` 方法来替代 `set()` 方法，就基本上可以保证任务能够准时执行了。

13.5.2 Doze 模式

虽然 Android 的每个系统版本都在手机电量方面努力进行优化，不过一直没能解决后台服务泛滥、手机电量消耗过快的问题。于是在 Android 6.0 系统中，谷歌加入了一个全新的 Doze 模式，从而可以大幅度地延长电池的使用寿命。本小节中我们就来了解一下这个模式，并且掌握一些编程时的注意事项。

首先看一下到底什么是 Doze 模式。当用户的设备是 Android 6.0 或以上系统时，如果该设备未插接电源，处于静止状态（Android 7.0 中删除了这一条件），且屏幕关闭了一段时间之后，就会进入到 Doze 模式。在 Doze 模式下，系统会对 CPU、网络、Alarm 等活动进行限制，从而延长了电池的使用寿命。

当然，系统并不会一直处于 Doze 模式，而是会间歇性地退出 Doze 模式一小段时间，在这段时间中，应用就可以去完成它们的同步操作、Alarm 任务，等等。图 13.9 完整描述了 Doze 模式的工作过程。

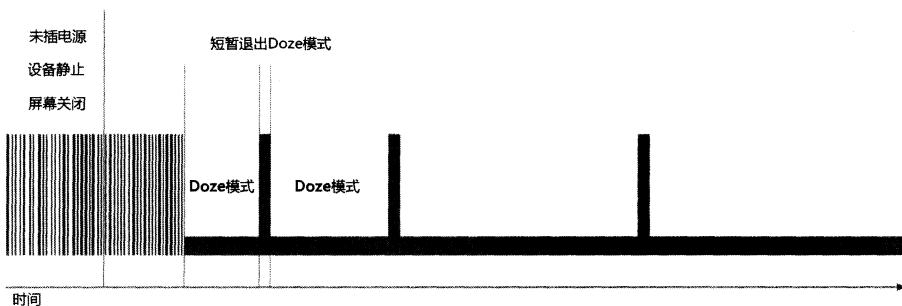


图 13.9 Doze 模式的工作过程

可以看到，随着设备进入 Doze 模式的时间越长，间歇性地退出 Doze 模式的时间间隔也会越长。因为如果设备长时间不使用的话，是没必要频繁退出 Doze 模式来执行同步等操作的，Android 在这些细节上的把控使得电池寿命进一步得到了延长。

接下来我们具体看一看在 Doze 模式下有哪些功能会受到限制吧。

- 网络访问被禁止。
- 系统忽略唤醒 CPU 或者屏幕操作。
- 系统不再执行 WIFI 扫描。
- 系统不再执行同步服务。
- Alarm 任务将会在下次退出 Doze 模式的时候执行。

注意其中的最后一条，也就是说，在 Doze 模式下，我们的 Alarm 任务将会变得不准时。当然，这在大多数情况下都是合理的，因为只有当用户长时间不使用手机的时候才会进入 Doze 模式，通常在这种情况下对 Alarm 任务的准时性要求并没有那么高。

不过，如果你真的有非常特殊的需求，要求 Alarm 任务即使在 Doze 模式下也必须正常执行，Android 还是提供了解决方案。调用 AlarmManager 的 `setAndAllowWhileIdle()` 或 `setExactAndAllowWhileIdle()` 方法就能让定时任务即使在 Doze 模式下也能正常执行了，这两个方法之间的区别和 `set()`、`setExact()` 方法之间的区别是一样的。

13.6 多窗口模式编程

由于手机屏幕大小的限制，传统情况下一个手机只能同时打开一个应用程序，无论是 Android、iOS 还是 Windows Phone 都是如此。我们也早就对此习以为常，认为这是理所当然的事情。而 Android 7.0 系统中却引入了一个非常有特色的功能——多窗口模式，它允许我们在同一个屏幕中同时打开两个应用程序。对于手机屏幕越来越大的今天，这个功能确实是越发重要了，那么本节中我们就将针对这一主题进行学习。

13.6.1 进入多窗口模式

首先你需要知道，我们不用编写任何额外的代码来让应用程序支持多窗口模式。事实上，本书中所编写的所有项目都是支持多窗口模式的。但是这并不意味着我们就不需要对多窗口模式进行学习，因为系统化地了解这些知识点才能编写出在多窗口模式下兼容性更好的程序。

那么先来看一下如何才能进入到多窗口模式。手机的导航栏你肯定是最熟悉不过了，上面一共有3个按钮，如图13.10所示。



图 13.10 手机导航栏

其中左边的 Back 按钮和中间的 Home 按钮我们都经常使用，但是右边的 Overview 按钮使用得就比较少了。这个按钮的作用是打开一个最近访问过的活动或任务的列表界面，从而能够方便地在多个应用程序之间进行切换，如图13.11所示。

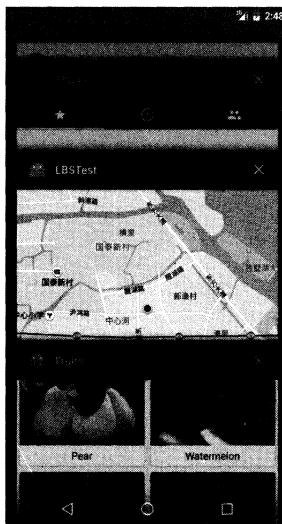


图 13.11 Overview 列表界面

我们可以通过以下两种方式进入多窗口模式。

- 在 Overview 列表界面长按任意一个活动的标题，将该活动拖动到屏幕突出显示的区域，则可以进入多窗口模式。
- 打开任意一个程序，长按 Overview 按钮，也可以进入多窗口模式。

比如说我们首先打开了 MaterialTest 程序，然后长按 Overview 按钮，效果如图 13.12 所示。



图 13.12 进入多窗口模式

可以看到，现在整个屏幕被分成了上下两个部分，MaterialTest 程序占据了上半屏，下半屏仍然还是一个 Overview 列表界面，另外 Overview 按钮的样式也有了变化。现在我们可以从 Overview 列表中选择任意一个其他程序，比如说这里点击 LBSTest，效果如图 13.13 所示。

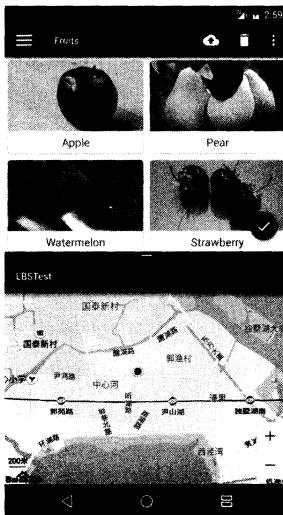


图 13.13 同时打开两个程序

我们还可以将模拟器旋转至水平方向，这样上下分屏的多窗口模式会自动切换成左右分屏的多窗口模式，如图 13.14 所示。

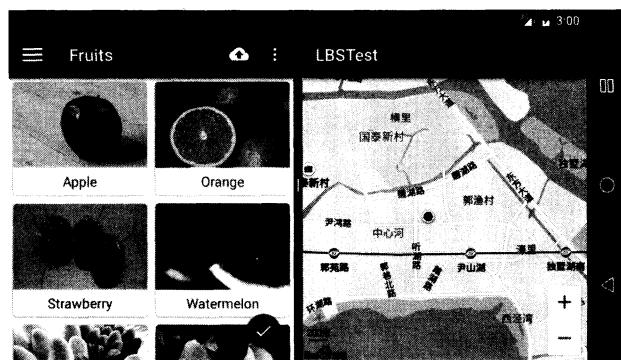


图 13.14 左右分屏的多窗口模式

多窗口模式的用法大概就是这个样子了，我们可以将任意两个应用同时打开，这样就能组合出许多更为丰富的使用场景。比如说刷微博的同时还能时刻关注 QQ 群消息，看电影的同时还能和别人一直聊着微信，等等。如果想要退出多窗口模式，只需要再次长按 Overview 按钮，或者将屏幕中央的分隔线向屏幕任意一个方向拖动到底即可。

可以看出，在多窗口模式下，整个应用的界面会缩小很多，那么编写程序时就应该多考虑使用 `match_parent` 属性、`RecyclerView`、`ListView`、`ScrollView` 等控件，来让应用的界面能够更好地适配各种不同尺寸的屏幕，尽量不要出现屏幕尺寸变化过大时界面就无法正常显示的情况。

13.6.2 多窗口模式下的生命周期

接下来我们学习一下多窗口模式下的生命周期。其实多窗口模式并不会改变活动原有的生命周期，只是会将用户最近交互过的那个活动设置为运行状态，而将多窗口模式下另外一个可见的活动设置为暂停状态。如果这时用户又去和暂停的活动进行交互，那么该活动就变成运行状态，之前处于运行状态的活动变成暂停状态。

下面我们还是通过一个例子来更加直观地理解多窗口模式下活动的生命周期。首先打开 `MaterialTest` 项目，修改 `MainActivity` 中的代码，如下所示：

```
public class MainActivity extends AppCompatActivity {

    private static final String TAG = "MaterialTest";

    ...

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

    @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");
}

...
}

```

这里我们在 Activity 的 7 个生命周期回调方法中分别打印了一句日志。

然后点击 Android Studio 导航栏上的 File→Open Recent→LBSTest，重新打开 LBSTest 项目。
修改 MainActivity 的代码，如下所示：

```

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "LBSTest";

    ...

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

```

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

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

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

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

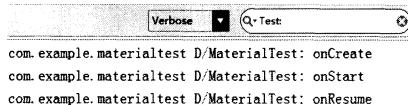
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.d(TAG, "onDestroy");
    mLocationClient.stop();
    mapView.onDestroy();
    baiduMap.setMyLocationEnabled(false);
}

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

...
}
```

这里同样也是在 Activity 的 7 个生命周期回调方法中分别打印了一句日志。注意这两处日志的 TAG 是不一样的，方便我们进行区分。

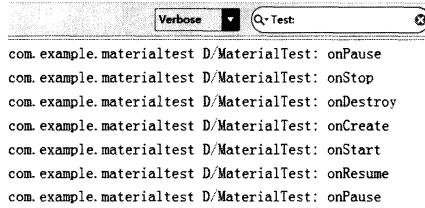
现在，先将 MaterialTest 和 LBSTest 这两个项目的最新代码都运行到模拟器上，然后启动 MaterialTest 程序。这时观察 logcat 中的打印日志（注意要将 logcat 的过滤器选择为 No Filters），如图 13.15 所示。



```
Verbose Q Test
com.example.materialtest D/MaterialTest: onCreate
com.example.materialtest D/MaterialTest: onStart
com.example.materialtest D/MaterialTest: onResume
```

图 13.15 启动 MaterialTest 时的打印日志

可以看到, `onCreate()`、`onStart()`和 `onResume()`方法会依次得到执行, 这个也是在我们意料之中的。然后长按 Overview 按钮, 进入多窗口模式, 此时的打印信息如图 13.16 所示。

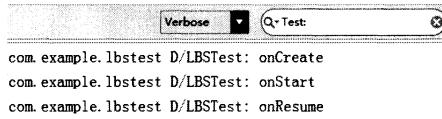


```
Verbose Q Test
com.example.materialtest D/MaterialTest: onPause
com.example.materialtest D/MaterialTest: onStop
com.example.materialtest D/MaterialTest: onDestroy
com.example.materialtest D/MaterialTest: onCreate
com.example.materialtest D/MaterialTest: onStart
com.example.materialtest D/MaterialTest: onResume
com.example.materialtest D/MaterialTest: onPause
```

图 13.16 进入多窗口模式时的打印日志

你会发现, `MaterialTest` 中的 `MainActivity` 经历了一个重新创建的过程。其实这个是正常现象, 因为进入多窗口模式后活动的大小发生了比较大的变化, 此时默认是会重新创建活动的。除此之外, 像横竖屏切换也是会重新创建活动的。进入多窗口模式后, `MaterialTest` 变成了暂停状态。

接着在 Overview 列表界面选中 `LBSTest` 程序, 打印信息如图 13.17 所示。

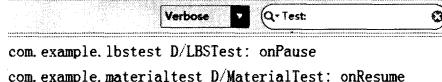


```
Verbose Q Test
com.example.lbstest D/LBSTest: onCreate
com.example.lbstest D/LBSTest: onStart
com.example.lbstest D/LBSTest: onResume
```

图 13.17 启动 LBSTest 时的打印日志

可以看到, 现在 `LBSTest` 的 `onCreate()`、`onStart()`和 `onResume()`方法依次得到了执行, 说明现在 `LBSTest` 变成了运行状态。

接下来我们可以随意操作一下 `MaterialTest` 程序, 然后观察 logcat 中的打印日志, 如图 13.18 所示。



```
Verbose Q Test
com.example.lbstest D/LBSTest: onPause
com.example.materialtest D/MaterialTest: onResume
```

图 13.18

现在 `LBSTest` 的 `onPause()`方法得到了执行, 而 `MaterialTest` 的 `onResume()`方法得到了执行, 说明 `LBSTest` 变成了暂停状态, `MaterialTest` 则变成了运行状态, 这和我们在本小节开头所分析的生命周期行为是一致的。

了解了多窗口模式下活动的生命周期规则，那么我们在编写程序的时候，就可以将一些关键性的点考虑进去了。比如说，在多窗口模式下，用户仍然可以看到处于暂停状态的应用，那么像视频播放器之类的应用在此时就应该能继续播放视频才对。因此，我们最好不要在活动的 `onPause()` 方法中去处理视频播放器的暂停逻辑，而是应该在 `onStop()` 方法中去处理，并且在 `onStart()` 方法恢复视频的播放。

另外，针对于进入多窗口模式时活动会被重新创建，如果你想改变这一默认行为，可以在 `AndroidManifest.xml` 中对活动进行如下配置：

```
<activity
    android:name=".MainActivity"
    android:label="Fruits"
    android:configChanges="orientation|keyboardHidden|screenSize|screenLayout">
    ...
</activity>
```

加入了这行配置之后，不管是进入多窗口模式，还是横竖屏切换，活动都不会被重新创建，而是会将屏幕发生变化的事件通知到 Activity 的 `onConfigurationChanged()` 方法当中。因此，如果你想在屏幕发生变化的时候进行相应的逻辑处理，那么在活动中重写 `onConfigurationChanged()` 方法即可。

13.6.3 禁用多窗口模式

多窗口模式虽然功能非常强大，但是未必就适用于所有的程序。比如说，手机游戏就非常不适合在多窗口模式下运行，很难想象我们如何一边玩着游戏，一边又操作着其他应用。因此，Android 还是给我们提供了禁用多窗口模式的选项，如果你非常不希望自己的应用能够在多窗口模式下运行，那么就可以将这个功能关闭掉。

禁用多窗口模式的方法非常简单，只需要在 `AndroidManifest.xml` 的 `<application>` 或 `<activity>` 标签中加入如下属性即可：

```
android:resizeableActivity=["true" | "false"]
```

其中，`true` 表示应用支持多窗口模式，`false` 表示应用不支持多窗口模式，如果不配置这个属性，那么默认值为 `true`。

现在我们将 `MaterialTest` 程序设置为不支持多窗口模式，如下所示：

```
<application
    ...
    android:resizeableActivity="false">
    ...
</application>
```

重新运行程序，然后长按 Overview 按钮，结果如图 13.19 所示。

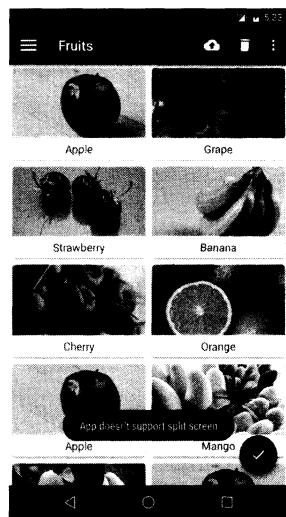


图 13.19 不支持多窗口模式时长按 Overview 按钮

可以看到，现在是无法进入到多窗口模式的，而且屏幕下方还会弹出一个 Toast 提示来告知用户，当前应用不支持多窗口模式。

虽说 `android:resizeableActivity` 这个属性的用法很简单，但是它还存在着一个问题，就是这个属性只有当项目的 `targetSdkVersion` 指定成 24 或者更高的时候才会有用，否则这个属性是无效的。那么比如说我们将项目的 `targetSdkVersion` 指定成 23，这个时候尝试进入多窗口模式，结果如图 13.20 所示。

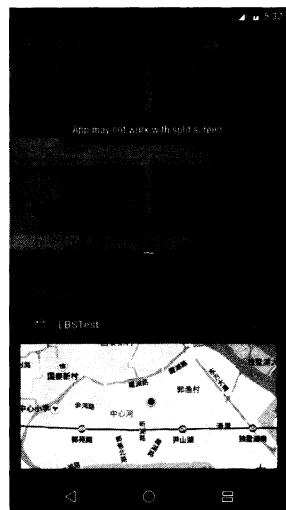


图 13.20 `targetSdkVersion` 指定成 23 时长按 Overview 按钮

可以看到，虽说界面上弹出了一个提示，告知我们此应用在多窗口模式下可能无法正常工作，但还是进入了多窗口模式。那这样我们就非常头疼了，因为有很多的老项目，它们的 targetSdkVersion 都没有指定到 24，岂不是这些老项目都无法禁用多窗口模式了？

针对这种情况，还有一种解决方案。Android 规定，如果项目指定的 targetSdkVersion 低于 24，并且活动是不允许横竖屏切换的，那么该应用也将不支持多窗口模式。

默认情况下，我们的应用都是可以随着手机的旋转自由地横竖屏切换的，如果想要让应用不允许横竖屏切换，那么就需要在 AndroidManifest.xml 的<activity>标签中加入如下配置：

```
android:screenOrientation=["portrait" | "landscape"]
```

其中，`portrait` 表示活动只支持竖屏，`landscape` 表示活动只支持横屏。当然 `android:screenOrientation` 属性中还有很多其他可选值，不过最常用的就是 `portrait` 和 `landscape` 了。

现在我们将 MaterialTest 的 MainActivity 设置为只支持竖屏，如下所示：

```
<activity
    android:name=".MainActivity"
    android:label="Fruits"
    android:screenOrientation="portrait">
    ...
</activity>
```

重新运行程序之后你会发现 MaterialTest 现在不支持横竖屏切换了，此时长按 Overview 按钮会弹出和图 13.19 中一样的提示，说明我们已经成功禁用多窗口模式了。

13.7 Lambda 表达式

Java 8 中着实引入了一些非常有特色的功能，如 Lambda 表达式、stream API、接口默认实现，等等。虽说我们本地安装的 JDK 就是 Java 8 的版本，不过本书中却一直没有使用过任何 Java 8 的新特性。这主要是因为我考虑到你对 Java 8 的新语法规则可能并不熟悉，如果直接应用到项目中的话，容易让代码难以理解，因此这里我就准备单独使用一节的篇幅来对 Java 8 的新特性进行讲解。

虽然刚才已经提到了几个 Java 8 中的新特性，不过现在能够立即应用到项目当中的也就只有 Lambda 表达式而已，因为 stream API 和接口默认实现等特性都只支持 Android 7.0 及以上的系统，我们显然不可能为了使用这些新特性而放弃兼容众多低版本的 Android 手机。而 Lambda 表达式却最低兼容到 Android 2.3 系统，基本上可以算是覆盖所有的 Android 手机了，那么本节中我们就来重点学习一下 Java 8 中的 Lambda 表达式。

Lambda 表达式本质上是一种匿名方法，它既没有方法名，也即没有访问修饰符和返回值类型，使用它来编写代码将会更加简洁，也更加易读。

如果想要在 Android 项目中使用 Lambda 表达式或者 Java 8 的其他新特性，首先我们需要在 app/build.gradle 中添加如下配置：

```

    android {
        ...
        defaultConfig {
            ...
            jackOptions.enabled = true
        }
        compileOptions {
            sourceCompatibility JavaVersion.VERSION_1_8
            targetCompatibility JavaVersion.VERSION_1_8
        }
        ...
    }
}

```

之后就可以开始使用 Lambda 表达式来编写代码了，比如说传统情况下开启一个子线程的写法如下：

```

new Thread(new Runnable() {
    @Override
    public void run() {
        // 处理具体的逻辑
    }
}).start();

```

而使用 Lambda 表达式则可以这样写：

```

new Thread(() -> {
    // 处理具体的逻辑
}).start();

```

是不是很神奇？不管是从代码行数上还是缩进结构上来看，Lambda 表达式的写法明显要更加精简。

那么为什么我们可以使用这么神奇的写法呢？这是因为 `Thread` 类的构造函数接收的参数是一个 `Runnable` 接口，并且该接口中只有一个待实现方法。我们查看一下 `Runnable` 接口的源码，如下所示：

```

public interface Runnable {

    /**
     * Starts executing the active part of the class' code. This method is
     * called when a thread is started that has been created with a class which
     * implements {@code Runnable}.
     */
    public void run();
}

```

凡是这种只有一个待实现方法的接口，都可以使用 Lambda 表达式的写法。比如说，通常创建一个类似于上述接口的匿名类实现需要这样写：

```

Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // 添加具体的实现
    }
}

```

```

    }
};
```

而有了 Lambda 表达式之后我们就可以这样写了：

```
Runnable runnable1 = () -> {
    // 添加具体的实现
};
```

了解了 Lambda 表达式的基本写法，接下来我们尝试自定义一个接口，然后再使用 Lambda 表达式的方式进行实现。

新建一个 MyListener 接口，代码如下所示：

```
public interface MyListener {
    String doSomething(String a, int b);
}
```

MyListener 接口中也只有一个待实现方法，这和 Runnable 接口的结构是基本一致的。唯一不同的是，MyListener 中的 doSomething() 方法是有参数并且有返回值的，那么我们就来看一看这种情况下该如何使用 Lambda 表达式进行实现。

其实写法也是比较相似的，使用 Lambda 表达式创建 MyListener 接口的匿名实现写法如下：

```
MyListener listener = (String a, int b) -> {
    String result = a + b;
    return result;
};
```

可以看到，doSomething() 方法的参数直接写在括号里面就可以了，而返回值则仍然像往常一样，写在具体实现的最后一行即可。

另外，Java 还可以根据上下文自动推断出 Lambda 表达式中的参数类型，因此上面的代码也可以简化成如下写法：

```
MyListener listener = (a, b) -> {
    String result = a + b;
    return result;
};
```

Java 将会自动推断出参数 a 是 String 类型，参数 b 是 int 类型，从而使得我们的代码变得更加精简了。

接下来举个具体的例子，比如说现在有一个方法是接收 MyListener 参数的，如下所示：

```
public void hello(MyListener listener) {
    String a = "Hello Lambda";
    int b = 1024;
    String result = listener.doSomething(a, b);
    Log.d("TAG", result);
}
```

我们在调用 `hello()` 这个方法的时候就可以这样写：

```
hello((a, b) -> {
    String result = a + b;
    return result;
});
```

那么 `doSomething()` 方法就会将 `a` 和 `b` 两个参数进行相加，从而最终的打印结果就会是“Hello Lambda1024”。

现在你已经将 Lambda 表达式的写法基本都掌握了，接下来我们看一看在 Android 当中有哪些常用的功能是可以使用 Lambda 表达式进行替换的。

其实只要是符合接口中只有一个待实现方法这个规则的功能，都是可以使用 Lambda 表达式来编写的。除了刚才举例说明的开启子线程之外，还有像设置点击事件之类的功能也是非常适合使用 Lambda 表达式的。

传统情况下，我们给一个按钮设置点击事件需要这样写：

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // 处理点击事件
    }
});
```

而使用 Lambda 表达式之后，就可以将代码简化成这个样子了：

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener((v) -> {
    // 处理点击事件
});
```

另外，当接口的待实现方法有且只有一个参数的时候，我们还可以进一步简化，将参数外面的括号去掉，如下所示：

```
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(v -> {
    // 处理点击事件
});
```

这样我们就将 Lambda 表达式的主要内容都掌握了。当然，有些人可能并不喜欢 Lambda 表达式这种极简主义的写法。不管你喜欢与否，Java 8 对于哪一种写法都是完全支持的，至于到底要不要使用 Lambda 表达式其实全凭个人，多一种选择总归不是一件坏事情。

13.8 总结

整整 13 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。

这 13 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、Material Design 等内容，本章中又学习了如全局获取 Context、定制日志工具、调试程序、多窗口模式编程、Lambda 表达式等高级技巧，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。

第 14 章

进入实战——开发酷欧天气

我们将要在本章中编写一个功能较为完整的天气预报程序，学习了这么久的 Android 开发，现在终于到了考核验收的时候了。那么第一步我们需要给这个软件起个好听的名字，这里就叫它酷欧天气吧，英文名就叫作 Cool Weather。确定了名字之后，下面就可以开始动手了。

14.1 功能需求及技术可行性分析

在开始编码之前，我们需要先对程序进行需求分析，想一想酷欧天气中应该具备哪些功能。将这些功能全部整理出来之后，我们才好动手去一一实现。这里我认为酷欧天气中至少应该具备以下功能：

- 可以罗列出全国所有的省、市、县；
- 可以查看全国任意城市的天气信息；
- 可以自由地切换城市，去查看其他城市的天气；
- 提供手动更新以及后台自动更新天气的功能。

虽然看上去只有 4 个主要的功能点，但如果想要全部实现这些功能却需要用到 UI、网络、数据存储、服务等技术，因此还是非常考验你的综合应用能力的。不过好在这些技术在前面的章节中我们全部都学习过了，只要你学得用心，相信完成这些功能对你来说并不难。

分析完了需求之后，接下来就要进行技术可行性分析了。首先需要考虑的一个问题就是，我们如何才能得到全国省市县的数据信息，以及如何才能获取到每个城市的天气信息。比较遗憾的是，现在网上免费的天气预报接口已经越来越少，很多之前可以使用的接口都慢慢关闭掉了，包括本书第 1 版中使用的中国天气网的接口。因此，这次我也是特意用心去找了一些更加稳定的天气预报服务，比如彩云天气以及和风天气都非常不错。这两个天气预报服务虽说都是收费的，但它们每天都提供了一定次数的免费天气预报请求。其中彩云天气的数据更加实时和专业，可以将天气预报精确到分钟级，每天提供 1000 次免费请求；和风天气的数据相对简单一些，比较适合新手学习，每天提供 3000 次免费请求。那么简单起见，这里我们就使用和风天气来作为天气预报的数据来源，每天 3000 次的免费请求对于学习而言已经是相当充足了。

解决了天气数据的问题，接下来还需要解决全国省市县数据的问题。同样，现在网上也没有一个稳定的接口可以使用，那么为了方便你的学习，我专门架设了一台服务器用于提供全国所有省市县的数据信息，从而帮你把道路都铺平了。

那么下面我们来看一下这些接口的具体用法。比如要想罗列出中国所有的省份，只需访问如下地址：

<http://guolin.tech/api/china>

服务器会返回我们一段 JSON 格式的数据，其中包含了中国所有的省份名称以及省份 id，如下所示：

```
[{"id":1,"name":"北京"}, {"id":2,"name":"上海"}, {"id":3,"name":"天津"}, {"id":4,"name":"重庆"}, {"id":5,"name":"香港"}, {"id":6,"name":"澳门"}, {"id":7,"name":"台湾"}, {"id":8,"name":"黑龙江"}, {"id":9,"name":"吉林"}, {"id":10,"name":"辽宁"}, {"id":11,"name":"内蒙古"}, {"id":12,"name":"河北"}, {"id":13,"name":"河南"}, {"id":14,"name":"山西"}, {"id":15,"name":"山东"}, {"id":16,"name":"江苏"}, {"id":17,"name":"浙江"}, {"id":18,"name":"福建"}, {"id":19,"name":"江西"}, {"id":20,"name":"安徽"}, {"id":21,"name":"湖北"}, {"id":22,"name":"湖南"}, {"id":23,"name":"广东"}, {"id":24,"name":"广西"}, {"id":25,"name":"海南"}, {"id":26,"name":"贵州"}, {"id":27,"name":"云南"}, {"id":28,"name":"四川"}, {"id":29,"name":"西藏"}, {"id":30,"name":"陕西"}, {"id":31,"name":"宁夏"}, {"id":32,"name":"甘肃"}, {"id":33,"name":"青海"}, {"id":34,"name":"新疆"}]
```

可以看到，这是一个 JSON 数组，数组中的每一个元素都代表着一个省份。其中，北京的 id 是 1，上海的 id 是 2。那么如何才能知道某个省内有哪些城市呢？其实也很简单，比如江苏的 id 是 16，访问如下地址即可：

<http://guolin.tech/api/china/16>

也就是说，只需要将省份 id 添加到 url 地址的最后面就可以了，现在服务器返回的数据如下：

```
[{"id":113,"name":"南京"}, {"id":114,"name":"无锡"}, {"id":115,"name":"镇江"}, {"id":116,"name":"苏州"}, {"id":117,"name":"南通"}, {"id":118,"name":"扬州"}, {"id":119,"name":"盐城"}, {"id":120,"name":"徐州"}, {"id":121,"name":"淮安"}, {"id":122,"name":"连云港"}, {"id":123,"name":"常州"}, {"id":124,"name":"泰州"}, {"id":125,"name":"宿迁"}]
```

这样我们就得到江苏省内所有城市的信息了，可以看到，现在返回的数据格式和刚才查看省份信息时返回的数据格式是一样的。相信此时你已经可以举一反三了，比如说苏州的 id 是 116，那么想要知道苏州市下又有哪些县和区的时候，只需访问如下地址：

<http://guolin.tech/api/china/16/116>

这次服务器返回的数据如下：

```
[{"id":937,"name":"苏州","weather_id":"CN101190401"}, {"id":938,"name":"常熟","weather_id":"CN101190402"}, {"id":939,"name":"张家港","weather_id":"CN101190403"}, {"id":940,"name":"昆山","weather_id":"CN101190404"}, {"id":941,"name":"吴中","weather_id":"CN101190405"},
```

```
{"id":942,"name":"吴江","weather_id":"CN101190407"},  
{"id":943,"name":"太仓","weather_id":"CN101190408"}]
```

通过这种方式，我们就能把全国所有的省、市、县都罗列出来了。那么解决了省市县数据的获取，我们又怎样才能查看到具体的天气信息呢？这就必须要用到每个地区对应的天气 id 了。观察上面返回的数据，你会发现每个县或区都会有一个 weather_id，拿着这个 id 再去访问和风天气的接口，就能够获取到该地区具体的天气信息了。

下面我们来看一下和风天气的接口该如何使用。首先你需要注册一个自己的账号，注册地址是 <http://guolin.tech/api/weather/register>。注册好了之后使用这个账号登录，就能看到自己的 API Key，以及每天剩余的访问次数了，如图 14.1 所示。

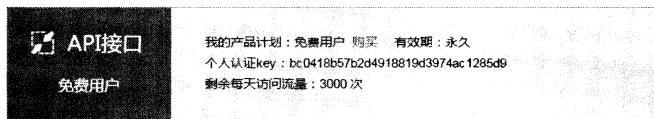


图 14.1 API Key 和每天剩余访问次数

有了 API Key，再配合刚才的 weather_id，我们就能获取到任意城市的天气信息了。比如说苏州的 weather_id 是 CN101190401，那么访问如下接口即可查看苏州的天气信息：

```
http://guolin.tech/api/weather?cityid=CN101190401&key=bc0418b57b2d4918819d3974ac1285d9
```

其中，cityid 部分填入的就是待查看城市的 weather_id，key 部分填入的就是我们申请到的 API Key。这样，服务器就会把苏州详细的天气信息以 JSON 格式返回给我们了。不过，由于返回的数据过于复杂，这里我做了一下精简处理，如下所示：

```
{
  "HeWeather": [
    {
      "status": "ok",
      "basic": {},
      "aqi": {},
      "now": {},
      "suggestion": {},
      "daily_forecast": []
    }
  ]
}
```

返回数据的格式大体上就是这个样子了，其中 status 代表请求的状态，ok 表示成功。basic 中会包含城市的一些基本信息，aqi 中会包含当前空气质量的情况，now 中会包含当前的天气信息，suggestion 中会包含一些天气相关的生活建议，daily_forecast 中会包含未来几天的天气信息。访问 <http://guolin.tech/api/weather/doc> 这个网址可以查看更加详细的文档说明。

数据都能获取到了之后，接下来就是 JSON 解析的工作了，这对于你来说应该很轻松了吧？

确定了技术完全可行之后，接下来就可以开始编码了。不过别着急，我们准备让酷欧天气成

为一个开源软件，并使用 GitHub 来进行代码托管，因此先让我们进入到本书最后一次的 Git 时间。

14.2 Git 时间——将代码托管到 GitHub 上

经过前面几章的学习，相信你已经可以非常熟练地使用 Git 了。本节依然是 Git 时间，这次我们将会把酷欧天气的代码托管到 GitHub 上面。

GitHub 是全球最大的代码托管网站，主要是借助 Git 来进行版本控制的。任何开源软件都可以免费地将代码提交到 GitHub 上，以零成本的代价进行代码托管。GitHub 的官网地址是 <https://github.com/>。官网的首页如图 14.2 所示。

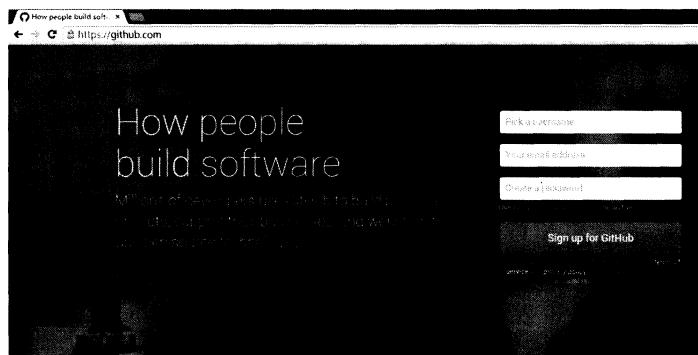


图 14.2 GitHub 首页

首先你需要有一个 GitHub 账号才能使用 GitHub 的代码托管功能，点击 Sign up for GitHub 按钮进行注册，然后填入用户名、邮箱和密码，如图 14.3 所示。

Create your personal account

Username
 ✓

This will be your username — you can enter your organization's username next.

Email Address
 ✓

You will occasionally receive account related emails. We promise not to share your email with anyone.

Password
 ✓

Use at least one lowercase letter, one numeral, and seven characters.

By clicking on "Create an account" below, you are agreeing to the Terms of Service and the Privacy Policy.

Create an account

图 14.3 注册账号

点击 Create an account 按钮来创建账户，接下来会让你选择个人计划，收费计划有创建私人版本库的权限，而我们的酷欧天气是开源软件，所以这里选择免费计划就可以了，如图 14.4 所示。

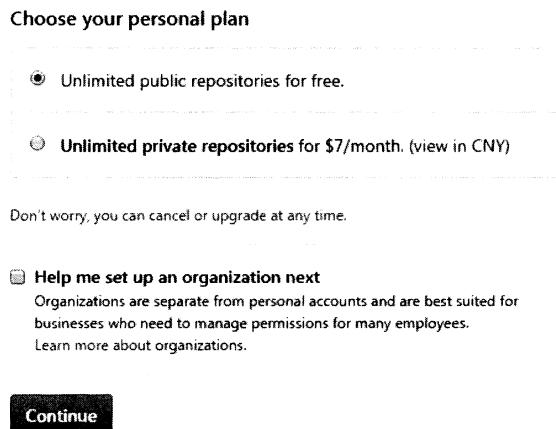


图 14.4 选择免费计划

接着点击 Continue 按钮会进入一个问卷调查界面，如图 14.5 所示。

How would you describe your level of programming experience?

Totally new to programming Somewhat experienced Very experienced

What do you plan to use GitHub for? (check all that apply)

School projects Research Design
 Development Project Management Other (please specify)

Which is closest to how you would describe yourself?

I'm a hobbyist I'm a professional I'm a student
 Other (please specify)

What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

Submit skip this step

图 14.5 问卷调查界面

如果你对这个有兴趣就填写一下，没兴趣的话直接点击最下方的 skip this step 跳过就可以了。这样我们就把账号注册好了，会自动跳转到 GitHub 的个人主页，如图 14.6 所示。

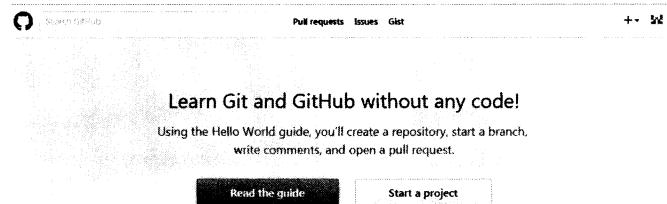


图 14.6 GitHub 个人主页

接下来就可以点击 Start a project 按钮来创建一个版本库了。由于我们是刚刚注册的账号，在创建版本库之前还需要做一下邮箱验证，验证成功之后就能开始创建了。这里将版本库命名为 coolweather，然后选择添加一个 Android 项目类型的.gitignore 文件，并使用 Apache License 2.0 来作为酷欧天气的开源协议，如图 14.7 所示。

The form fields include:

- Owner:** guolindev
- Repository name:** coolweather
- Description (optional):** (empty)
- Visibility:**
 - Public**: Anyone can see this repository. You choose who can commit.
 - Private**: You choose who can see and commit to this repository.
- Initialize this repository with a README**: checked
- Add .gitignore:** Android
- Add a license:** Apache License 2.0
- Create repository** button

图 14.7 创建版本库

接着点击 Create repository 按钮，coolweather 这个版本库就创建完成了，如图 14.8 所示。版本库主页地址是 <https://github.com/guolindev/coolweather>。

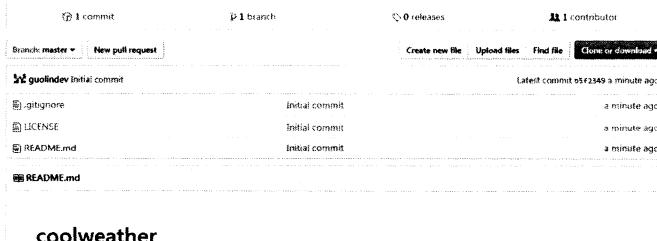


图 14.8 版本库主页

可以看到, GitHub 已经自动帮我们创建了 .gitignore、LICENSE 和 README.md 这 3 个文件, 其中编辑 README.md 文件中的内容可以修改酷欧天气版本库主页的描述。

创建好了版本库之后, 我们就需要创建酷欧天气这个项目了。在 Android Studio 中新建一个 Android 项目, 项目名叫作 CoolWeather, 包名叫作 com.coolweather.android, 如图 14.9 所示。

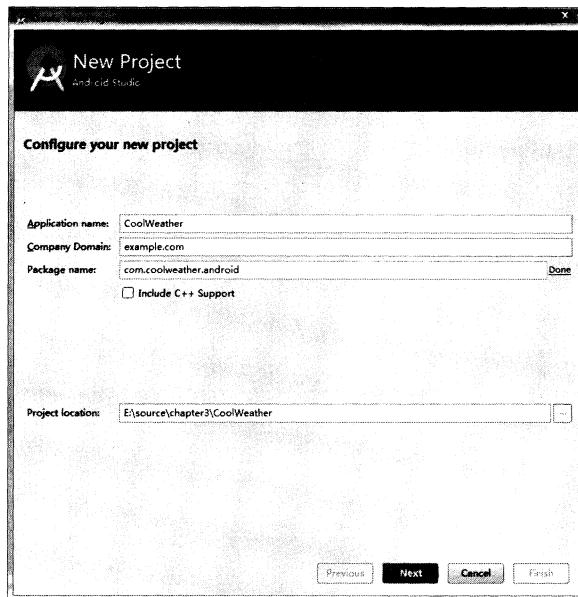


图 14.9 创建 CoolWeather 项目

之后的步骤不用多说, 一直点击 Next 就可以完成项目的创建, 所有选项都使用默认的就好。

接下来的一步非常重要, 我们需要将远程版本库克隆到本地。首先必须知道远程版本库的 Git 地址, 点击 Clone or download 按钮就能够看到了, 如图 14.10 所示。



图 14.10 查看版本库的 Git 地址

点击右边的复制按钮可以将版本库的 Git 地址复制到剪贴板, 酷欧天气版本库的 Git 地址是 <https://github.com/guolindev/coolweather.git>。

然后打开 Git Bash 并切换到 CoolWeather 的工程目录下, 如图 14.11 所示。



图 14.11 在 Git Bash 中进入 CoolWeather 工程目录

接着输入 `git clone https://github.com/guolindev/coolweather.git` 来把远程版本库克隆到本地，如图 14.12 所示。



图 14.12 将远程版本库克隆到本地

看到图中所给的文字提示就表示克隆成功了，并且 `.gitignore`、`LICENSE` 和 `README.md` 这 3 个文件也已经被复制到了本地，可以进入到 `coolweather` 目录，并使用 `ls -al` 命令查看一下，如图 14.13 所示。



图 14.13 查看克隆到本地的文件

现在我们需要将这个目录中的所有文件全部复制粘贴到上一层目录中，这样就能将整个 `CoolWeather` 工程目录添加到版本控制中去了。注意 `.git` 是一个隐藏目录，在复制的时候千万不要漏掉。另外，上一层目录中也有一个 `.gitignore` 文件，我们直接将其覆盖即可。复制完之后可以把 `coolweather` 目录删除掉，最终 `CoolWeather` 工程的目录结构如图 14.14 所示。



图 14.14 CoolWeather 工程的目录结构

接下来我们应该把 CoolWeather 项目中现有的文件提交到 GitHub 上面，这就很简单了，先将所有文件添加到版本控制中，如下所示：

```
git add .
```

然后在本地执行提交操作：

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

最后将提交的内容同步到远程版本库，也就是 GitHub 上面：

```
git push origin master
```

注意，在最后一步的时候 GitHub 要求输入用户名和密码来进行身份校验，这里输入我们注册时填入的用户名和密码就可以了，如图 14.15 所示。



图 14.15 将提交的内容同步到远程版本库

这样就已经同步完成了，现在刷新一下酷欧天气版本库的主页，你会看到刚才提交的那些文件已经存在了，如图 14.16 所示。

		Latest commit 8632138 6 minutes ago
idea	First commit.	6 minutes ago
app	First commit.	6 minutes ago
gradle/wrapper	First commit.	6 minutes ago
.gitignore	Initial commit.	an hour ago
LICENSE	Initial commit	an hour ago
README.md	Initial commit	an hour ago
build.gradle	First commit.	6 minutes ago
gradle.properties	First commit.	6 minutes ago
gradlew	First commit.	6 minutes ago
gradlew.bat	First commit.	6 minutes ago
settings.gradle	First commit.	6 minutes ago

图 14.16 在 GitHub 上查看提交的内容

14.3 创建数据库和表

从本节开始，我们就要真正地动手编码了，为了要让项目能够有更好的结构，这里需要在 com.coolweather.android 包下再新建几个包，如图 14.17 所示。

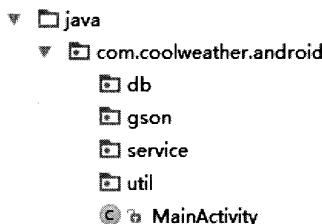


图 14.17 项目的新结构

其中 db 包用于存放数据库模型相关的代码, gson 包用于存放 JSON 模型相关的代码, service 包用于存放服务相关的代码, util 包用于存放工具相关的代码。

根据 14.1 节进行的技术可行性分析, 第一阶段我们要做的就是创建好数据库和表, 这样从服务器获取到的数据才能够存储到本地。关于数据库和表的创建方式, 我们早在第 6 章中就已经学过了。那么为了简化数据库的操作, 这里我准备使用 LitePal 来管理酷欧天气的数据库。

首先需要将项目所需的各种依赖库进行声明, 编辑 app/build.gradle 文件, 在 dependencies 闭包中添加如下内容:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:24.2.1'
    testCompile 'junit:junit:4.12'
    compile 'org.litepal.android:core:1.3.2'
    compile 'com.squareup.okhttp3:okhttp:3.4.1'
    compile 'com.google.code.gson:gson:2.7'
    compile 'com.github.bumptech.glide:glide:3.7.0'
}
```

这里声明的 4 个库我们之前都是使用过的, LitePal 用于对数据库进行操作, OkHttp 用于进行网络请求, GSON 用于解析 JSON 数据, Glide 用于加载和展示图片。酷欧天气将会对这几个库进行综合运用, 这里直接一次性将它们都添加进来。

然后我们来设计一下数据库的表结构, 表的设计当然是仁者见仁智者见智, 并不是说哪种设计就是最规范最完美的。这里我准备建立 3 张表: province、city、county, 分别用于存放省、市、县的数据信息。对应到实体类中的话, 就应该建立 Province、City、County 这 3 个类。

那么, 在 db 包下新建一个 Province 类, 代码如下所示:

```
public class Province extends DataSupport {
    private int id;
    private String provinceName;
    private int provinceCode;
    public int getId() {
```

```
        return id;
    }

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

    public String getProvinceName() {
        return provinceName;
    }

    public void setProvinceName(String provinceName) {
        this.provinceName = provinceName;
    }

    public int getProvinceCode() {
        return provinceCode;
    }

    public void setProvinceCode(int provinceCode) {
        this.provinceCode = provinceCode;
    }
}
```

其中，`id` 是每个实体类中都应该有的字段，`provinceName` 记录省的名字，`provinceCode` 记录省的代号。另外，LitePal 中的每一个实体类都是必须要继承自 `DataSupport` 类的。

接着在 `db` 包下新建一个 `City` 类，代码如下所示：

```
public class City extends DataSupport {

    private int id;

    private String cityName;

    private int cityCode;

    private int provinceId;

    public int getId() {
        return id;
    }

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

    public String getCityName() {
        return cityName;
    }

    public void setCityName(String cityName) {
        this.cityName = cityName;
    }
}
```

```
public int getCityCode() {
    return cityCode;
}

public void setCityCode(int cityCode) {
    this.cityCode = cityCode;
}

public int getProvinceId() {
    return provinceId;
}

public void setProvinceId(int provinceId) {
    this.provinceId = provinceId;
}

}
```

其中，`cityName`记录市的名字，`cityCode`记录市的代号，`provinceId`记录当前市所属省的 id 值。

然后在 `db` 包下新建一个 `County` 类，代码如下所示：

```
public class County extends DataSupport {

    private int id;

    private String countyName;

    private String weatherId;

    private int cityId;

    public int getId() {
        return id;
    }

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

    public String getCountyName() {
        return countyName;
    }

    public void setCountyName(String countyName) {
        this.countyName = countyName;
    }

    public String getWeatherId() {
        return weatherId;
    }

    public void setWeatherId(String weatherId) {
```

```

        this.weatherId = weatherId;
    }

    public int getCityId() {
        return cityId;
    }

    public void setCityId(int cityId) {
        this.cityId = cityId;
    }

}

```

其中，`countyName` 记录县的名字，`weatherId` 记录县所对应的天气 id，`cityId` 记录当前县所属市的 id 值。

可以看到，实体类的内容都非常简单，就是声明了一些需要的字段，并生成相应的 `getter` 和 `setter` 方法就可以了。

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：

```

<litepal>

    <dbname value="cool_weather" />

    <version value="1" />

    <list>
        <mapping class="com.coolweather.android.db.Province" />
        <mapping class="com.coolweather.android.db.City" />
        <mapping class="com.coolweather.android.db.County" />
    </list>

</litepal>

```

这里我们将数据库名指定成 `cool_weather`，数据库版本指定成 1，并将 `Province`、`City` 和 `County` 这 3 个实体类添加到映射列表当中。

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.coolweather.android">

    <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>
```

这样我们就将所有的配置都完成了，数据库和表会在首次执行任意数据库操作的时候自动创建。

好了，第一阶段的代码写到这里就差不多了，我们现在提交一下。首先将所有新增的文件添加到版本控制中：

```
git add .
```

接着执行提交操作：

```
git commit -m "加入创建数据库和表的各项配置。"
```

最后将提交同步到 GitHub 上面：

```
git push origin master
```

OK！第一阶段完工，下面让我们赶快进入到第二阶段的开发工作中吧。

14.4 遍历全国省市县数据

在第二阶段中，我们准备把遍历全国省市县的功能加入，这一阶段需要编写的代码量比较大，你一定要跟上脚步。

我们已经知道，全国所有省市县的数据都是从服务器端获取到的，因此这里和服务器的交互是必不可少的，所以我们可以在 util 包下先增加一个 `HttpUtil` 类，代码如下所示：

```

public class HttpUtil {

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

}
```

由于 OkHttp 的出色封装，这里和服务器进行交互的代码非常简单，仅仅 3 行就完成了。现在我们发起一条 HTTP 请求只需要调用 `sendOkHttpRequest()` 方法，传入请求地址，并注册一个回调来处理服务器响应就可以了。

另外，由于服务器返回的省市县数据都是 JSON 格式的，所以我们最好再提供一个工具类来解析和处理这种数据。在 util 包下新建一个 `Utility` 类，代码如下所示：

```

public class Utility {

    /**
     * 解析和处理服务器返回的省级数据
     */
    public static boolean handleProvinceResponse(String response) {
        if (!TextUtils.isEmpty(response)) {
            try {
                JSONArray allProvinces = new JSONArray(response);
                for (int i = 0; i < allProvinces.length(); i++) {
                    JSONObject provinceObject = allProvinces.getJSONObject(i);
                    Province province = new Province();
                    province.setProvinceName(provinceObject.getString("name"));
                    province.setProvinceCode(provinceObject.getInt("id"));
                    province.save();
                }
                return true;
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
        return false;
    }

    /**
     * 解析和处理服务器返回的市级数据
     */
    public static boolean handleCityResponse(String response, int provinceId) {
        if (!TextUtils.isEmpty(response)) {
            try {
                JSONArray allCities = new JSONArray(response);
                for (int i = 0; i < allCities.length(); i++) {
                    JSONObject cityObject = allCities.getJSONObject(i);
                    City city = new City();
                    city.setCityName(cityObject.getString("name"));
                    city.setCityCode(cityObject.getInt("id"));
                    city.setProvinceId(provinceId);
                    city.save();
                }
                return true;
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
        return false;
    }

    /**
     * 解析和处理服务器返回的县级数据
     */
    public static boolean handleCountyResponse(String response, int cityId) {
        if (!TextUtils.isEmpty(response)) {
            try {
                JSONArray allCounties = new JSONArray(response);

```

```

        for (int i = 0; i < allCounties.length(); i++) {
            JSONObject countyObject = allCounties.getJSONObject(i);
            County county = new County();
            county.setCountyName(countyObject.getString("name"));
            county.setWeatherId(countyObject.getString("weather_id"));
            county.setCityId(cityId);
            county.save();
        }
        return true;
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
return false;
}

}

```

可以看到，我们提供了 `handleProvincesResponse()`、`handleCitiesResponse()`、`handleCountiesResponse()` 这 3 个方法，分别用于解析和处理服务器返回的省级、市级和县级数据。处理的方式都是类似的，先使用 `JSONArray` 和 `JSONObject` 将数据解析出来，然后组装成实体类对象，再调用 `save()` 方法将数据存储到数据库当中。由于这里的 JSON 数据结构比较简单，我们就不使用 GSON 来进行解析了。

需要准备的工具类就这么多，现在可以开始写界面了。由于遍历全国省市县的功能我们在后面还会复用，因此就不写在活动里面了，而是写在碎片里面，这样需要复用的时候直接在布局里面引用碎片就可以了。

在 `res/layout` 目录中新建 `choose_area.xml` 布局，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:background="#fff">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary">

        <TextView
            android:id="@+id/title_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerInParent="true"
            android:textColor="#fff"
            android:textSize="20sp"/>

        <Button

```

```

        android:id="@+id/back_button"
        android:layout_width="25dp"
        android:layout_height="25dp"
        android:layout_marginLeft="10dp"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:background="@drawable/ic_back"/>
    
```

```

<ListView
    android:id="@+id/list_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>

```

```

</LinearLayout>

```

布局文件中的内容并不复杂，我们先是定义了一个头布局来作为标题栏，将布局高度设置为 actionBar 的高度，背景色设置为 colorPrimary。然后在头布局中放置了一个 TextView 用于显示标题内容，放置了一个 Button 用于执行返回操作，注意我已经提前准备好了一张 ic_back.png 图片用于作为按钮的背景图。这里之所以要自己定义标题栏，是因为碎片中最好不要直接使用 ActionBar 或 Toolbar，不然在复用的时候可能会出现一些你不想看到的效果。

接下来在头布局的下面定义了一个 ListView，省市县的数据就将显示在这里。之所以这次使用了 ListView，是因为它会自动给每个子项之间添加一条分隔线，而如果使用 RecyclerView 想实现同样的功能则会比较麻烦，这里我们总是选择最优的实现方案。

接下来也是最关键一步，我们需要编写用于遍历省市县数据的碎片了。新建 ChooseAreaFragment 继承自 Fragment，代码如下所示：

```

public class ChooseAreaFragment extends Fragment {
    public static final int LEVEL_PROVINCE = 0;
    public static final int LEVEL_CITY = 1;
    public static final int LEVEL_COUNTY = 2;
    private ProgressDialog progressDialog;
    private TextView titleText;
    private Button backButton;
    private ListView listView;
    private ArrayAdapter<String> adapter;
    private List<String> dataList = new ArrayList<>();
    /**
     * 省列表
    
```

```
/*
private List<Province> provinceList;

/**
 * 市列表
 */
private List<City> cityList;

/**
 * 县列表
 */
private List<County> countyList;

/**
 * 选中的省份
 */
private Province selectedProvince;

/**
 * 选中的城市
 */
private City selectedCity;

/**
 * 当前选中的级别
 */
private int currentLevel;

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.choose_area, container, false);
    titleText = (TextView) view.findViewById(R.id.title_text);
    backButton = (Button) view.findViewById(R.id.back_button);
    listView = (ListView) view.findViewById(R.id.list_view);
    adapter = new ArrayAdapter<>(getContext(), android.R.layout.simple_list_
        item_1, dataList);
    listView.setAdapter(adapter);
    return view;
}

@Override
public void onActivityResult(Bundle savedInstanceState) {
    super.onActivityResult(savedInstanceState);
    listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position,
            long id) {
            if (currentLevel == LEVEL_PROVINCE) {
                selectedProvince = provinceList.get(position);
                queryCities();
            } else if (currentLevel == LEVEL_CITY) {
                selectedCity = cityList.get(position);
                queryCounties();
            }
        }
    });
}
```

```

        }
    });
    backButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (currentLevel == LEVEL_COUNTY) {
                queryCities();
            } else if (currentLevel == LEVEL_CITY) {
                queryProvinces();
            }
        }
    });
    queryProvinces();
}

/**
 * 查询全国所有的省，优先从数据库查询，如果没有查询到再去服务器上查询
 */
private void queryProvinces() {
    titleText.setText("中国");
    backButton.setVisibility(View.GONE);
    provinceList = DataSupport.findAll(Province.class);
    if (provinceList.size() > 0) {
        dataList.clear();
        for (Province province : provinceList) {
            dataList.add(province.getProvinceName());
        }
        adapter.notifyDataSetChanged();
        listView.setSelection(0);
        currentLevel = LEVEL_PROVINCE;
    } else {
        String address = "http://guolin.tech/api/china";
        queryFromServer(address, "province");
    }
}

/**
 * 查询选中省内所有的市，优先从数据库查询，如果没有查询到再去服务器上查询
 */
private void queryCities() {
    titleText.setText(selectedProvince.getProvinceName());
    backButton.setVisibility(View.VISIBLE);
    cityList = DataSupport.where("provinceid = ?", String.valueOf(selected
        Province.getId())).find(City.class);
    if (cityList.size() > 0) {
        dataList.clear();
        for (City city : cityList) {
            dataList.add(city.getCityName());
        }
        adapter.notifyDataSetChanged();
        listView.setSelection(0);
        currentLevel = LEVEL_CITY;
    } else {
        int provinceCode = selectedProvince.getProvinceCode();
    }
}

```

```
        String address = "http://guolin.tech/api/china/" + provinceCode;
        queryFromServer(address, "city");
    }

    /**
     * 查询选中市内所有的县，优先从数据库查询，如果没有查询到再去服务器上查询
     */
    private void queryCounties() {
        titleText.setText(selectedCity.getCityName());
        backButton.setVisibility(View.VISIBLE);
        countyList = DataSupport.where("cityid = ?", String.valueOf(selectedCity.getId())).find(County.class);
        if (countyList.size() > 0) {
            dataList.clear();
            for (County county : countyList) {
                dataList.add(county.getCountyName());
            }
            adapter.notifyDataSetChanged();
            listView.setSelection(0);
            currentLevel = LEVEL_COUNTY;
        } else {
            int provinceCode = selectedProvince.getProvinceCode();
            int cityCode = selectedCity.getCityCode();
            String address = "http://guolin.tech/api/china/" + provinceCode + "/" +
                cityCode;
            queryFromServer(address, "county");
        }
    }

    /**
     * 根据传入的地址和类型从服务器上查询省市县数据
     */
    private void queryFromServer(String address, final String type) {
        showProgressDialog();
        HttpUtil.sendOkHttpRequest(address, new Callback() {
            @Override
            public void onResponse(Call call, Response response) throws IOException {
                String responseText = response.body().string();
                boolean result = false;
                if ("province".equals(type)) {
                    result = Utility.handleProvinceResponse(responseText);
                } else if ("city".equals(type)) {
                    result = Utility.handleCityResponse(responseText,
                        selectedProvince.getId());
                } else if ("county".equals(type)) {
                    result = Utility.handleCountyResponse(responseText,
                        selectedCity.getId());
                }
                if (result) {
                    getActivity().runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            closeProgressDialog();
                            if ("province".equals(type)) {
                                // 处理省
                            } else if ("city".equals(type)) {
                                // 处理市
                            } else if ("county".equals(type)) {
                                // 处理县
                            }
                        }
                    });
                }
            }
        });
    }
}
```

```
        queryProvinces();
    } else if ("city".equals(type)) {
        queryCities();
    } else if ("county".equals(type)) {
        queryCounties();
    }
}
});
}
}

@Override
public void onFailure(Call call, IOException e) {
    // 通过 runOnUiThread()方法回到主线程处理逻辑
    getActivity().runOnUiThread(new Runnable() {
        @Override
        public void run() {
            closeProgressDialog();
            Toast.makeText(getContext(), "加载失败", Toast.LENGTH_SHORT).show();
        }
    });
}
});

}

/**
 * 显示进度对话框
 */
private void showProgressDialog() {
    if (progressDialog == null) {
        progressDialog = new ProgressDialog(getActivity());
        progressDialog.setMessage("正在加载...");
        progressDialog.setCancelable(false);
    }
    progressDialog.show();
}

/**
 * 关闭进度对话框
 */
private void closeProgressDialog() {
    if (progressDialog != null) {
        progressDialog.dismiss();
    }
}
```

这个类里的代码虽然非常多，可是逻辑却不复杂，我们来慢慢理一下。在 `onCreateView()` 方法中先是获取到了一些控件的实例，然后去初始化了 `ArrayAdapter`，并将它设置为 `ListView` 的适配器。接着在 `onActivityCreated()` 方法中给 `ListView` 和 `Button` 设置了点击事件，到这里我们的初始化工作就算是完成了。

在 `onActivityCreated()` 方法的最后，调用了 `queryProvinces()` 方法，也就是从这里开始加载省级数据的。`queryProvinces()` 方法中首先会将头布局的标题设置成中国，将返回按钮隐藏起来，因为省级列表已经不能再返回了。然后调用 LitePal 的查询接口来从数据库中读取省级数据，如果读取到了就直接将数据显示到界面上，如果没有读取到就按照 14.1 节讲述的接口组装出一个请求地址，然后调用 `queryFromServer()` 方法来从服务器上查询数据。

`queryFromServer()` 方法中会调用 `HttpUtil` 的 `sendOkHttpRequest()` 方法来向服务器发送请求，响应的数据会回调到 `onResponse()` 方法中，然后我们在这里去调用 `Utility` 的 `handleProvincesResponse()` 方法来解析和处理服务器返回的数据，并存储到数据库中。接下来的一步很关键，在解析和处理完数据之后，我们再次调用了 `queryProvinces()` 方法来重新加载省级数据，由于 `queryProvinces()` 方法牵扯到了 UI 操作，因此必须要在主线程中调用，这里借助了 `runOnUiThread()` 方法来实现从子线程切换到主线程。现在数据库中已经存在了数据，因此调用 `queryProvinces()` 就会直接将数据显示到界面上了。

当你点击了某个省的时候会进入到 `ListView` 的 `onItemClick()` 方法中，这个时候会根据当前的级别来判断是去调用 `queryCities()` 方法还是 `queryCounties()` 方法，`queryCities()` 方法是去查询市级数据，而 `queryCounties()` 方法是去查询县级数据，这两个方法内部的流程和 `queryProvinces()` 方法基本相同，这里就不重复讲解了。

另外还有一点需要注意，在返回按钮的点击事件里，会对当前 `ListView` 的列表级别进行判断。如果当前是县级列表，那么就返回到市级列表，如果当前是市级列表，那么就返回到省级表列表。当返回到省级列表时，返回按钮会自动隐藏，从而也就不再需要再做进一步的处理了。

这样我们就把遍历全国省市县的功能完成了，可是碎片是不能直接显示在界面上的，因此我们还需要把它添加到活动里才行。修改 `activity_main.xml` 中的代码，如下所示：

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

    <fragment
        android:id="@+id/choose_area_fragment"
        android:name="com.coolweather.android.ChooseAreaFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</FrameLayout>
```

布局文件很简单，只是定义了一个 `FrameLayout`，然后将 `ChooseAreaFragment` 添加进来，并让它充满整个布局。

另外，我们刚才在碎片的布局里面已经自定义了一个标题栏，因此就不再需要原生的 `ActionBar` 了，修改 `res/values/styles.xml` 中的代码，如下所示：

```

<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        ...
    </style>

</resources>

```

现在第二阶段的开发工作也完成得差不多了，我们可以运行一下来看看效果。不过在运行之前还有一件事没有做，那就是声明程序所需要的权限。修改 AndroidManifest.xml 中的代码，如下所示：

```

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

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

    ...

```

</manifest>

由于我们是通过网络接口来获取全国省市县数据的，因此必须要添加访问网络的权限才行。

现在可以运行一下程序了，结果如图 14.18 所示。

可以看到，全国所有省级数据都显示出来了。我们还可以继续查看市级数据，比如点击江苏省，结果如图 14.19 所示。

这个时候标题栏上会出现一个返回按钮，用于返回上一级列表。

然后再点击苏州市查看县级数据，结果如图 14.20 所示。

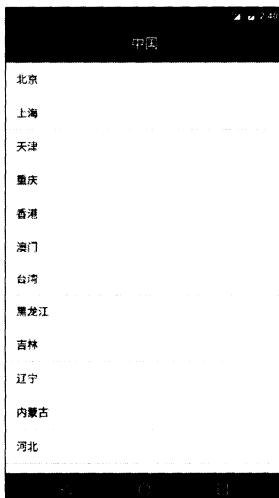


图 14.18 显示省级数据

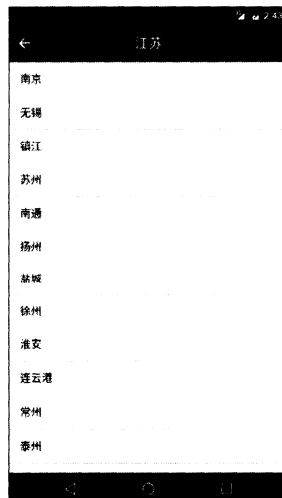


图 14.19 显示市级数据

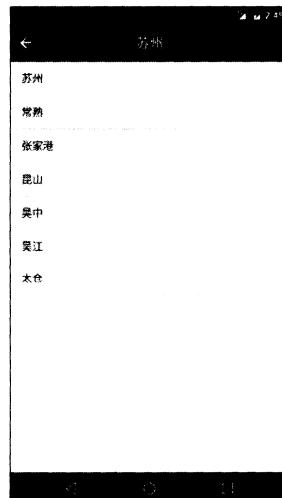


图 14.20 显示县级数据

好了，这样第二阶段的开发工作也都完成了，我们仍然要把代码提交一下。

```
git add .
git commit -m "完成遍历省市县三级列表的功能。"
git push origin master
```

到目前为止进度算是相当不错啊，那么我们就趁热打铁，来进行第三阶段的开发工作。

14.5 显示天气信息

在第三阶段中，我们就要开始去查询天气，并且把天气信息显示出来了。由于和风天气返回的 JSON 数据结构非常复杂，如果还使用 `JSONObject` 来解析就会很麻烦，这里我们就准备借助 `JSON` 来对天气信息进行解析了。

14.5.1 定义 `JSON` 实体类

`JSON` 的用法很简单，解析数据只需要一行代码就能完成了，但前提是要先将数据对应的实体类创建好。由于和风天气返回的数据内容非常多，这里我们不可能将所有的内容都利用起来，因此我筛选了一些比较重要的数据来进行解析。

首先我们回顾一下返回数据的大致格式：

```
{
    "HeWeather": [
        {
            "status": "ok",
            "basic": {},
            "aqi": {},
            "now": {},
            "suggestion": {},
            "daily_forecast": []
        }
    ]
}
```

其中，`basic`、`aqi`、`now`、`suggestion` 和 `daily_forecast` 的内部又都会有具体的内容，那么我们就可以将这 5 个部分定义成 5 个实体类。

下面开始来一个个看，`basic` 中具体内容如下所示：

```
"basic": {
    "city": "苏州",
    "id": "CN101190401",
    "update": {
        "loc": "2016-08-08 21:58"
    }
}
```

其中，`city` 表示城市名，`id` 表示城市对应的天气 `id`，`update` 中的 `loc` 表示天气的更新时

间。我们按照此结构就可以在 gson 包下建立一个 `Basic` 类，代码如下所示：

```
public class Basic {  
  
    @SerializedName("city")  
    public String cityName;  
  
    @SerializedName("id")  
    public String weatherId;  
  
    public Update update;  
  
    public class Update {  
  
        @SerializedName("loc")  
        public String updateTime;  
  
    }  
}
```

由于 JSON 中的一些字段可能不太适合直接作为 Java 字段来命名，因此这里使用了 `@SerializedName` 注解的方式来让 JSON 字段和 Java 字段之间建立映射关系。

这样我们就将 `Basic` 类定义好了，还是挺容易理解的吧？其余的几个实体类也是类似的，我们使用同样的方式来定义就可以了。比如 api 中的具体内容如下所示：

```
"aqi":{  
    "city":{  
        "aqi":"44",  
        "pm25":"13"  
    }  
}
```

那么，在 gson 包下新建一个 `AQI` 类，代码如下所示：

```
public class AQI {  
  
    public AQICity city;  
  
    public class AQICity {  
  
        public String aqi;  
        public String pm25;  
  
    }  
}
```

`now` 中的具体内容如下所示：

```
"now":{
```

```

    "tmp":"29",
    "cond":{
        "txt":"阵雨"
    }
}
}

```

那么，在 gson 包下新建一个 Now 类，代码如下所示：

```

public class Now {

    @SerializedName("tmp")
    public String temperature;

    @SerializedName("cond")
    public More more;

    public class More {

        @SerializedName("txt")
        public String info;

    }
}

```

suggestion 中的具体内容如下所示：

```

"suggestion":{

    "comf":{

        "txt":"白天天气较热，虽然有雨，但仍然无法削弱较高气温给人们带来的暑意，  

        这种天气会让您感到不很舒适。"
    },
    "cw":{

        "txt":"不宜洗车，未来 24 小时内有雨，如果在此期间洗车，雨水和路上的泥水  

        可能会再次弄脏您的爱车。"
    },
    "sport":{

        "txt":"有降水，且风力较强，推荐您在室内进行低强度运动；若坚持户外运动，  

        请选择避雨防风的地点。"
    }
}

```

那么，在 gson 包下新建一个 Suggestion 类，代码如下所示：

```

public class Suggestion {

    @SerializedName("comf")
    public Comfort comfort;

    @SerializedName("cw")
    public CarWash carWash;

    public Sport sport;
}

```

```

public class Comfort {
    @SerializedName("txt")
    public String info;
}

public class CarWash {
    @SerializedName("txt")
    public String info;
}

public class Sport {
    @SerializedName("txt")
    public String info;
}

```

到目前为止都还比较简单，不过接下来的一项数据就有点特殊了，`daily_forecast` 中的具体内容如下所示：

```

"daily_forecast": [
    {
        "date": "2016-08-08",
        "cond": {
            "txt_d": "阵雨"
        },
        "tmp": {
            "max": "34",
            "min": "27"
        }
    },
    {
        "date": "2016-08-09",
        "cond": {
            "txt_d": "多云"
        },
        "tmp": {
            "max": "35",
            "min": "29"
        }
    },
    ...
]

```

可以看到，`daily_forecast` 中包含的是一个数组，数组中的每一项都代表着未来一天的天气信息。针对于这种情况，我们只需要定义出单日天气的实体类就可以了，然后在声明实体类引用的时候使用集合类型来进行声明。

那么在 gson 包下新建一个 Forecast 类，代码如下所示：

```
public class Forecast {  
    public String date;  
  
    @SerializedName("tmp")  
    public Temperature temperature;  
  
    @SerializedName("cond")  
    public More more;  
  
    public class Temperature {  
  
        public String max;  
  
        public String min;  
    }  
  
    public class More {  
  
        @SerializedName("txt_d")  
        public String info;  
    }  
}
```

这样我们就把 basic、aqi、now、suggestion 和 daily_forecast 对应的实体类全部都创建好了，接下来还需要再创建一个总的实例类来引用刚刚创建的各个实体类。在 gson 包下新建一个 Weather 类，代码如下所示：

```
public class Weather {  
    public String status;  
  
    public Basic basic;  
  
    public AQI aqi;  
  
    public Now now;  
  
    public Suggestion suggestion;  
  
    @SerializedName("daily_forecast")  
    public List<Forecast> forecastList;  
}
```

在 Weather 类中，我们对 Basic、AQI、Now、Suggestion 和 Forecast 类进行了引用。其中，由于 daily_forecast 中包含的是一个数组，因此这里使用了 List 集合来引用 Forecast 类。

另外，返回的天气数据中还会包含一项 `status` 数据，成功返回 `ok`，失败则会返回具体的原因，那么这里也需要添加一个对应的 `status` 字段。

现在所有的 JSON 实体类都定义好了，接下来我们开始编写天气界面。

14.5.2 编写天气界面

首先创建一个用于显示天气信息的活动。右击 `com.coolweather.android` 包 → New → Activity → Empty Activity，创建一个 `WeatherActivity`，并将布局名指定成 `activity_weather.xml`。

由于所有的天气信息都将在同一个界面上显示，因此 `activity_weather.xml` 会是一个很长的布局文件。那么为了让里面的代码不至于混乱不堪，这里我准备使用 3.4.1 小节学过的引入布局技术，即将界面的不同部分写在不同的布局文件里面，再通过引入布局的方式集成到 `activity_weather.xml` 中，这样整个布局文件就会显得非常工整。

右击 `res/layout` → New → Layout resource file，新建一个 `title.xml` 作为头布局，代码如下所示：

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize">

    <TextView
        android:id="@+id/title_city"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:textColor="#fff"
        android:textSize="20sp" />

    <TextView
        android:id="@+id/title_update_time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dp"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:textColor="#fff"
        android:textSize="16sp" />

</RelativeLayout>
```

这段代码还是比较简单的，头布局中放置了两个 `TextView`，一个居中显示城市名，一个居右显示更新时间。

然后新建一个 `now.xml` 作为当前天气信息的布局，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="wrap_content"
    android:layout_margin="15dp">>

    <TextView
        android:id="@+id/degree_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:textColor="#fff"
        android:textSize="60sp" />

    <TextView
        android:id="@+id/weather_info_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="end"
        android:textColor="#fff"
        android:textSize="20sp" />

</LinearLayout>
```

当前天气信息的布局中也是放置了两个 TextView，一个用于显示当前气温，一个用于显示天气概况。

然后新建 forecast.xml 作为未来几天天气信息的布局，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="wrap_content"
    android:layout_margin="15dp"
    android:background="#8000">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dp"
        android:layout_marginTop="15dp"
        android:text="预报"
        android:textColor="#fff"
        android:textSize="20sp"/>

    <LinearLayout
        android:id="@+id/forecast_layout"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
    </LinearLayout>

</LinearLayout>
```

这里最外层使用 LinearLayout 定义了一个半透明的背景，然后使用 TextView 定义了一个标

题，接着又使用一个 `LinearLayout` 定义了一个用于显示未来几天天气信息的布局。不过这个布局中并没有放入任何内容，因为这是要根据服务器返回的数据在代码中动态添加的。

为此，我们还需要再定义一个未来天气信息的子项布局，创建 `forecast_item.xml` 文件，代码如下所示：

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="15dp">

    <TextView
        android:id="@+id/date_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_weight="2"
        android:textColor="#fff"/>

    <TextView
        android:id="@+id/info_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_weight="1"
        android:gravity="center"
        android:textColor="#fff"/>

    <TextView
        android:id="@+id/max_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="1"
        android:gravity="right"
        android:textColor="#fff"/>

    <TextView
        android:id="@+id/min_text"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_weight="1"
        android:gravity="right"
        android:textColor="#fff"/>

</LinearLayout>
```

子项布局中放置了 4 个 `TextView`，一个用于显示天气预报日期，一个用于显示天气概况，另外两个分别用于显示当天的最高温度和最低温度。

然后新建 `aqi.xml` 作为空气质量信息的布局，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="wrap_content"
    android:layout_margin="15dp"
    android:background="#8000">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dp"
        android:layout_marginTop="15dp"
        android:text="空气质量"
        android:textColor="#fff"
        android:textSize="20sp"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="15dp">

        <RelativeLayout
            android:layout_width="0dp"
            android:layout_height="match_parent"
            android:layout_weight="1">

            <LinearLayout
                android:orientation="vertical"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_centerInParent="true">

                <TextView
                    android:id="@+id/aqi_text"
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_gravity="center"
                    android:textColor="#fff"
                    android:textSize="40sp"
                    />

                <TextView
                    android:layout_width="wrap_content"
                    android:layout_height="wrap_content"
                    android:layout_gravity="center"
                    android:text="AQI 指数"
                    android:textColor="#fff"/>

            </LinearLayout>
        </RelativeLayout>
    <RelativeLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
```

```

        android:layout_weight="1"

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true">

        <TextView
            android:id="@+id/pm25_text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:textColor="#fff"
            android:textSize="40sp"
        />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:text="PM2.5 指数"
            android:textColor="#fff"
        />

    </LinearLayout>
</RelativeLayout>
</LinearLayout>
</LinearLayout>

```

这个布局中的代码虽然看上去有点长，但是并不复杂。首先前面都是一样的，使用 `LinearLayout` 定义了一个半透明的背景，然后使用 `TextView` 定义了一个标题。接下来，这里使用 `LinearLayout` 和 `RelativeLayout` 嵌套的方式实现了一个左右平分并且居中对齐的布局，分别用于显示 AQI 指数和 PM 2.5 指数。相信你只要仔细看一看，这个布局还是很好理解的。

然后新建 `suggestion.xml` 作为生活建议信息的布局，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="wrap_content"
    android:layout_margin="15dp"
    android:background="#800000">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="15dp"
        android:layout_marginTop="15dp"
        android:text="生活建议"
    />

```

```
    android:textColor="#fff"  
    android:textSize="20sp"/>  
  
<TextView  
    android:id="@+id/comfort_text"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_margin="15dp"  
    android:textColor="#fff" />  
  
<TextView  
    android:id="@+id/car_wash_text"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_margin="15dp"  
    android:textColor="#fff" />  
  
<TextView  
    android:id="@+id/sport_text"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_margin="15dp"  
    android:textColor="#fff" />  
  
</LinearLayout>
```

这里同样也是先定义了一个半透明的背景和一个标题，然后下面使用了 3 个 TextView 分别用于显示舒适度、洗车指数和运动建议的相关数据。

这样我们就把天气界面上每个部分的布局文件都编写好了，接下来的工作就是将它们引入到 activity_weather.xml 当中，如下所示：

```
<FrameLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:background="@color/colorPrimary">  
  
<ScrollView  
    android:id="@+id/weather_layout"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:scrollbars="none"  
    android:overScrollMode="never">  
  
<LinearLayout  
    android:orientation="vertical"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content">  
  
    <include layout="@layout/title" />  
    <include layout="@layout/now" />
```

```

<include layout="@layout/forecast" />
<include layout="@layout/aqi" />
<include layout="@layout/suggestion" />
</LinearLayout>
</ScrollView>
</FrameLayout>

```

可以看到，首先最外层布局使用了一个 `FrameLayout`，并将它的背景色设置成 `colorPrimary`。然后在 `FrameLayout` 中嵌套了一个 `ScrollView`，这是因为天气界面中的内容比较多，使用 `ScrollView` 可以允许我们通过滚动的方式查看屏幕以外的内容。

由于 `ScrollView` 的内部只允许存在一个直接子布局，因此这里又嵌套了一个垂直方向的 `LinearLayout`，然后在 `LinearLayout` 中将刚才定义的所有布局逐个引入。

这样我们就将天气界面编写完成了，接下来开始编写业务逻辑，将天气显示到界面上。

14.5.3 将天气显示到界面上

首先需要在 `Utility` 类中添加一个用于解析天气 JSON 数据的方法，如下所示：

```

public class Utility {
    ...
    /**
     * 将返回的 JSON 数据解析成 Weather 实体类
     */
    public static Weather handleWeatherResponse(String response) {
        try {
            JSONObject jsonObject = new JSONObject(response);
            JSONArray jsonArray = jsonObject.getJSONArray("HeWeather");
            String weatherContent = jsonArray.getJSONObject(0).toString();
            return new Gson().fromJson(weatherContent, Weather.class);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}

```

可以看到，`handleWeatherResponse()`方法中先是通过 `JSONObject` 和 `JSONArray` 将天气数据中的主体内容解析出来，即如下内容：

```
{
    "status": "ok",
    "basic": {},
    "aqi": {}
}
```

```
    "now": {},
    "suggestion": {},
    "daily_forecast": []
}
```

然后由于我们之前已经按照上面的数据格式定义过相应的 JSON 实体类，因此只需要通过调用 `fromJson()` 方法就能直接将 JSON 数据转换成 `Weather` 对象了。

接下来的工作是我们如何在活动中去请求天气数据，以及将数据展示到界面上。修改 `WeatherActivity` 中的代码，如下所示：

```
public class WeatherActivity extends AppCompatActivity {

    private ScrollView weatherLayout;
    private TextView titleCity;
    private TextView titleUpdateTime;
    private TextView degreeText;
    private TextView weatherInfoText;
    private LinearLayout forecastLayout;
    private TextView aqiText;
    private TextView pm25Text;
    private TextView comfortText;
    private TextView carWashText;
    private TextView sportText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_weather);
        // 初始化各控件
        weatherLayout = (ScrollView) findViewById(R.id.weather_layout);
        titleCity = (TextView) findViewById(R.id.title_city);
        titleUpdateTime = (TextView) findViewById(R.id.title_update_time);
        degreeText = (TextView) findViewById(R.id.degree_text);
        weatherInfoText = (TextView) findViewById(R.id.weather_info_text);
        forecastLayout = (LinearLayout) findViewById(R.id.forecast_layout);
        aqiText = (TextView) findViewById(R.id.aqi_text);
        pm25Text = (TextView) findViewById(R.id.pm25_text);
        comfortText = (TextView) findViewById(R.id.comfort_text);
        carWashText = (TextView) findViewById(R.id.car_wash_text);
        sportText = (TextView) findViewById(R.id.sport_text);
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences
            (this);
        String weatherString = prefs.getString("weather", null);
        if (weatherString != null) {
```

```

        // 有缓存时直接解析天气数据
        Weather weather = Utility.handleWeatherResponse(weatherString);
        showWeatherInfo(weather);
    } else {
        // 无缓存时去服务器查询天气
        String weatherId = getIntent().getStringExtra("weather_id");
        weatherLayout.setVisibility(View.INVISIBLE);
        requestWeather(weatherId);
    }
}

/**
 * 根据天气 id 请求城市天气信息
 */
public void requestWeather(final String weatherId) {

    String weatherUrl = "http://guolin.tech/api/weather?cityid=" +
        weatherId + "&key=bc0418b57b2d4918819d3974ac1285d9";
    HttpUtil.sendOkHttpRequest(weatherUrl, new Callback() {
        @Override
        public void onResponse(Call call, Response response) throws IOException {
            final String responseText = response.body().string();
            final Weather weather = Utility.handleWeatherResponse(responseText);
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    if (weather != null && "ok".equals(weather.status)) {
                        SharedPreferences.Editor editor = PreferenceManager.
                            getDefaultSharedPreferences(WeatherActivity.this).
                            edit();
                        editor.putString("weather", responseText);
                        editor.apply();
                        showWeatherInfo(weather);
                    } else {
                        Toast.makeText(WeatherActivity.this, "获取天气信息失败",
                            Toast.LENGTH_SHORT).show();
                    }
                }
            });
        }
        @Override
        public void onFailure(Call call, IOException e) {
            e.printStackTrace();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Toast.makeText(WeatherActivity.this, "获取天气信息失败",
                        Toast.LENGTH_SHORT).show();
                }
            });
        }
    });
}

```

```

    /**
     * 处理并展示 Weather 实体类中的数据
     */
    private void showWeatherInfo(Weather weather) {
        String cityName = weather.basic.cityName;
        String updateTime = weather.basic.update.updateTime.split(" ")[1];
        String degree = weather.now.temperature + "℃";
        String weatherInfo = weather.now.more.info;
        titleCity.setText(cityName);
        titleUpdateTime.setText(updateTime);
        degreeText.setText(degree);
        weatherInfoText.setText(weatherInfo);
        forecastLayout.removeAllViews();
        for (Forecast forecast : weather.forecastList) {
            View view = LayoutInflater.from(this).inflate(R.layout.forecast_
                item, forecastLayout, false);
            TextView dateText = (TextView) view.findViewById(R.id.date_text);
            TextView infoText = (TextView) view.findViewById(R.id.info_text);
            TextView maxText = (TextView) view.findViewById(R.id.max_text);
            TextView minText = (TextView) view.findViewById(R.id.min_text);
            dateText.setText(forecast.date);
            infoText.setText(forecast.more.info);
            maxText.setText(forecast.temperature.max);
            minText.setText(forecast.temperature.min);
            forecastLayout.addView(view);
        }
        if (weather.aqi != null) {
            aqiText.setText(weather.aqi.city.aqi);
            pm25Text.setText(weather.aqi.city.pm25);
        }
        String comfort = "舒适度：" + weather.suggestion.comfort.info;
        String carWash = "洗车指数：" + weather.suggestion.carWash.info;
        String sport = "运动建议：" + weather.suggestion.sport.info;
        comfortText.setText(comfort);
        carWashText.setText(carWash);
        sportText.setText(sport);
        weatherLayout.setVisibility(View.VISIBLE);
    }
}

```

这个活动中的代码也比较长，我们还是一步步梳理下。在 `onCreate()`方法中仍然先是去获取一些控件的实例，然后会尝试从本地缓存中读取天气数据。那么第一次肯定是没有缓存的，因此就会从 Intent 中取出天气 id，并调用 `requestWeather()`方法来从服务器请求天气数据。注意，请求数据的时候先将 ScrollView 进行隐藏，不然空数据的界面看上去会很奇怪。

`requestWeather()`方法中先是使用了参数中传入的天气 id 和我们之前申请好的 API Key 拼装出一个接口地址，接着调用 `HttpUtil.sendOkHttpRequest()`方法来向该地址发出请求，服务器会将相应城市的天气信息以 JSON 格式返回。然后我们在 `onResponse()`回调中先调用 `Utility.handleWeatherResponse()`方法将返回的 JSON 数据转换成 `Weather` 对象，再将当前线程切换到主线程。然后进行判断，如果服务器返回的 status 状态是 ok，就说明请求天气成功了，此时将返

回的数据缓存到 SharedPreferences 当中，并调用 showWeatherInfo()方法来进行内容显示。

showWeatherInfo()方法中的逻辑就比较简单了，其实就是从 Weather 对象中获取数据，然后显示到相应的控件上。注意在未来几天天气预报的部分我们使用了一个 for 循环来处理每天的天气信息，在循环中动态加载 forecast_item.xml 布局并设置相应的数据，然后添加到父布局当中。设置完了所有数据之后，记得要将 ScrollView 重新变成可见。

这样我们就将首次进入 WeatherActivity 时的逻辑全部梳理完了，那么当下一次再进入 WeatherActivity 时，由于缓存已经存在了，因此会直接解析并显示天气数据，而不会再次发起网络请求了。

处理完了 WeatherActivity 中的逻辑，接下来我们要做的，就是如何从省市县列表界面跳转到天气界面了，修改 ChooseAreaFragment 中的代码，如下所示：

```
public class ChooseAreaFragment extends Fragment {
    ...
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position,
                    long id) {
                if (currentLevel == LEVEL_PROVINCE) {
                    selectedProvince = provinceList.get(position);
                    queryCities();
                } else if (currentLevel == LEVEL_CITY) {
                    selectedCity = cityList.get(position);
                    queryCounties();
                } else if (currentLevel == LEVEL_COUNTY) {
                    String weatherId = countyList.get(position).getWeatherId();
                    Intent intent = new Intent(getActivity(), WeatherActivity.class);
                    intent.putExtra("weather_id", weatherId);
                    startActivity(intent);
                    getActivity().finish();
                }
            }
        });
        ...
    }
}
```

非常简单，这里在 onItemClick()方法中加入了一个 if 判断，如果当前级别是 LEVEL_COUNTY，就启动 WeatherActivity，并把当前选中县的天气 id 传递过去。

另外，我们还需要在 MainActivity 中加入一个缓存数据的判断才行。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);
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
        if (prefs.getString("weather", null) != null) {
            Intent intent = new Intent(this, WeatherActivity.class);
            startActivity(intent);
            finish();
        }
    }
}
```

可以看到，这里在 `onCreate()` 方法的一开始先从 `SharedPreferences` 文件中读取缓存数据，如果不为 `null` 就说明之前已经请求过天气数据了，那么就没必要让用户再次选择城市，而是直接跳转到 `WeatherActivity` 即可。

好了，现在重新运行一下程序，然后选择江苏→苏州→昆山，结果如图 14.21 所示。

然后我们还可以向下滑动查看更多天气信息，如图 14.22 所示。

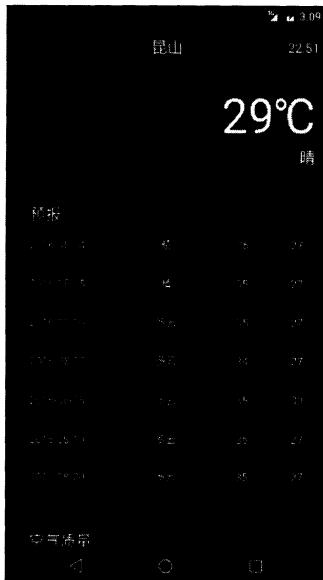


图 14.21 显示天气信息



图 14.22 查看更多天气信息

14.5.4 获取必应每日一图

虽说现在我们已经把天气界面编写得非常不错了，不过和市场上的一些天气软件的界面相比，仍然还是有一定差距的。出色的天气软件不会像我们现在这样使用一个固定的背景色，而是会根据不同的城市或者天气情况展示不同的背景图片。

当然实现这个功能并不复杂，最重要的是需要有服务器的接口支持。不过我实在是没有精力去准备这样一套完善的服务器接口，那么为了不让我们的天气界面过于单调，这里我准备使用一个巧妙的办法。

必应想必你肯定不会陌生，这是一个由微软开发的搜索引擎网站。这个网站除了提供强大的搜索功能之外，还有一个非常有特色的地方，就是它每天都会在首页展示一张精美的背景图片，如图 14.23 所示。

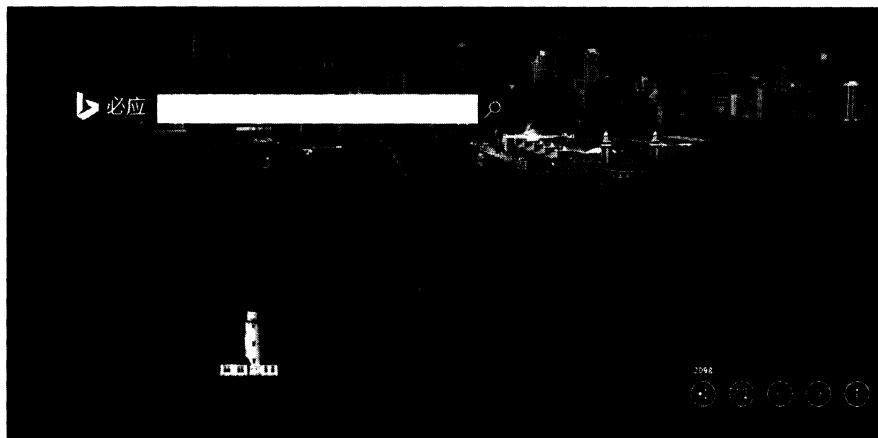


图 14.23 必应的首页

由于这些图片都是由必应精挑细选出来的，并且每天都会变化，如果我们使用它们来作为天气界面的背景图，不仅可以让界面变得更加美观，而且解决了界面一成不变、过于单调的问题。

为此我专门准备了一个获取必应每日一图的接口：http://guolin.tech/api/bing_pic。

访问这个接口，服务器会返回今日的必应背景图链接：

http://cn.bing.com/az/hprichbg/rb/ChicagoHarborLH_ZH-CN9974330969_1920x1080.jpg。

然后我们再使用 Glide 去加载这张图片就可以了。

总体思路就是这么简单，下面开始来动手实现吧。首先修改 activity_weather.xml 中的代码，如下所示：

```
<FrameLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:background="@color/colorPrimary">

    <ImageView
        android:id="@+id/bing_pic_img"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop" />

    <ScrollView
        android:id="@+id/weather_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="none"
        android:overScrollMode="never">

        ...

    </ScrollView>
</FrameLayout>
```

这里我们在 `FrameLayout` 中添加了一个 `ImageView`，并且将它的宽和高都设置成 `match_parent`。由于 `FrameLayout` 默认情况下会将控件都放置在左上角，因此 `ScrollView` 会完全覆盖住 `ImageView`，从而 `ImageView` 也就成为背景图片了。

接着修改 `WeatherActivity` 中的代码，如下所示：

```
public class WeatherActivity extends AppCompatActivity {

    ...

    private ImageView bingPicImg;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_weather);
        // 初始化各控件
        bingPicImg = (ImageView) findViewById(R.id.bing_pic_img);
        ...
        String bingPic = prefs.getString("bing_pic", null);
        if (bingPic != null) {
            Glide.with(this).load(bingPic).into(bingPicImg);
        } else {
            loadBingPic();
        }
    }

    /**
     * 根据天气 id 请求城市天气信息
     */
    public void requestWeather(final String weatherId) {
```

```

    ...
    loadBingPic();
}

/**
 * 加载必应每日一图
 */
private void loadBingPic() {
    String requestBingPic = "http://guolin.tech/api/bing_pic";
    HttpUtil.sendOkHttpRequest(requestBingPic, new Callback() {
        @Override
        public void onResponse(Call call, Response response) throws IOException {
            final String bingPic = response.body().string();
            SharedPreferences.Editor editor = PreferenceManager.
                getDefaultSharedPreferences(WeatherActivity.this).edit();
            editor.putString("bing_pic", bingPic);
            editor.apply();
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    Glide.with(WeatherActivity.this).load(bingPic).into
                        (bingPicImg);
                }
            });
        }

        @Override
        public void onFailure(Call call, IOException e) {
            e.printStackTrace();
        }
    });
}

...
}

```

可以看到，首先在 `onCreate()` 方法中获取了新增控件 `ImageView` 的实例，然后尝试从 `SharedPreferences` 中读取缓存的背景图片。如果有缓存的话就直接使用 `Glide` 来加载这张图片，如果没有的话就调用 `loadBingPic()` 方法去请求今日的必应背景图。

`loadBingPic()` 方法中的逻辑就非常简单了，先是调用了 `HttpUtil.sendOkHttpRequest()` 方法获取到必应背景图的链接，然后将这个链接缓存到 `SharedPreferences` 当中，再将当前线程切换到主线程，最后使用 `Glide` 来加载这张图片就可以了。另外需要注意，在 `requestWeather()` 方法的最后也需要调用一下 `loadBingPic()` 方法，这样在每次请求天气信息的时候同时也会刷新背景图片。现在重新运行一下程序，效果如图 14.24 所示。

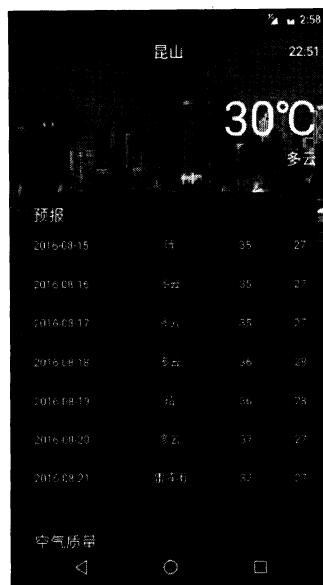


图 14.24 在天气界面显示必应背景图

怎么样？虽说只是换了一张背景图而已，但是整个界面的视觉体验就完全不一样了，瞬间提升了好几个档次。而且我们的背景图并不是一成不变的，每天都会是不同的图片，永远给人一种耳目一新的感觉。

不过如果你仔细观察图 14.24，你会发现背景图并没有和状态栏融合到一起，这样的话视觉体验就还是没有达到最佳的效果。虽说我们在 12.7.2 小节已经学习过如何将背景图和状态栏融合到一起，但当时是借助 Design Support 库完成的，而我们这个项目中并没有引入 Design Support 库。

当然如果还是模仿 12.7.2 小节的做法，引入 Design Support 库，然后嵌套 CoordinatorLayout、AppBarLayout、CollapsingToolbarLayout 等布局，也能实现背景图和状态栏融合到一起的效果，不过这样做就过于麻烦了，这里我准备教你另外一种更简单的实现方式。修改 WeatherActivity 中的代码，如下所示：

```
public class WeatherActivity extends AppCompatActivity {
    ...
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (Build.VERSION.SDK_INT >= 21) {
            View decorView = getWindow().getDecorView();
            decorView.setSystemUiVisibility(
                View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                | View.SYSTEM_UI_FLAG_LAYOUT_STABLE);
        }
    }
}
```

```
        getWindow().setStatusBarColor(Color.TRANSPARENT);
    }
    setContentView(R.layout.activity_weather);
    ...
}

...
}
```

由于这个功能是 Android 5.0 及以上的系统才支持的，因此我们先在代码中做了一个系统版本号的判断，只有当版本号大于或等于 21，也就是 5.0 及以上系统时才会执行后面的代码。

接着我们调用了 `getWindow().getDecorView()` 方法拿到当前活动的 DecorView，再调用它的 `setSystemUiVisibility()` 方法来改变系统 UI 的显示，这里传入 `View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN` 和 `View.SYSTEM_UI_FLAG_LAYOUT_STABLE` 就表示活动的布局会显示在状态栏上面，最后调用一下 `setStatusBarColor()` 方法将状态栏设置成透明色。

仅仅这些代码就可以实现让背景图和状态栏融合到一起的效果了。不过，如果运行一下程序，你会发现还是有些问题，天气界面的头布局几乎和系统状态栏紧贴到一起了，如图 14.25 所示。

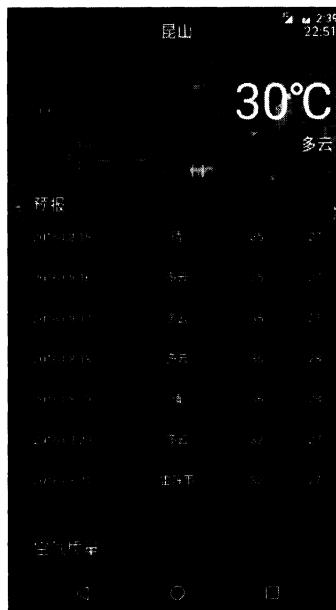


图 14.25 头布局和状态栏紧贴在一起

这是由于系统状态栏已经成为我们布局的一部分，因此没有单独为它留出空间。当然，这个问题也是非常好解决的，借助 `android:fitsSystemWindows` 属性就可以了。修改 `activity_weather.xml` 中的代码，如下所示：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/colorPrimary">

    ...

    <ScrollView
        android:id="@+id/weather_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scrollbars="none"
        android:overScrollMode="never">

        <LinearLayout
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:fitsSystemWindows="true">

            ...

        </LinearLayout>
    </ScrollView>
</FrameLayout>
```

这里在 ScrollView 的 LinearLayout 中增加了 `android:fitsSystemWindows` 属性，设置成 `true` 就表示会为系统状态栏留出空间。现在重新运行一下代码，效果如图 14.26 所示。



图 14.26 为系统状态栏留出空间

OK，这样第三阶段的开发工作也都完成了，我们把代码提交一下。

```
git add .
git commit -m "加入显示天气信息的功能。"
git push origin master
```

14.6 手动更新天气和切换城市

经过第三阶段的开发，现在酷欧天气的主体功能已经有了，不过你会发现目前存在着一个比较严重的 bug，就是当你选中了某一个城市之后，就没法再去查看其他城市的天气了，即使退出程序，下次进来的时候还会直接跳转到 WeatherActivity。

因此，在第四阶段中我们要加入切换城市的功能，并且为了能够实时获取到最新的天气，我们还会加入手动更新天气的功能。

14.6.1 手动更新天气

先来实现一下手动更新天气的功能。由于我们在上一节中对天气信息进行了缓存，目前每次展示的都是缓存中的数据，因此现在非常需要一种方式能够让用户手动更新天气信息。

至于如何触发更新事件呢？这里我准备采用下拉刷新的方式，正好我们之前也学过下拉刷新的用法，实现起来会比较简单。

首先修改 activity_weather.xml 中的代码，如下所示：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/colorPrimary">

    ...

    <android.support.v4.widget.SwipeRefreshLayout
        android:id="@+id/swipe_refresh"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <ScrollView
            android:id="@+id/weather_layout"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:scrollbars="none"
            android:overScrollMode="never">

            ...
    
```

```
</android.support.v4.widget.SwipeRefreshLayout>
```

```
</FrameLayout>
```

可以看到，这里在 ScrollView 的外面又嵌套了一层 SwipeRefreshLayout，这样 ScrollView 就自动拥有下拉刷新功能了。

然后修改 WeatherActivity 中的代码，加入更新天气的处理逻辑，如下所示：

```
public class WeatherActivity extends AppCompatActivity {

    public SwipeRefreshLayout swipeRefresh;

    ...

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ...
        swipeRefresh = (SwipeRefreshLayout) findViewById(R.id.swipe_refresh);
        swipeRefresh.setColorSchemeResources(R.color.colorPrimary);
        SharedPreferences prefs = PreferenceManager.
            getDefaultSharedPreferences(this);
        String weatherString = prefs.getString("weather", null);
        final String weatherId;
        if (weatherString != null) {
            // 有缓存时直接解析天气数据
            Weather weather = Utility.handleWeatherResponse(weatherString);
            weatherId = weather.basic.weatherId;
            showWeatherInfo(weather);
        } else {
            // 无缓存时去服务器查询天气
            weatherId = getIntent().getStringExtra("weather_id");
            weatherLayout.setVisibility(View.INVISIBLE);
            requestWeather(weatherId);
        }
        swipeRefresh.setOnRefreshListener(new SwipeRefreshLayout.
            OnRefreshListener() {
                @Override
                public void onRefresh() {
                    requestWeather(weatherId);
                }
            });
        ...
    }

    /**
     * 根据天气 id 请求城市天气信息
     */
    public void requestWeather(final String weatherId) {

        String weatherUrl = "http://guolin.tech/api/weather?cityid=" +
            weatherId + "&key=bc0418b57b2d4918819d3974ac1285d9";
```

```

HttpUtil.sendOkHttpRequest(weatherUrl, new Callback() {
    @Override
    public void onResponse(Call call, Response response) throws IOException {
        ...
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                if (weather != null && "ok".equals(weather.status)) {
                    SharedPreferences.Editor editor = PreferenceManager.
                        getDefaultSharedPreferences(WeatherActivity.
                            this).edit();
                    editor.putString("weather", responseText);
                    editor.apply();
                    showWeatherInfo(weather);
                } else {
                    Toast.makeText(WeatherActivity.this, "获取天气信息失败",
                        Toast.LENGTH_SHORT).show();
                }
                swipeRefresh.setRefreshing(false);
            }
        });
    }
}

@Override
public void onFailure(Call call, IOException e) {
    e.printStackTrace();
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(WeatherActivity.this, "获取天气信息失败",
                Toast.LENGTH_SHORT).show();
            swipeRefresh.setRefreshing(false);
        }
    });
}
});

loadBingPic();
}

...
}

```

修改的代码并不算多，首先在 `onCreate()` 方法中获取到了 `SwipeRefreshLayout` 的实例，然后调用 `setColorSchemeResources()` 方法来设置下拉刷新进度条的颜色，这里我们就使用主题中的 `colorPrimary` 作为进度条的颜色了。接着定义了一个 `weatherId` 变量，用于记录城市的天气 id，然后调用 `setOnRefreshListener()` 方法来设置一个下拉刷新的监听器，当触发了下拉刷新操作的时候，就会回调这个监听器的 `onRefresh()` 方法，我们在这里去调用 `requestWeather()` 方法请求天气信息就可以了。

另外不要忘记，当请求结束后，还需要调用 `SwipeRefreshLayout` 的 `setRefreshing()` 方法

并传入 `false`，用于表示刷新事件结束，并隐藏刷新进度条。

现在重新运行一下程序，并在屏幕的主界面向下拖动，效果如图 14.27 所示。

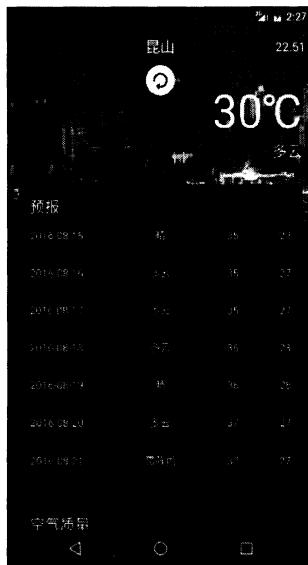


图 14.27 手动更新天气

更新完天气信息之后，下拉进度条会自动消失。

14.6.2 切换城市

完成了手动更新天气的功能，接下来我们继续实现切换城市功能。

既然是要切换城市，那么就肯定需要遍历全国省市县的数据，而这个功能我们早在 14.4 节就已经完成了，并且当时考虑为了方便后面的复用，特意选择了在碎片当中实现。因此，我们其实只需要在天气界面的布局中引入这个碎片，就可以快速集成切换城市功能了。

虽说实现原理很简单，但是显然我们也不可能让引入的碎片把天气界面遮挡住，这又该怎么办呢？还记得 12.3 节学过的滑动菜单功能吗？将碎片放入到滑动菜单中真是再合适不过了，正常情况下它不占据主界面的任何空间，想要切换城市的时候只需要通过滑动的方式将菜单显示出来就可以了。

下面我们就按照这种思路来实现。首先按照 Material Design 的建议，我们需要在头布局中加入一个切换城市的按钮，不然的话用户可能根本就不知道屏幕的左侧边缘是可以拖动的。修改 `title.xml` 中的代码，如下所示：

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
```

```
    android:layout_height="?attr/actionBarSize">

    <Button
        android:id="@+id/nav_button"
        android:layout_width="30dp"
        android:layout_height="30dp"
        android:layout_marginLeft="10dp"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:background="@drawable/ic_home" />

    ...
</RelativeLayout>
```

这里添加了一个 Button 作为切换城市的按钮，并且让它居左显示。另外，我提前准备好了 一张图片来作为按钮的背景图。

接着修改 activity_weather.xml 布局来加入滑动菜单功能，如下所示：

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/colorPrimary">

    ...
<android.support.v4.widget.DrawerLayout
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v4.widget.SwipeRefreshLayout
        android:id="@+id/swipe_refresh"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        ...
    </android.support.v4.widget.SwipeRefreshLayout>

    <fragment
        android:id="@+id/choose_area_fragment"
        android:name="com.coolweather.android.ChooseAreaFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        />
    </android.support.v4.widget.DrawerLayout>

</FrameLayout>
```

可以看到，我们在 SwipeRefreshLayout 的外面又嵌套了一层 DrawerLayout。DrawerLayout 中的第一个子控件用于作为主屏幕中显示的内容，第二个子控件用于作为滑动菜单中显示的内容，因此这里我们在第二个子控件的位置添加了用于遍历省市县数据的碎片。

接下来需要在 WeatherActivity 中加入滑动菜单的逻辑处理，修改 WeatherActivity 中的代码，如下所示：

```
public class WeatherActivity extends AppCompatActivity {

    public DrawerLayout drawerLayout;

    private Button navButton;

    ...

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

        drawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
        navButton = (Button) findViewById(R.id.nav_button);
        ...

        navButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                drawerLayout.openDrawer(GravityCompat.START);
            }
        });
    }

    ...
}
```

很简单，首先在 `onCreate()` 方法中获取到新增的 `DrawerLayout` 和 `Button` 的实例，然后在 `Button` 的点击事件中调用 `DrawerLayout` 的 `openDrawer()` 方法来打开滑动菜单就可以了。

不过现在还没有结束，因为这仅仅是打开了滑动菜单而已，我们还需要处理切换城市后的逻辑才行。这个工作就必须要在 `ChooseAreaFragment` 中进行了，因为之前选中了某个城市后是跳转到 `WeatherActivity` 的，而现在由于我们本来就是在 `WeatherActivity` 当中的，因此并不需要跳转，只是去请求新选择城市的天气信息就可以了。

那么很显然这里我们需要根据 `ChooseAreaFragment` 的不同状态来进行不同的逻辑处理，修改 `ChooseAreaFragment` 中的代码，如下所示：

```
public class ChooseAreaFragment extends Fragment {

    ...

    @Override
```

```

public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    listView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position,
                long id) {
            if (currentLevel == LEVEL_PROVINCE) {
                selectedProvince = provinceList.get(position);
                queryCities();
            } else if (currentLevel == LEVEL_CITY) {
                selectedCity = cityList.get(position);
                queryCounties();
            } else if (currentLevel == LEVEL_COUNTY) {
                String weatherId = countyList.get(position).getWeatherId();
                if (getActivity() instanceof MainActivity) {
                    Intent intent = new Intent(getActivity(), WeatherActivity.
                            class);
                    intent.putExtra("weather_id", weatherId);
                    startActivity(intent);
                    getActivity().finish();
                } else if (getActivity() instanceof WeatherActivity) {
                    WeatherActivity activity = (WeatherActivity) getActivity();
                    activity.drawerLayout.closeDrawers();
                    activity.swipeRefresh.setRefreshing(true);
                    activity.requestWeather(weatherId);
                }
            }
        });
    ...
}
...
}

```

这里我使用了一个 Java 中的小技巧, `instanceof` 关键字可以用来判断一个对象是否属于某个类的实例。我们在碎片中调用 `getActivity()` 方法, 然后配合 `instanceof` 关键字, 就能轻松判断出该碎片是在 `MainActivity` 当中, 还是在 `WeatherActivity` 当中。如果是在 `MainActivity` 当中, 那么处理逻辑不变。如果是在 `WeatherActivity` 当中, 那么就关闭滑动菜单, 显示下拉刷新进度条, 然后请求新城市的天气信息。

这样我们就把切换城市的功能全部完成了, 现在可以重新运行一下程序, 效果如图 14.28 所示。

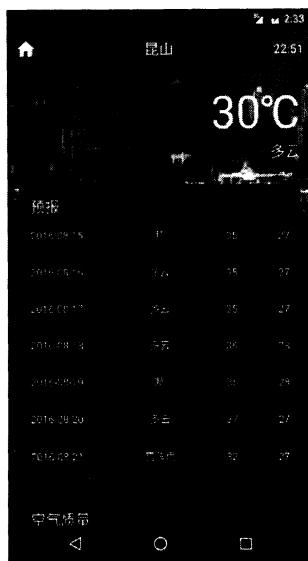


图 14.28 拥有切换城市按钮的天气界面

可以看到，标题栏上多出了一个用于切换城市的按钮。点击该按钮，或者在屏幕的左侧边缘进行拖动，就能让滑动菜单界面显示出来了，如图 14.29 所示。

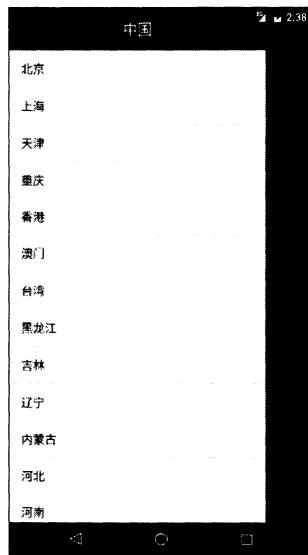


图 14.29 显示滑动菜单界面

然后我们就可以在这里切换其他城市了。选中城市之后滑动菜单会自动关闭，并且主界面上的天气信息也会更新成你选择的那个城市。

这样，第四阶段的开发任务也完成了。当然，仍然不要忘记提交代码。

```
git add .
git commit -m "新增切换城市和手动更新天气的功能。"
git push origin master
```

14.7 后台自动更新天气

为了要让酷欧天气更加智能，在第五阶段我们准备加入后台自动更新天气的功能，这样就可以尽可能地保证用户每次打开软件时看到的都是最新的天气信息。

要想实现上述功能，就需要创建一个长期在后台运行的定时任务，这个功能肯定是最难不倒你的，因为我们在13.5节中就已经学习过了。

首先在service包下新建一个服务，右击com.coolweather.android.service→New→Service→Service，创建一个AutoUpdateService，并将Exported和Enabled这两个属性都勾中。然后修改AutoUpdateService中的代码，如下所示：

```
public class AutoUpdateService extends Service {

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

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        updateWeather();
        updateBingPic();
        AlarmManager manager = (AlarmManager) getSystemService(ALARM_SERVICE);
        int anHour = 8 * 60 * 60 * 1000; // 这是8小时的毫秒数
        long triggerAtTime = SystemClock.elapsedRealtime() + anHour;
        Intent i = new Intent(this, AutoUpdateService.class);
        PendingIntent pi = PendingIntent.getService(this, 0, i, 0);
        manager.cancel(pi);
        manager.set(AlarmManager.ELAPSED_REALTIME_WAKEUP, triggerAtTime, pi);
        return super.onStartCommand(intent, flags, startId);
    }

    /**
     * 更新天气信息
     */
    private void updateWeather(){
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
        String weatherString = prefs.getString("weather", null);
        if (weatherString != null) {
            // 有缓存时直接解析天气数据
            Weather weather = Utility.handleWeatherResponse(weatherString);
            String weatherId = weather.basic.weatherId;

            String weatherUrl = "http://guolin.tech/api/weather?cityid=" +
```

```

        weatherId + "&key=bc0418b57b2d4918819d3974ac1285d9";
    HttpUtil.sendOkHttpRequest(weatherUrl, new Callback() {
        @Override
        public void onResponse(Call call, Response response) throws
                IOException {
            String responseText = response.body().string();
            Weather weather = Utility.handleWeatherResponse(responseText);
            if (weather != null && "ok".equals(weather.status)) {
                SharedPreferences.Editor editor = PreferenceManager.
                    getDefaultSharedPreferences(AutoUpdateService.this).
                    edit();
                editor.putString("weather", responseText);
                editor.apply();
            }
        }
        @Override
        public void onFailure(Call call, IOException e) {
            e.printStackTrace();
        }
    });
}
}

/**
 * 更新必应每日一图
 */
private void updateBingPic() {
    String requestBingPic = "http://guolin.tech/api/bing_pic";
    HttpUtil.sendOkHttpRequest(requestBingPic, new Callback() {
        @Override
        public void onResponse(Call call, Response response) throws IOException {
            String bingPic = response.body().string();
            SharedPreferences.Editor editor = PreferenceManager.getDefault
                SharedPreferences(AutoUpdateService.this).edit();
            editor.putString("bing_pic", bingPic);
            editor.apply();
        }
        @Override
        public void onFailure(Call call, IOException e) {
            e.printStackTrace();
        }
    });
}
}

```

可以看到，在 `onStartCommand()`方法中先是调用了 `updateWeather()`方法来更新天气，然后调用了 `updateBingPic()`方法来更新背景图片。这里我们将更新后的数据直接存储到 `SharedPreferences` 文件中就可以了，因为打开 `WeatherActivity` 的时候都会优先从 `SharedPreferences` 缓存中读取数据。

之后就是我们学习过的创建定时任务的技巧了，为了保证软件不会消耗过多的流量，这里将时间间隔设置为 8 小时，8 小时后 AutoUpdateReceiver 的 `onStartCommand()` 方法就会重新执行，这样也就实现后台定时更新的功能了。

不过，我们还需要在代码某处去激活 AutoUpdateService 这个服务才行。修改 WeatherActivity 中的代码，如下所示：

```
public class WeatherActivity extends AppCompatActivity {
    ...
    /**
     * 处理并展示 Weather 实体类中的数据。
     */
    private void showWeatherInfo(Weather weather) {
        if (weather != null && "ok".equals(weather.status)) {
            ...
            Intent intent = new Intent(this, AutoUpdateService.class);
            startService(intent);
        } else {
            Toast.makeText(WeatherActivity.this, "获取天气信息失败", Toast.LENGTH_SHORT).show();
        }
    }
}
```

可以看到，这里在 `showWeather()` 方法的最后加入启动 AutoUpdateService 这个服务的代码，这样只要一旦选中了某个城市并成功更新天气之后，AutoUpdateService 就会一直在后台运行，并保证每 8 小时更新一次天气。

现在可以再提交一下代码：

```
git add .
git commit -m "增加后台自动更新天气的功能。"
git push origin master
```

14.8 修改图标和名称

目前的酷欧天气看起来还不太像是一个正式的软件，为什么呢？因为都还没有一个像样的图标呢。一直使用 Android Studio 自动生成的图标确实不太合适，是时候需要换一下了。

这里我事先准备好了一张图片来作为软件图标，由于我也不是搞美术的，因此图标设计得非常简单，如图 14.30 所示。



图 14.30 酷欧天气的图标

理论上来讲，我们应该给这个图标提供几种不同分辨率的版本，然后分别放入到相应分辨率的 mipmap 目录下，这里简单起见，我就都使用同一张图了。将这张图片命名成 logo.png，放入到所有以 mipmap 开头的目录下，然后修改 AndroidManifest.xml 中的代码，如下所示：

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

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

    <application
        android:name="org.litepal.LitePalApplication"
        android:allowBackup="true"
        android:icon="@mipmap/logo"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        ...
    </application>

</manifest>
```

这里将<application>标签的 android:icon 属性指定成@mipmap/logo 就可以修改程序图标了。接下来我们还需要修改一下程序的名称，打开 res/values/string.xml 文件，其中 app_name 对应的就是程序名称，将它修改成酷欧天气即可，如下所示：

```
<resources>
    <string name="app_name">酷欧天气</string>
</resources>
```

现在重新运行一遍程序，这时观察酷欧天气的桌面图标，如图 14.31 所示。



图 14.31 手机桌面图标

养成良好的习惯，仍然不要忘记提交代码。

```
git add .
git commit -m "修改程序图标和名称。"
git push origin master
```

这样我们就终于大功告成了！

14.9 你还可以做的事情

经过五个阶段的开发，酷欧天气已经是一个完善、成熟的软件了吗？嘿嘿，还差得远呢！现在的酷欧天气只能说是具备了一些最基本的功能，和那些商用的天气软件比起来还有很大的差

距，因此你仍然还有非常巨大的发挥空间来对它进行完善。

比如说以下功能是你可以考虑加入到酷欧天气中的。

- 增加设置选项，让用户选择是否允许后台自动更新天气，以及设定更新的频率。
- 优化软件界面，提供多套与天气对应的图片，让程序可以根据不同的天气自动切换背景图。
- 允许选择多个城市，可以同时观察多个城市的天气信息，不用来回切换。
- 提供更加完整的天气信息，目前我们只使用了和风天气返回的一小部分数据而已。

另外，由于酷欧天气的源码已经托管在了 GitHub 上面，如果你想在现有代码的基础上继续对这个项目进行完善，就可以使用 GitHub 的 Fork 功能。

首先登录你自己的 GitHub 账号，然后打开酷欧天气版本库的主页：<https://github.com/guolindev/coolweather>，这时在页面头部的最右侧会有一个 Fork 按钮，如图 14.32 所示。



图 14.32 GitHub Fork 按钮

点击一下 Fork 按钮就可以将酷欧天气这个项目复制一份到你的账号下，再使用 `git clone` 命令将它克隆到本地，然后你就可以在现有代码的基础上随心所欲地添加任何功能并提交了。

第 15 章

最后一步——将应用发布到 360 应用商店

应用已经开发出来了，下一步我们需要思考推广方面的工作。那么如何才能让更多的用户知道并使用我们的应用程序呢？在手机领域，最常见的做法就是将程序发布到某个应用商店中，这样用户就可以通过商店找到我们的应用程序，然后轻松地进行下载和安装。

说到应用商店，在Android领域真的可以称得上是百家争鸣，除了谷歌官方推出的Google Play之外，在中国还有像360、豌豆荚、百度、应用宝等知名的应用商店。当然，这些商店所提供的功能都是比较类似的，发布应用的方法也大同小异，因此这里我们就只学习如何将应用发布到360应用商店，其他应用商店的发布方法相信你完全可以自己摸索出来。

15.1 生成正式签名的 APK 文件

之前我们一直都是通过Android Studio来将程序安装到手机上的，而它背后实际的工作流程是，Android Studio会将程序代码打包成一个APK文件，然后将这个文件传输到手机上，最后再执行安装操作。Android系统会将所有的APK文件识别为应用程序的安装包，类似于Windows系统上的EXE文件。

但并不是所有的APK文件都能成功安装到手机上，Android系统要求只有签名后的APK文件才可以安装，因此我们还需要对生成的APK文件进行签名才行。那么你可能会有疑问了，直接通过Android Studio来运行程序的时候好像并没有进行过签名操作啊，为什么还能将程序安装到手机上呢？这是因为Android Studio使用了一个默认的keystore文件帮我们自动进行了签名。点击Android Studio右侧工具栏的Gradle→项目名→:app→Tasks→android，双击signingReport，结果如图15.1所示。

```
Variant: debug
Config: debug
Store: C:\Users\Administrator\.android\debug.keystore
```

图 15.1 查看默认的 keystore 文件

也就是说，我们所有通过 Android Studio 来运行的程序都是使用了这个 debug.keystore 文件来进行签名的。不过这仅仅适用于开发阶段而已，现在酷欧天气已经快要发布了，要使用一个正式的 keystore 文件来进行签名才行。下面我们就来学习一下，如何生成一个带有正式签名的 APK 文件。

15.1.1 使用 Android Studio 生成

先学习一下如何使用 Android Studio 来生成正式签名的 APK 文件。点击 Android Studio 导航栏上的 Build→Generate Signed APK，首次点击可能会提示让我们输入操作系统的密码，如图 15.2 所示。

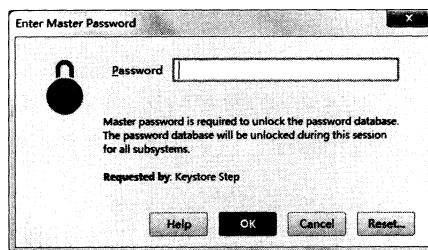


图 15.2 输入操作系统密码提示框

输入密码之后点击 OK，则会弹出如图 15.3 所示的创建签名 APK 对话框。

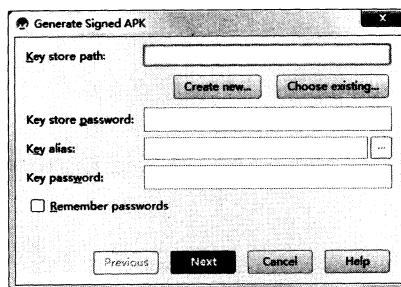


图 15.3 创建签名 APK 对话框

由于目前我们还没有一个正式的 keystore 文件，所以应该点击 Create new 按钮，然后会弹出一个新的对话框来让我们填写创建 keystore 文件所必要的信息。根据自己的实际情况进行填写就行了，如图 15.4 所示。

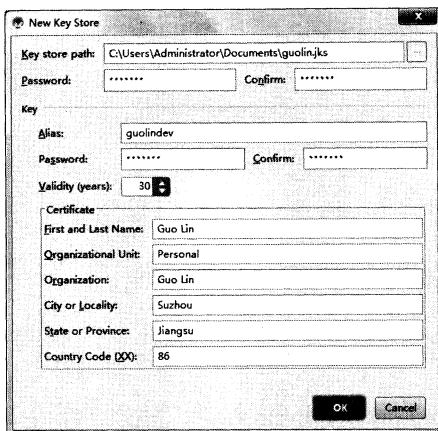


图 15.4 填写 keystore 文件信息

这里需要注意，在 Validity 那一栏填写的是 keystore 文件的有效时长，单位是年，一般建议时间可以填得长一些，比如我填了 30 年。然后点击 OK，这时我们刚才填写的信息会自动填充到创建签名 APK 对话框当中，如图 15.5 所示。

如果你希望以后都不用再输 keystore 的密码了，可以将 Remember passwords 选项勾上。然后点击 Next，这时就要选择 APK 文件的输出地址了，如图 15.6 所示。

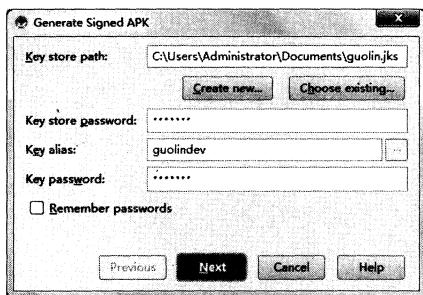


图 15.5 信息自动填充完整

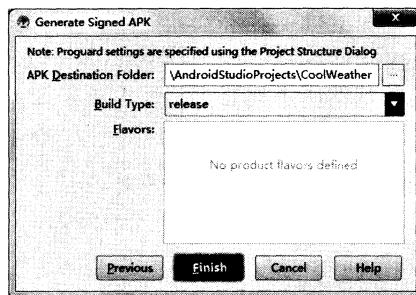


图 15.6 选择 APK 文件的输出地址

这里默认是将 APK 文件生成到项目的根目录下，我就不做修改了。现在点击 Finish，然后稍等一段时间，APK 文件就都会生成好了，并且会在右上角弹出一个如图 15.7 所示的提示。

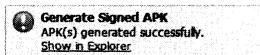


图 15.7 提示 APK 文件生成成功

我们点击提示上的 Show in Explorer 可以立刻查看生成的 APK 文件，如图 15.8 所示。



图 15.8 查看生成的 APK 文件

这里的 app-release.apk 就是带有正式签名的 APK 文件了。

15.1.2 使用 Gradle 生成

上一小节中我们使用了 Android Studio 提供的可视化工具来生成带有正式签名的 APK 文件，除此之外，Android Studio 其实还提供了另外一种方式——使用 Gradle 生成，下面我们就来学习一下。

Gradle 是一个非常先进的项目构建工具，在 Android Studio 中开发的所有项目都是使用它来构建的。在之前的项目中，我们也体验过了 Gradle 带来的很多便利之处，比如说当需要添加依赖库的时候不需要自己再去手动下载了，而是直接在 dependencies 闭包中添加一句引用声明就可以了。

不过这里我要提醒你一句，如果你想将 Gradle 完全精通的话，这个难度就比较大了。Gradle 的用法极为丰富，想要完全掌握它的用法，其复杂程度并不亚于学习一门新的语言（Gradle 是使用 Groovy 语言编写的）。而 Android 中主要只是使用 Gradle 来构建项目而已，因此这里我们掌握一些它的基本用法就好了，重点还是要放在功能开发上面，不要本末倒置了。当然，如果你对 Gradle 非常感兴趣，也可以到网上去查询它的更多用法。

下面我们开始学习如何使用 Gradle 来生成带有正式签名的 APK 文件。编辑 app/build.gradle 文件，在 android 闭包中添加如下内容：

```
android {
    compileSdkVersion 24
    buildToolsVersion "24.0.2"
    defaultConfig {
        applicationId "com.coolweather.android"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
    }
    signingConfigs {
        config {
            storeFile file('C:/Users/Administrator/Documents/guolin.jks')
            storePassword '1234567'
            keyAlias 'guolindev'
            keyPassword '1234567'
        }
    }
}
```

```

        }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
        }
    }
}

```

可以看到，这里在 android 闭包中添加了一个 signingConfigs 闭包，然后在 signingConfigs 闭包中又添加了一个 config 闭包。接着在 config 闭包中配置 keystore 文件的各种信息，storeFile 用于指定 keystore 文件的位置，storePassword 用于指定密码，keyAlias 用于指定别名，keyPassword 用于指定别名密码。

将签名信息都配置好了之后，接下来只需要在生成正式版 APK 的时候去应用这个配置就可以了。继续编辑 app/build.gradle 文件，如下所示：

```

android {
    ...
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                'proguard-rules.pro'
            signingConfig signingConfigs.config
        }
    }
}

```

这里我们在 buildTypes 下面的 release 闭包中应用了刚才添加的签名配置，这样当生成正式版 APK 文件的时候就会自动使用我们刚才配置的签名信息来进行签名了。

现在 build.gradle 文件已经配置完成，那么我们如何才能生成 APK 文件呢？其实非常简单，Android Studio 中内置了很多的 Gradle Tasks，其中就包括了生成 APK 文件的 Task。点击右侧工具栏的 Gradle→项目名→:app→Tasks→build，如图 15.9 所示。

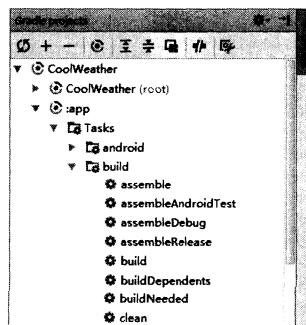


图 15.9 查看内置 Gradle Tasks

其中 assembleDebug 用于生成测试版的 APK 文件, assembleRelease 用于生成正式版的 APK 文件, assemble 用于同时生成测试版和正式版的 APK 文件。在生成 APK 之前, 先要双击 clean 这个 Task 来清理一下当前项目, 然后双击 assembleRelease, 结果如图 15.10 所示。

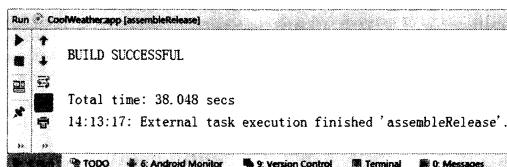


图 15.10 assembleRelease 执行成功

可以看到, 这里提示我们 BUILD SUCCESSFUL, 说明 assembleRelease 执行成功了。APK 文件会自动生成在 app/build/outputs/apk 目录下, 如图 15.11 所示。

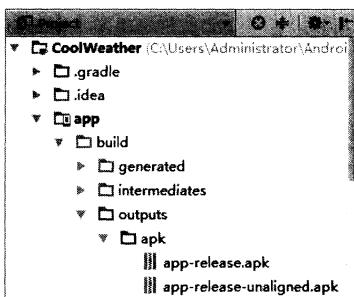


图 15.11 查看生成的 APK 文件

其中, app-release.apk 就是带有正式签名的 APK 文件了。另外还有一个 app-release-unaligned.apk, 这是一个没有经过对齐的正式版 APK 文件, 我们直接忽略它就可以了。

虽说现在 APK 文件已经成功生成了, 不过还有一个小细节需要注意一下。目前 keystore 文件的所有信息都是以明文的形式直接配置在 build.gradle 中的, 这样就不太安全。Android 推荐的做法是将这类敏感数据配置在一个独立的文件里面, 然后再在 build.gradle 中去读取这些数据。

下面我们来按照这种方式实现。Android Studio 项目的根目录下有一个 gradle.properties 文件, 它是用来配置全局键值对数据的, 我们在 gradle.properties 文件中添加如下内容:

```
KEY_PATH=C:/Users/Administrator/Documents/guolin.jks
KEY_PASS=1234567
ALIAS_NAME=guolindev
ALIAS_PASS=1234567
```

可以看到, 这里将 keystore 文件的各种信息以键值对的形式进行了配置, 然后我们在 build.gradle 中去读取这些数据就可以了。编辑 app/build.gradle 文件, 如下所示:

```
android {
    ...
}
```

```

signingConfigs {
    config {
        storeFile file(KEY_PATH)
        storePassword KEY_PASS
        keyAlias ALIAS_NAME
        keyPassword ALIAS_PASS
    }
}
...
}

```

这里只需要将原来的明文配置改成相应的键值，一切就完工了。这样直接查看 build.gradle 文件是无法看到 keystore 文件的各种信息的，只有查看 gradle.properties 文件才能看得到。然后我们只需要将 gradle.properties 文件保护好就行了，比如说将它从 Git 版本控制中排除。这样 gradle.properties 文件就只会保留在本地，从而也就不用担心 keystore 文件的信息会泄漏了。

15.1.3 生成多渠道 APK 文件

现在你已经掌握了两种生成带有正式签名的 APK 文件的方式，从简易程度上来讲，两种方式差不多，基本都还是比较简单的，选择使用哪一种全凭你自己的喜好。

现在 APK 文件已经生成好了，可能在大多数情况下，我们都只需要一个 APK 文件就足够了，不过本小节中我们再来讨论一种比较特殊的情况——生成多渠道 APK 文件。

在本章的开头就已经提到过，目前 Android 领域的应用商店非常多，不像苹果只有一个 App Store。当然我们完全可以使用同一个 APK 文件来上架不同的应用商店，但是如果你有一些特殊需求的话，比如说针对不同的应用商店渠道来定制不同的界面，这就比较头疼了。

传统情况下，开发这种差异性需求非常痛苦，通常需要维护多份代码版本，然后逐个打成相应渠道的 APK 文件。一旦有任何功能变更就苦不堪言，因为每份代码版本里面都需要逐个修改一遍。

幸运的是，现在 Android Studio 提供了一种非常方便的方法来应对这种差异性需求，极大地解决了之前版本维护困难的问题，下面我们就来学习一下。

比如说这里我们准备生成 360 和百度两个渠道的 APK 文件，那么修改 app/build.gradle 文件，如下所示：

```

android {
    compileSdkVersion 24
    buildToolsVersion "24.0.2"
    defaultConfig {
        applicationId "com.coolweather.android"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 1
        versionName "1.0"
    }
}

```

```

    }
productFlavors {
    qihoo {
        applicationId "com.coolweather.android.qihoo"
    }
    baidu {
        applicationId "com.coolweather.android.baidu"
    }
}
...
}

```

可以看到，这里添加了一个 `productFlavors` 闭包，然后在该闭包中添加所有的渠道配置就可以了。注意 Gradle 中的配置规定不能以数字开头，因此这里我将 360 的渠道名配置成了 `qihoo`。渠道名的闭包中可以覆写 `defaultConfig` 中的任何一个属性，比如说这里将 `applicationId` 属性进行了覆写，那么最终生成的各渠道 APK 文件的包名也将各不相同。

接下来我们开始针对不同渠道编写差异性需求。在 `app/src` 目录下（`main` 的平级目录）新建一个 `baidu` 目录，然后在 `baidu` 目录下再新建 `java` 和 `res` 这两个目录，如图 15.12 所示。

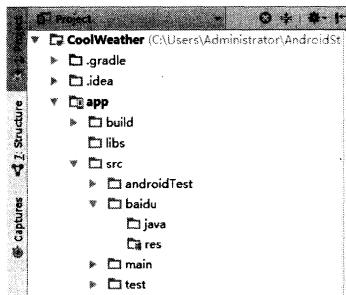


图 15.12 创建渠道专属目录

这样我们就可以在这里编写百度渠道特有的功能了，`java` 目录用于存放代码，`res` 目录用于存放资源，如果需要覆写 `AndroidManifest` 文件中的内容，还可以在 `baidu` 目录下再新建一个 `AndroidManifest.xml` 文件。

当然，实际上我们并没有什么渠道差异性的需求，因此这里也只是为了演示一下，我们就给不同渠道的 APK 起一个不同的应用名吧。

应用名之前是定义在 `main/res/values/string.xml` 文件中的，那么我们在 `baidu` 目录下也建立一个相同的目录结构，然后将 `baidu/res/values/string.xml` 中的内容进行如下修改：

```

<resources>
    <string name="app_name">酷欧百度版</string>
</resources>

```

这样百度渠道的 APK 就会使用 `baidu/res/values/string.xml` 中定义的应用名来覆盖原有的应用名。同样的道理，我们再新建一个 `qihoo` 目录，然后在 `qihoo` 目录下也建立相同的目录结构，并

将 string.xml 中的内容进行如下修改：

```
<resources>
    <string name="app_name">酷欧 360 版</string>
</resources>
```

这样我们就以一个简单的示例实现渠道差异性需求了，下面开始来生成多渠道的 APK 文件。观察右侧工具栏的 Gradle Tasks 列表，你会发现里面多出了几个新的 Task，如图 15.13 所示。

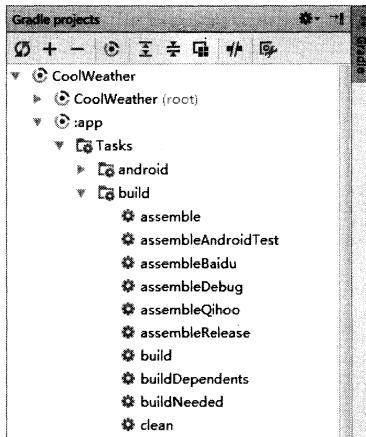


图 15.13 查看新的 Task

其中，如果你只想生成百度渠道的 APK 文件，那么就执行 assembleBaidu；如果你只想生成 360 渠道的 APK 文件，那么就执行 assembleQihoo；如果你想一次性生成所有渠道的 APK 文件，那么就还是执行 assembleRelease。

除了使用 Gradle 的方式生成之外，使用 Android Studio 提供的可视化工具也是能生成多渠道 APK 文件的，如图 15.14 所示。

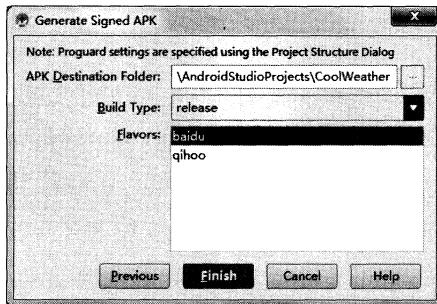


图 15.14 使用可视化工具生成多渠道 APK

这里我们可以选择是生成百度渠道的 APK 文件，还是生成 360 渠道的 APK 文件，如果你想

一次性生成多个渠道的 APK 文件，按住 CTRL 键就可以进行多选了。

接下来我们可以通过 `adb install` 命令将生成好的 APK 文件安装到模拟器上，如图 15.15 所示。

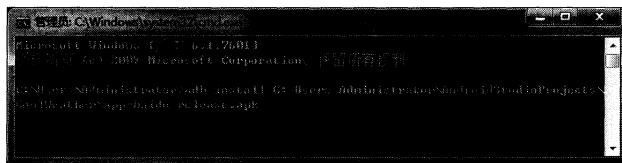


图 15.15 将生成的 APK 安装到模拟器上

`adb install` 命令的后面加上 APK 文件的路径，就可以将该 APK 文件安装到模拟器上了。我们使用同样的方法将百度和 360 这两个渠道的 APK 文件都安装到模拟器上，结果如图 15.16 所示。



图 15.16 模拟器上的安装结果

可以看到，目前模拟器上有 3 个版本的酷欧天气，这是由于之前我们在 `productFlavors` 中覆盖了各渠道的 `applicationId` 属性，保证每个 APK 文件的包名都不相同，因而它们才能安装到同一个设备上面。另外，从应用名上来看，渠道差异性开发工作也顺利完成了。

不过，上面的例子只是为了演示生成多渠道 APK 功能而特意编写的，实际上我们并没有这个需求。现在将 `productFlavors` 闭包删除，恢复成之前的 APK 文件，我们准备进行上架操作。

15.2 申请 360 开发者账号

目前，酷欧天气的 APK 安装包已经准备好了，但如果想要把它发布到 360 应用商店，还需要去申请一个 360 开发者账号才行，申请地址是：<http://dev.360.cn>。

打开该网页，在页面顶部有登录和注册按钮。如果你还没有 360 账号，则需要在这里注册一个新的账号，如果你之前已经有 360 账号了，那么直接登录就可以了。

登录成功之后打开 <http://dev.360.cn/mod/developer> 这个网址，来申请成为开发者，如图 15.17 所示。

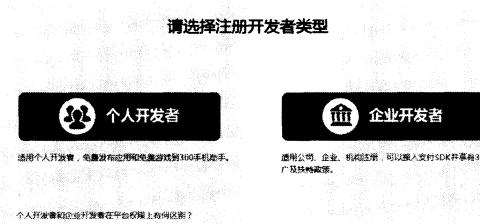


图 15.17 选择注册开发者类型

这里可以选择是申请成为个人开发者还是企业开发者。很显然，我们是以个人的身份来发布应用的，那么点击个人开发者就可以了。

接下来需要填写一些基本信息和联系方式，如图 15.18 所示。

This screenshot shows the 'Basic Information' section of the registration form. It includes fields for '注册账号' (Account), '开发者姓名' (Developer Name), '出品人' (Publisher), '上传证件照' (Upload ID Card Photo), and '个人身份证件' (Personal ID Card). Each field has specific instructions and placeholder text.

图 15.18 填基本信息和联系方式

填写完基本信息之后向下滚动继续填写联系方式，全部填写完成之后，点击屏幕最下方的“同意并注册开发者”按钮来完成注册，如图 15.19 所示。

我已阅读并同意《360移动开放平台服务条款》

图 15.19 完成开发者注册

这样你就成功成为一名 360 开发者了！

15.3 发布应用程序

接下来我们开始发布酷欧天气这个应用，还是在浏览器访问地址：<http://dev.360.cn>，你会在界面上看到如图 15.20 所示的内容。

然后点击软件发布，就会显示如图 15.21 所示的界面。



图 15.20 软件发布和游戏发布

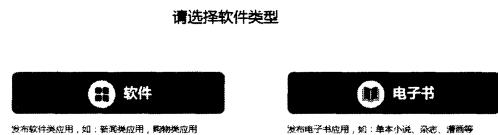


图 15.21 选择软件类型

我们需要选择是发布软件类应用还是电子书类应用，这里点击软件。接下来会弹出一个新的界面让我们上传 APK 以及填写应用信息。首先来上传 APK 吧，点击上传按钮，选择带有正式签名的 APK 文件，然后就会自动开始上传了，上传完成之后会显示如图 15.22 所示的界面。

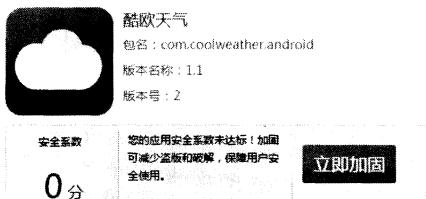


图 15.22 上传 APK 完成

这个界面提醒我们，目前应用的安全系数较低，建议对 APK 进行加固。实际上这个是 360 应用商店的特殊需求，并不是所有应用商店都要求进行加固的。但是我们还是得按照它的要求来修改，不然审核可能会不通过。

这里点击立即加固按钮，360 会帮忙我们将原 APK 文件进行加固，并生成一个新的 APK 文件，如图 15.23 所示。

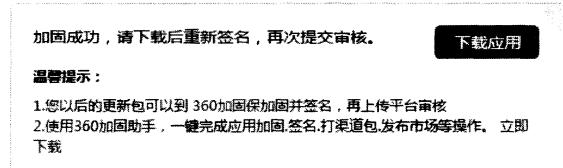


图 15.23 加固成功提示

不过这个加固后的 APK 文件是没有经过签名的，也就是说我们还需要将它下载下来，然后手动进行签名才行。

点击下载应用按钮，先将加固后的 APK 文件下载下来。接下来的工作就有点烦琐了，因为 Android Studio 中并没有提供对一个未签名的 APK 直接进行签名的功能，因此我们只能通过最原始的方式，使用 jarsigner 命令来进行签名。

在命令行界面按照以下格式输入签名命令：

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore [keystore文件路径]  
-storepass [keystore文件密码] [待签名APK路径] [keystore文件别名]
```

将[]中的描述替换成 keystore 文件的具体信息就能签名成功了，注意[]符号是不需要的。接着我们将签名后的 APK 文件重新上传就可以了。

APK 上传成功之后，接下来需要选择应用的分类，如图 15.24 所示。

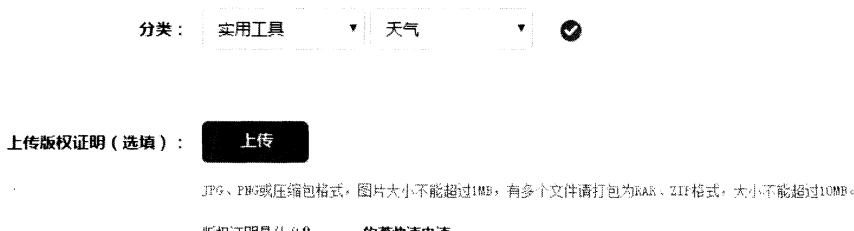


图 15.24 选择应用分类

这里我们将应用分类选择成实用工具→天气。下面还有一个上传版权证明的选项，这是一个选填项，我们直接忽略就可以了。

接着向下滚动网页，设置支持的语言以及资费类型，如图 15.25 所示。

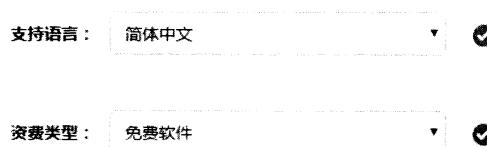


图 15.25 设置支持语言和资费类型

继续滚动网页，下面需要填写应用简介以及当前版本介绍，如图 15.26 所示。

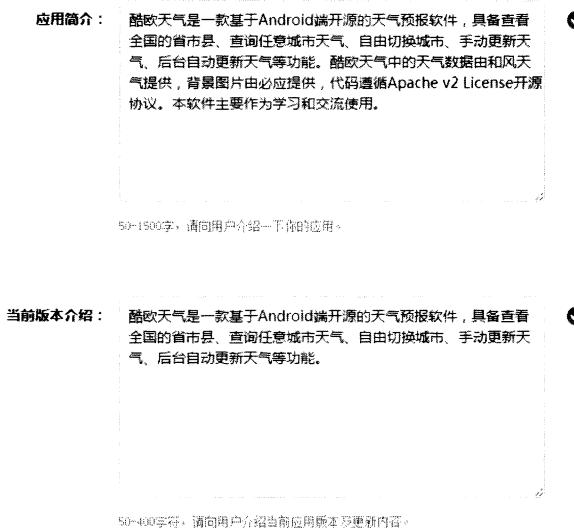


图 15.26 填写应用简介和当前版本介绍

在版本介绍的下面，360 还要求填写一项隐私权限说明，由于酷欧天气只申请了一个网络权限，因此没什么需要说明的，我们直接忽略这一项就可以了。

继续向下滚动网页，接下来需要上传一张高分辨率的应用图标，图标要求是 512×512 像素的 PNG 格式图片，如图 15.27 所示。

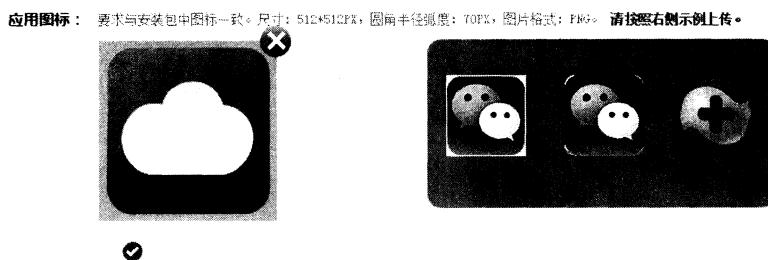


图 15.27 上传高分辨率的应用图标

上传好了图标，我们还需要提供 5 张酷欧天气的屏幕截图，点击上传截图按钮，然后选择准备好的图片即可，如图 15.28 所示。

应用截图：请上传4-5张截图（尺寸保持一致），支持JPG、PNG格式。每张图片要求：不小于800*480(480*800)，单张图片不能超过3M。请去除截图中的顶部通知栏。[查看示例](#)

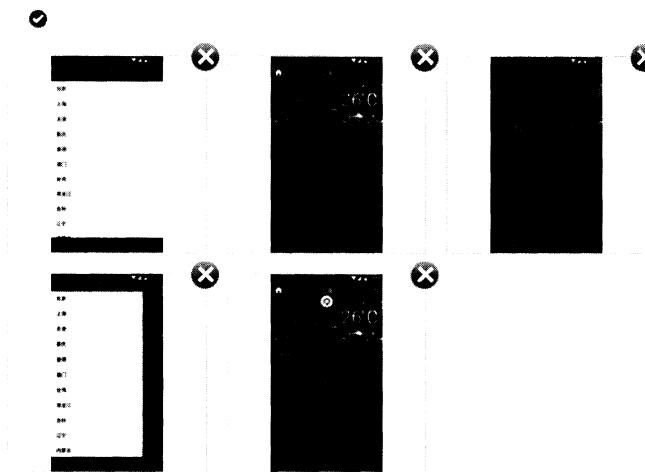


图 15.28 上传屏幕截图

继续向下滚动，还有一个审核辅助说明的选填项，我们也直接忽略就可以了。最后就是一些额外的定制选项，如图 15.29 所示。

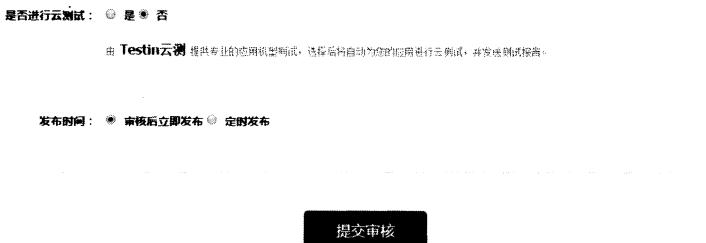


图 15.29 额外的辅助选项

这里我们选择不进行云测试，并在审核后立即发布。

激动人心的时刻终于到了，现在点击一下提交审核按钮就可以将酷欧天气发布到 360 应用商店了，这时会显示如图 15.30 所示的提示。

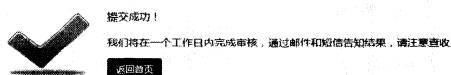


图 15.30 提交成功提示

由于 360 会对我们的应用程序进行审核，接下来又进入了等待当中。不过还好，根据提示来看，这次也许不需要等太久。

果不其然，过了几个小时之后在 360 手机助手上搜索酷欧天气关键字，就可以看到这个应用已经成功上线了，如图 15.31 所示。

点击进去可以查看应用的详情，如图 15.32 所示。



图 15.31 搜索酷欧天气关键字



图 15.32 查看应用详情

到了这里，我们就将应用程序的发布工作全部完成了，之后你应该尽可能地多为你的应用进行宣传，因为用户越多，你能得到的回报就越大。那么如何才能从我们辛辛苦苦编写的程序中得到回报呢？方式有很多种，其中较为常见的做法就是通过广告来进行盈利，因此下一节我们就学习一下，如何在应用程序中嵌入广告。

15.4 嵌入广告进行盈利

谷歌充分考虑到了可以在 Android 应用程序中嵌入广告来让开发者获得收入，因此早早地就收购了 AdMob 公司。AdMob 创立于 2006 年，是全球最早致力于在移动设备上提供广告服务的公司之一，如今成为了谷歌的子公司，AdMob 的广告更加适合在 Android 系统以及 Google Play 上面进行投放。

不过对于国内开发者来说，AdMob 可能就不是那么适合了。因为 AdMob 平台上的广告大多都是英文的，中文广告数有限，并且将 AdMob 账户中的钱提取到银行账户中也比较麻烦，因此这里我们就不准备使用 AdMob 了，而是将眼光放在一些国内的移动广告平台上面。在国内的这一领域，做得比较好的移动广告平台也不少，其中我个人认为腾讯广告联盟（原广点通）特别专业，因此我们就选择它来为酷欧天气提供广告服务吧。

15.4.1 注册腾讯广告联盟账号

下面开始动手，首先第一步我们需要注册一个腾讯广告联盟的账号，注册地址为：<http://e.qq.com>。

com/dev/index.html。

打开该网页，选择使用QQ号登录，然后就会自动跳转到腾讯广告联盟的注册界面，如图15.33所示。图15.33中的所有内容都是必填项，我们按照实际情况来填写就可以了，填写完成之后点击下一步，如图15.34所示。

图 15.33 填写个人信息

图 15.34 填写银行卡信息

由于腾讯广告联盟涉及提现服务，因此我们还需要填写银行卡信息，并上传银行卡照片。填写完成之后继续点击下一步，如图15.35所示。

最后，将你的身份证正反面照片上传，点击提交按钮，就能提交审核了，如图15.36所示。

图 15.35 上传身份证照片

图 15.36 提交审核

只要你前面填写的内容都是真实有效的，审核一般都会很快通过，这里我们只需要耐心等待几个小时就好了。

15.4.2 新建媒体和广告位

审核通过之后，我们就可以进入到腾讯广告联盟的后台，开始给酷欧天气添加广告了。首先需要进入媒体管理界面，点击新建媒体按钮，这时会显示一个页面来让你填写应用的相关信息，我们根据提示一一填好即可，如图 15.37 所示。

系统平台： Android程序 iOS程序

媒体名称： 酷欧天气

关键词： 天气

媒体类别： 生活实用 天气

媒体简介：
酷欧天气是一款基于Android端开发的天气预报软件，具备查看全国的省市县、查询任意城市天气、自由切换城市、手动更新天气、后台自动更新天气等功能

程序主包名： com.coolweather.android

媒体状态： 已上架 未上架

详情页地址： <http://zhushou.360.cn/detail/index/soft/1105585573>

下一步 **取消**

图 15.37 填写应用的相关信息

注意这里需要填写一个详情页地址，也就是酷欧天气在 360 应用商店上的详情页地址。打开 <http://zhushou.360.cn>，在搜索框上输入“酷欧天气”，就能找到该地址了。

填写完成之后点击下一步，接下来需要下载 SDK，如图 15.38 所示。

点击 Android SDK 下载按钮，先把 SDK 下载下来，我们稍后就会进行接入。继续点击下一步，如图 15.39 所示。

媒体名称： 酷欧天气

媒体类型： Android程序

应用ID： 1105585573

媒体名称： 酷欧天气

媒体类型： Android程序

应用ID： 1105585573

应用程序： 上传 输入下载URL
<http://zhushou.360.cn/detail/index/soft/1105585573>

Android SDK 下载

上一步 **下一步** **完成**

图 15.38 下载 SDK

图 15.39 完成媒体创建

这里要求填入一个 APK 的下载的地址，我们直接就填入图 15.37 当中的详情页地址就可以

了。点击完成按钮，现在又会进入到审核等待当中。

为什么新建媒体也需要进行审核呢？这是因为腾讯为了防止某些开发者在垃圾软件上面投放广告，因此要求开发者必须提交应用程序的 APK 文件进行审核，只有审核通过的应用才允许进行广告投放。那么我们只能继续等待。审核通过之后，在媒体管理界面查看新建媒体的状态，如图 15.40 所示。

应用ID	媒体名称	联盟开通状态	业务状态	系统平台	操作
1105585573	酷软天气	已开通	正常	Android	修改 新建广告位

图 15.40 查看新建媒体的状态

可以看到，联盟开通状态显示已开通，业务状态显示正常，说明新建的媒体已经通过审核了。注意这里还自动生成了一个应用 ID，我们稍后就会用到。

现在点击新建广告位，就可以来创建一个广告位了，如图 15.41 所示。

新建广告位

媒体选择： 酷软天气

广告位名称：启动广告

广告位类型： Banner广告 应用墙 插屏广告 开屏广告

敏感行业： 成人用品类
 医疗健康类
 高耗类
 心理健康类
 星座运势类
 P2P网贷平台

如果对于某些类型广告素材中的成人用品过于敏感，您可以进行行挂广告屏蔽

创建 取消

图 15.41 新建广告位

首先要输入广告位的名称，然后选择广告位的类型。腾讯广告联盟支持 Banner、应用墙、插屏和开屏这 4 种广告类型，具体每种广告类型的区别你可以通过查阅文档进行了解，这里我们选择开屏广告。接下来还可以对一些敏感行业的广告进行屏蔽，选择完成之后点击创建按钮完成广告位创建。

现在进入到广告位管理界面，就能查看到我们刚刚新建的广告位了，如图 15.42 所示。

名称&ID	广告位类型	所属应用ID	业务状态	广告位状态	操作
启动广告 4010212448179536	开屏	酷软天气 1105585573	正常	启用中	修改

图 15.42 查看新建广告位

其中，4010212448179536 是广告位 ID，1105585573 是应用 ID。有了这两个数据之后，我们就可以开始接入广告 SDK 了。

15.4.3 接入广告SDK

首先将刚才下载的广告SDK压缩包解压，里面的内容非常简单，如图15.43所示。

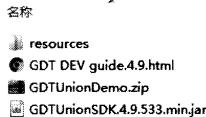


图15.43 广告SDK压缩包中的内容

其中resources文件夹中放的是一些资源图片，我们使用不到。GDT DEV guide.4.9.html是广告SDK的对接文档，GDTUnionDemo.zip是广告SDK的对接示例，GDTUnionSDK4.9.533.min.jar则是广告SDK中最主要的一个Jar包文件了。

由于腾讯广告SDK中的功能还是挺多的，这里不可能面面俱到，将每一个功能都进行详细地讲解。因此我准备只讲解开屏广告这一种类型的广告用法，剩下的其他功能你可以通过阅读文档来进行学习。

我们先将GDTUnionSDK4.9.533.min.jar复制到app/libs目录当中，并点击一下Android Studio顶部工具栏中的Sync按钮完成同步。

接着在AndroidManifest.xml中声明以下权限，其中网络访问权限是之前声明过的，不需要声明两遍。

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_UPDATES" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

注意，其中READ_PHONE_STATE、ACCESS_COARSE_LOCATION和WRITE_EXTERNAL_STORAGE这3个权限是危险权限，因此我们待会还需要进行运行时权限处理。

接下来在<application>标签中添加如下内容：

```
<activity
    android:name="com.qq.e.ads.ADAActivity"
    android:configChanges="keyboard|keyboardHidden|orientation|screenSize" />
<service
    android:name="com.qq.e.comm.DownloadService"
    android:exported="false" />
```

这样就将配置工作完成了。

然后我们还需要创建一个用于显示开屏广告的活动，右击com.coolweather.android包→New→Activity→Empty Activity，创建一个SplashActivity，并将布局名指定成activity_splash.xml。修改activity_splash.xml中的代码，如下所示：

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

</RelativeLayout>
```

这里只有一个空的 RelativeLayout，我们并不需要在 RelativeLayout 当中放入什么内容，但是必须给它定义一个 id。

接着修改 SplashActivity 中的代码，如下所示：

```
public class SplashActivity extends AppCompatActivity {

    private RelativeLayout container;

    /**
     * 用于判断是否可以跳过广告，进入 MainActivity
     */
    private boolean canJump;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_splash);
        container = (RelativeLayout) findViewById(R.id.container);
        // 进行运行时权限处理
        List<String> permissionList = new ArrayList<>();
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_PHONE_
            STATE) != PackageManager.PERMISSION_GRANTED) {
            permissionList.add(Manifest.permission.READ_PHONE_STATE);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_-
            COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            permissionList.add(Manifest.permission.ACCESS_COARSE_LOCATION);
        }
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.WRITE_-
            EXTERNAL_STORAGE) != PackageManager.PERMISSION_GRANTED) {
            permissionList.add(Manifest.permission.WRITE_EXTERNAL_STORAGE);
        }
        if (!permissionList.isEmpty()) {
            String [] permissions = permissionList.toArray(new String[permissionList.
                size()]);
            ActivityCompat.requestPermissions(this, permissions, 1);
        } else {
            requestAds();
        }
    }

    /**
     * 请求开屏广告
     */
    private void requestAds() {
        String appId = "1105585573";
```

```
String adId = "4010212448179536";
new SplashAD(this, container, appId, adId, new SplashADListener() {
    @Override
    public void onADDismissed() {
        // 广告显示完毕
        forward();
    }

    @Override
    public void onNoAD(int i) {
        // 广告加载失败
        forward();
    }

    @Override
    public void onADPresent() {
        // 广告加载成功
    }

    @Override
    public void onADClicked() {
        // 广告被点击
    }
});
}

@Override
protected void onPause() {
    super.onPause();
    canJump = false;
}

@Override
protected void onResume() {
    super.onResume();
    if (canJump) {
        forward();
    }
    canJump = true;
}

private void forward() {
    if (canJump) {
        // 跳转到MainActivity
        Intent intent = new Intent(this, MainActivity.class);
        startActivity(intent);
        finish();
    } else {
        canJump = true;
    }
}

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

```

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

```

可以看到，在`onCreate()`方法中，我们先是获取到了`RelativeLayout`的实例，紧接着就开始进行运行时权限处理。由于这里也是需要在运行时一次性申请多个权限，因此采用了和11.3.2小节同样的写法，相信你一定不会陌生吧。

当用户同意了所有的权限申请之后，就会调用`requestAds()`方法来请求广告数据。在`requestAds()`方法中我们先是定义了`appId`和`adId`这两个变量，它们的值就是在腾讯广告联盟后台生成的应用ID和广告位ID，然后创建`SplashAD`的实例来获取广告数据。`SplashAD`的构造函数接收5个参数，第1个参数是当前活动的实例，第2个参数是`RelativeLayout`的实例，第3个参数是应用ID，第4个参数是广告位ID，相信前4个参数都没什么需要解释的。第5个参数是一个`SplashADListener`的实例，用于监听广告数据的回调。其中`onADDismissed()`方法会在广告显示完毕时回调，`onNoAD()`方法会在广告加载失败时回调，`onADPresent()`方法会在广告加载成功时回调，`onADClicked()`方法会在广告被点击时回调。当广告显示完毕或者广告加载失败时，我们调用`forward()`方法跳转到`MainActivity`，并将当前活动关闭即可。

另外注意这里还使用了一个`canJump`变量用于对活动跳转进行控制。这是因为如果用户点击了广告，会启动一个新的活动来展示广告的详细内容，这个时候即使回调了`onADDismissed()`方法，显然也不应该启动`MainActivity`，因此我们在`onPause()`方法中将`canJump`设置成了`false`。然后在`forward()`方法中发现`canJump`是`false`，因此不会进行跳转，但是会将`canJump`设置成`true`。最后，当用户看完了广告回到`SplashActivity`时，`onResume()`方法将会执行，这个时候发现`canJump`是`true`，因此就会调用`forward()`方法来启动`MainActivity`。

整体流程大概就是这个样子了，接下来我们还需要将主活动设置成`SplashActivity`而不再是`MainActivity`，否则广告界面将无法得到展示。修改`AndroidManifest.xml`中的代码，如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.coolweather.android">
    ...
    <application
        android:name="org.litepal.LitePalApplication"
        android:allowBackup="true"
        android:icon="@mipmap/logo"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
        </activity>
        <activity android:name=".SplashActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        ...
    </application>
</manifest>
```

这样我们就将广告 SDK 全部对接完成了，现在可以重新运行一下程序来看一看效果。不过需要注意，广告在模拟器上是不会显示的，我们要用真正的手机测试才行。程序启动后首先会弹出运行时权限的申请对话框，全部都点击允许之后就能看到广告数据了，如图 15.44 所示。



图 15.44 显示广告数据

开屏广告会持续 5 秒钟时间，然后就会自动跳转到 MainActivity 中，后面的流程就和之前是完全一样的了。

15.4.4 重新发布应用程序

现在我们已经成功在酷欧天气中接入了广告功能，那么是时候将这个新版本更新到 360 应用商店上了。

由于即将发布的会是新版的酷欧天气，因此在生成安装包之前还需要修改一下应用程序的版本号信息。编辑 `app/build.gradle` 文件，如下所示：

```
android {
    compileSdkVersion 24
    buildToolsVersion "24.0.2"
    defaultConfig {
        applicationId "com.coolweather.android"
        minSdkVersion 15
        targetSdkVersion 24
        versionCode 2
        versionName "1.1"
    }
    ...
}
```

可以看到，这里将 `versionCode` 改成了 2，`versionName` 改成了 1.1。需要注意的是，每个版本的 `versionCode` 和 `versionName` 都不能和其他版本相同，且新版应用的版本号必须大于老版应用的版本号。

接下来我们就可以使用在 15.1 节学习的技术来生成新的 APK 文件，具体的步骤就不再重复介绍了，最终会生成一个新的 `app-release.apk` 文件，下面我们将它重新发布到 360 应用商店。

打开 360 开发者后台的管理中心页面，然后点击酷欧天气的更新管理按钮，如图 15.45 所示。



图 15.45 更新酷欧天气

点击编辑与更新按钮，就能上传新的 APK 文件了。注意上传之后仍然会提醒应用的安全系数较低，我们只需要使用和之前同样的方式进行加固就可以了。

另外，由于新版的酷欧天气中增加了一些敏感隐私权限，因此我们还需要在这一项上面做出说明，如图 15.46 所示。

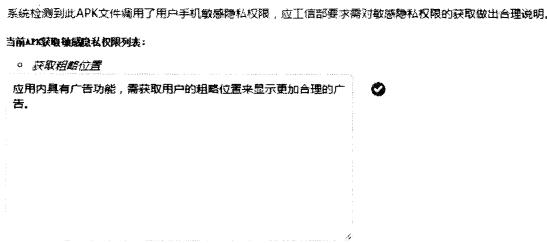


图 15.46 对敏感隐私权限进行说明

现在只需要点击页面最下方的提交审核按钮，新版本的酷欧天气就发布成功了，当然还需要通过360的审核才行。以后每当有用户观看或点击了应用程序中的广告时，我们就能真正地得到收益。在腾讯广告联盟的后台管理界面可以查看到每天的收益情况，如图15.47所示。



图 15.47 查看广告收益情况

你的应用越成功，所获得的广告收益也会越多，因此赶快去编写更多优秀应用程序来赚更多的钱吧，相信通过整本书的学习，你已经有足够的能力做到了！

15.5 结束语

就这样，本书所有的内容你都学完了，现在你已经成功毕业，并且成为了一名合格的Android开发者。但是如果想要成为一名出色的Android开发者，光靠本书中的这些理论知识以及少量的实践还是不够的，你需要真正步入到工作岗位当中，通过更多的项目实战来不断地历练和提升自己。

唠叨了整本书的话，但是到了最后却不知道该说点什么好，我不想说我能教你的就只有这些了，因为实际上我想教你或者和你一起探讨的内容还有很多很多，不过限于篇幅的原因，本书的内容就只能到此为止了。但我会长期在博客和微信公众号上面分享更多Android相关的技术文章，如果感兴趣的话，可以到我的博客和公众号中继续学习。当然，如果对本书中的内容有疑问，也可以到博客或公众号中给我留言，我的博客地址如下：

<http://guolin.tech>

扫一扫下方二维码即可关注我的公众号：



好了，就到这里吧，祝愿你未来的Android之旅都能愉快。