Java 平台反应式编程（Reactive Programming）入门

CSDN技术头条 发表于 CSDN技术头条

反应式编程（Reactive Programming）对有些人来说可能相对陌生一点。反应式编程是一套完整的编程体系，既有其指导思想，又有相应的框架和库的支持，并且在生产环境中有大量实际的应用。在支持度方面，既有大公司参与实践，也有强大的开源社区的支持。

反应式编程出现的时间并不短，不过在最近的一段时间内，它得到了很大的关注。这主要体现在主流编程平台和框架增强了对它的支持，使它得到了更多的受众，同时也反映了其在开发中的价值。

就 Java 平台来说，几个突出的事件包括：Java 9中把反应式流规范以 java.util.concurrent.Flow 类的方式添加到了标准库中；Spring 5对反应式编程模型提供了内置支持，并增加了新的 WebFlux 模块来支持反应式 Web 应用的开发。在前端开发中，Angular 框架也内置使用了 RxJS。

反应式编程所涵盖的内容很多。本 Chat 作为反应式编程的入门，主要侧重在 Java 平台。与其他编程范式一样，反应式编程要求开发人员改变其固有的思维模式，以不同的角度来看问题。对于熟悉了传统面向对象编程范式的人来说，这样的思想转变可能并不那么容易。

反应式编程在解决某些问题时有其先天的优势。在对应用性能要求很高的今天，反应式编程有更大的用武之地。作为开发人员来说，根据项目的需求和特征，选择最适合的编程模型可以达到事半功倍的效果。这也是本 Chat 的出发点。

需要注意的是，反应式编程相关的术语目前并没有非常统一的翻译方法，本文中尽量使用较为常见的译法或英文原文。

概述

在讨论反应式编程之前，首先必须要提到的是《反应式宣言（The Reactive Manifesto）》。反应式宣言中对反应式系统（Reactive Systems）的特征进行了定义，有如下四个：

- 及时响应（Responsive）：系统在尽可能的情况下及时响应请求。
- 有韧性（Resilient）：系统在出现失败时仍然可以及时响应。
- 有弹性（Elastic）：在不同的负载下，系统仍然保持及时响应。
- 消息驱动（Message Driven）：系统使用异步消息传递来确定不同组件之间的边界，并确保松散耦合、隔离和位置透明性。

这四个特征互相关联和影响。及时响应是核心价值，是反应式系统所追求的目标。有韧性和有弹性是反应式系统的外在表现形式，通过它们才能实现及时响应这个核心价值。消息驱动则是实现手段。

反应式编程的重要概念之一是负压（back-pressure），是系统在负载过大时的重要反馈手段。当一个组件的负载过大时，可能导致该组件崩溃。为了避免组件失败，它应该通过负压来通知其上游组件减少负载。负压可能会一直级联往上传递，最终到达用户处，进而影响响应的及时性。

这是在系统整体无法满足过量需求时的自我保护手段，可以保证系统的韧性，不会出现失败的情况。此时系统应该通过增加资源等方式来做出调整。

反应式流

口很简单。在其 Java API 中，只定义了4个接口。在下面介绍 Java 9 的 Flow 类时会具体介绍这4个接口。

数据传递方式

随着反应式流的出现，我们可以对 Java 平台上常见的几种数据传递方式做一下总结和比较。

1. 直接的方法调用。数据使用者直接调用提供者的方法来获取数据。这种方式是同步的，调用者在方法返回前会被阻塞。调用者和提供者之间的耦合最紧。每次方法调用只能返回一个数据（虽然可以使用集合类来返回多个数据，但从概念上来说，集合类仍然只能视为一个数据）。
2. 使用 Iterable。Iterable 表示一个可以被枚举的数据的集合，通常用不同的集合类型来表示，如 List、Set 和 Map 等。Iterable 定义了可以对集合的数据所进行的操作。这些操作是同步的。Iterable 所包含的数据数量是有限的。
3. 使用 Future。Future 表示的是一个可以在未来获取的结果，由一个异步操作来负责给出这个结果。在获取到 Future 对象之后，可以使用 get 方法来获取到所需要的结果。虽然计算的过程是异步的，get 方法使用时仍然是阻塞的。Future 只能表示一个结果。
4. 反应式流。反应式流表示的是异步无阻塞的数据流，其中包含的元素数量可能是无限的。

Java 8 的 java.util.stream.Stream 可以看成是对 Iterable 的一种扩展，可以包含无限元素。Stream 同时又有一部分反应式流实现的特征，主要体现在其流式接口（Fluent interface）上，也可以做并行处理。不过 Stream 缺乏最根本的对负压的支持。

Future 和 CompletableFuture

Java中的 Future 把异步操作进行了抽象，但是只解决了一半的问题。虽然 Future 所表示的计算是异步的，但是对计算结果的获取仍然是同步阻塞的。Future 原本的设计思路是：当需要执行耗时的计算时，提交该计算任务到 ExecutorService，并得到一个 Future 对象作为返回值。

接着就可以执行其他任务，然后再使用之前得到的 Future 对象来获取到所需的计算的结果值，再继续下面的计算。这样的设计思路有一个突出的问题，那就是在实际中很难找到一个合适的时机来获取 Future 对象的计算结果。因为 get 方法是阻塞的，如果调用早了，主线程仍然会被阻塞；如果调用晚了，在某种程度上降低了并发的效率。

除此之外，如果需要在代码的不同部分之间传递计算的结果，需要把 Future 对象在不同的对象之间进行传递，也增加了系统的耦合性。

Java 8 的 CompletableFuture 的出现解决了上面提到的 Future 的问题。而解决的办法是允许异步操作进行级联。比如有一个服务用来生成报表，另外一个服务用来发送电子邮件。

生成报表的服务返回的是 CompletableFuture 对象，只需要通过 thenApply 或 thenRun 就可以调用发送电子邮件的服务，得到的结果是另外一个 CompletableFuture 对象。在使用 CompletableFuture 时，不需要考虑获取异步操作结果的时机，只需要以声明式的方式定义出对结果的操作即可。这也避免了不必要的 CompletableFuture 对象传递。

CompletableFuture 仍然只能表示一个结果。如果把 CompletableFuture 的思路进一步扩展，就是反应式流解决问题的思路。在实际中，异步服务通常都是处理数据流。比如上面提到的发送电子邮件的服务，会接受来自不同源的数据。

反应式流的一个重要目标是确保流的消费者不会因为负载过重而崩溃。

在具体介绍反应式流之前，我们先看一下反应式流会带来的思维方式的转变。

流式思考（Thinking in Streams）

环三种控制结构来完成不同的行为。

开发人员在程序中编写的是执行的步骤；以数据为中心侧重的是数据在不同组件的流动。开发人员在程序中编写的是对数据变化的声明式反应。

我们通过一个具体的示例来说明以流为中心的思维模式。在电子商务网站中都有购物车这个功能。用户在购物车界面可以看到所有已经添加的商品，还可以进一步修改商品的数量。

当数量更新之后，购物车界面上要显示更新后的订单总价。按照一般的面向对象的思路，我们会有一个订单对象，里面包含了当前全部的商品，并有一个属性来表示订单的总价。当商品数量更新之后，订单对象中的商品被更新，同时需要重新调用计算总价的方法来更新总价属性值。

下面是按照命令式思路的基本 Java 代码。updateQty 用来更新订单商品数量，calculateTotal 用来计算总价。典型的运行流程是先调用 updateQty，再调用 calculateTotal。

```
class Order {
    public void updateQty(LineItem item, int qty) {
        item.qty = qty;        this.calculateTotal();
    }    private void calculateTotal() {        this.total = ...
    }
}
```

如果采用事件驱动的方式，比如典型的 Web 界面中，情况并不会好太多。我们可以为不同的动作创建相应的事件。每个事件有自己的类型和相应的数据（payload）。比如，商品数量更新事件的数据中会包含商品的 ID 和新的数量。

系统对不同的事件有不同的处理方式。商品数量更新事件其实是对之前的 updateQty 方法调用的封装。引入事件的好处是可以把调用者和处理者进行解耦。直接使用方法调用 order.updateQty() 的方式，把调用者和处理者紧密耦合在一起。

在引入了事件之后，原来的一个步骤被划分成3个小步骤：

1. 调用者创建事件并发布。
2. 事件中间件负责传递事件，通常采用事件总线（Event Bus）来完成。
3. 处理者接收到事件进行处理。

事件驱动的方式增加了一定的灵活性，那对数据的处理仍然不是很自然。再回到最初的问题，问题的本质在于订单的总价是会随着商品的数量而改变的。当商品的数量变化时，订单对象本身并不会对该变化作出反应来更新自身的总价属性。如果以反应式的思维模式，那会是不一样的情况。

在以流为中心是思维模式中，值可能产生变化的变量都是一个流。流中的元素代表了变量在不同时刻的值。如果一个变量的值的变化会引起另外一个变量的变化，则把前一个变量所表示的流作为它所能引起变化另外一个变量对应的流的上游。我们可以把每个商品的数量看成一个流。

当数量更新时，流中会产生一个新的元素。流中的元素可能是“1 -> 2 -> 3 -> 2”，也可能是其他合法的序列。每个元素表示了用户的一次操作的结果。订单的总价也是一个流，它的元素表示了由于商品数量变化所对应的总价。总价对应的流中的元素是根据所有商品数量流的元素来产生的。

每当任意一个商品数量中产生了新的元素，都会在总价流中产生一个对应的新元素。对总价的计算逻辑使用流的运算符来表示。

接着我们来具体看看怎么以反应式流的方式来实现购物车。为了更加直观的展示，这里我使用的是 JavaScript 上的反应式库 RxJS。下面的代码是一个简单的购物车页面。页面上有3个固定的商品。

每个商品有对应的 input 元素。input 元素的 data-price 属性表明了商品的单价。函数 calculateItemPrice 的作用是根据一个 input 元素来计算其对应商品的价格，也就是单价乘以数量。

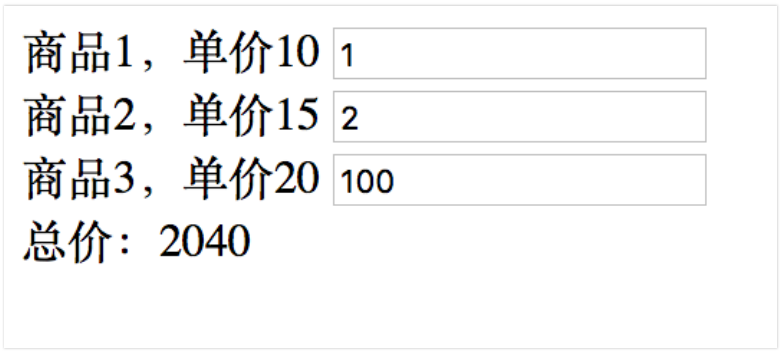
对于事件对象，可以通过 target 属性获取到对应的 input 元素，再使用 calculateItemPrice 进行计算。在经过 map 操作符之后，流的元素变成了每个商品的价格。流中的初始元素是数量为 1 时的价格。

Rx.Observable.combineLatest 方法的作用是把每个 input 所对应的流进行合并，从每个流中获取最新的元素，组合成一个数组，作为它所对应的流的元素。我们只需要把数组的值进行累加，就得到了总价。

```
<!DOCTYPE html><html>
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>使用反应式编程的购物车示例</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.5/lodash.js"></scr
    <script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/5.5.6/Rx.js"></script>
    <script>
      let calculateItemPrice = node => $(node).val() * parseFloat($(node).data('price'));

      $(function() {
        Rx.Observable.combineLatest(
          $('input').map((index, node) => Rx.Observable.fromEvent(node, 'change')
              .map(e => calculateItemPrice(e.target))
              .startWith(calculateItemPrice(node))).get())
        .map(values => _.sum(values))
        .subscribe(total => $('#total').html(total))
      });
    </script>
  </head>
  <body>
    <div>
      <span>商品1，单价10</span>
      <input type="number" value="1" data-price="10">
    </div>
    <div>
      <span>商品2，单价15</span>
      <input type="number" value="1" data-price="15">
    </div>
    <div>
      <span>商品3，单价20</span>
      <input type="number" value="1" data-price="20">
    </div>
    <div>总价：<span id="total"></span></div>
  </body></html>
```

下图是运行起来的效果图。



从上述代码可以看到，反应式流采用了与传统编程不同的思路，更加注重的是数据层面上的抽象，淡化了状态。

下面我们结合 Java 9 中的 java.util.concurrent.Flow 类来说明反应式流规范。Java 9 中的 Flow 只是简单的把反应式流规范的4个接口整合到了一个类中。

Publisher

顾名思义，Publisher 是数据的发布者。Publisher 接口只有一个方法 subscribe 来添加数据的订阅者，也就是下面的 Subscriber。

Future 和 CompletableFuture

Subscriber

Subscriber 是数据的订阅者。Subscriber 接口有4个方法，都是作为不同事件的处理器。在订阅者成功订阅到发布者之后，其 onSubscribe(Subscription s) 方法会被调用。Subscription 表示的是当前的订阅关系。

当订阅成功后，可以使用 Subscription 的 request(long n) 方法来请求发布者发布 n 条数据。发布者可能产生3种不同的消息通知，分别对应 Subscriber 的另外3个回调方法。

- 数据通知：对应 onNext 方法，表示发布者产生的数据。
- 错误通知：对应 onError 方法，表示发布者产生了错误。
- 结束通知：对应 onComplete 方法，表示发布者已经完成了所有数据的发布。

在上述3种通知中，错误通知和结束通知都是终结通知，也就是在终结通知之后，不会再有其他通知产生。

Subscription

Subscription 表示的是一个订阅关系。除了之前提到的 request 方法之外，还有 cancel 方法用来取消订阅。需要注意的是，在 cancel 方法调用之后，发布者仍然有可能继续发布通知。但订阅最终会被取消。

Processor

Processor 表示的一种特殊的对象，既是生产者，又是订阅者。

Publisher 只有在收到请求之后，才会产生数据。这就保证了订阅者可以根据自己的处理能力，确定要 Publisher 产生的数据量，这就是负压的实现方式。

Reactor

反应式流规范所提供的 API 是很简单的，并不能满足日常开发的需求。反应式流的价值在于对流以声明式的方式进行的各种操作，以及不同流之间的整合。这些都需要通过第三方库来完成。

目前 Java 平台上主流的反应式库有两个，分别是 Netflix 维护的 RxJava 和 Pivotal 维护的 Reactor。RxJava 是 Java 平台反应式编程的鼻祖。反应式流规范在很大程度上借鉴了 RxJava 的理念。

由于 RxJava 的产生早于反应式流规范，与规范的兼容性并不是特别好。

Reactor 是一个完全基于反应式流规范的全新实现，也是 Spring 5 默认的反应式框架。

Reactor 的两个最核心的类是 Flux 和 Mono。Reactor 采用了两个不同的类来表示流。Flux 表示的包含0到无限个元素的流，而 Mono 则表示最多一个元素的流。虽然从逻辑上来说，Mono 表示的流都可以用 Flux 来表示，这样的区分使得很多操作的语义更容易理解。

比如对一个 Flux 进行 reduce 操作的结果是一个 Mono。而对一个 Mono 进行 repeat 操作得到的是一个 Flux。

第一类是创建 Flux 和 Mono 的静态方法。比如 Flux 的 fromArray、fromIterable 和 fromStream 方法分别从数组、Iterable 和 Stream 中创建 Flux 。interval 可以根据时间间隔生成从0开始的递增序列。Mono 还可以从 Runnable、Callable 和 CompletableFuture 中创建。

```
Flux.fromArray(new String[] {"a", "b", "c"})
  .subscribe(System.out::println);

Mono.fromFuture(CompletableFuture.completedFuture("Hello World"))
  .subscribe(System.out::println);
```

第二类是缓冲上游流中的元素的操作符，包括 buffer、bufferTimeout、bufferWhen、bufferUntil、bufferWhile、window、windowTimeout、windowWhen、windowUntil 和 windowWhile 等。

buffer 等方法按照元素数量和/或间隔时间来收集元素，把原始的Flux<T>转换成 `Flux<List<T>>` 。window 等方法与 buffer 作用类似，只不过是把原始的 `Flux<T>` 转换成 `Flux<Flux<T>>` 。

使用 bufferTimeout 可以用简洁的方式解决一些复杂的问题。比如，有一个执行批量处理的服务，我们需要在请求数量达到某个阈值时马上执行批量处理，或者给定的时间间隔过去之后也要执行批量处理。这样既可以在负载高时降低批量处理的压力，又可以在负载低时保证及时性。

在下面的代码中，Flux.interval 用来生成递增的序列，其中第一个 Flux 的时间间隔是100毫秒，第二个 Flux 的时间间隔是10毫秒，并有一秒的延迟。两个 Flux 表示的流被 merge 合并。

bufferTimeout 的设置是最多10个元素和最长500毫秒。由于生成的流是无限的，我们使用 take(3) 来取前面3个元素。toStream() 是把 Flux 转换成 Java 8 的 Stream ，这样可以阻止主线程退出直到流中全部元素被消费。在最初的 500 毫秒，只有第一个 Flux 产生数据，因此得到的 List 中只包含5个元素。

在接着的 500 毫秒，由于时间精确度的原因，在 List 中仍然是可能有来自第二个 Flux 的元素。第三个 List 则包含10个元素。

```
Flux.merge(    Flux.interval(Duration.ofMillis(100)).map(v -> "a" + v),    Flux.interval(
).bufferTimeout(10, Duration.ofMillis(500))    .take(3)    .toStream()    .forEach(System
```

第三类是收集操作符，包括collect、collectList、collectMap、collectMultimap 和 collectSortedList 等，用来把流中的元素收集到不同的集合对象中。

第四类是流合并操作符，包括 concat 和 merge 等。concat 和 merge 都可以合并多个流，不同之处在于 concat 会在完全消费前一个流之后，才开始消费下一个流；而 merge 则同时消费所有流，来自不同流的元素会交织在一起。

第五类是流转换合并操作符，包括 concatMap 和 flatMap。这些操作符都把原始流的每个元素转换成一个新的流，再合并这些新生成的流。在合并流时，concatMap 的语义与 concat 相似，而 flatMap 的语义与 merge 相似。下面代码的输出结果是：0、0、1、0、1、2。

```
Flux.just(1, 2, 3).concatMap(v ->     Flux.interval(Duration.ofMillis(100)).take(v))    .
```

第六类是对流元素进行处理的操作符。这一类操作符种类很多，也比较容易理解。比如对流中元素进行转换的 map，对元素进行过滤的 filter，去掉重复元素的 distinct，从流中抽取给定数量元素的 take 和跳过流中给定数量元素的 skip。

WebFlux

WebFlux 是 Spring 5 中新引入的开发反应式 Web 应用的模块。该模块中包含了对反应式 HTTP、服务器推送事件（Server-sent Events）和 WebSocket 的客户端和服务器端的支持。在服务器端，WebFlux 支持两种不同的编程模型：第一种是 Spring MVC 中使用的基于 Java 注解的方式；第二种是基于 Java 8 的 Lambda 表达式的函数式编程模型。

基于 Java 注解的编程模型与之前的 Spring MVC 的注解方式并没有太大的区别，容易上手。函数式编程模型功能强大，也更灵活，可以实现动态路由等复杂场景，相应的也更难上手。

与传统 Spring MVC 的区别在于，WebFlux 的请求和响应使用的都是 Flux 或 Mono 对象。一般的 REST API 使用 Mono 来表示请求和响应对象；服务器推送事件使用 Flux 来表示从服务器端推送的事件流；WebSocket 则使用 Flux 来表示客户端和服务器之间的双向数据传递。

为了最大程度的发挥反应式流和负压的作用，WebFlux 应用的各个部分都应该是支持反应式的，也就是说各个部分都应该是异步非阻塞的。要做到这一点，需要其他的库提供支持，主要是与外部系统和服务整合的部分。

比如在数据访问层，可以通过 Spring Data 的反应式支持来访问不同类型的数据源。当然这也需要底层驱动的支持。越来越多的数据源驱动已经提供了对反应式流规范的支持，还有很多开源库可以使用。

小结

反应式编程在解决某些问题时有其独到之处，可以作为传统编程范式的良好补充，也可以从头开发一个完整的反应式应用。要了解反应式编程，最重要的是思维模式的转变。这不可能一蹴而就，只能通过大量的实战开发来获取相关经验。大胆在你的下一个项目中使用反应式编程吧，肯定会有不一样的体验。

原文发布于微信公众号 - CSDN技术头条（CSDN_Tech）
原文发表时间：2018-03-20
本文参与腾讯云自媒体分享计划，欢迎正在阅读的你也加入，一起分享。
发表于 2018-03-26

Java

相关文章

来自专栏 用户2442861的专栏

Python基础学习笔记之（一）（华工大神）

前段时间参加微软的windows Azure云计算的一个小培训，其中Python被用的还是蛮多的。另外，一些大公司如Google（实现web爬虫...

126 1

来自专栏 互联网杂技

如何去了解JavaScript引擎的工作原理

1. 什么是JavaScript解析引擎？ 简单地说，JavaScript解析引擎就是能够“读懂”JavaScript代码，并准确地给出代码运行结果的一段程序。...

366 7

来自专栏 ThoughtWorks

在项目中透明地引入特性开关

之前曾经推荐过崔立强的《使用功能开关更好地实现持续部署》，介绍Feature Toggle的实践。北京办公室的孟宇现在对这个问题有了新的思考，当我们抛却Spri...

343 6

来自专栏 点滴积累

geotrellis使用（十）缓冲区分析以及多种类型要素栅格化

目录 前言 缓冲区分析 多种类型要素栅格化 总结 参考链接 一、前言 上两篇文章介绍了如何使用Geotrellis进行矢量数据栅格化以及栅格渲染，...

355 8

来自专栏 蓝天

代码规范：换行对齐问题

对一于单行代码过长时，采取换行，这个没有什么可争议的，主要焦点在新的一行从哪开始，通常有两派，一派就是如上述两段代码所示，另一派则采用如下规范：

72 2

来自专栏 何俊林

Android Multimedia框架总结（八）Stagefright框架之Awesom…

前言：前面一篇分析了mediaplayerservice及MediaPlayer中的CS模型，但是对于如何能把数据解析出来，渲染到最终的SurfaceView上...

235 9

来自专栏 编舟记

Clojure集合管道函数练习

TDD讨论组里的申导最近在B站直播了Martin Fowler的经典文章Refactoring with Loops and Collection Pipeli...

来自专栏 玩转全栈

flutter全局数据共享通知方案

让我们先抛开Flutter这个平台说话，如果让你实现数据共享，你能想到的基础方案有哪些。

1.6K 18

来自专栏 FreeBuf

又见卡死图，竟然一个“小黑点”就能干掉QQ？

前一阵子QQ群里流行了一个东西，一段话加上一个黑点（表情），点击之后QQ就会卡死。(文内链接请点击“阅读原文”查看)

124 5

来自专栏 Web 开发

知呼前端工程师面试题目

第三题的答案（我的答案有点问题，等弄好了再上传 要想完美，先出主体，通过负margin+双容器）

80 0