链接地址：
Android官网

android在线源码

我的CSDN博客

我的Github

开源项目分析

强大的android工具资料网站

AndroidTraning中文版

Android开发技术周报（中文版）

Android开发技术周报（英文版）

知乎Android开发技术周报（中文版）专栏

JAVA 设计模式

云在千峰

Trinea的github

android-cn github

Google利器Android Studio从入门到精通

使用Gradle构建Android程序

昵称：Leo的银弹
园龄：1年10个月
粉丝：7
关注：0
+加关注

< 2016年4月 >

日	一	二	三	四	五	六
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

搜索

找找看

谷歌搜索

常用链接

我的随笔

我的评论

RxJava开发精要7 – Schedulers-解决Android主线程问题

- 原文出自《RxJava Essentials》
- 原文作者：Ivan Morgillo
- 译文出自：开发技术前线 www.devtf.cn
- 转载声明: 本译文已授权开发者头条享有独家转载权，未经允许，不得转载!
- 译者：yuxingxin
- 项目地址：RxJava-Essentials-CN

前面一章是最后一章关于RxJava的Observable的创建和操作的章节。我们学习到了如何将两个或更多的Observables合并在一起，join它们，zip它们，merge它们以及如何创建一个新的Observable来满足我们特殊的需求。

本章中，我们提升标准看看如何使用RxJava的调度器来处理多线程和并发编程的问题。我们将学习到如何以响应式的方式创建网络操作，内存访问，以及耗时任务。

StrictMode

为了获得更多出现在代码中的关于公共问题的信息，我们激活了StrictMode模式。

StrictMode帮助我们侦测敏感的活动，如我们无意的在主线程执行磁盘访问或者网络调用。正如你所知道的，在主线程执行繁重的或者长时的任务是不可取的。因为Android应用的主线程时UI线程，它被用来处理和UI相关的操作：这也是获得更平滑的动画体验和响应式App的唯一方法。

为了在我们的App中激活StrictMode，我们只需要在MainActivity中添加几行代码，即onCreate()方法中这样：

```
@Override
public void onCreate() {
   super.onCreate();
   if (BuildConfig.DEBUG) {
      StrictMode.setThreadPolicy(new
StrictMode.ThreadPolicy.Builder().detectAll().penaltyLog().build());
      StrictMode.setVmPolicy(new
StrictMode.VmPolicy.Builder().detectAll().penaltyLog().build());
   }
}
```

我们并不想它总是激活着，因此我们只在debug构建时使用。这种配置将报告每一种关于主线程用法的违规做法，并且这些做法都可能与内存泄露有关：Activities、BroadcastReceivers、Sqlite等对象。

选择了penaltyLog()，当违规做法发生时，StrictMode将会在logcat打印一条信息。

避免阻塞I/O的操作

阻塞I/O的操作将使App能够进行下一步操作前会强制使其等待结果的返回。在UI线程上执行一个阻塞操作将强制使UI卡住，这将直接产生不好的用户体验。

我们激活StrictMode后，我们开始收到了关于我们的App错误操作磁盘I/O的不友好信息。

```
D/StrictMode  StrictMode policy violation; ~duration=998 ms:
android.os.StrictMode$StrictModeDiskReadViolation: policy=31 violation=2
at android.os.StrictMode$AndroidBlockGuardPolicy.onReadFromDisk
```

我的参与

最新评论

我的标签

更多链接

最新随笔

1. RxJava学习(三)

2. RxJava学习(二)

3. Android性能优化之如何避免Over draw

4. Android 应用开发性能优化完全分析

5. Android性能优化典范 - 第2季

6. Android界面性能调优手册

7. 15个必知的Android开发者选项

8. Retrofit初识

9. RxJava学习(一)

10. Git push错误non-fast-forward后的冲突解决

我的标签

线程安全(1)

随笔分类(265)

android Training(9)

android知识点复习与总结(77)

JAVA 知识点复习与总结(2)

RxJava(12)

读书笔记(8)

开源项目学习(38)

设计模式(14)

算法与数据结构(2)

项目经验谈(80)

性能优化(17)

学习大牛(6)

随笔档案(259)

2016年4月 (12)

2016年3月 (27)

2016年2月 (29)

```
3 (StrictMode.java:1135)
4 at libcore.io.BlockGuardOs.open(BlockGuardOs.java:106) at
5 libcore.io.IoBridge.open(IoBridge.java:393)
6 at java.io.FileOutputStream.<init>(FileOutputStream.java:88)
7 at android.app.ContextImpl.openFileOutput(ContextImpl.java:918)
8 at android.content.ContextWrapper.openFileOutput(ContextWrapper. java:185)
  at com.packtpub.apps.rxjava_essentials.Utils.storeBitmap (Utils.java:30)
```

上一条信息告诉我们`Utils.storeBitmap()`函数执行完耗时998ms：在UI线程上近1秒的不必要的工作和App上近1秒不必要的迟钝。这是因为我们以阻塞的方式访问磁盘。我们的`storeBitmap()`函数包含了：

```
1 FileOutputStream fOut = context.openFileOutput(filename, Context.MODE_PRIVATE);
2
```

它直接访问智能手机的固态存储然后就慢了。我们该如何提高访问速度呢？`storeBitmap()`函数保存了已安装App的图标。他返回了`void`，因此在执行下一个操作前我们毫无理由去等待直到它完成。我们可以启动它并让它执行在不同的线程。Android中这些年线程管理的变化产生了App诡异的行为。我们可以使用AsyncTask，但是我们要避免掉入前几章里的`onPrehttps://github.com/yuxingxin/RxJava-Essentials-CN/raw/master.onPosthttps://github.com/yuxingxin/RxJava-Essentials-CN/raw/master.doInBackGround`地狱。我们将使用RxJava的方式;万岁的调度器！

Schedulers

调度器以一种最简单的方式将多线程用在你的Apps的中。它们时RxJava重要的一部分并能很好地与Observables协同工作。它们无需处理实现、同步、线程、平台限制、平台变化而可以提供一种灵活的方式来创建并发程序。

RxJava提供了5种调度器：

- `.io()`
- `.computation()`
- `.immediate()`
- `.newThread()`
- `.trampoline()`

让我们一个一个的来看下它们：

Schedulers.io()

这个调度器时用于I/O操作。它基于根据需要，增长或缩减来自适应的线程池。我们将使用它来修复我们之前看到的`StrictMode`违规做法。由于它专用于I/O操作，所以并不是RxJava的默认方法；正确的使用它是由开发者决定的。

重点需要注意的是线程池是无限制的，大量的I/O调度操作将创建许多个线程并占用内存。一如既往的是，我们需要在性能和简捷两者之间找到一个有效的平衡点。

Schedulers.computation()

这个是计算工作默认的调度器，它与I/O操作无关。它也是许多RxJava方法的默认调度器：`buffer()`,`debounce()`,`delay()`,`interval()`,`sample()`,`skip()`。

Schedulers.immediate()

这个调度器允许你立即在当前线程执行你指定的工作。它是`timeout()`,`timeInterval()`,以及`timestamp()`方法默认的调度器。

Schedulers.newThread()

这个调度器正如它所看起来的那样：它为指定任务启动一个新的线程。

Schedulers.trampoline()

当我们想在当前线程执行一个任务时，并不是立即，我们可以用`.trampoline()`将它入队。这个调度器将会处理它的队列并且按序运行队列中每一个任务。它是`repeat()`和`retry()`方法默认的调度器。

2016年1月 (5)
2015年12月 (21)
2015年11月 (8)
2015年10月 (1)
2015年7月 (1)
2015年5月 (1)
2015年4月 (4)
2015年1月 (1)
2014年12月 (2)
2014年10月 (4)
2014年9月 (1)
2014年8月 (3)
2014年5月 (27)
2014年4月 (7)
2014年3月 (32)
2014年2月 (17)
2014年1月 (7)
2013年11月 (2)
2013年10月 (10)
2013年9月 (4)
2013年8月 (28)
2013年7月 (5)

文章分类

android知识点

java 设计模式

java设计模式

积分与排名

积分 - 8262

排名 - 20181

最新评论

1. Re:Github上更新自己Fork的代码

可以在线pull request同步

--Michael Jiang

阅读排行榜

非阻塞I/O操作

现在我们知道如何在一个指定I/O调度器上来调度一个任务，我们可以修改`storeBitmap()`函数并再次检查`StrictMode`的不合规做法。为了这个例子，我们可以在新的`blockingStoreBitmap()`函数中重排代码。

```
private static void blockingStoreBitmap(Context context, Bitmap bitmap, String filename) {
    FileOutputStream fOut = null;
    try {
        fOut = context.openFileOutput(filename, Context.MODE_PRIVATE);
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, fOut);
        fOut.flush();
        fOut.close();
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        try {
            if (fOut != null) {
                fOut.close();
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

现在我们可以使用`Schedulers.io()`创建非阻塞的版本：

```
public static void storeBitmap(Context context, Bitmap bitmap, String filename) {
    Schedulers.io().createWorker().schedule(() -> {
        blockingStoreBitmap(context, bitmap, filename);
    });
}
```

每次我们调用`storeBitmap()`，RxJava处理创建所有它需要从I / O线程池一个特定的I/ O线程执行我们的任务。所有要执行的操作都避免在UI线程执行并且我们的App比之前要快上1秒：logcat上也不再有`StrictMode`的不合规做法。

下图展示了我们在`storeBitmap()`场景看到的两种方法的不同：

1. 自定义滚轮效果选择器spinnerwheel的使用总结(445)

2. Android开源项目发现---ViewPager 、Gallery 篇（持续更新）(224)

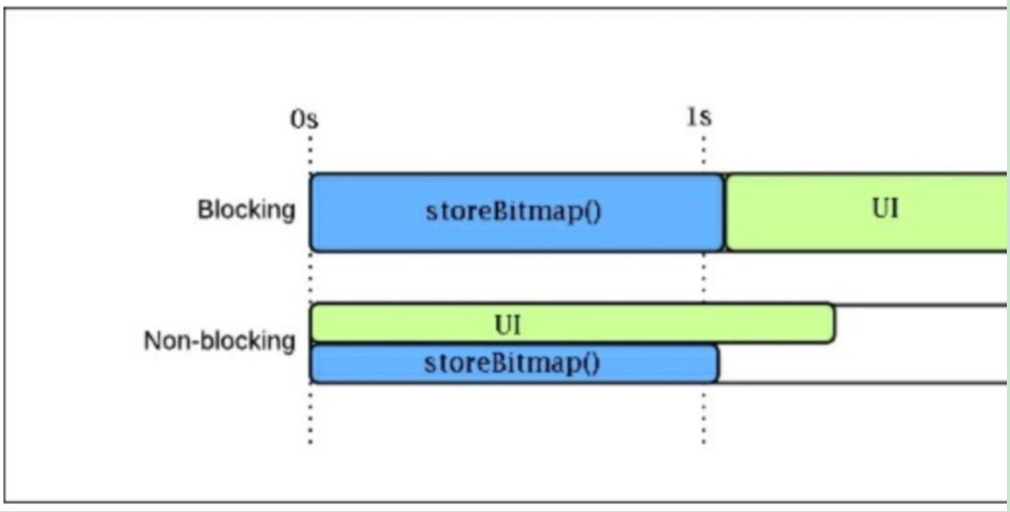
3. 实现ImageView中两张图片重叠显示(209)

4. Android用户界面 UI组件--AdapterView及其子类(一) ListView及各种Adapter详解(145)

5. Android代码中使用Ping命令(128)

评论排行榜

1. Github上更新自己Fork的代码(1)



SubscribeOn and ObserveOn

我们学到了如何在一个调度器上运行一个任务。但是我们如何利用它来和Observables一起工作呢？RxJava提供了subscribeOn()方法来用于每个Observable对象。subscribeOn()方法用Scheduler来作为参数并在这个Scheduler上执行Observable调用。

在"真实世界"这个例子中，我们调整loadList()函数。首先，我们需要一个新的getApps()方法来检索已安装的应用列表：

```
1
2
3
4  private Observable<AppInfo> getApps() {
5      return Observable.create(subscriber -> {
6          List<AppInfo> apps = new ArrayList<>();
7          SharedPreferences sharedPref =
8  getActivity().getPreferences(Context.MODE_PRIVATE);
9          Type appInfoType = new TypeToken<List<AppInfo>>(){}.getType();
1          String serializedApps = sharedPref.getString("APPS", "");
0          if (!"".equals(serializedApps)) {
1              apps = new Gson().fromJson(serializedApps,appInfoType);
1          }
1          for (AppInfo app : apps) {
2              subscriber.onNext(app);
1          }
3          subscriber.onCompleted();
1      });
4  }
1
5
1
6
```

getApps()方法返回一个AppInfo的Observable。它先从Android的SharePreferences读取到已安装的应用程序列表。反序列化，并一个接一个的发射AppInfo数据。使用新的方法来检索列表，loadList()函数改成下面这样：

```
1
2
3
4
5
6
7  private void loadList() {
8      mRecyclerView.setVisibility(View.VISIBLE);
9      getApps().subscribe(new Observer<AppInfo>() {
1          @Override
0          public void onCompleted() {
1              mSwipeRefreshLayout.setRefreshing(false);
1              Toast.makeText(getActivity(), "Here is the list!",
1  Toast.LENGTH_LONG).show();
```

```
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
            mSwipeRefreshLayout.setRefreshing(false);
        }

        @Override
        public void onNext(AppInfo appInfo) {
            mAddedApps.add(appInfo);
                mAdapter.addApplication(mAddedApps.size() - 1, appInfo);
        }
    });
}
```

如果我们运行代码，StrictMode将会报告一个不合规操作，这是因为SharePreferences会减慢I/O操作。我们所需要做的是指定getApps()需要在调度器上执行：

```
<br />getApps().subscribeOn(Schedulers.io())
        .subscribe(new Observer<AppInfo>() { [https://github.com/yuxingxin/RxJava-
Essentials-CN/raw/master.]
```

Schedulers.io()将会去掉StrictMode的不合规操作，但是我们的App现在崩溃了是因为：

```
at rx.internal.schedulers.ScheduledAction.run(ScheduledAction.jav a:58)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors. java:422)
at java.util.concurrent.FutureTask.run(FutureTask.java:237)
at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutu
reTask.access$201(ScheduledThreadPoolExecutor.java:152)
at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutu
reTask.run(ScheduledThreadPoolExecutor.java:265)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolEx
ecutor.java:1112)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolE
xecutor.java:587)
at java.lang.Thread.run(Thread.java:841) Caused by:
    android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original
thread that created a view hierarchy can touch its views.
```

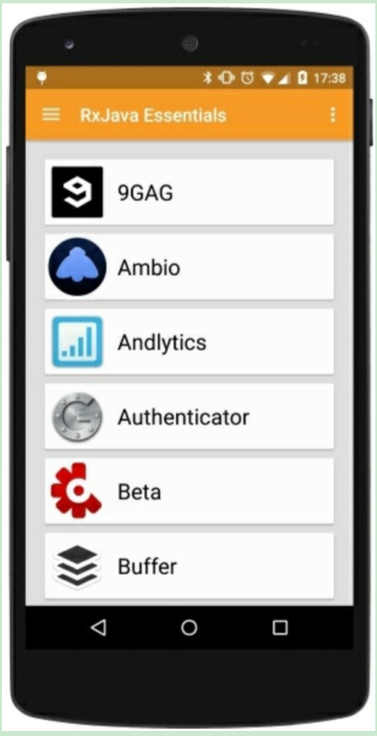
Only the original thread that created a view hierarchy can touch its views.

我们再次回到Android的世界。这条信息简单的告诉我们我们试图在一个非UI线程来修改UI操作。意思是我们需要在I/O调度器上执行我们的代码。因此我们需要和I/O调度器一起执行代码，但是当结果返回时我们需要在UI线程上操作。RxJava让你能够订阅一个指定的调度器并观察它。我们只需在loadList()函数添加几行代码，那么每一项就都准备好了：

```
getApps()
.onBackpressureBuffer()
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
```

```
5 .subscribe(new Observer<AppInfo>() { [https://github.com/yuxingxin/RxJava-
6 Essentials-CN/raw/master.]
```

observeOn()方法将会在指定的调度器上返回结果：如例子中的UI线程。onBackpressureBuffer()方法将告诉Observable发射的数据如果比观察者消费的数据要更快的话，它必须把它们存储在缓存中并提供一个合适的时间给它们。做完这些工作之后，如果我们运行App，就会出现已安装的程序列表：



处理耗时的任务

我们已经知道如何处理缓慢的I/O操作。让我们看一个与I/O无关的耗时的任务。例如，我们修改loadList()函数并创建一个新的slow函数发射我们已安装的app数据。

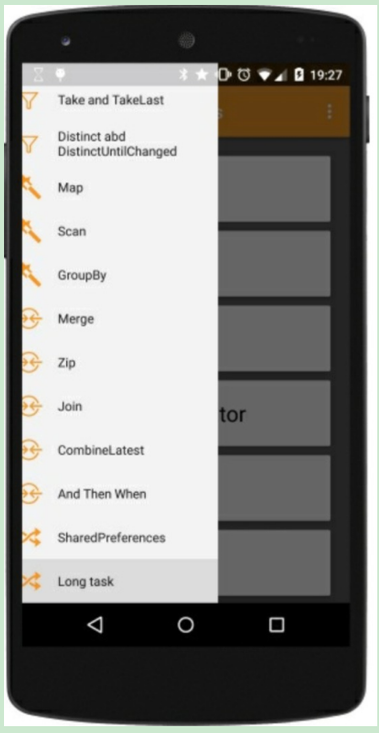
```
1  private Observable<AppInfo> getObservableApps(List<AppInfo> apps) {
2      return Observable .create(subscriber -> {
3          for (double i = 0; i < 1000000000; i++) {
4              double y = i * i;
5          }
6          for (AppInfo app : apps) {
7              subscriber.onNext(app);
8          }
9          subscriber.onCompleted();
10     });
11 }
12
```

正如你看到的，这个函数执行了一些毫无意义的计算，只是针对这个例子消耗时间，然后从List<AppInfo>对象中发射我们的AppInfo数据，现在，我们重排loadList()函数如下：

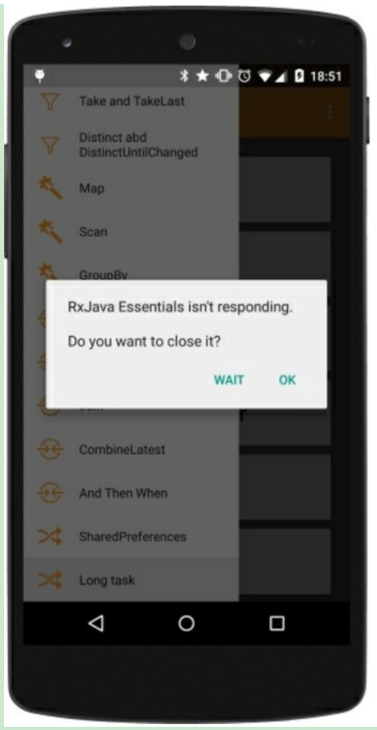
```
1
2
3
4
5
6
7  private void loadList(List<AppInfo> apps) {
8      mRecyclerView.setVisibility(View.VISIBLE);
9      getObservableApps(apps)
10         .subscribe(new Observer<AppInfo>() {
11             @Override
               public void onCompleted() {
                   mSwipeRefreshLayout.setRefreshing(false);
```

```
            Toast.makeText(getActivity(), "Here is the list!",
Toast.LENGTH_LONG).show();
        }

        @Override
        public void onError(Throwable e) {
            Toast.makeText(getActivity(), "Something went wrong!",
Toast.LENGTH_SHORT).show();
            mSwipeRefreshLayout.setRefreshing(false);
        }

        @Override
        public void onNext(AppInfo appInfo) {
            mAddedApps.add(appInfo);
            mAdapter.addApplication(mAddedApps.size() - 1, appInfo);
        }
    });
}
```

如果我们运行这段代码，当我们点击Navigation Drawer菜单项时App将会卡住一会，然后你能看到下图中半关闭的菜单：



如果我们不够走运的话，我们可以看到下图中经典的ANR信息框：



可以确定的是，我们将会看到下面在logcat中不愉快的信息：

```
I/Choreographer  Skipped 598 frames! The application may be doing too much work on its main thread.
```

这条信息比较清楚，Android在告诉我们用户体验非常差的原因是我们用不必要的工作量阻塞了UI线程。但是我们已经知道了如何处理它：我们有调度器！我们只须添加几行代码到我们的Observable链中就能去掉加载慢和Choreographer信息：

```
getObservableApps(apps)
    .onBackpressureBuffer()
    .subscribeOn(Schedulers.computation())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<AppInfo>() { [https://github.com/yuxingxin/RxJava-Essentials-CN/raw/master.]
```

用这几行代码，我们将可以快速关掉Navigation Drawer,一个漂亮的进度条，一个工作在独立的线程缓慢执行的计算任务，并在主线程返回结果让我们更新已安装的应用列表。

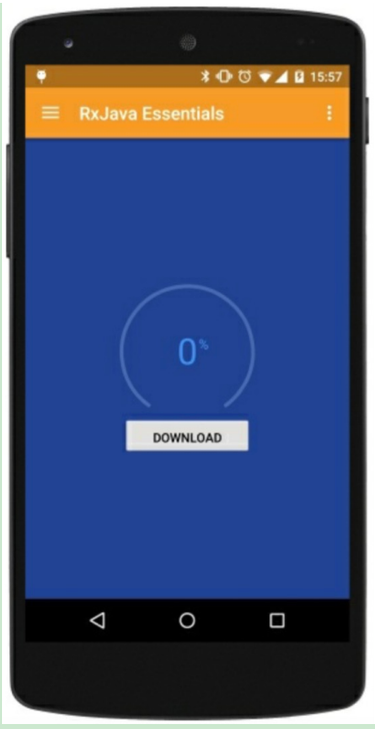
执行网络任务

网络在今天是99%的移动应用的一部分：我们总是连接远端服务器来检索我们App需要的信息。

作为网络访问的第一个方法，我们将创建下面这样一个场景:

- 加载一个进度条。
- 用一个按钮开始文件下载。
- 下载过程中更新进度条。
- 下载完后开始视频播放。

我们的用户界面非常简单，我们只需要一个有趣的进度条和一个下载按钮。



首先，我们创建mDownloadProgress

```
1 private PublishSubject<Integer>mDownloadProgress = PublishSubject.create();
2
```

这个主题我们用来管理进度的更新，它和download函数协同工作。

```
1  private boolean downloadFile(String source, String destination) {
2      boolean result = false;
3      InputStream input = null;
4      OutputStream output = null;
5      HttpURLConnection connection = null;
6      try {
7          URL url = new URL(source);
8          connection = (HttpURLConnection) url.openConnection();
9          connection.connect();
10         if (connection.getResponseCode() != HttpURLConnection.HTTP_OK) {
11             return false;
12         }
13         int fileLength = connection.getContentLength();
14         input = connection.getInputStream();
15         output = new FileOutputStream(destination);
16         byte data[] = new byte[4096];
17         long total = 0;
18         int count;
19         while ((count = input.read(data)) != -1) {
20             total += count;
21             if (fileLength >0) {
22                 int percentage = (int) (total * 100 / fileLength);
23                 mDownloadProgress.onNext(percentage);
24             }
25             output.write(data, 0, count);
26         }
27         mDownloadProgress.onCompleted();
28         result = true;
29     } catch (Exception e) {
30         mDownloadProgress.onError(e);
31     } finally {
32         try {
33             if (output != null) {
34                 output.close();
35             }
36             if (input != null) {
37                 input.close();
```

```
            }
        } catch (IOException e) {
            mDownloadProgress.onError(e);
        }
        if (connection != null) {
            connection.disconnect();
            mDownloadProgress.onCompleted();
        }
    }
    return result;
}

```

上面的这段代码将会触发NetworkOnMainThreadException异常。我们可以创建RxJava版本的函数进入我们挚爱的响应式世界来解决这个问题：

```


private Observable<Boolean> obserbableDownload(String source, String
destination) {
    return Observable.create(subscriber -> {
        try {
            boolean result = downloadFile(source, destination);
            if (result) {
                subscriber.onNext(true);
                subscriber.onCompleted();
            } else {
                subscriber.onError(new Throwable("Download failed."));
            }
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}


```

现在我们需要触发下载操作，点击下载按钮:

```


@OnClick(R.id.button_download)
void download() {
    mButton.setText(getString(R.string.downloading));
    mButton.setClickable(false);
    mDownloadProgress.distinct()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<Integer>() {

        @Override
```

```
        public void onCompleted() {
            App.L.debug("Completed");
        }

        @Override
        public void onError(Throwable e) {
            App.L.error(e.toString());
        }

        @Override
        public void onNext(Integer progress) {
            mArcProgress.setProgress(progress);
        }
    });

    String destination = "sdcardsoftboy.avi";
    obserbableDownload("http://archive.blender.org/fileadmin/movies/softboy.avi",
destination)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(success -> {
            resetDownloadButton();
            Intent intent = new Intent(android.content.Intent.ACTION_VIEW);
            File file = new File(destination);
            intent.setDataAndType(Uri.fromFile(file),"video/avi");
            intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            startActivity(intent);
        }, error -> {
            Toast.makeText(getActivity(), "Something went south",
Toast.LENGTH_SHORT).show();
            resetDownloadButton();
        });
}
```

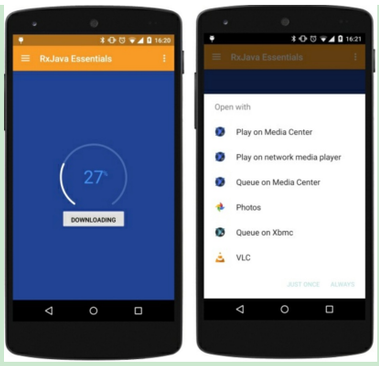
我们使用Butter Knife的注解@OnClick来绑定按钮的方法并更新按钮信息和点击状态：我们不想让用户点击多次从而触发多次下载事件。

然后，我们创建一个subscription来观察下载进度并相应的更新进度条。很明显，我们我们观测主线程是因为进度条是UI元素。

```
obserbableDownload("http://archive.blender.org/fileadmin/movies/softboy.avi",
"sdcardsoftboy.avi";)
```

这是一个下载Observable。网络调用是一个I/O任务和我们预料的那样使用I/O调度器。当下载完成时，我们在onNext()启动视频播放器，并且播放器将会在目的URL找到下载的文件.。

下图展示了下载进度和视频播放器对话框：



总结

这一章中，我们学习了如何简单的将多线程应用在我们的App中。RxJava为此提供了极其有用的工具：调度器。调度器来自不同的指定优化场景并且我们也不避免了`StrictMode`不合法操作以及阻塞I/O函数。我们现在可以用简单的，响应式的并在整个App中保持一致的方式来访问内存和网络。

下一章中，我们将会提高风险并创建一个真实世界App，并使用Square公司开源的REST API库Retrofit从不同的远程资源获取数据来创建一个复杂的material design UI。

分类: RxJava

Leo的银弹
关注 - 0
粉丝 - 7
+加关注

« 上一篇：RxJava开发精要6 – Observables组合
» 下一篇：RxJava开发精要8 – 与REST无缝结合-RxJava和Retrofit

posted @ 2016-02-16 22:35 Leo的银弹 阅读(22) 评论(0) 编辑 收藏