响应式编程入门(RxJava) (http://www.liuhaihua.cn/archives/533405.html)

zhuangli 发布于 2018-08-29 分类：Java (http://www.liuhaihua.cn/archives/category/ittechnology/java) / 编程技术 (http://www.liuhaihua.cn/archives/category/ittechnology)
阅读(261) 评论(0)

随着时间 (http://www.liuhaihua.cn/archives/tag/%e6%97%b6%e9%97%b4)的发展，编程领域不断地推出新的技术来尝试解决已有的问题，**响应式 (http://www.liuhaihua.cn/archives/tag/%e5%93%8d%e5%ba%94%e5%bc%8f)编程(流式编程)**正是近几年来非常流行的一个解决方案，如果我们关注一下业界动态，就会发现绝大多数的语言和框架都纷纷开始支持这一编程模型 (http://www.liuhaihua.cn/archives/tag/%e6%a8%a1%e5%9e%8b)：

- Java 8 => 引入Stream流，Observable 和 Observer 模式
- Spring 5 => 引入WebFlux，底层全面采用了响应式模型
- RxJava => 去年长期霸占git (http://www.liuhaihua.cn/archives/tag/git)hub最受欢迎的项目第一名

可以预见，响应式编程未来必将大规模的应用于开发 (http://www.liuhaihua.cn/archives/tag/%e5%bc%80%e5%8f%91)领域，阿里内部也已经开始了相关的改造，目前淘宝应用架构 (http://www.liuhaihua.cn/archives/tag/%e5%ba%94%e7%94%a8%e6%9e%b6%e6%9e%84)已经走上了流式架构升级之路，作为开发，响应式的编程范式还是很有必要掌握的，下文将基于RxJava 2.0给出相关概念的简单介绍和基本编程思路的讲解（基于《Learning Rx Java》一书前三章总结 (http://www.liuhaihua.cn/archives/tag/%e6%80%bb%e7%bb%93)）

基本思路

关于响应式编程（Reactive Programming, 下文简称RP）的定义，众说纷纭。维基百科将其定义为一种编程范式，ReactiveX将其定义为一种设计模式 (http://www.liuhaihua.cn/archives/tag/%e8%ae%be%e8%ae%a1%e6%a8%a1%e5%bc%8f)的强化（观察者模式），也有大牛认为RP只不过是已有各种的轮子的一个组装...有关RP的本质 (http://www.liuhaihua.cn/archives/tag/%e6%9c%ac%e8%b4%a8)，我们可以在文章 (http://www.liuhaihua.cn/archives/tag/%e6%96%87%e7%ab%a0)的最后进行简单的讨论，但从学习的角度而言，我认为最好的方式是将RP看做是一种面向事件和流的编程思想，如下：

Java推崇OOP的编程思想，因此对于Java程序员 (http://www.liuhaihua.cn/archives/tag/%e7%a8%8b%e5%ba%8f%e5%91%98)而言，程序就是各种对象的组合，编程就是控制对象状态和行为的一个过程。

RP推崇面向流的编程的思想，因此对于开发人员而言，无论是事件还是数据 (http://www.liuhaihua.cn/archives/tag/%e6%95%b0%e6%8d%ae)，全部都将以流的方式呈现，这些流时而并行，时而交叉，编程就是观察和调控这些流的一个过程。

从现实世界出发

在现实世界中有一个显而易见的物理现象，那就是 **一切都在运动(变化)** ，无论是交通、天气、人，即便是一块岩石，也会随着地球 (http://www.liuhaihua.cn/archives/tag/%e5%9c%b0%e7%90%83)的自转而运动。而不同物体的运动之间可能互不干扰，比如运动在不同道路上的车辆和行人，也可能出现交叉，比如在同一个十字路口相遇的行人和车辆。

回归到我们的编程当中，我们往往将程序抽象成多个过程，和现实世界一样，这些过程也在变化和运动，它们之间有时可以并行，有时会产生依赖（前置、后置条件），传统编程模型中，我们往往会采用多线程 (http://www.liuhaihua.cn/archives/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)或者异步的方式来处理这些过程，但这样的方式 **并不自然** 。

因此RP从现实世界进行抽象和采样，将程序分为了以下三种组成：

1. Observable：可被观察的事物，也就是事件和数据流
2. Observer：观察流的事物
3. Operator：操作符，对流进行连接和过滤等等操作的事物

让我们用一个最简单的例子来引入这三个概念：

```
Observable<String> myStrings =
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
    myStrings.map (http://www.liuhaihua.cn/archives/tag/map)(s -> s.length())
        .subscribe(s -> System.out.println(s));
```

复制代码 (http://www.liuhaihua.cn/archives/tag/%e4%bb%a3%e7%a0%81)

在上面的例子中， `myStrings` 就是Observable， `map(s -> s.length())` 就是Operator， `subscribe(s -> System.out.println(s))` 就是Observer。

这个例子将会把几个字符串先取长度，然后再逐个输出。

Observable

Observable简单来说，就是流，在RxJava中只有三个最核心的API (http://www.liuhaihua.cn/archives/tag/api):

```
onNext()
onComplete()
onError()
```

基本上所有的流都是这三种方法的一个包装。

创建

1. 使用 `Observable.create()`

```
Observable<String> myStrings = Observable.create(emitter -> {
    emitter.onNext("apple");
    emitter.onNext("bear");
    emitter.onNext("change");
    emitter.onComplete();
});
```
复制代码

2. 使用 `Observable.just()`

```
Observable<String> myStrings =
        Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon");
```
复制代码

注意这种方法创建的元素数量必须是有限的

3. 从其他数据源创建，例如 `Observable.fromIterable()` ， `Observable.range()`

Hot & Cold Observables

Cold的流生产的数据是静态的，好比一张CD，无论什么时候，无论什么人来听，都可以听到完整的内容。

```
Observable<String> source =
      Observable.just("Alpha","Beta","Gamma","Delta","Epsilon");
//first observer
source.subscribe(s -> System.out.println("Observer 1 Received: " + s));
//second observer
source.subscribe(s -> System.out.println("Observer 2 Received: " + s));
```
复制代码

上面两个subscribe注册的observer将会得到完全相同的一串流

Hot的流产生的数据是动态的，好比收音机电台，错过了播放的时段，过去的数据就取不到了。直接创建Hot流需要用到Listener，官方给出了一个JavaFx的例子，但并不合适放在这里。

事实上，通常通过Connectab (http://www.liuhaihua.cn/archives/tag/tab)leObservable的方式来将一个Cold的流转换成一个Hot的流：

```
ConnectableObservable<String> source =
   Observable.just("Alpha","Beta","Gamma","Delta","Epsilon")
   .publish();
   //Set up observer 1
   source.subscribe(s -> System.out.println("Observer 1: " + s));
  //Set up observer 2
  source.map(String::length)
    .subscribe(i -> System.out.println("Observer 2: " + i));
//Fire!
  source.connect();
```
复制代码

通过 `publish()` 方法可以创建一个 `ConnectableObservable` ，然后通过 `connect()` 方法启动流的传输，这时source将会把所有的数据都传递给两个observer。此后如果再给source注册新的Observer，将不会得到任何数据，因为Hot流不允许数据被重复消费。

Observers

观察者本身是比较简单的结构，主要功能由调用方自己去实现。

可以通过实现 `Observer` 接口的方式去创建一个观察者，当然更常见的case是通过lambda (http://www.liuhaihua.cn/archives/tag/lambda)表达式来创建一个观察者，就像之前用到的例子一样，这里不详细展开了。

注册的方式是通过调用Observerbale的 `subscribe` 方法。

Operators

只创建流和观察者并不能有什么太大的作用，大多时候我们需要通过操作符(Operator)来对流进行各种各样的操作才能使RP变得有实际意义。

RxJava中提供了非常丰富的操作符，大致分为以下几类：

1. 创建操作符，用于创建流，刚才我们已经用到了其中的几个，比如Create, Just, Range和Interval
2. 变换操作符，用于将一个流变换成另一种形式，比如Buffer（将流中的元素打包转换成集合）、Map（对流中的每个元素执行一个函数），Window（将流中的元素拆分成不同的窗口再发射）
3. 过滤操作符，过滤掉流中的部分数据以获取指定的数据元素，比如Filter、First、Distinct
4. 组合操作符，将多个流融合成一个流，比如And、Then、Merge、Zip (http://www.liuhaihua.cn/archives/tag/ip)
5. 条件/算术/聚合/转换操作符 … 起到各种运算辅助作用
6. 自定义操作符，由用户自己创建

如果把每个操作符都展开讲一遍，差不多就能出一本书了，可见操作符提供的功能之丰富。下文只展开一些和 **背压** 有关的操作符。

背压

所谓背压，是指 **异步环境** 中，生产者的速度大于消费者速度时，消费者反过来控制生产者速度的策略（也称为回压），这是一种流控的方式，可以更有效的利用资源、同时防止错误的雪崩。

为了实现背压策略，可以使用以下几种操作符

1. Throttling 节流类

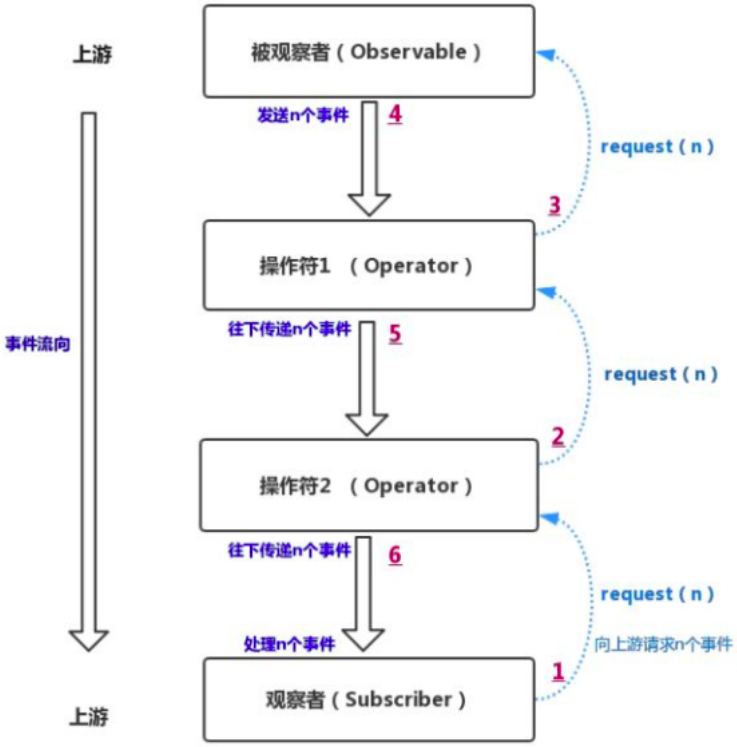
   通过操作符来调节Observable发射消息的速度，比如使用 `sample()` 来定期采样，并发 (http://www.liuhaihua.cn/archives/tag/%e5%b9%b6%e5%8f%91)出最后一个数据，或者使用 `throttleWithTimeout()` 来丢弃超时的数据。但这样做会丢弃一部分数据，最终消费者拿不到完整的消息。

2. Buffer & Window 缓冲和窗口类

   使用缓冲 `buffer()` 和窗口 `window()` 暂存那些发送速度过快的 (http://www.liuhaihua.cn/archives/tag/%e5%bf%ab%e7%9a%84)消息，等到消息发送速度下降的时候再释放它们。这主要应用于Observable发送速率不均匀的场景。

除了使用操作符之外，还有一种方式可以实现背压，那就是响应式拉取（Reactive pull）。

通过在Observer中调用 `request(n)` 方法，可以实现由消费者来反控生产者的生产，也就是说只有当消费者请求的时候，生产者才去产生数据，当消费者消费完了数据再去请求新的数据，这样就不会出现生产速度过快的情况了，如下图



但是这样做需要上游的被观察者能够对request请求做出响应，这时候又可以用到几个操作符来控制Observable的行为：

1. `onBackPressureBuffer`

   为Observable发出来的数据制作缓存 (http://www.liuhaihua.cn/archives/tag/%e7%bc%93%e5%ad%98)，产生的数据先放在缓存中，每当有request请求过来时，就从缓存里取出对应数量的事件返回。

2. `onBackPressureDrop`

   命令Observable丢弃后来的时间，直到Subscriber再次调用request(n)方法的时候，就发送给该subscriber调用时间以后的n个时间。

背压策略是一个值得深入研究和探讨的领域，基于消费者消息的回压让 **动态限流 (http://www.liuhaihua.cn/archives/tag/%e9%99%90%e6%b5%81)**、**断路** 成为可能，也因为有了背压的感知，应用有机会做到 **动态的缩扩容** 。

思考：为什么需要RP

RP的基本概念介绍的差不多了，现在需要思考一下为什么需要RP，在服务端 (http://www.liuhaihua.cn/archives/tag/%e6%9c%8d%e5%8a%a1%e7%ab%af)怎么应用RP，以及它所能够带来的优势。

首先，有关RP的本质，我觉得它就是一个异步编程的轮子，用观察者模式的API把异步编程的过程变得更加清晰和简单，这就好比go使用CSP来简化并发操作一样，底层其实还是对已有技术的封装。

那么问题在于，为什么要封装异步编程，使用异步编程能带来什么好处，为了解决这个问题我们又需要回归原点。

如果要问服务端最大的性能瓶颈是什么，那答案一定是IO (http://www.liuhaihua.cn/archives/tag/io)，因为处理一个请求的过程中最耗时的部分就是等待IO，而等待就会造成阻塞，所以如果要提升性能，就不能写出阻塞的代码来。

如何才能让代码不阻塞？以Java服务端来说，传统的处理方式无外乎以下两种：

1. 使用Thread，把业务代码和IO代码放到不同的线程 (http://www.liuhaihua.cn/archives/tag/%e7%ba%bf%e7%a8%8b)里跑。但你需要面对并发问题（资源竞争），同时根据之前对Java线程调度的分析我们知道这样对CPU的资源利用率并不高效（上下文切换消耗比较大）。
2. 使用异步回调，你可以用Callback或者Future来实现，但需要自己去实现调度逻辑，同时Callback这样的模式写出来的代码是不好理解的，有可能出现Callbcak Hell。

所以最终为了解决性能瓶颈，RP给出的办法就是：

提供一个优秀的异步处理框架，同时简化编写异步代码的流程，最终实现减少阻塞，提升性能的大目标。

原文

https (http://www.liuhaihua.cn/archives/tag/https)://juejin.im/post/5b8552e86fb9a019ea0208c8

上一篇
Java — Hotspot虚拟机调优与GC垃圾回收策略
(http://www.liuhaihua.cn/archives/533383.html)

下一篇
Bulk 异常引发的 Elasticsearch 内存泄漏
(http://www.liuhaihua.cn/archives/533407.html)