赵凯强

专注Android移动开发，热爱分享，支持开源

用RxJava.Observable取代AsyncTask和AsyncTaskLoader-RxJava Android模版

标签： RxJava

2015-03-30 14:54 3394人阅读 评论(2) 收藏 举报

分类： Android开发经验（71）

目录(?) [+]

转载请注明出处http://blog.csdn.net/zhaokaiqiang1992

> 欢迎关注ndroid-tech-frontier开源项目，定期翻译国外Android优质的技术、开源库、软件架构设计、测试等文章

- 译者 : ZhaoKaiQiang
- 校对者: chaossss
- 状态 : 校对完成

在网上有很多关于RxJava入门指南的帖子，其中一些是基于Android环境的。但是，我想到目前为止，很多人只是沉迷于他们所看到的这些，当要解决在他们的Android项目中出现的具体问题时，他们并不知道如何或者是为什么要使用RxJava。在这一系列的文章中，我想要探索在我工作过的一些依赖于RxJava架构的Android项目中的模式。

在文章的开始，我想要处理一些Android开发者在使用RxJava的时候，很容易遇到的状况。从这个角度，我将提供更高级和更合适的解决方案。在这一系列的文章中，我希望可以听到其他开发者在使用RxJava的过程中解决类似的问题，或许他们和我发现的一样呢。

问题一：后台任务

Android开发者首先遇到的挑战就是如何有效的在后台线程中工作，然后在UI线程中更新UI。这经常是因为需要从web service中获取数据。对于已经有相关经验的你可能会说：“这有什么挑战性？你只需要启动一个AsyncTask，然后所有的工作它就都给你做了。”如果你是这样想的，那么你有一个机会去改善这种状况。这是因为你已经习惯了这种复杂的方式并且忘记这本应该是很简洁的，或者是说你没有处理所有应该处理的边界情况。让我们来谈谈这个。

默认的解决方案：AsyncTask

AsyncTask是在Android里面默认的处理工具，开发者可以做里面一些长时间的处理工作，而不会阻塞用户界面。(注意：最近，AsyncTaskLoader用来处理一些更加具体的数据加载任务，我们以后会再谈谈这个)

表面上，这似乎很简单，你定义一些代码在后台线程中运行，然后定义一些代码运行在UI线程中，在后台任务处理完之后，它在UI线程会处理从后台任务传递过来的执行结果。

```
private class CallWebServiceTask extends AsyncTask<String, Result, Void> {

    protected Result doInBackground(String... someData) {
        Result result = webService.doSomething(someData);
        return result;
    }

    protected void onPostExecute(Result result) {
        if (result.isSuccess() {
            resultText.setText("It worked!");
        }
    }
}
```

使用AsyncTask的最大的问题是在细节的处理上，让我们谈谈这个问题。

错误处理

这种简单用法的第一个问题就是："如果出现了错误怎么办？"不幸的是，暂时没有非常好的解决方案。所以很多的开发者最终要继承AsyncTask，然后在doInBackground()中包裹一个try/catch，返回一个

Activity和Fragment的生命周期

另外一个你必须面对的问题是："当AsyncTask正在运行的时候，如果我退出Activity或者是旋转设备的话会发生什么？"嗯，如果你只是发送一些数据，之后就不再关心发送结果，那可能是没有问题的，但是如果你需要根据Task的返回结果更新UI呢？如果你不做一些事情阻止的话，那么当你试图去调用Activity或者是view的话，你将得到一个空指针异常导致程序崩溃，因为他们现在是不可见或者是null的。

同样，在这个问题上AsyncTask没有做很多工作去帮助你。作为一个开发者，你需要确保保持一个Task的引用，并且要么当Activity正在被销毁的时候取消Task，要么当你试图在onPostExecute()里面更新UI的时候，确保Activity是在一个可达状态。当你只想明确的做一些工作，并且让项目容易维护的时候，这将会继续提高维护项目的难度。

旋转时的缓存(或是其他情况)

当你的用户还是待在当前Activity，仅仅是旋转屏幕会发生什么？在这种情况下，取消Task没有任何意义，因为在旋转之后，你最终还是需要重新启动Task。或者是你不想重启Task，因为状况在一些地方以非幕等的方式发生了突变(because it mutates some state somewhere in a non-idempotent way),但是你确实想要得到结果，因为这样你就可以更新UI来反映这种情况。

如果你专门的做一个只读的加载操作，你可以使用AsyncTaskLoader去解决这个问题。但是对于标准的Android方式来说，它还是很沉重，因为缺少错误处理，在Activity中没有缓存，还有很多独有的其他怪癖。

组合的多个Web Server调用

现在，假如说我们已经想办法把上面的问题都解决了，但是我们现在需要做一些连续的网络求求，每一个请求都需要基于前一个请求的结果。或者是，我们想做一些并发的网络请求，然后把结果合并在一起发送到UI线程，但是，再次抱歉，AsyncTask在这里帮不到你。

一旦你开始做这样的事情，随着更多的复杂线程模型的增长，之前的问题会导致处理这样的事情非常的痛苦和苍白无力。如果想要这些线程一起运行，要不你就让它们单独运行，然后回调，或者在其他情况下，让它们在一个后台线程中同步运行，最后复制组成不同。(To chain calls together, you either keep them separate and end up in callback hell, or run them synchronously together in one background task end up duplicating work to compose them differently in other situations.)

如果要并行运行，你必须创建一个自定义的executor然后传递给AsyncTaskTask，因为默认的AsyncTask不支持并行。并且为了协调并行线程，你需要使用像是CountDownLatchs, Threads, Executors 和 Futures去降低更复杂的同

步模式。

可测试性

抛开这些不说，如果你喜欢测试你的代码，AsyncTask并不能给你带来什么帮助。如果我们不做额外的工作，测试AsyncTask非常困难，它很脆弱并且难以维持。这是一篇有关如何成功测试AsyncTask的帖子。

更好的解决方案：RxJava's Observable

幸运的是，一旦我们决定使用RxJava依赖库的时候，我们讨论的这些问题就都迎刃而解了。下面我们看看它能为我们做什么。

下面我们将会使用Observables写一个请求代码来替代上面的AsyncTask方式。(如果你使用Retrofit，那么你应该很容易使用，其次它还支持Observable 返回值，并且它工作在一个后台的线程池，无需你额外的工作)

```
webService.doSomething(someData)
.observeOn(AndroidSchedulers.mainThread())
.subscribe(
    result -> resultText.setText("It worked!")),
    e -> handleError(e)
);
```

错误处理

你可能会注意到，没有做额外的工作，我们已经处理了AsyncTask不会处理的成功和错误的情况，并且我们写了很少的代码。你看到的额外的组件是我们想要Observer 在UI主线程中处理的结果。这样可以让我们前进一点点。并且如果你的sebService对象不想在后台线程中运行，你也可以在这里通过使用.subscribeOn(…) 声明。(注意：这些例子是使用Java 8的lambda语法，使用Retrolambda就可以在Android项目中进行使用了，但在我看来，这样做的回报是高于风险的，和写这篇文章相比，我们更喜欢在我们的项目中使用。)

Activity和Fragment的生命周期

现在，我们想在这里利用RxAndroid解决上面提到的生命周期的问题，我们不需要指定mainThread() scheduler(顺便说一句，你只需要导入RxAndroid)。就像下面这样

```
AppObservable.bindFragment(this, webService.doSomething(someData))
.subscribe(
    result -> resultText.setText("It worked!")),
    e -> handleError(e)
);
```

我通常的做法是在应用的Base Fragment里面创建一个帮助方法来简化这一点，你可以参考我维护的一个Base RxFragment 获得一些指导。

```
bind(webService.doSomething(someData))
.subscribe(
    result -> resultText.setText("It worked!")),
    e -> handleError(e)
);
```

如果我们的Activity或者是Fragment不再是可达状态，那么AppObservable.bindFragment()可以在调用链中插入一个垫片，来阻止onNext()运行。如果当Task试图运行的时候，Activity、Fragment是不可达状态，subscription 将会取消订阅并且停止运行，所以不会存在空指针的风险，程序也不会崩溃。一个需要注意的是，当我们离开Activity和Fragment时，我们会暂时或者是永久的泄露，这取决于问题中的Observable 的行为。所以在bind()方法中，我也会调用LifeCycleObservable机制，当Fragment销毁的时候自动取消。这样做的好处是一劳永逸。

所以，这解决了首要的两个问题，但是下面这一个才是RxJava大发神威的地方。

组合的多个Web Server调用

在这里我不会详细的说明，因为这是一个复杂的问题，但是通过使用Observables，你可以用非常简单和易于理解的方式完成复杂的事情。这里是一个链式Web Service调用的例子，这些请求互相依赖，在线程池中运行第二批并行调用，然后在将结果返回给Observer之前，对数据进行合并和排序。为了更好的测量，我甚至在里面放置了一个过滤器。所有的业务逻辑都在下面这短短五行代码里面…

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
return cityDirectory.getUsCapitals()
    .flatMap(cityList -> Observable.from(cityList))
    .filter(city -> city.getPopulation() > 500,000)
    .flatMap(city -> weatherService.getCurrentWeather(city)) //each runs in paralle
    .toSortedList((cw1,cw2) -> cw1.getCityName().compare(cw2.getCityName()));
}
```

旋转时的缓存(或是其他情况)

既然这是一个加载的数据，那么我们可能需要将数据进行缓存，这样当我们旋转设备的时候，就不会触发再次调用全部web service的事件。一种实现的方式是保留Fragment，并且保存一个对Observable 的缓存的引用，就像下面这样：

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

    setRetainInstance(true);
    weatherObservable = weatherManager.getWeatherForLargeUsCapitals().cache();
}

public void onViewCreated(...) {
    super.onViewCreated(...)
    bind(weatherObservable).subscribe(this);
}
```

在旋转之后，订阅者的缓存实例就会立即发出和第一次请求相同的请求，防止真实的Web Service请求发生。

如果你想要避免缓存的Fragment(并且有很充足的理由去避免它)，我们可以通过使用AsyncSubject实现缓存。无论何时被订阅，AsyncSubject 都会重新发出最后的项目。或者我们可以使用BehaviorSubject获得最后的值和新值改变整个应用程序。

WeatherListFragment.java

```
public void onViewCreated() {
super.onViewCreated()
bind(weatherManager.getWeatherForLargeUsCapitals()).subscribe(this);
}
```

WeatherManager.java

```
public Observable<List<CityWeather>> getWeatherForLargeUsCapitals() {
if (weatherSubject == null) {
    weatherSubject = AsyncSubject.create();

    cityDirectory.getUsCapitals()
        .flatMap(cityList -> Observable.from(cityList))
        .filter(city -> city.getPopulation() > 500,000)
        .flatMap(city -> weatherService.getCurrentWeather(city))
        .toSortedList((cw1,cw2) -> cw1.getCityName().compare(cw2.getCityName()))
        .subscribe(weatherSubject);
```

```
}
return weatherSubject;
}
```

因为“缓存”是由Manager单独管理的，它不会与Fragment/Activity的周期绑定，并且在Activity/Fragment中将持续存在。如果你想强迫刷新基于以类似的方式来保留Fragment缓存实例的生命周期事件，你可以这样做：

```
public void onCreate() {
super.onCreate()
if (savedInstanceState == null) {
    weatherManager.invalidate(); //invalidate cache on fresh start
}
}
```

这件事情的伟大之处在于，它不像是Loaders，我们可以很灵活的缓则缓存很多Activity和Services中的结果。只需要去掉oncreate()中的invalidate()调用，并让你的Manager对象决定何时发出新的气象数据就可以了。可能是一个Timer，或者是用户定位改变，或者是其他任何时刻，这真的没关系。你现在可以控制什么时候如何去更新缓存和重新加载。并且当你的缓存策略发生改变的时候，Fragment和你的Manager对象之间的接口不需要进行改变。它只不过是一个 List的Observer。

可测试性

测试是我们想要实现干净、简单的最后一个挑战。(让我们忽略一个事实，在测试期间,我们可能想要模拟出实际的Web服务。这样做很简单，下面通过一个接口注入到那些依赖你可能已经正在做的标准模式中。)

幸运的是，Observables给我们一个简单的方式来将一个异步方法变成同步，你要做的就是使用toblocking()方法。我们看一个测试例子。

```
List results = getWeatherForLargeUsCapitals().toBlocking().first();
assertEquals(12, results.size());
```

就像这样，我们没有必要去使用Futures或者是CountDownLatchs让做一些脆弱的操作，比如线程睡眠或者是让我们的测试变得很复杂，我们的测试现在是简单、干净、可维护的。

结论

更新：我已经创建了一对简单的项目来演示AsyncTask风格和AsyncTaskLoader风格。

RxJava，你值得拥有。我们使用rx.Observable来替换AsyncTask和AsyncTaskLoader可实现更加强大和清晰的代码。使用RxJava Observables很快乐，而且我期待能够呈现更多的Android问题的解决方案。

原文链接

Replace AsyncTask and AsyncTaskLoader with rx.Observable – RxJava Android Patterns

上一篇 【凯子哥带你夯实应用层】新手必备的常用代码片段整理（二）
下一篇 【凯子哥带你夯实应用层】还在用XListView？试试更漂亮的AutoLoadListView吧！

我的同类文章

猜你在找

威哥全套Android开发课程【基础与UI技术】

Web无障碍——测试与工具

Android自动化测试第二季（提高篇）

Android开源项目实践之UI篇

Android自动化测试第三季

查看评论

2楼 麦田里的守望者-Rye 前天 10:37发表

谢谢分享，受益了

1楼 任晓天 2016-02-23 23:11发表

期待更多的rxJAVA的文章

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap