

```

32     Bitmap internet_bitmap = (Bitmap)msg.obj;
33     imageView.setImageBitmap(internet_bitmap);
34   }
35 }
36 }
37
38 //使用线程下载图片数据，之后通过handle更新
39 public void showImageByThread(ImageView imageView,final String imageUrl,
40     final LruDiskCacheManger lruDiskCacheManger) {
41     this.imageView = imageView;
42     this.imageViewTag = (String) imageView.getTag();
43
44     new Thread()
45     {
46         @Override
47         public void run() {
48             // TODO Auto-generated method stub
49             super.run();
50             Bitmap bitmap = getBitmapByUrl(imageUrl);
51             Message message = Message.obtain();
52             message.obj = bitmap;
53             handler.sendMessage(message);
54
55             lruDiskCacheManger.writeBitmap2Cache(imageUrl, bitmap);
56         }
57     }.start();
58 }
59
60 //通过HttpURLConnection下载图片并显示
61 private Bitmap getBitmapByUrl(String url_bitmap)
62 {
63     Bitmap bitmap = null;
64     InputStream is = null;

```

图 11B-28 修改 showImageByThread 方法,加入写入图片缓存



图 11B-29 在 XML 中添加菜单项

```

182     @Override
183     public boolean onOptionsItemSelected(MenuItem item) {
184         // TODO Auto-generated method stub
185         switch (item.getItemId()) {
186             case R.id.action_watchCacheSize:
187                 showCacheSizeDialog();
188                 break;
189             }
190             default:
191                 return super.onOptionsItemSelected(item);
192             }
193             return true;
194         }
195
196     private void showCacheSizeDialog() {
197         //获取缓存大小，保留两位小数
198         double cacheSize = new BigDecimal(
199             (double)lruDiskCacheManger.getCacheSize() / 1024.0 / 1024.0).setScale(2,
200             BigDecimal.ROUND_HALF_UP).doubleValue();
201
202         String cacheSize_str = "当前缓存大小：" + cacheSize + "M";
203         AlertDialog.Builder builder = new AlertDialog.Builder(this);
204         builder.setTitle("提示");
205         builder.setMessage(cacheSize_str);
206         builder.create().show();
207     }
208

```

图 11B-30 操纵菜单项

B.3.4 删除所有缓存

同样地,把删除所有缓存功能的入口放在菜单项中,代码如图 11B-31、图 11B-32 所示。

如图 11B-32 所示,当用户单击菜单并选择“删除所有缓存”项时,将会调用 showDeleteAllCacheDialog 方法,该方法将会显示对话框供用户是否选择删除缓存,如果选择

确定，则会调用 LruDiskCacheManger 的 deleteBitmapCacheAll 方法删除所有缓存。

B.3.5 效果图

运行程序，可看到效果图如图 11B-33 所示。当快速滑动 ListView 时，也不会出现显示默认图片的情况。当单击右上角的菜单时并选择查看“缓存大小”时，将显示提示对话框，如图 11B-34 所示。



图 11B-31 在 XML 中添加菜单项

```
>MainActivity.java ② LruDiskCacheManger
183  @Override
184  public boolean onOptionsItemSelected(MenuItem item) {
185      // TODO Auto-generated method stub
186      switch (item.getItemId()) {
187          case R.id.action_watchCacheSize: {
188              showCacheSizeDialog();
189              break;
190          }
191          case R.id.action_deleteAllCache: {
192              showDeleteAllCacheDialog();
193              break;
194          }
195          default:
196              return super.onOptionsItemSelected(item);
197      }
198      return true;
199  }
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
private void showDeleteAllCacheDialog()
{
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle("提示");
    builder.setMessage("真的删除所有缓存 ?");
    builder.setPositiveButton("确定", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface arg0, int arg1) {
            // TODO Auto-generated method stub
            lruDiskCacheManger.deleteBitmapCacheAll();
            Toast.makeText(MainActivity.this, "删除成功", Toast.LENGTH_SHORT);
        }
    });
    builder.setNegativeButton("取消", null);
    builder.create().show();
}
```

图 11B-32 响应菜单项

当然，选择“删除所有缓存”时，如图 11B-35 所示，如单击确认，所有缓存将会被删除（等于第一次使用），此时快速滑动 ListView，将再次出现默认图的情况（需要从网络下载图片）。



图 11B-33 效果图



图 11B-34 缓存大小提示对话框



图 11B-35 显示是否删除缓存的提示

B.4 项目心得

引用电影的一句台词：“念念不忘，必有回响。”任何技术或者产品，就是同一个功能，如本例实现缓存部分，反复琢磨，找寻业界大牛的解决方案(github 或者 git 上)总有值得优化，提高，学习的地方。

B.5 相关资料

Android DiskLruCache 完全解析，硬盘缓存的最佳方案：

http://blog.csdn.net/guolin_blog/article/details/28863651

C Android 函数响应式链编程 RxJava

搜索关键字

- (1) RxJava
- (2) RxAndroid
- (3) 观察者模式

本章难点

RxJava 在 Android 开发者中变得越来越流行,它的魅力在于:在异步处理中,无论是多么复杂的逻辑,使用 RxJava 都可以让代码保持足够的简洁。本章将会讲解 RxJava 的使用,并完成以下任务:

- (1) 使用 RxJava 在控制台输出一组字符串。
- (2) 使用 RxJava 优化一个图片下载的 Android 案例。

第一个任务让大家了解使用 RxJava 的步骤,以及学习几个重要的操作符。第二个任务会使用 RxJava 对一个 Android 案例进行优化,让大家直观地感受到 RxJava 的处理异步时的便利性。

C.1 简介

C.1.1 RxJava 到底是什么

一个词:异步。

RxJava 在 GitHub 上的自我介绍是“a library for composing a synchronous and event-based programs using observable sequences for the Java VM”(一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库)。这就是 RxJava 的精准概括。

也许对于初学者来说,这段话还是很难看得懂。其实, RxJava 的本质是一个实现异步操作的库。

读者可能有疑问:Android 中, AsyncTask、Handler 同样是异步,那为何要用 RxJava 呢? RxJava 的优势在哪里? 这个问题的答案也可以总结为两个字:简洁。在 Android 中, AsyncTask、Handler 的出现也是为了让异步的过程更加简洁,而 RxJava 的优势在于,随着程序业务逻辑越来越复杂,它同样可以保持简洁。

那 RxJava 是否可以完全替代 Handler 呢? 个人认为是不可以。Handler 除了和 Thread 配合实现异步更新 UI 外,还可以实现执行计划任务、线程间通信等功能,实现的功能可以更加精细,颗粒度更小,而 RxJava 更适合专门处理异步更新 UI 这一功能。

因此,如果读者需要实现异步加载数据,然后在 UI 线程中刷新界面这一类的需要,使用 RxJava 比 Handler 和 AsyncTask 更加合适、方便。

C.1.2 RxJava 的实现原理——观察者模式

1. 观察者模式介绍

RxJava 实现异步原理,本质上是一种扩展的观察者模式。那么问题来了,什么是观察者模式呢?

举个通俗易懂的例子,如“监控抓小偷”,监控需要在小偷伸手作案的时候实施抓捕。在这个例子里,监控是观察者,小偷是被观察者,监控需要时刻盯着小偷的一举一动,才能保证不会漏过任何瞬间。当然,程序设计里的观察者模式和这种生活中真正的观察是有所不同的:观察者不需要时时刻刻盯着被观察者(例如 A 不需要每 2ms 就检查一次 B 的状态),而是采用注册(Register)或者订阅(Subscribe)的方式,告诉被观察者:我要你的状态变化,你要在它变化的时候通知我。

类比“监控抓小偷”的例子,程序中的观察者模式就像:监控事先告诉警察,小偷在作案的时候,就会向警察汇报,然后便采取行动。在程序设计中,就是这样的流程。

在 Android 开发中,一个经典的例子就是单击事件监听器 OnClickListener。对设置 OnClickListener 来说,View 是被观察者,OnClickListener 是观察者,两者通过 setOnClickListener() 方法达成订阅关系。订阅之后用户单击按钮的瞬间,Android Framework 就会将单击事件发送给已经注册的 OnClickListener。采取这样被动的观察方式,既省去了反复检索状态的资源消耗,也能够得到最高的反馈速度。

OnClickListener 的模式大致如图 11C-1 所示。

另外,如果把这张图中的概念抽象出来(Button→被观察者、OnClickListener→观察者、setOnClickListener()→订阅,onClick()→事件),就由专用的观察者模式(例如只用于监听控件单击)转变成了通用的观察者模式,如图 11C-2 所示。



图 11C-1 OnClickListener 的观察者模式



图 11C-2 通用的观察者模式

而 RxJava 作为一个工具库,使用的就是通用形式的观察者模式。

【知识点】观察者模式,是设计模式的一种,通常被称为订阅—发布模式。采取观察者模式,既省去了反复检索状态的资源消耗,也能够得到最高的反馈速度。

2. RxJava 中的观察者模式

了解 RxJava 的观察者模式,这一步十分重要,希望大家能够仔细阅读,否则可能影响理解后面案例的实现。

RxJava 有四个基本概念:Observable(可观察者,即被观察者,即前文说的小偷)、Observer(观察者,即前文说的监控)、subscribe(订阅,小偷被监控抓住了!)、事件(小偷作案的事件)。

被观察者 Observable 和观察者 Observer 通过 subscribe() 方法实现订阅关系,从而观察者 Observable 可以在需要的时候发出事件来通知被观察者 Observer。与传统观察者模式不同,RxJava 的事件回调方法除了普通事件 onNext()(小偷每一件的作案事件)之外,还定义了两个特殊的事件:onCompleted() 和 onError()。

onCompleted():事件队列完结(小偷作案完成事件)。RxJava 不仅把每个事件单独处理,还会把它们看作一个队列。RxJava 规定,当不会再有新的 onNext() 发出时,需要触发

onCompleted() 方法作为标志。

onError(): 事件队列异常(小偷作案失败事件)。在事件处理过程中出异常时, onError() 会被触发, 同时队列自动终止, 不允许再有事件发出。



图 11C-3 RxJava 的观察者模式

在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个, 并且是事件序列中的最后一个。因为 onCompleted() 和 onError() 两者也是互斥的, 即在队列中调用了其中一个, 就不应该再调用另一个。

RxJava 的观察者模式大致如图 11C-3 所示。

C.2 案例实现

C.2.1 RxJava 案例

在本例中, 实现一个很简单的例子: 使用 RxJava 在控制台中依次打印一组字符串。让大家对 RxJava 的使用流程及重要的操作符有个初步认识。

我们使用的平台是 Android Studio。首先创建一个 Android 项目, 名为 RxJavaDemo, 并在 gradle 文件中引入 RxJava 和 RxAndroid 的依赖, 如图 11C-4 所示。

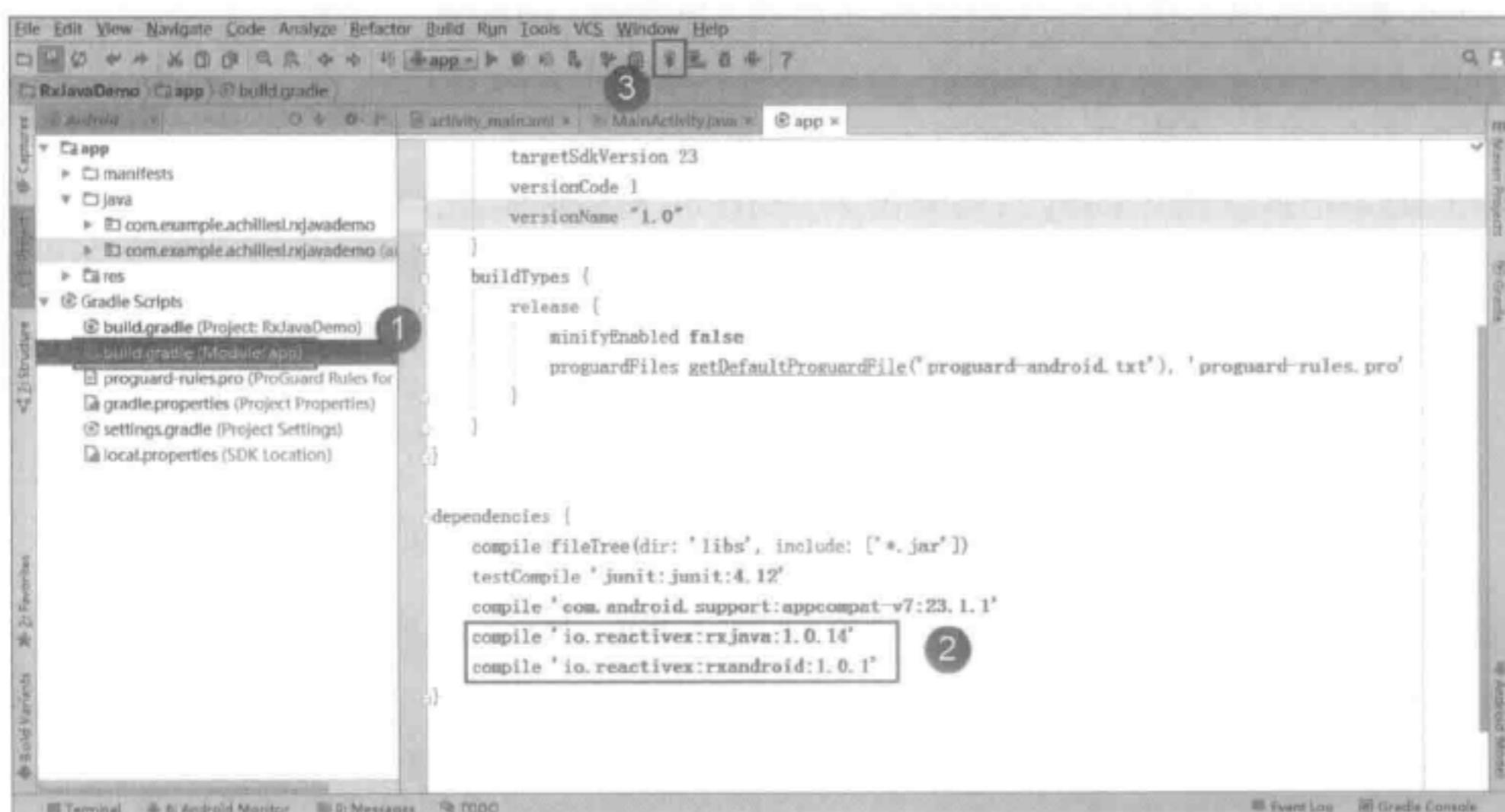


图 11C-4 创建项目 RxJavaDemo, 并添加依赖

添加依赖后, 单击如方框 3 所示的按钮, 更新 gradle 文件, 完成后就可以在项目中使用 RxJava 及 RxAndroid。

【注意】 本章全部案例均使用 Android Studio 作为开发平台, 对于使用 eclipse 的读者, 可以自行到 maven 官网上下载相关 jar 包, 导入后即可使用 RxJava 及 RxAndroid。

(1) RxJava:

<https://repo1.maven.org/maven2/io/reactive/rxjava/1.1.0/rxjava-1.1.0.jar>

(2) RxAndroid:

<http://search.maven.org/#search|ga|1|rxandroid>

1. 创建被观察者 Observable

Observable 即被观察者, 它决定什么时候触发事件以及触发怎样的事件。通俗地讲, 这里

的 Observable 相当于 1.2 节中“RxJava 的实现原理——观察者模式”中的小偷。因此，要使用 RxJava，先在程序中创建一个被观察者(小偷)。

RxJava 使用 create() 方法来创建一个 Observable，并重写 call 方法为它定义事件触发规则。代码如图 11C-5 所示。

如图 11C-5 方框所示，在 Observable.create 方法中传入了一个 OnSubscribe 对象作为参数，这个 OnSubscribe 对象，可以理解成一个计划表(小偷作案的记录表)。当 Observable 被订阅的时候(小偷被监控抓住了!)，OnSubscribe 的 call() 方法会自动被调用(向监控汇报作案过程)，事件序列就会依照设定依次触发(被抓后，小偷向监控汇报作案内容：我偷了一个 Hello 字符串、一个 world 字符串、一个 Hi 字符串、一个 You are my World too! 字符串、已经全部偷完了)。

一般来讲，我们会调用 onNext 方法来传递事件，当所有事件传输完成时，就调用 onComplete 方法。此外，还有一个 onError 方法，该方法在事件传输中断时调用。

这样，由被观察者(小偷)调用了观察者(监控)的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

2. 创建观察者 Observer

Observer 即观察者，它决定事件触发的时候将有怎样的行为。通俗来讲，下面准备实现的这个 Observer，即是前文案例中的监控，它会根据小偷的作案事件做出不同的响应。代码如图 11C-6 所示。

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

        Observable observable = Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<String> subscriber) {
                subscriber.onNext("Hello");
                subscriber.onNext("World");
                subscriber.onNext("Hi");
                subscriber.onNext("You are my World too!");
                subscriber.onCompleted();
            }
        });
    }
}
```

图 11C-5 创建 Observable

```
ObserverString observer = new ObserverString() {
    @Override
    public void onComplete() {
        Log.d("TAG", "onCompleted");
    }

    @Override
    public void onError(Throwable e) {
        Log.d("TAG", "onError");
    }

    @Override
    public void onNext(String s) {
        Log.d("TAG", "Item is :" + s);
    }
};
```

图 11C-6 创建观察者 Observer

下面需要重写了观察者 Observer 的 onComplete、onError、onNext 方法。这些方法会在事件传输完成时(小偷偷完时)、事件传输中断时(小偷失败时)、事件传输时(小偷正在作案时)得到回调，对应于被观察者在 call 方法中调用 onComplete、onError、onNext 等方法。这些回调方法的详细作用，可见上文中阐述的“RxJava 中的观察者模式”。

在图 11C-6 中，当事件传输完成时，将打印 onComplete 字符串，当传输事件失败时，将打印 onError 字符串，当传输有效的字符串时，就直接打印。

实现观察者，除了 Observer 接口之外，RxJava 还内置了另一个抽象类：Subscriber。Subscriber 对 Observer 接口进行了一些扩展，但它们的基本使用方式是完全一样的，如图 11C-7 所示。

Subscribe 接口相对于 Observer 接口,增加了两个方法:

`onStart()`方法：这是 `Subscriber` 增加的方法。它会在 `subscribe` 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。

`unsubscribe()`方法：这是也 `Subscriber` 增加的方法，用于取消订阅。在这个方法被调用后，`Subscriber` 将不再接收事件。

3. 使用 Subscribe 链接观察者与被观察者

创建了被观察者 Observable 和观察者 Observer 之后,需要用 subscribe()方法将它们联结起来,此时整条响应链(RxJava 基于链式编程)就可以工作了(相当于监控抓住了小偷,小偷准备向监控汇报)。代码很简单,如图 11C-8 方框中所示。

```
47
48     Subscriber subscriber = new Subscriber<String>() {
49         @Override
50         public void onCompleted() {
51             Log.d("TAG", "onCompleted");
52         }
53         @Override
54         public void onError(Throwable e) {
55             Log.d("TAG", "onError");
56         }
57         @Override
58         public void onNext(String s) {
59             Log.d("TAG", "Item is :" + s);
60         }
61         @Override
62         public void onStart() {
63             super.onStart();
64             Log.d("TAG", "onStart");
65         }
66     };
67 }
```

图 11C-7 Subscriber 用法展示

```
1 package com.example.android.miwok;
2
3 import android.os.Bundle;
4 import androidx.appcompat.app.AppCompatActivity;
5 import androidx.recyclerview.widget.RecyclerView;
6 import androidx.recyclerview.widget.LinearLayoutManager;
7 import androidx.recyclerview.widget.RecyclerView.Adapter;
8
9 import java.util.ArrayList;
10
11 public class MainActivity extends AppCompatActivity {
12
13     private RecyclerView miwokRecyclerView;
14     private MiwokAdapter miwokAdapter;
15
16     @Override
17     protected void onCreate(Bundle savedInstanceState) {
18         super.onCreate(savedInstanceState);
19         setContentView(R.layout.activity_main);
20
21         miwokRecyclerView = findViewById(R.id.miWokRecyclerView);
22
23         ArrayList<Word> wordList = new ArrayList<Word>();
24
25         wordList.add(new Word("WATER", "H2O", R.drawable.ic_water));
26         wordList.add(new Word("MOUNTAIN", "MOUNTAIN", R.drawable.ic_mountain));
27         wordList.add(new Word("SUN", "SUN", R.drawable.ic_sun));
28         wordList.add(new Word("MOON", "MOON", R.drawable.ic_moon));
29         wordList.add(new Word("FIRE", "FIRE", R.drawable.ic_fire));
30
31         miwokAdapter = new MiwokAdapter(wordList);
32
33         miwokRecyclerView.setLayoutManager(new LinearLayoutManager(this));
34         miwokRecyclerView.setAdapter(miwokAdapter);
35
36         miwokAdapter.setOnItemClickListener(new MiwokAdapter.OnItemClickListener() {
37             @Override
38             public void onError(Throwable e) {
39                 Log.d("TAG", "onError");
40             }
41
42             @Override
43             public void onNext(String s) {
44                 Log.d("TAG", "Item is :" + s);
45             }
46         });
47     }
48
49 }
```

图 11C-8 使用 subscribe 方法链接 observable 和 observer

读者可能会注意到,subscribe() 这个方法有点怪:它看起来是:被观察者订阅了观察者(即小偷主动向监控自首了,而不是监控抓住了小偷)。这让人读起来有点别扭,不过如果把 API 设计成 observer.subscribe(observable),虽然更加符合思维逻辑,但对流式 API 的设计就造成影响了(读者可以带着这个问题,在 C.2.2 节“Android 中实战”中寻找答案。在那一节中可以看到:获取图片 URL→根据 URL 得到图片→过滤→显示图片这个流程是在被观察者.subscribe(观察者)模式下实现的,如果 API 改成观察者.subscribe(被观察者)模式,上述的流程将会颠倒,代码可读性会更加差),比较起来,开销变大,可读性变差,明显是得不偿失的。此时,编译运行程序,可以看到在控制台中,依次打印了字符串,效果如图 11C-9 所示。

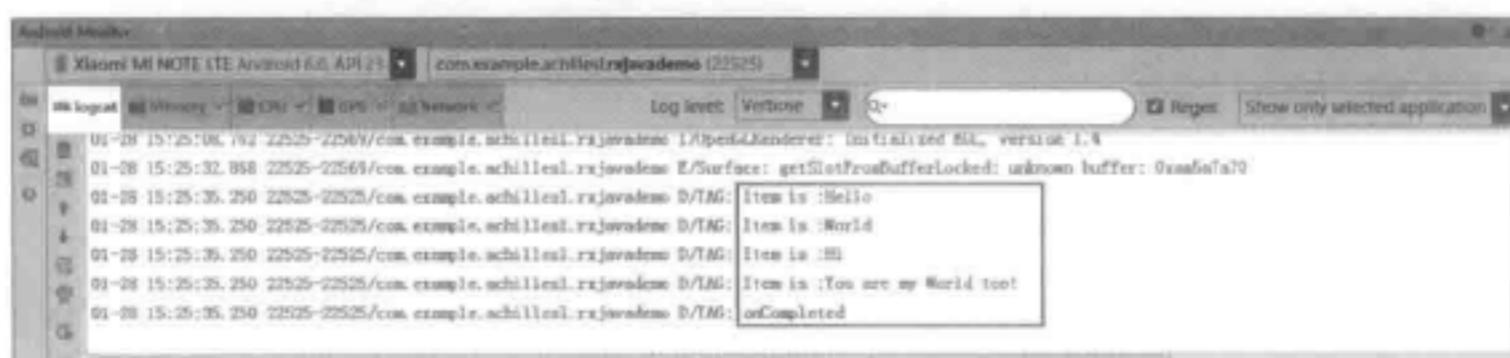


图 11C-9 在控制台中，依次打印字符串

4. 订阅特定的事件回调

上一节中,已经学习了 subscribe 方法的用法,subscribe 方法可以将观察者和被观察者链接起来。当被观察者 observable 开始事件流传输的,观察者可以通过 onCompleted、onError、onNext 方法对被观察者传输的事件进行监听处理。

于是问题来了，作为开发者可能只对某个事件回调感兴趣，如仅想监听 `onNext` 事件（好

比监控只想听小偷的作案具体事件,不想了解作案失败和作案结束事件),此时怎么办呢?

实际上,subscribe方法还可以支持不完整定义的回调,形式如图 11C-10 所示。

The screenshot shows an Android Studio code editor with the file `MainActivity.java` open. The code defines three Action1<String> objects: `onNextAction`, `onErrorAction`, and `onCompletedAction`. Each has an overridden `call` method that logs a message. Below these, there are three comments explaining how to use them with the `subscribe` method:

```
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66 // 自动创建 Subscriber，并使用 onNextAction 来定义 onNext()
67 observable.subscribe(onNextAction);
68 // 自动创建 Subscriber，并使用 onNextAction 和 onErrorAction 来定义 onNext() 和 onError()
69 observable.subscribe(onNextAction, onErrorAction);
70 // 自动创建 Subscriber，并使用 onNextAction、onErrorAction 和 onCompletedAction 来定义 onNext()、onError() 和 onCompleted()
71 observable.subscribe(onNextAction, onErrorAction, onCompletedAction);
```

图 11C-10 在 subscribe 方法中定义不完全的回调

简单解释一下这段代码中出现的 Action1 和 Action0。

Action0 是 RxJava 的一个接口,它只有一个 `call()`方法,这个方法是无参数无返回值的。由于观察者的 `onCompleted()` 方法也是无参数无返回值的,因此 Action0 可以被当成一个包装对象,将 `onCompleted()` 的内容打包起来将自己作为一个参数传入 `subscribe()` 以实现不完整定义的回调。

图 11C-10 中,定义的 `onCompletedAction` 对象实现了 Action0 接口,图中虽然在第 71 行才将 `onCompletedAction` 对象传入。实际上,也可以像第 67 行代码那样,单独传入 `onCompletedAction` 对象对 `onCompleted` 事件单独进行监听。

Action1 也是一个接口,它同样只有一个方法 `call(T param)`,这个方法也无返回值,但有一个参数。与 Action0 同理,由于观察者的 `onNext(T obj)` 和 `onError(Throwable error)` 方法也是单参数无返回值的,因此 Action1 可以将 `onNext(obj)` 和 `onError(error)` 打包起来传入 `subscribe()` 以实现不完整定义的回调。

图 11C-10 中,定义的 `onNextAction` 和 `onErrorAction` 对象,都是实现了 Action1 接口,在第 67 和第 69 行代码中,可以将 `onNextAction` 和 `onErrorAction` 单独传入 `subscribe` 方法,此时可以实现业务只需关注的 `onNext` 或 `onError` 事件即可。

这样子,在实现观察者的时候,只需要实现感兴趣的事件回调即可,不必为每个事件重写回调方法。

5. filter 操作符

单词 filter,翻译成中文是过滤意思,在 RxJava 中 filter 操作符可以根据需求过滤掉不符合条件的事件。

比如在上一节的例子中,需要加入一个功能:在控制台中只打印长度大于 3 的字符串(好比小偷想,我要把字符串长度小于 3 的扔掉)。此时,用 filter 操作符可以很轻松地实现这个功能,代码如图 11C-11 所示。

在图 11C-11 中,可以看到下面给 filter 方法传入了一个匿名对象,该对象实现了 Func1 的接口。实际上,Func1 接口类似于前文的 Action0 和 Action1 接口。Func1 接口有两个泛型参数,图中第一个参数是 String,表示事件的类型是 String 类型,第二个参数类型为布尔型,表

```
40
41     @Override
42     public void onNext(String s) {
43         Log.d("TAG", "Item is :" + s);
44     }
45 }
46
47 observable.filter(new Func1<String, Boolean>() {
48     @Override
49     public Boolean call(String s) {
50         return s.length() > 3;
51     }
52 }
53 .subscribe(observer);
54 }
55
56 }
```

图 11C-11 filter 的使用

示 call 方法返回值的类型。

使用该接口需要实现 call 方法,当 call 方法返回 true 时,表示该事件满足条件不必过滤,否则将该事件过滤,不再往下传递。可以看到第 50 行的代码中,当字符串长度大于 3 时,则不过滤,否则过滤该事件。

此时,编译运行程序,看到控制台中打印了长度大于 3 的字符串,而字符串 Hi 则被过滤掉了,因此没有显示,如图 11C-12 所示。

```
01-28 19:38:22.276 13869-13869/com.example.achilles1.rxjavademo D/TAG: Item is :Hello
01-28 19:38:22.276 13869-13869/com.example.achilles1.rxjavademo D/TAG: Item is :World
01-28 19:38:22.276 13869-13869/com.example.achilles1.rxjavademo D/TAG: Item is :You are my World too!
01-28 19:38:22.276 13869-13869/com.example.achilles1.rxjavademo D/TAG: onCompleted
```

图 11C-12 控制台输出信息

6. map 操作符

RxJava 提供了对事件序列进行变换的支持,这是它的核心功能之一。所谓变换,就是将事件序列中的对象或整个序列进行加工处理,转换成不同的事件或事件序列(好比小偷偷了赃物,然后把它变现成钱,于是赃物的类型从“物体”转变成“货币”)。

上一个在例子中,将需求变成:将长度大于 3 的字符串转成对应的 hashCode,然后在控制台输出。此时,map 操作符可以派上用场了,代码如图 11C-13 所示。

如图 11C-13 方框所示,可以看到 map()方法将参数中的 String 类型的对象转换成一个 Integer 类型的对象(hashCode 是 Integer 类型)后再作为返回值,这样经过 map()方法变换后,事件的参数类型也由 String 型转化为了 Integer 型。

对于 map 操作符,可以理解成:需要将序列中事件的类型,转换成另一种类型,至于是何种类型转成何种类型、具体怎么转变,这些是需要开发者去实现的。

编译运行程序,效果如图 11C-14 所示。

```
40
41
42
43
44
45
46
47 observable.filter(new Func1<String, Boolean>() {
48     @Override
49     public Boolean call(String s) {
50         return s.length() > 3;
51     }
52 }
53 .map(new Func1<String, Integer>() {
54     @Override
55     public Integer call(String s) {
56         return s.hashCode();
57     }
58 }
59 )
60 .subscribe(new Action1<Integer>() {
61     @Override
62     public void call(Integer integer) {
63         Log.d("TAG", "Item hashCode is :" + integer);
64     }
65 }
66 );
67 }
```

图 11C-13 map 操作符的运用

```
01-28 20:23:40.697 22521-22521/com.example.achilles1.rxjavademo D/TAG: Item hashCode is :89609650
01-28 20:23:40.697 22521-22521/com.example.achilles1.rxjavademo D/TAG: Item hashCode is :83766130
01-28 20:23:40.697 22521-22521/com.example.achilles1.rxjavademo D/TAG: Item hashCode is :1879971202
```

图 11C-14 输出字符串的 hashCode

C. 2.2 Android 中实战

“RxJava 的优势在于，随着程序业务逻辑越来越复杂，它同样可以保持代码简洁易读。”在这个案例中，下面将对这句话进行验证。

在上一节中，已经学习了 RxJava 的基础用法。在这一小节中，将使用 RxJava 对一个 Android 小案例进行优化。该案例实现的功能十分简单：

根据图片的 URL 来下载图片，并展示到 ImageView 上。其中图片 URL 也需要从网络下载（本例用线程休眠来模拟图片 URL 的下载），如果得到的 URL 是合法的，则下载对应的图片，否则就显示默认图。同时，单击 CHANGEPIC 按钮可以切换图片，并且在图片下载的过程中不可再次单击，效果如图 11C-15、图 11C-16 所示。

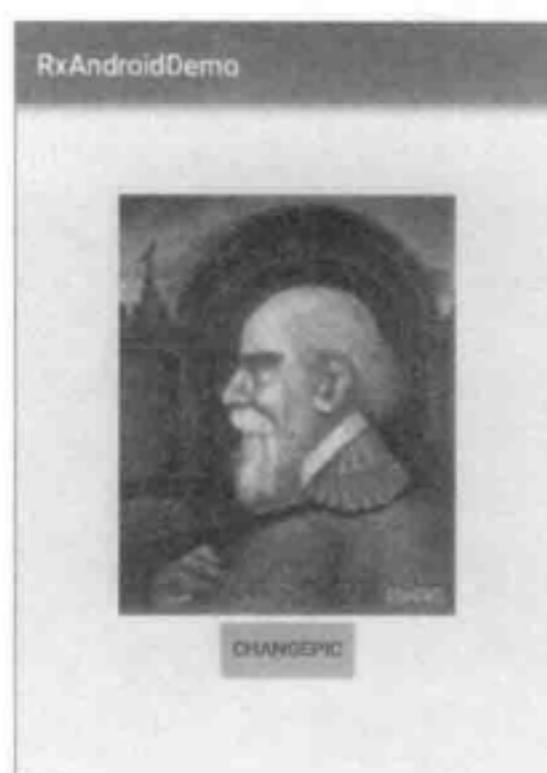


图 11C-15 显示网络图片



图 11C-16 若图片 URL 非法，显示默认图片

希望通过这个案例，可以让大家直观地看到，实现异步处理时，在代码的简洁与可读性层面上，使用 RxJava 方式要远远优于传统的 Thread+Handle 的方式。

1. 案例的关键代码

代码如图 11C-17 所示，在 MainActivity 类的方框 1 中，定义了一个字符串数据，包含了一组图片的 URL，其中最后一个 URL 内容为 error，用于模拟图片 URL 为非法的情况。

The screenshot shows the Android Studio code editor with the file "MainActivity.java" open. The code defines a class "MainActivity" extending "AppCompatActivity". It contains fields for an ImageView and a Button, and a private String array "img_urls" containing five URLs and one "error" string. A circled "1" highlights the array definition. Below it, an "onClick" listener for the button is defined, which calls a "showImage()" method. A circled "2" highlights the "showImage()" call.

```
19
20 public class MainActivity extends AppCompatActivity {
21
22     private ImageView iv_show;
23     private Button btn_changePic;
24
25     private String[] img_urls = new String[]
26     {
27         "https://ss0.baidu.com/73t1bjeh1BF3odCf/it/u=2580698788,1699392913&fm=73",
28         "https://ss0.baidu.com/73t1bjeh1BF3odCf/it/u=818970105,1773418521&fm=73",
29         "https://ss0.baidu.com/73t1bjeh1BF3odCf/it/u=1515523246,2441596091&fm=73",
30         "https://ss0.baidu.com/73t1bjeh1BF3odCf/it/u=268183895,1481180843&fm=96&s=7534877488175&SFEDAAE1C203005087",
31         "error"
32     };
33
34     @Override
35     protected void onCreate(Bundle savedInstanceState) {
36         super.onCreate(savedInstanceState);
37         setContentView(R.layout.activity_main);
38
39         iv_show = (ImageView) findViewById(R.id.iv_show);
40         btn_changePic = (Button) findViewById(R.id.btn_changePic);
41         btn_changePic.setOnClickListener(new View.OnClickListener() {
42             @Override
43             public void onClick(View v) {
44                 showImage();
45             }
46         });
47     }
48 }
```

图 11C-17 部分关键代码

在方框 2 中,当单击按钮时,将调用 showImage 方法进行图片下载。

实际上,这个案例最关键的部分就在于 showImage 方法,也是后续需要用 RxJava 进行优化的部分。

看一下在用 RxJava 优化“前”showImage 方法时如何实现的,代码如图 11C-18 所示。可以看到在 showImage 方法中,创建了一个线程并调用 getImageUrl 方法模拟图片 URL 的下载。如果得到合法的 URL 后,则调用 getBitmapByUrl 方法进行图片下载,并在 runOnUiThread(runOnUIThread 的其实也是使用了 Handle)中更新图片(图 11C-18 中方框 3 与方框 4 部分)。如果得到非法的 URL,则通过 runOnUiThread 方法显示默认图。

```
49     private void showImage() {
50         btn_changePic.setEnabled(false);
51         new Thread(new Runnable() {
52             @Override
53             public void run() {
54                 final String imageUrl = getImageUrl(); ① //下载图片URL
55
56                 if (imageUrl.startsWith("https://")) { ② //判断图片URL是否合法
57                     new Thread(new Runnable() {
58                         @Override
59                         public void run() {
60                             final Bitmap bitmap = getBitmapByUrl(imageUrl);
61                             if (bitmap != null) {
62                                 runOnUiThread(new Runnable() {
63                                     @Override
64                                     public void run() {
65                                         btn_changePic.setEnabled(true);
66                                         iv_show.setImageBitmap(bitmap);
67                                     }
68                                 });
69                             }
70                         }
71                     }).start();
72                 } else {
73                     runOnUiThread(new Runnable() {
74                         @Override
75                         public void run() {
76                             btn_changePic.setEnabled(true);
77                             iv_show.setImageResource(R.mipmap.ic_launcher);
78                             Toast.makeText(MainActivity.this, "BitmapUrl is illegal", Toast.LENGTH_SHORT).show();
79                         }
80                     });
81                 }
82             }
83         }).start();
84     }
85
86     //模拟网络延时，模拟图片的下载地址
87     private String getImageUrl() {
88         try {
89             Thread.sleep(200);
90         } catch (InterruptedException e) {
91             e.printStackTrace();
92         }
93         int index = new Random().nextInt(5);
94         return img_urls[index];
95     }
96
97     //使用 HttpURLConnection下载图片
98     private Bitmap getBitmapByUrl(String url_bitmap) {
99         Bitmap bitmap = null;
100        InputStream is = null;
101        try {
102            URL url = new URL(url_bitmap);
103            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
104            is = new BufferedInputStream(connection.getInputStream());
105            bitmap = BitmapFactory.decodeStream(is);
106            connection.disconnect();
107            return bitmap;
108        } catch (MalformedURLException e) {
109            e.printStackTrace();
110        } catch (IOException e) {
111            e.printStackTrace();
112        } finally {
113            try {
114                if (is != null) is.close();
115            } catch (IOException e) {
116                e.printStackTrace();
117            }
118        }
119        return null;
120    }
121 }
```

图 11C-18 showImage 方法的实现

```
84     //模拟网络延时，模拟图片的下载地址
85     private String getImageUrl() {
86         try {
87             Thread.sleep(200);
88         } catch (InterruptedException e) {
89             e.printStackTrace();
90         }
91         int index = new Random().nextInt(5);
92         return img_urls[index];
93     }
94
95     //使用 HttpURLConnection下载图片
96     private Bitmap getBitmapByUrl(String url_bitmap) {
97         Bitmap bitmap = null;
98         InputStream is = null;
99         try {
100            URL url = new URL(url_bitmap);
101            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
102            is = new BufferedInputStream(connection.getInputStream());
103            bitmap = BitmapFactory.decodeStream(is);
104            connection.disconnect();
105            return bitmap;
106        } catch (MalformedURLException e) {
107            e.printStackTrace();
108        } catch (IOException e) {
109            e.printStackTrace();
110        } finally {
111            try {
112                if (is != null) is.close();
113            } catch (IOException e) {
114                e.printStackTrace();
115            }
116        }
117        return null;
118    }
119 }
```

图 11C-19 getImageUrl 和 getBitmapByUrl 方法的实现

虽然这段代码运行起来没有什么问题,但是由于逻辑负责,各种处理、回调糅合在一起,给人“谜”一般的感觉。有一句话叫:代码“迷之缩进”毁一生,做过大型项目的人都明白。

再来看一下上图两个关键的方法(图 11C-18 方框 1 与方框 2 所标注的方法):getImageUrl 和 getBitmapByUrl,实现的代码如图 11C-19 所示。

getImageUrl 方法实现模拟了下载图片 URL 的过程,主要通过 Thread.Sleep 休眠 200 毫秒,模拟网络下载耗时,然后随机返回 img_urls 数组里的一个 Url 来实现的。

getBitmapByUrl 方法实现了网络图片的下载,主要通过 HttpURLConnection 类来实现的。

关键代码已经讲解完毕,大家应该有所感悟:在异步处理的过程中,由于业务逻辑的复杂,往往使得代码变得晦涩难懂,往往在一段时间过后,自己也看不懂当初的想法。在下一节,将使用 RxJava 对这部分代码进行优化。

2. 使用 RxJava 进行案例优化

1) 在 gradle 文件中加上相关的依赖

如图 11C-20 所示,在案例的 gradle 文件中加入相关的依赖。

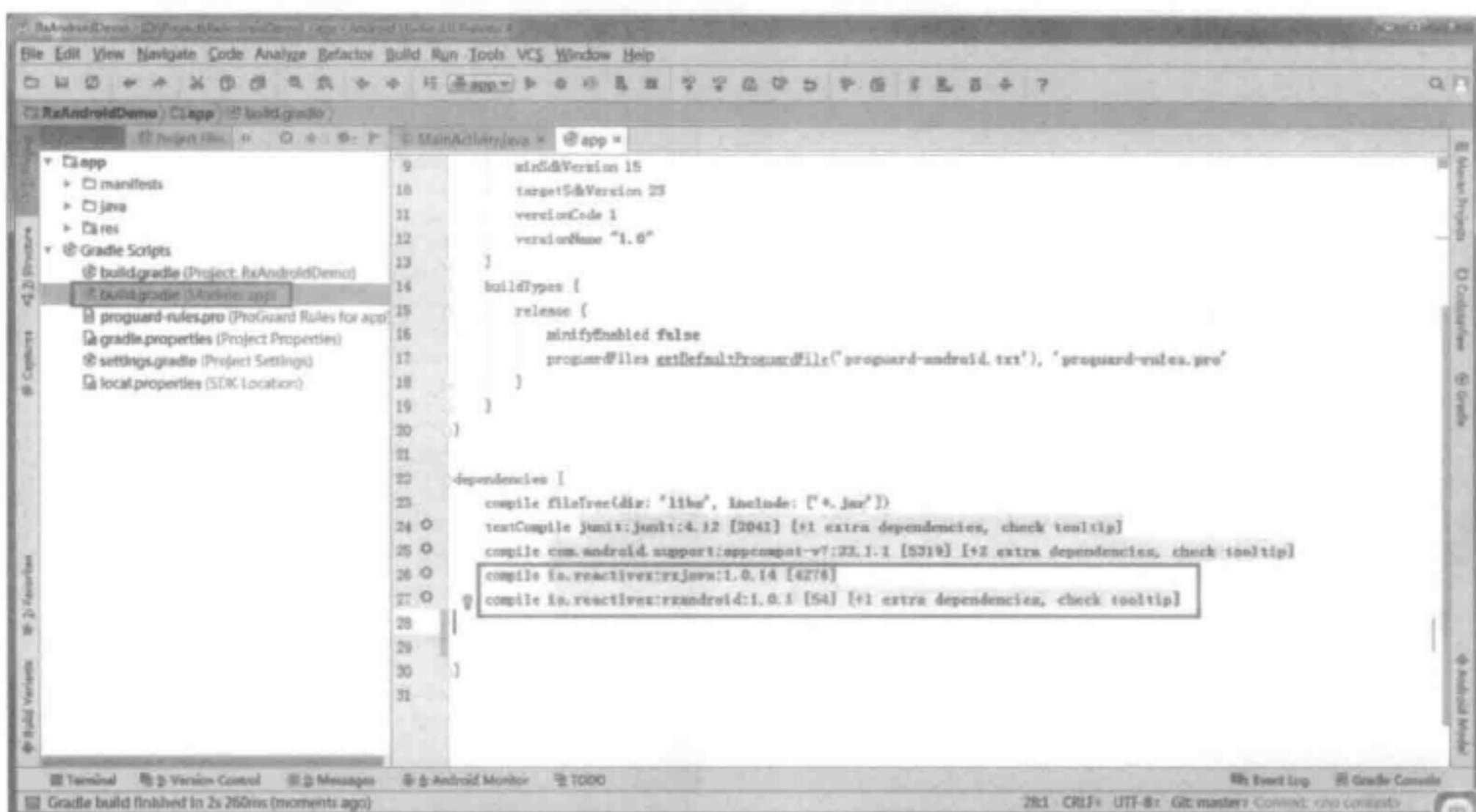


图 11C-20 在 gradle 中加入依赖

2) 创建观察者与被观察者

在这一步中,编写 showImage4RxJava 方法,创建观察者与被观察者并使用 subscribe 方法将它们链接起来,这里其中使用 Subscriber 类作为观察者,代码如图 11C-21 所示。

3) 重写 Observable, OnSubscribe<String> 类的 call 方法

代码如图 11C-22 所示,在该方法中,调用了 getImageUrl 方法获得图片 URL,如果 URL 合法则调用观察者的 onNext 方法将事件往下传递,若 URL 非法则调用观察者的 onError 方法终止事件传输。

4) 使用 map 操作符实现 URL 到图片的变换

这一步,使用了 map 操作符,将图片 URL 转成 Bitmap 对象,实际上是重写了 Fun1 的 call 方法,并使用 getBitmapByUrl 方法作为返

```
private void showImage4RxJava() {
    Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            subscriber.onCompleted();
        }
    }).subscribe(new Subscriber<String>() {
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
        }

        @Override
        public void onNext(String s) {
        }
    });
}

private void showImage() {
    btn_changePic.setEnabled(false);
}
```

图 11C-21 创建观察者与被观察者

回值，通过该方法可以将图片 URL 转换成 Bitmap 对象。代码如图 11C-23 所示。

5) 加入线程控制 Scheduler

在不指定线程的情况下，RxJava 遵循的是线程不变的原则，即：在哪个线程调用 subscribe()，就在哪个线程生产事件，在哪个线程生产事件，就在哪个线程消费事件。

由此可知，到目前为止，案例修改后的所有方法都是在 UI 线程运行的，但是那么像本例的 getImageUrl 和 getBitmapByUrl 方法，都是比较耗时的，希望将这两个方法在一个新的线程中执行，那应该怎么办呢？很简单，使用 subscribeOn 和 observeOn 操作符即可。代码如图 11C-24 所示。

```
private void showImageRxJava() {
    Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            String imageUrl = getImageUrl();
            if (imageUrl.startsWith("https://")) {
                subscriber.onNext(imageUrl);
            } else {
                subscriber.onError(new Throwable("BitmapUrl is illegal"));
            }
        }
    }).subscribe(new Subscriber<String>() {
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
        }
    });
}
```

图 11C-22 重写 call 方法

```
Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        String imageUrl = getImageUrl();
        if (imageUrl.startsWith("https://")) {
            subscriber.onNext(imageUrl);
        } else {
            subscriber.onError(new Throwable("BitmapUrl is illegal"));
        }
    }
}).map(new Func1<String, Bitmap>() {
    @Override
    public Bitmap call(String s) {
        return getBitmapByUrl(s);
    }
}).subscribe(new Subscriber<Bitmap>() {
    @Override
    public void onCompleted() {
    }

    @Override
    public void onError(Throwable e) {
    }
});
```

图 11C-23 使用 map 操作符进行 URL 转 Bitmap

subscribeOn 操作符：指定了 subscribe() 方法所发生的线程，即 Observable. OnSubscribe 被激活时所处的线程，或者称为事件产生的线程。

observeOn 操作符：指定观察者 Subscriber 所运行在的线程，或者称为事件消费的线程。

在图 11C-24 中，使用 subscribeOn() 方法并传入 Schedulers. io() 参数指定事件的产生的线程，可以使得 subscribeOn 操作符前的代码都在一个新的线程中运行。然后接着使用了 observeOn() 方法并传入 AndroidSchedulers. mainThread() 参数指定了事件的消费线程，可以使观察者的回调方法（即事件的消费）在 UI 线程中运行。

【知识点】RxJava 已经内置了几个线程调度器 Scheduler，它们适合大多数的使用场景：

(1) Schedulers. immediate()：直接在当前线程运行，相当于不指定线程。这是默认的 Scheduler。

(2) Schedulers. newThread()：总是启用新线程，并在新线程执行操作。

(3) Schedulers. io()：I/O 操作（读写文件、读写数据库、网络信息交互等）所使用的 Scheduler。行为模式和 newThread() 差不多，区别在于 io() 的内部实现是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下 io() 比 newThread() 更有效率。不要把计算工作放在 io() 中，可以避免创建不必要的线程。

(4) AndroidSchedulers. mainThread()，Android 专门的一个操作符，操作将在 Android 主线程运行。

```
62     .subscribeOn(Schedulers.io())
63     .observeOn(AndroidSchedulers.mainThread())
64     .map(new Func1<String, Bitmap>() {
65         @Override
66         public Bitmap call(String s) {
67             return getBitmapByUrl(s);
68         }
69     })
70     .subscribeOn(Schedulers.io()) // subscribeOn前的代码将在新线程中执行
71     .observeOn(AndroidSchedulers.mainThread()) // observeOn后的代码在UI线程中执行
72     .subscribe(new Subscriber<Bitmap>() {
73         @Override
74         public void onCompleted() {
75         }
76         @Override
77         public void onError(Throwable e) {
78         }
79     });
80 }
```

图 11C-24 加入线程控制

6) 使用 filter 操作符

在原来的案例中,可以看到,若请求得到的图片为 null,则过滤并不处理。在 RxJava 中,这个需求可以用 filter 操作符来实现,如图 11C-25 所示。

```
62     .map(new Func1<String, Bitmap>() {
63         @Override
64         public Bitmap call(String s) {
65             return getBitmapByUrl(s);
66         }
67     })
68     .subscribeOn(Schedulers.io())
69     .observeOn(AndroidSchedulers.mainThread())
70     .filter(new Func1<Bitmap, Boolean>() {
71         @Override
72         public Boolean call(Bitmap bitmap) {
73             return bitmap != null;
74         }
75     })
76     .subscribe(new Subscriber<Bitmap>() {
77         @Override
78         public void onCompleted() {
79         }
80         @Override
81         public void onError(Throwable e) {
82         }
83     });
84 }
```

图 11C-25 使用 filter 过滤掉为 null 的图片

在图 11C-25 方框所示中,使用了 filter 操作符,传入了 Func1 对象,并重写了该对象的 call 方法:若 Bitmap 为空时,返回 false,此时事件将不再往下传递。

7) 实现观察者 Subscriber 的回调方法

这一步,需要重写观察者 Subscriber 的回调方法,代码如图 11C-26 所示。

如上图 11C-26 所示,重写了 onError、onNext、onStart 等方法。若 onError 方法得到回调,说明图片 URL 非法,此时 ImageView 显示默认图。若 onNext 方法得到回调,表明得到有效的图片,此时 ImageView 直接显示下载得到的图片。

为了避免在图片下载过程中,用户可能重复单击按钮。因此也实现了 onStart 方法。它会在 subscribe 刚开始,而事件还未发送之前被调用,在这个时候将 CHANGEPICTURE 按钮设置为

```
79     return bitmap != null;
80   }
81 }
82 .subscribe(new Subscriber<Bitmap>() {
83   @Override
84   public void onCompleted() {
85   }
86 }
87   @Override
88   public void onError(Throwable e) {
89     btn_changePic.setEnabled(true);
90     iv_show.setImageResource(R.mipmap.ic_launcher);
91     Toast.makeText(MainActivity.this, e.getMessage(), Toast.LENGTH_SHORT).show();
92   }
93   @Override
94   public void onNext(Bitmap bitmap) {
95     btn_changePic.setEnabled(true);
96     iv_show.setImageBitmap(bitmap);
97   }
98   @Override
99   public void onStart() {
100    super.onStart();
101    btn_changePic.setEnabled(false);
102  }
103})
```

图 11C-26 重写观察者 Subscriber 的回调方法

不可单击状态,同时当 onError、onNext 方法得到调用时重新将 CHANGEPIC 按钮设置为可单击状态。

8) 运行程序

最后,在 CHANGEPIC 按钮的单击事件回调方法中,改成使用 showImage4RxJava 方法。编译,运行程序。可看到效果如图 11C-27、图 11C-28 所示。



图 11C-27 单击按钮,显示图片



图 11C-28 若 URL 非法,显示默认图

3. 对比

此时,已经使用 RxJava 重写了一遍异步加载图片的案例,回顾一下前面编写的 RxJava 代码,如图 11C-29 所示,在对比一下使用 RxJava 之前的代码,如图 11C-30 所示。大家可以感受到 RxJava 的便利之处了吗?

RxJava 实现的代码,是一条从上到下的链式调用,没有任何嵌套,这在逻辑的简洁性上是十分具有优势的。当需求变得复杂时,这种优势将更加明显(试想一下,当你一两个月后翻回这里看到自己当初写下的那一片各种判断、嵌套的代码,你能保证自己将迅速看懂,而不是对着代码重新捋一遍思路?)

学习完这个案例，亲自证实了 RxJava 的优势：无论多复杂的业务逻辑，使用 RxJava 实现的代码都可以保持足够的简洁性。

```
private void showImage0() {
    Observable.create(new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> subscriber) {
            String imageUrl = getImageUrl();
            if (imageUrl.startsWith("https://")) {
                subscriber.onNext(imageUrl);
            } else {
                subscriber.onError(new Throwable("BitmapUrl is illegal"));
            }
        }
    })
    .map(new Func1<String, Bitmap>() {
        @Override
        public Bitmap call(String s) {
            return getBitmapByUrl(s);
        }
    })
    .subscribeOn(Schedulers.io()) //subscribeOn的代码将在新线程中执行
    .observeOn(AndroidSchedulers.mainThread()) //observeOn后的代码将在主线程中执行
    .filter(new Func1<Bitmap, Boolean>() {
        @Override
        public Boolean call(Bitmap bitmap) {
            return bitmap != null;
        }
    })
    .subscribe(new Subscriber<Bitmap>() {
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable e) {
            btn_changePic.setEnabled(true);
            iv_show.setImageResource(R.drawable.ic_launcher);
            Toast.makeText(MainActivity.this, e.getMessage(), Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onNext(Bitmap bitmap) {
            btn_changePic.setEnabled(false);
            iv_show.setImageBitmap(bitmap);
        }
    });
}
```

图 11C-29 使用 RxJava 的效果

```
private void showImage0() {
    btn_changePic.setEnabled(false);
    new Thread(new Runnable() {
        @Override
        public void run() {
            final String imageUrl = getImageUrl(); //红框，模拟下载图片URL

            if (imageUrl.startsWith("https://")) { //判断图片URL是否合法
                new Thread(new Runnable() { //创建线程下载图片
                    @Override
                    public void run() {
                        final Bitmap bitmap = getBitmapByUrl(imageUrl); //蓝框，根据URL下载图片
                        if (bitmap != null) {
                            runOnUiThread(new Runnable() {
                                @Override
                                public void run() {
                                    btn_changePic.setEnabled(true);
                                    iv_show.setImageBitmap(bitmap);
                                }
                            });
                        }
                    }
                }).start(); //绿框，在runOnUiThread方法中刷新界面
            } else {
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        btn_changePic.setEnabled(true);
                        iv_show.setImageResource(R.drawable.ic_launcher);
                        Toast.makeText(MainActivity.this, "BitmapUrl is illegal", Toast.LENGTH_SHORT).show();
                    }
                });
            }
        }
    }).start();
}
```

图 11C-30 使用 RxJava 前的效果

C.3 项目心得

本文对 RxJava 做出了一个入门级别的介绍，并没有对 RxJava 实现的原理做出过多讲解，希望学有余力的读者继续深入，参考“参考资料”的文章链接，不断学习、进步。

C.4 参考资料

(1) 给 Android 开发者的 RxJava 详解：

<http://gank.io/post/560e15be2dca930e00da1083>

(2) NotRxJava 懒人专用指南：

<http://www.devtf.cn/?p=323>

(3) 深入浅出 RxJava：

<http://blog.csdn.net/lzyzsd/article/details/41833541>

(4) Awesome-RxJava：

<https://github.com/lzyzsd/Awesome-RxJava/blob/master/README.md?from=timeline&isappinstalled=1>

D Android 自动化持续集成环境 CI

搜索关键字

(1) 持续集成

(2) GitHub

(3) UI Automator

本章难点

本章节将介绍 Android 自动化持续集成环境构建与操作。

软件集成，是软件开发过程中经常发生的事情，选择代码、构建、测试和发布都属于软件集成的环节。这些过程通常由人为地完成。“聪明”的程序员擅长把通常需要人工操作的任务变为自动化。持续集成表示自动化的可持续的软件集成，它一种软件开发的过程。

D.1 项目简介

集成(Integration)在软件开发的日常工作中频繁地发生，它包括构建、测试、部署等几个环节。

本次实训介绍了继续集成的理论，并介绍了自动构建工具和自动化测试框架的使用。通过本实训将认识持续集成的基本概念，并能动手搭建基于 Android 的持续集成环境。

Android 的集成开发工具(IDE)例如 Eclipse 或 Android Studio 等为开发人员提供了很方便的构建安装包(下文简称 APK)的功能，在开发的过程中，开发人员经常会使用它们来构建生成 APK，进行调试或者发布版本。

然而，利用集成开发工具进行构建 Android 软件，不仅依赖开发人员的经验，而且也考验开发人员的耐心和细心。如果开发人员在软件发布上市的时候，忘记修改一些调试的选项，或

者遗忘了一部分代码,将软件发布出去,将对产品形象造成负面影响,甚至可能造成经济上的损失。这种情况,对于团队协作开发而言,尤为明显。

为了解决以上问题,本文引入了搭建 Android 持续集成环境的概念。

本文提到的集成环境,只需要使用一台电脑(不限操作系统,Windows、OS X 或 Linux 均可),可选设备为一台 Android 硬件设备。

阅读本文后,可以掌握:

- (1) 安装 Jenkins 自动化构建系统;
- (2) 应用 Jenkins 创建 Android 的持续集成任务;
- (3) 应用 Android 自动化测试框架 UI Automator。

D.2 案例设计与实现

D.2.1 需求分析

搭建适用于 Android 的持续集成环境,整个持续集成的过程由程序自动完成,不需要人工参与。包含以下环节:

- (1) 自动获取合适的代码。
- (2) 自动化构建,输出软件版本。
- (3) 自动化测试,输出测试报告。

D.2.2 示例说明

1. 用于构建的 App 示例

本文将使用一个名为“Actions”的 Android 端的 GTD(Getting Thinks Done)软件来演示如何搭建持续集成的 Android 开发环境。

值得注意是,如何编写 App 不是本章的重点内容(源代码可下载),该 App 仅用于演示如何实现自动化构建以及自动化测试,如图 11D-1 所示。

Actions 的功能包括:

- (1) 新增和编辑任务(Action);
- (2) 查看任务列表和任务详情;
- (3) 将标记为完成(Achieved)。

2. 持续集成工具

持续集成工具用于管理持续集成的流程,包括配置代码库,触发构建等功能,如图 11D-2 所示。

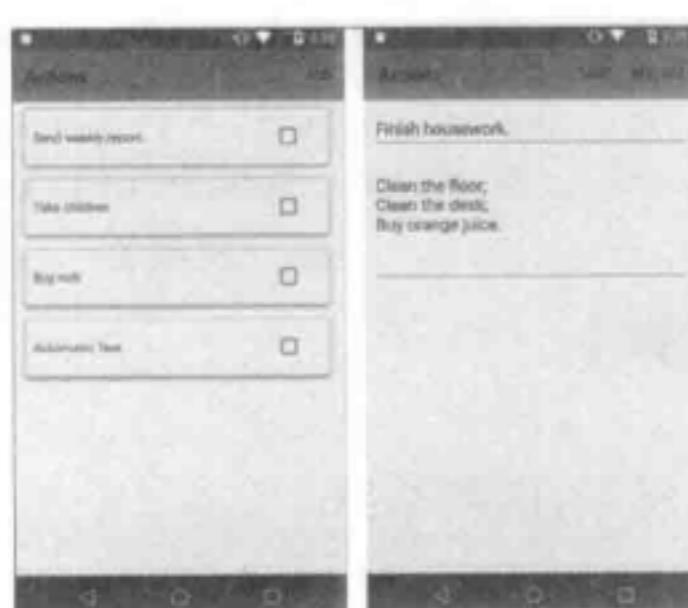


图 11D-1 本文的示例程序:Actions



图 11D-2 持续集成工具