# 响应式编程总览

## 响应式编程总览

In computing, reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. - [Reactive programming - Wikipedia](https://en.wikipedia.org/wiki/Reactive_programming)

在上述响应式编程（以下简称RP）的定义中，除了异步编程，还包含两个重要的关键词：

Data streams: 即数据流，分为静态数据流（比如数组，文件）和动态数据流（比如事件流，日志流）两种。基于数据流模型，RP得以提供一套统一的Stream风格的数据处理接口。和Java 8中的Stream API相比，RP API除了支持静态数据流，还支持动态数据流，并且允许复用和同时接入多个订阅者。

The propagation of change: 变化传播，简单来说就是以一个数据流为输入，经过一连串操作转化为另一个数据流，然后分发给各个订阅者的过程。这就有点像函数式编程中的组合函数，将多个函数串联起来，把一组输入数据转化为格式迥异的输出数据。

一个容易混淆的概念是响应式设计，虽然它的名字中也包含了“响应式”三个字，但其实和RP完全是两码事。响应式设计是指网页能够自动调整布局和样式以适配不同尺寸的屏幕，属于网站设计的范畴，而RP是一种关注系统可响应性，面向数据流的编程思想或者说编程框架。

### 特性

从本质上说，RP是一种异步编程框架，和其他框架相比，RP至少包含了以下三个特性：

描述而非执行：在你最终调用subscribe()方法之前，从发布端到订阅端，没有任何事会发生。就好比无论多长的水管，只要水龙头不打开，水管里的水就不会流动。为了提高描述能力，RP提供了比Stream丰富的多的多的API，比如buffer(), merge(), onErrorMap()等。

提高吞吐量: 类似于HTTP/2中的连接复用，RP通过线程复用来提高吞吐量。在传统的Servlet容器中，每来一个请求就会发起一个线程进行处理。受限于机器硬件资源，单台服务器所能支撑的线程数是存在一个上限的，假设为T，那么应用同时能处理的请求数（吞吐量）必然也不会超过T。但对于一个使用[Spring 5](http://emacoo.cn/backend/spring5-overview/)开发的RP应用，如果运行在像Netty这样的异步容器中，无论有多少个请求，用于处理请求的线程数是相对固定的，因此最大吞吐量就有可能超过T。

背压（Backpressure）支持：简单来说，背压就是一种反馈机制。在一般的Push模型中，发布者既不知道也不关心订阅者的处理速度，当数据的发布速度超过处理速度时，需要订阅者自己决定是缓存还是丢弃。如果使用RP，决定权就交回给发布者，订阅者只需要根据自己的处理能力问发布者请求相应数量的数据。你可能会问这不就是Pull模型吗？其实是不同的。在Pull模型中，订阅者每次处理完数据，都要重新发起一次请求拉取新的数据，而使用背压，订阅者只需要发起一次请求，就能连续不断的重复请求数据。

### 适用场景

了解了RP的这些特性，你可能已经猜想到RP有哪些适用场景了。一般来说，RP适用于高并发、带延迟操作的场景，比如以下这些情况（的组合）：

一次请求涉及多次外部服务调用

非可靠的网络传输

高并发下的消息处理

弹性计算网络

### 代价

Every coin has two sides.

和任何框架一样，有优势必然就有劣势。RP的两个比较大的问题是：

虽然复用线程有助于提高吞吐量，但一旦在某个回调函数中线程被卡住，那么这个线程上所有的请求都会被阻塞，最严重的情况，整个应用会被拖垮。

难以调试。由于RP强大的描述能力，在一个典型的RP应用中，大部分代码都是以链式表达式的形式出现，比如flux.map(String::toUpperCase).doOnNext(s -> LOG.info("UC String {}", s)).next().subscribe()，一旦出错，你将很难定位到具体是哪个环节出了问题。所幸的是，RP框架一般都会提供一些工具方法来辅助进行调试。

2 Reactor实战

为了帮助你理解上面说的一些概念，下面我就通过几个测试用例，演示RP的两个关键特性：提高吞吐量和背压。完整的代码可参见我GitHub上的[示例工程](https://github.com/emac/demo-reactor" \t "_blank)。

|  |  |
| --- | --- |
| 提高吞吐量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 | @Test public void testImperative() throws InterruptedException {  \_runInParallel(CONCURRENT\_SIZE, () -> {  ImperativeRestaurantRepository.INSTANCE.insert(load);  }); }  private void \_runInParallel(int nThreads, Runnable task) throws InterruptedException {  ExecutorService executorService = Executors.newFixedThreadPool(nThreads);  for (int i = 0; i < nThreads; i++) {  executorService.submit(task);  }  executorService.shutdown();  executorService.awaitTermination(1, TimeUnit.MINUTES); }  @Test public void testReactive() throws InterruptedException {  CountDownLatch latch = new CountDownLatch(CONCURRENT\_SIZE);  for (int i = 0; i < CONCURRENT\_SIZE; i++) {  ReactiveRestaurantRepository.INSTANCE.insert(load).subscribe(s -> {  }, e -> latch.countDown(), latch::countDown);  }  latch.await(); } |

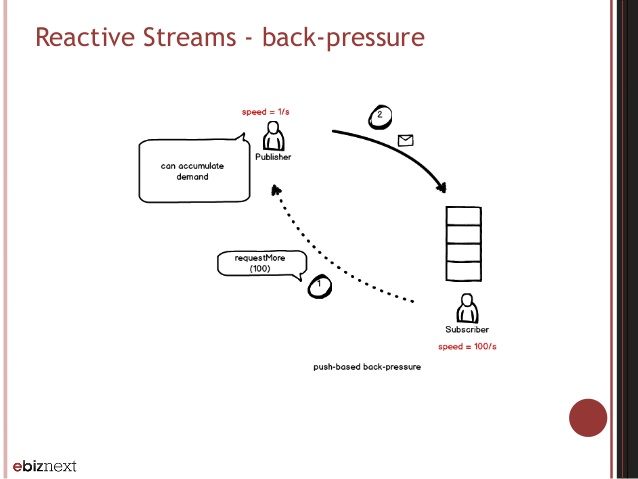
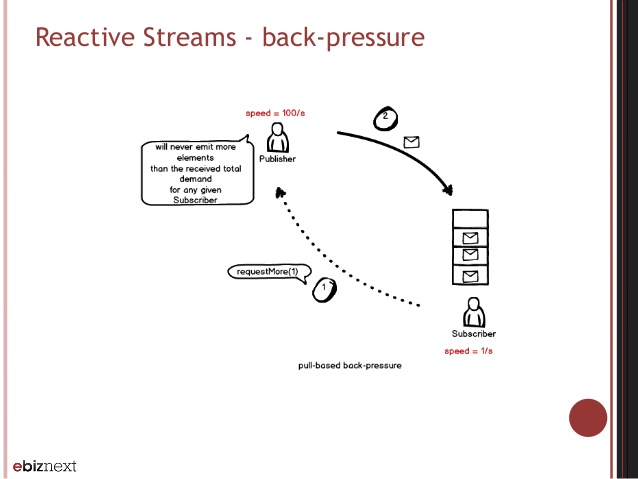
用例解读：

第一个测试用例使用的是多线程+MongoDB Driver，同时起100个线程，每个线程往MongoDB中插入10000条数据，总共100万条数据，平均用时15秒左右。

第二个测试用例使用的是Reactor+MongoDB Reactive Streams Driver，同样是插入100万条数据，平均用时不到10秒，吞吐量提高了50%！

背压

在演示测试用例之前，先看两张图，帮助你更形象的理解什么是背压。



图片出处：[Dataflow and simplified reactive programming](https://www.slideshare.net/StephaneManciot/psug-52-dataflow-and-simplified-reactive-programming-with-akkastreams)

两张图乍一看没啥区别，但其实是完全两种不同的背压策略。第一张图，发布速度（100/s）远大于订阅速度（1/s），但由于背压的关系，发布者严格按照订阅者的请求数量发送数据。第二张图，发布速度（1/s）小于订阅速度（100/s），当订阅者请求100个数据时，发布者会积满所需个数的数据再开始发送。可以看到，通过背压机制，发布者可以根据各个订阅者的能力动态调整发布速度。

|  |  |
| --- | --- |
| 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 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 | @BeforeEach public void beforeEach() {  // initialize publisher  AtomicInteger count = new AtomicInteger();  timerPublisher = Flux.create(s ->  new Timer().schedule(new TimerTask() {  @Override  public void run() {  s.next(count.getAndIncrement());  if (count.get() == 10) {  s.complete();  }  }  }, 100, 100)  ); }  @Test public void testNormal() throws InterruptedException {  CountDownLatch latch = new CountDownLatch(1);  timerPublisher  .subscribe(r -> System.out.println("Continuous consuming " + r),  e -> latch.countDown(),  latch::countDown);  latch.await(); }  @Test public void testBackpressure() throws InterruptedException {  CountDownLatch latch = new CountDownLatch(1);  AtomicReference<Subscription> timerSubscription = new AtomicReference<>();  Subscriber<Integer> subscriber = new BaseSubscriber<Integer>() {  @Override  protected void hookOnSubscribe(Subscription subscription) {  timerSubscription.set(subscription);  }   @Override  protected void hookOnNext(Integer value) {  System.out.println("consuming " + value);  }   @Override  protected void hookOnComplete() {  latch.countDown();  }   @Override  protected void hookOnError(Throwable throwable) {  latch.countDown();  }  };  timerPublisher.onBackpressureDrop().subscribe(subscriber);  new Timer().schedule(new TimerTask() {  @Override  public void run() {  timerSubscription.get().request(1);  }  }, 100, 200);  latch.await(); } |

用例解读：

第一个测试用例演示了在理想情况下，即订阅者的处理速度能够跟上发布者的发布速度（以100ms为间隔产生10个数字），控制台从0打印到9，一共10个数字，和发布端一致。

第二个测试用例故意调慢了订阅者的处理速度（每200ms处理一个数字），同时发布者采用了Drop的背压策略，结果控制台只打印了一半的数字（0，2，4，6，8），另外一半的数字由于背压的原因被发布者Drop掉了，并没有发给订阅者。

## 通俗解释什么是响应式编程

1. 响应式编程(Reactive Programming 或称反应式编程)是一种流行的编程方法，编写代码是基于对变化的反应。它的灵感来自于我们的日常生活，也即我们如何采取行动以及与他人沟通。  
     
   我们在执行日常生活活动时，我们会尽可能多任务，但大脑无法处理多任务，不管我们如何努力去做。我们人类实现多任务的唯一办法是在时间线上在任务之间切换。事实上，我们总是切换任务，即使我们没有意识到它。  
     
   例如，要执行一个任务：在星巴克喝一杯咖啡饮料，你需要发出一个命令，等待它准备好，然后接受你的饮料。当你在等待的时候，你很可能会找到别的事情做。这是最简单的执行任务的反应(响应)形式，你会在你等待来自咖啡师的“响应”时做别的事情，当你的咖啡已经准备好后，会叫你的名字时。  
     
   响应编程能够简化编程，它依赖于事件，代码运行的顺序不是代码行的顺序，而是和一个以上的事件有关，这些事件发生是以随着时间的推移的序列。我们把这一系列事件称为“流”。  
     
   何为事件？例如，你知道某个名人总是在发送有趣微博，每次他推发一条微博我们可以称之为一个“事件”。如果你看看这位名人微博系列，你会发现其实是一个随着时间的推移（一系列的事件）发生的一序列的“事件”。  
     
   响应式编程就是因为我们得“响应”这些事件而得以命名。例如，想象一下，你在等待某人发送一个很酷商品的促销码，当这条促销码发出时，你会立即响应，去购买这个很酷的商品。这正是什么响应性编程的原理。  
     
   为了能够对事件作出反应，我们必须要监督它。 如果我们没有监听的情况下，我们永远不会知道什么时候它会有事件反应。 在微博上，可以设置监测微博的事件，我们遵循简并设置我们的电话，每次她发微博事件时就会第一时间通知我们。   
     
   在响应式编程中，监视事件被称为侦听或订阅该事件。 这一点其实很相似订阅简讯。 当您订阅Web上发布的新闻，您得提供您的电子邮件地址，每次有新的时事通讯时候，您的电子邮件地址将得到该新闻的一个副本。 同样，我们使用某个函数订阅事件流，每当有一个新的事件时，流将激活这个函数，以使我们的代码能够对事件作出响应。 在这个比喻中，新闻通讯平台是事件流，电子报的每一个新闻是一个事件，你的电子邮件是您使用订阅事件流的函数。  
     
   现在想象一个动态的通讯，您可以选择主题，并只会发送匹配主题的新闻项目。 你可根据自己的喜好过滤的通讯新闻，而这一点我们可以在事件流做的一样好。 想像你已经使用不同的电子邮件地址订阅多个通讯，以及后来决定你想要的通讯的所有新闻发送到一个新的单电子邮件地址。 简单可以做的是设置通讯新闻发送到新的电子邮件地址的转发邮件规则。 我们可以用事件流同样做到。  
     
   将事件流可以和通常数组比较。 他们其实很相似。 数组是在空间值的序列，而事件流是随时间的值的序列。 在响应式编程中，所有函数操作都可以针对一个数组阵列上 - 比如过滤，reduce，mapping，结合，管道 - 也可以在事件流完成！ 我们可以过滤的事件流，reduce合并事件流的值，将事件流映射到另一个另外一个，结合组合事件流，将一个流的输入输出到另一个。

# 第二章 Reactor：构建响应式快速数据应用程序的开发库



Reactor是第二代响应式开发库，支持构建基于JVM的非阻塞式应用程序，遵循[Reactive Streams规范](https://github.com/reactive-streams/reactive-streams-jvm" \t "_blank)。

## 主要特性

### REACTIVE CORE

Reactor**完全非阻塞式**通信基础框架，支持高效按需管理。它可以用Java 8 functional API、*Completable Future*、*Stream* 和 *Duration*直接交互。

### TYPED [0|1|N] 序列

Reactor提供了**两种响应式AP组合** [Flux [N]](http://projectreactor.io/core/docs/api/reactor/core/publisher/Flux.html)和 [Mono [0|1]](http://projectreactor.io/core/docs/api/reactor/core/publisher/Mono.html) 实现了丰富的[Reactive扩展](http://reactivex.io/)。Reactor还通过**Bus**扩展支持可扩展的内存路由。

### 非阻塞IO

适用**微服务**架构， Reactor IO提供了**带反向压力（backpressure）的网络引擎**，支持HTTP（包括Websockets）、TCP和UDP。非阻塞IO还完全支持响应式编码和解码。

### 高效消息传输

Reactor Processors、Operators 和 Timers 可以**持续提供高传输吞吐率**，每秒传输的消息数量可达到10的百万次方。**低内存开销**也是Reactor让大家青睐的原因之一。不仅如此，Reactor还是[正在进行的研究项目](http://github.com/reactor/reactive-streams-commons" \t "_blank)的一部分，该项目旨在对传输流程做进一步优化。

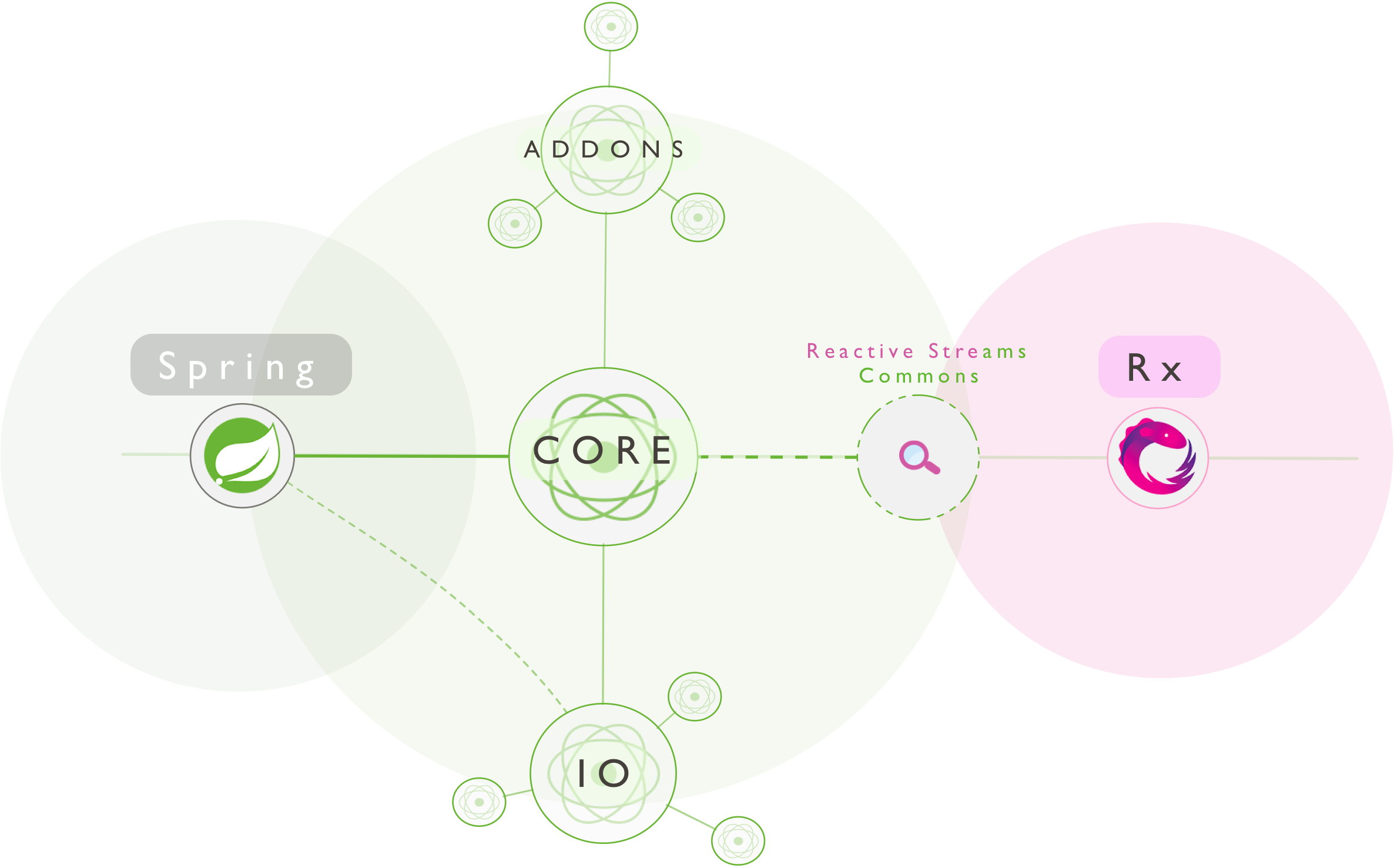
### 易于使用

作为响应式引擎和SPI，Reactor Core 和 IO模块可以根据具体用例构建响应式数据流，并且还支持与[Spring](http://spring.io/" \t "_blank)、[RxJava](http://reactivex.io/" \t "_blank)、[Akka Streams](http://doc.akka.io/docs/akka-stream-and-http-experimental/current/java.html" \t "_blank)、[Ratpack](http://ratpack.io/" \t "_blank)等框架集成。作为响应式API，Reactor框架模块还提供了像组合（composition）和事件发布订阅这样的丰富特性。

### 延迟守护者

水平扩展是降低延迟、解决微服务速度慢的一个重要工具。通过支持异步结构的水平扩展，Cloud Native app可以获得更好的效果。为了支持这样的设计，Reactor提供了非阻塞式和带有反向压力的嵌入式解决方案，包括本地和远程单播、多播消息通信，或者TCP、HTTP、UDP客户端和服务器。

## 项目结构



### Reactor Core

APP和框架响应式支持基础，受 [Mono](http://projectreactor.io/core/docs/api/reactor/core/publisher/Mono.html)（1元素）和[Flux](http://projectreactor.io/core/docs/api/reactor/core/publisher/Flux.html" \t "_blank) （n元素）启发提供了丰富的响应式API扩展：

* <https://github.com/reactor/reactor-core>
* 文档：[http://projectreactor.io/core/docs/reference](http://projectreactor.io/core/docs/reference" \t "_blank)
* API：[http://projectreactor.io/core/docs/api](http://projectreactor.io/core/docs/api" \t "_blank)

### Reactor IO

2.5.0.BUILD-SNAPSHPOT

* <https://github.com/reactor/reactor-io>:
* 文档：[http://projectreactor.io/io/docs/reference](http://projectreactor.io/io/docs/reference" \t "_blank)
* API：[http://projectreactor.io/io/docs/api](http://projectreactor.io/io/docs/api" \t "_blank)
* 子模块：
  + reactor-ipc：内存和InterProcessCommunication抽象。
  + reactor-aeron：高效的单播或多播通信。
  + reactor-codec：JSON、压缩、Kryo、Protobuf编解码器。
  + reactor-netty：TCP和HTTP客户端及服务器。

### Reactor Addons

2.5.0.BUILD-SNAPSHPOT

* <https://github.com/reactor/reactor-addons>
* 文档： <http://projectreactor.io/ext/docs/reference>
* API：[http://projectreactor.io/ext/docs/api](http://projectreactor.io/ext/docs/api" \t "_blank)
* 子模块：
  + reactor-alloc
  + reactor-bus
  + reactor-logback
  + reactor-pylon
  + reactor-pipes

### Reactor Incubator

2.5.0.BUILD-SNAPSHPOT

* <https://github.com/reactor/reactor-incubator>
* 子模块：
  + reactor-amqp
  + reactor-chronicle
  + reactor-groovy
  + reactor-net-0mq

### Reactive Streams Commons

在为Reactive Streams设计高效并发操作符的工作中提出了公用基础代码项目，即[Reactive Streams Commons](https://github.com/reactor/reactive-streams-commons" \t "_blank)。Reactor完全符合RSC设计理念，在reactor-core稳定版API中直接使用了RSC。Reactive Streams Commons旨在为大家提供了一种高效流处理解决方案，因此从设计开始就确保不依赖其它框架。

#### 注册

可以随时[加入](https://support.springsource.com/spring_committer_signup" \t "_blank)、fork、讨论和PR。在项目的路线图中就采取了协作方式，我们欢迎新点子、简化建议、文档、反馈。

#### Maven

最新的快照和发布文件在 repo.spring.io 中提供。稳定版会同步到Maven中央仓库。把这个仓库加入到Gradle构建，指定下面的URL：

XHTML



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | repositories {        //maven { url 'http://repo.spring.io/libs-release' }        //maven { url 'http://repo.spring.io/libs-milestone' }        maven { url 'http://repo.spring.io/libs-snapshot' }        mavenCentral()      }        dependencies {        // Reactor Core        compile "io.projectreactor:reactor-core:2.5.0.BUILD-SNAPSHOT"          // Reactor Aeron        // compile "io.projectreactor:reactor-aeron:2.5.0.BUILD-SNAPSHOT"           // Reactor Pipes        // compile "io.projectreactor:reactor-pipes:2.5.0.BUILD-SNAPSHOT"           // Reactor Netty4         // compile "io.projectreactor:reactor-netty:2.5.0.BUILD-SNAPSHOT"           // Reactor Codecs (Jackson, Kryo...)         // compile "io.projectreactor:reactor-codec:2.5.0.BUILD-SNAPSHOT"        } |

## 开始

将Reactor添加到现有项目，只需把配置好的文件添加到项目中。像Fluxand Mono 这样的Reactor基础组件已经包含在reactor-core 中。附加组件由reactor-aeron、reactor-netty、reactor-pipes、reactor-bus和reactor-pylon提供。

build.gradle



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | repositories {      // maven { url 'http://repo.spring.io/release' } // 1      // maven { url 'http://repo.spring.io/milestone' } // 2      maven { url 'http://repo.spring.io/snapshot' }  // 3  }    dependencies {      //Core stuff (Flux, Mono, Schedulers...)      compile "io.projectreactor:reactor-core:2.5.0.M3"        //Everything Routing (Event Bus)      //compile "io.projectreactor:reactor-bus:2.5.0.BUILD-SNAPSHOT"        //Netty TCP/HTTP/UDP client/servers      //compile "io.projectreactor:reactor-netty:2.5.0.BUILD-SNAPSHOT"      //compile "io.netty:netty-all:4.0.34.final"        //Reactive Aeron client/servers      //compile "io.projectreactor:reactor-aeron:2.5.0.BUILD-SNAPSHOT"  } |

1. 取消注释使用发布版本。
2. 取消注释使用里程碑版本。
3. 取消注释使用快照版本。

添加reactor-core到项目后，可以使用Flux、 Mono和availableProcessor创建事件序列执行带有反向压力（backpressure）的异步操作。

## 开发资源

* 文档
  + [Reactor Core](http://projectreactor.io/core/docs/reference/)
  + [Reactor IO](http://projectreactor.io/io/docs/reference/)
  + [Reactor Addons](http://projectreactor.io/ext/docs/reference/)
  + [Reactor 2.0（早期版本）](http://projectreactor.io/old/reference/)
* 资源
  + [ReactiveX](http://reactivex.io/)
  + [Reactive Streams](http://www.reactive-streams.org/)
  + [Spring指南](https://spring.io/guides)
* 社区
  + [Github](https://github.com/reactor/reactor)
  + [Twitter](https://twitter.com/projectreactor)
  + [Gitter讨论组](https://gitter.im/reactor/reactor)
  + [Google小组](https://groups.google.com/forum/#!forum/reactor-framework)

## 协议

Reactor遵循[Apache 2.0](http://www.apache.org/licenses/LICENSE-2.0.html" \t "_blank)开源协议发布。

官方网站：<http://projectreactor.io/>  
开源地址：[https://github.com/reactor](https://github.com/reactor" \t "_blank)

# Spring 5 响应式编程

近年来，响应式编程在开发者社区和客户中很受欢迎，由于其以声明的方式构建应用程序的能力，而不是强制，形成更加敏感和有弹性的应用。Spring 5 将反应系统纳入其核心框架的事实已经显示出向声明式编程的范式转变。

响应式编程管理数据生产者和消费者之间的异步数据流，它们需要以流畅的方式对数据进行响应。所以，响应式编程都是异步和事件驱动的流畅应用程序，需要少量的线程进行缩放。

响应式编程很难构建基于线程的架构，由于在基于共享可变状态、线程和锁的应用程序扩展过程中涉及到高度复杂性。

在响应式编程的上下文中，“在流中有数据时，所有事物都是流并以流畅方式行为。”

## 为什么是响应式编程

高层次的抽象与响应式编程导致了代码可读性的提高，因此开发人员可以主要关注定义业务逻辑的事件的相互依存性。

在高度并发的环境中，响应模式自然地适合于消息处理，这是一个常见的企业用例。

由于执行反压力的特性，响应式方法最适合控制生产者和消费者之间的流量，这将有助于避免内存不足。

对于一个或几个线程，IO绑定任务可以通过异步和非阻塞方式执行，而且不阻塞当前线程。

在高交互和实时应用程序或任何操作/事件时，都可能触发多个连接子系统的通知，在这种情况下响应式编程可以更有效的进行管理。

## 用于响应式编程实现的理想案例

大量的交易处理服务，如银行部门。

大型在线购物应用程序的通知服务，如亚马逊。

股票价格同时变动的股票交易业务。

## 响应流 (Reactive Streams)

“响应流”定义一个 API 规范，其中包含一组最小的接口，这些接口公开了定义非阻塞反压力的数据流的操作和实体的方法。

随着反压力的引入，响应流允许用户控制来自出版商的数据交换速率。

响应流 API java.util.concurrent.flow 已正式成为 java 9 的一部分。

响应流主要用作互操作层。

## Spring 5 提供的响应式编程

Spring-Web-Reactive 模块和 Spring MVC 都支持相同的 @Controller 编程，但 Spring-Web-Reactive 是在 Reactive 和非阻塞引擎上执行的。  
Spring-Web-Reactive 模块和 Spring MVC 共享许多通用的算法，但 Spring-Web-Reactive 模块已经重新定义了许多 Spring MVC 契约，例如 HandlerMapping和HandlerAdapter，以使它们变得异步和非阻塞，以及启用响应式 HTTP 请求和响应（以 RouterFunction 和 HandlerFunction 的形式）。  
除了现有的 RestTemplate 外，新的响应式 WebClient 也在 Spring 5 中引入。

支持响应式编程的 HTTP 客户端（例如 Reactor、Netty、Undertow）已经采用了一系列响应式的 ClientHttpRequest 和 ClientHttpResponse 抽象，它们将请求和响应体作为 Flux <DataBuffer> 公开，并且在读写端提供全面的背压支持。  
Spring 5 Framework 引入了 Reactor，作为 Reactive Streams 规范的实现。  
Reactor 是用于在 JVM 上构建非阻塞应用程序的下一代 Reactive 库。  
Reactor 扩展了基本的 Reactive Streams Publisher 契约，并定义了 Flux 和 Mono API 类型，分别为 0..N 和 0..1 的数据序列提供声明性操作。

Spring Web Reactive 使用 Servlet 3.1 提供了非阻塞 I/O，并运行在 Servlet 3.1 容器上。  
Spring WebFlux 提供了两种编程模型可供选择。

注解式控制器：这些与 Spring MVC 相同，在使用一些 Spring-Web 模块提供的附加注解的情况下。Spring MVC 和 WebFlux 控制器都支持 Reactive 返回类型。另外，WebFlux 还支持 Reactive @RequestBody 参数。

函数式编程模型：基于 lambda 的轻量级小型库，用于给出路由和处理请求的实用程序。

## Spring Web Reactive vs. Spring Web MVC

Spring 5 包含了 Spring Web Reactive（在 spring-web-reactive 模块下）和 Spring Web MVC（在 spring-webmvc 模块下）。

虽然 Spring Web Reactive 和 Spring Web MVC 模块共享了很多算法，但是由于 Spring Web Reactive 能够在响应式的非阻塞 Reactive Streams HTTP 适配器层上运行，因此它们之间不共享代码。

Spring MVC 执行需要 Servlet 容器，而 Spring Web Reactive 也可以在非 Servlet 运行时上运行，比如在 Netty 和 Undertow 上运行。

如果对具有轻量级、函数式的 Web 框架的非阻塞 Web 栈的必要要求，并与 Java 8 lambda 或 Kotlin一起使用，则应该考虑从 Spring MVC 应用程序切换到 Spring Web Reactive。

## 响应式编程的基本结构

这是一个 2.0.0 M5 版本的  pom.xml ，依赖于 WebFiux。

<parent>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-parent</artifactId>

    <version>2.0.0.M5</version>

</parent>

<dependencies>

    <dependency>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-webflux</artifactId>                               </dependency>

</dependencies>

### 传统方法 VS. 响应方法

在传统方法中，执行将被阻塞，并将一直等到您的服务执行完成为止。在下面的代码中，在第一个 print 语句之后，程序执行将被阻塞并等待服务执行完成。在服务执行完成后，将恢复程序执行，并执行第二个 print 语句。

@GetMapping("/traditional")

public List < Product > getAllProducts() {

    System.out.println("Traditional way started");

    List < Product > products = prodService.getProducts("traditional");

    System.out.println("Traditional way completed");

    return products;

}

在响应方法中，程序执行将继续，而不用等待服务执行的完成。在下面的代码中，在第一个 print 语句之后，第二个 print 语句将以非阻塞的方式执行，而无需等待服务执行的完成。Flux stream 将随着产品数据的可用性而被填充。

@GetMapping(value = "/reactive", .TEXT\_EVENT\_STREAM\_VALUE)

public Flux < Product > getAll() {

    System.out.println("Reactive way using Flux started");

    Flux < Product > fluxProducts = prodService.getProductsStream("Flux");

    System.out.println("Reactive way using Flux completed");

    return fluxProducts;

}

### Reactive Web Client

除了现有的 RestTemplate 之外，Spring 5 还引入了 Reactive WebClient。  
 ClientHttpRequest 和 ClientHttpResponse 抽象将请求和响应主体使用 Flux <DataBuffer> 公开，并在读和写端提供完全背压支持。  
 来自 Spring Core 的编码器和解码器抽象也用在客户端，用于将字节序的 Flus 序列化或发序列化成类型对象。  
 下面是一个 Reactive WebClient 的示例，它调用终端并接收及处理 Reactive Stream Flux 对象。

@GetMapping("/accounts/{id}/alerts")

public Flux < Alert > getAccountAlerts(@PathVariable Long id) {

    WebClient webClient = new WebClient(new ReactorClientHttpConnector());

    return this.repository.getAccount(id).flatMap(account -> webClient.perform(get("/alerts/{key}", account.getKey())).extract(bodyStream(Alert.class)));

}

## Spring 5 的局限

对 Reactive 应用程序进行故障诊断有点困难，并且你有可能在解决问题的同时，意外地阻塞了已经引入的代码。

大多数传统的基于 Java 的集成库仍然是阻塞的。

可用于 Reactive 数据存储的选项是非常有限的，少数 NoSQL 数据库（如 MongoDB）除外。

Spring Security 仍然不被支持。

# Spring 5响应式Web框架实战（上）

引子：被誉为“中国大数据第一人”的涂子沛先生在其成名作《数据之巅》里提到，摩尔定律、社交媒体、数据挖掘是大数据的三大成因。IBM的研究称，整个人类文明所获得的全部数据中，有90%是过去两年内产生的。在此背景下，包括NoSQL，Hadoop, Spark, Storm, Kylin在内的大批新技术应运而生。其中以[RxJava](https://github.com/ReactiveX/RxJava" \t "_blank)和[Reactor](http://projectreactor.io/" \t "_blank)为代表的响应式（Reactive）编程技术针对的就是经典的大数据4V定义（Volume，Variety，Velocity，Value）中的Velocity，即高并发问题，而在即将发布的Spring 5中，也引入了响应式编程的支持。在接下来的几周，我会围绕响应式编程分三期与你分享我的一些学习心得。本篇是第三篇，通过一个简单的Spring 5示例应用，探一探即将于下月底发布的Spring 5的究竟。

## 回顾

通过前两篇的介绍，相信你对响应式编程和Spring 5已经有了一个初步的了解。下面我将以一个简单的Spring 5应用为例，介绍如何使用Spring 5快速搭建一个响应式Web应用（以下简称RP应用）。

## 实战

### 环境准备

首先，从GitHub下载我的这个示例应用，地址是

<https://github.com/emac/spring5-features-demo>。

然后，从MongoDB[官网](https://www.mongodb.com/download-center#community)下载最新版本的MongoDB，然后在命令行下运行mongod &启动服务。

现在，可以先试着跑一下项目中自带的测试用例。

./gradlew clean build

### 依赖介绍

接下来，看一下这个示例应用里的和响应式编程相关的依赖。

compile('org.springframework.boot:spring-boot-starter-webflux')

compile('org.springframework.boot:spring-boot-starter-data-mongodb-reactive')

testCompile('io.projectreactor.addons:reactor-test')

spring-boot-starter-webflux: 启用Spring 5的RP（Reactive Programming）支持，这是使用Spring 5开发RP应用的必要条件，就好比spring-boot-starter-web之于传统的Spring MVC应用。

spring-boot-starter-data-mongodb-reactive: Spring 5中新引入的针对MongoDB的Reactive Data扩展库，允许通过统一的RP风格的API操作MongoDB。

io.projectreactor.addons:reactor-test: [Reactor](http://projectreactor.io/)（Spring 5默认使用的RP框架）提供的官方测试工具库。

### 示例代码

不知道你是否还记得，在本系列第一篇[【Spring 5】响应式Web框架前瞻](http://emacoo.cn/backend/spring5-overview/" \t "_blank)里提到，Spring 5提供了Spring MVC注解和Router Functions两种方式来编写RP应用。本篇我就先用大家最熟悉的MVC注解来展示如何编写一个最简单的RP Controller。

@RestController

public class RestaurantController {

/\*\*

\* 扩展ReactiveCrudRepository接口，提供基本的CRUD操作

\*/

private final RestaurantRepository restaurantRepository;

/\*\*

\* spring-boot-starter-data-mongodb-reactive提供的通用模板

\*/

private final ReactiveMongoTemplate reactiveMongoTemplate;

public RestaurantController(RestaurantRepository restaurantRepository, ReactiveMongoTemplate reactiveMongoTemplate) {

this.restaurantRepository = restaurantRepository;

this.reactiveMongoTemplate = reactiveMongoTemplate;

}

@GetMapping("/reactive/restaurants")

public Flux<Restaurant> findAll() {

return restaurantRepository.findAll();

}

@GetMapping("/reactive/restaurants/{id}")

public Mono<Restaurant> get(@PathVariable String id) {

return restaurantRepository.findById(id);

}

@PostMapping("/reactive/restaurants")

public Flux<Restaurant> create(@RequestBody Flux<Restaurant> restaurants) {

return restaurants

.buffer(10000)

.flatMap(rs -> reactiveMongoTemplate.insert(rs, Restaurant.class));

}

@DeleteMapping("/reactive/restaurants/{id}")

public Mono<Void> delete(@PathVariable String id) {

return restaurantRepository.deleteById(id);

}

}

可以看到，实现一个RP Controller和一个普通的Controller是非常类似的，最核心的区别是，优先使用RP中最基础的两种数据类型，Flux（对应多值）和Mono（单值），尤其是方法的参数和返回值。即便是空返回值，也应封装为Mono<Void>。这样做的目的是，使得应用能够以一种统一的符合RP规范的方式处理数据，最理想的情况是从最底层的数据库（或者其他系统外部调用），到最上层的Controller层，所有数据都不落地，经由各种Flux和Mono铺设的“管道”，直供调用端。就像农夫山泉那句著名的广告词，我们不生产水，我们只是大自然的搬运工。

### 单元测试

和非RP应用的单元测试相比，RP应用的单元测试主要是使用了一个Spring 5新引入的测试工具类，WebTestClient，专门用于测试RP应用。

@RunWith(SpringRunner.class)

@SpringBootTest

public class RestaurantControllerTests {

@Test

public void testNormal() throws InterruptedException {

// start from scratch

restaurantRepository.deleteAll().block();

// prepare

WebTestClient webClient = WebTestClient.bindToController(new RestaurantController(restaurantRepository, reactiveMongoTemplate)).build();

Restaurant[] restaurants = IntStream.range(0, 100)

.mapToObj(String::valueOf)

.map(s -> new Restaurant(s, s, s))

.toArray(Restaurant[]::new);

// create

webClient.post().uri("/reactive/restaurants")

.accept(MediaType.APPLICATION\_JSON\_UTF8)

.syncBody(restaurants)

.exchange()

.expectStatus().isOk()

.expectHeader().contentType(MediaType.APPLICATION\_JSON\_UTF8)

.expectBodyList(Restaurant.class)

.hasSize(100)

.consumeWith(rs -> Flux.fromIterable(rs)

.log()

.subscribe(r1 -> {

// get

webClient.get()

.uri("/reactive/restaurants/{id}", r1.getId())

.accept(MediaType.APPLICATION\_JSON\_UTF8)

.exchange()

.expectStatus().isOk()

.expectHeader().contentType(MediaType.APPLICATION\_JSON\_UTF8)

.expectBody(Restaurant.class)

.consumeWith(r2 -> Assert.assertEquals(r1, r2));

})

);

}

}

创建WebTestClient实例时，首先要绑定一下待测试的RP Controller。可以看到，和业务类一样，编写RP应用的单元测试，同样也是数据不落地的流式风格。

在示例应用中可以找到更多的单元测试。

# Spring 5】响应式Web框架实战（下）

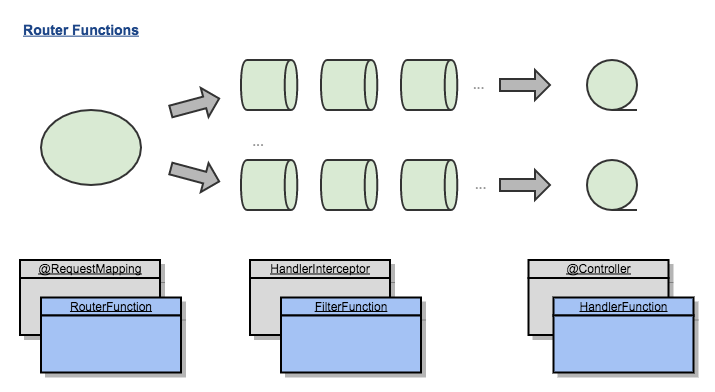
引子：被誉为“中国大数据第一人”的涂子沛先生在其成名作《数据之巅》里提到，摩尔定律、社交媒体、数据挖掘是大数据的三大成因。IBM的研究称，整个人类文明所获得的全部数据中，有90%是过去两年内产生的。在此背景下，包括NoSQL，Hadoop, Spark, Storm, Kylin在内的大批新技术应运而生。其中以[RxJava](https://github.com/ReactiveX/RxJava" \t "_blank)和[Reactor](http://projectreactor.io/" \t "_blank)为代表的响应式（Reactive）编程技术针对的就是经典的大数据4V定义（Volume，Variety，Velocity，Value）中的Velocity，即高并发问题，而在即将发布的Spring 5中，也引入了响应式编程的支持。在接下来的几周，我会围绕响应式编程分三期与你分享我的一些学习心得。本篇是第三篇（下），通过一个简单的Spring 5示例应用，探一探即将于下月底发布的Spring 5的究竟。

## 回顾

[上篇](http://emacoo.cn/backend/spring5-reactive-tutorial/)介绍了如何使用Spring MVC注解实现一个响应式Web应用（以下简称RP应用），本篇接着介绍另一种实现方式——Router Functions。

## 实战

### Router Functions



Router Functions是Spring 5新引入的一套Reactive风格（基于Flux和Mono）的函数式接口，主要包括RouterFunction，HandlerFunction和HandlerFilterFunction，分别对应Spring MVC中的@RequestMapping，@Controller和HandlerInterceptor（或者Servlet规范中的Filter）。

和Router Functions搭配使用的是两个新的请求/响应模型，ServerRequest和ServerResponse，这两个模型同样提供了Reactive风格的接口。

### 示例代码

#### 自定义RouterFunction和HandlerFilterFunction

@Configuration

public class RestaurantServer implements CommandLineRunner {

@Autowired

private RestaurantHandler restaurantHandler;

/\*\*

\* 注册自定义RouterFunction

\*/

@Bean

public RouterFunction<ServerResponse> restaurantRouter() {

RouterFunction<ServerResponse> router = route(GET("/reactive/restaurants").and(accept(APPLICATION\_JSON\_UTF8)), restaurantHandler::findAll)

.andRoute(GET("/reactive/delay/restaurants").and(accept(APPLICATION\_JSON\_UTF8)), restaurantHandler::findAllDelay)

.andRoute(GET("/reactive/restaurants/{id}").and(accept(APPLICATION\_JSON\_UTF8)), restaurantHandler::get)

.andRoute(POST("/reactive/restaurants").and(accept(APPLICATION\_JSON\_UTF8)).and(contentType(APPLICATION\_JSON\_UTF8)), restaurantHandler::create)

.andRoute(DELETE("/reactive/restaurants/{id}").and(accept(APPLICATION\_JSON\_UTF8)), restaurantHandler::delete)

// 注册自定义HandlerFilterFunction

.filter((request, next) -> {

if (HttpMethod.PUT.equals(request.method())) {

return ServerResponse.status(HttpStatus.BAD\_REQUEST).build();

}

return next.handle(request);

});

return router;

}

@Override

public void run(String... args) throws Exception {

RouterFunction<ServerResponse> router = restaurantRouter();

// 转化为通用的Reactive HttpHandler

HttpHandler httpHandler = toHttpHandler(router);

// 适配成Netty Server所需的Handler

ReactorHttpHandlerAdapter httpAdapter = new ReactorHttpHandlerAdapter(httpHandler);

// 创建Netty Server

HttpServer server = HttpServer.create("localhost", 9090);

// 注册Handler并启动Netty Server

server.newHandler(httpAdapter).block();

}

}

可以看到，使用Router Functions实现RP应用时，你需要自己创建和管理容器，也就是说Spring 5并没有针对Router Functions提供IoC支持，这是Router Functions和Spring MVC相比最大的不同。除此之外，你需要通过RouterFunction的API（而不是注解）来配置路由表和过滤器。对于简单的应用，这样做问题不大，但对于上规模的应用，就会导致两个问题：1）Router的定义越来越庞大；2）由于URI和Handler分开定义，路由表的维护成本越来越高。那为什么Spring 5会选择这种方式定义Router呢？接着往下看。

#### 自定义HandlerFunction

@Component

public class RestaurantHandler {

/\*\*

\* 扩展ReactiveCrudRepository接口，提供基本的CRUD操作

\*/

private final RestaurantRepository restaurantRepository;

/\*\*

\* spring-boot-starter-data-mongodb-reactive提供的通用模板

\*/

private final ReactiveMongoTemplate reactiveMongoTemplate;

public RestaurantHandler(RestaurantRepository restaurantRepository, ReactiveMongoTemplate reactiveMongoTemplate) {

this.restaurantRepository = restaurantRepository;

this.reactiveMongoTemplate = reactiveMongoTemplate;

}

public Mono<ServerResponse> findAll(ServerRequest request) {

Flux<Restaurant> result = restaurantRepository.findAll();

return ok().contentType(APPLICATION\_JSON\_UTF8).body(result, Restaurant.class);

}

public Mono<ServerResponse> findAllDelay(ServerRequest request) {

Flux<Restaurant> result = restaurantRepository.findAll().delayElements(Duration.ofSeconds(1));

return ok().contentType(APPLICATION\_JSON\_UTF8).body(result, Restaurant.class);

}

public Mono<ServerResponse> get(ServerRequest request) {

String id = request.pathVariable("id");

Mono<Restaurant> result = restaurantRepository.findById(id);

return ok().contentType(APPLICATION\_JSON\_UTF8).body(result, Restaurant.class);

}

public Mono<ServerResponse> create(ServerRequest request) {

Flux<Restaurant> restaurants = request.bodyToFlux(Restaurant.class);

Flux<Restaurant> result = restaurants

.buffer(10000)

.flatMap(rs -> reactiveMongoTemplate.insert(rs, Restaurant.class));

return ok().contentType(APPLICATION\_JSON\_UTF8).body(result, Restaurant.class);

}

public Mono<ServerResponse> delete(ServerRequest request) {

String id = request.pathVariable("id");

Mono<Void> result = restaurantRepository.deleteById(id);

return ok().contentType(APPLICATION\_JSON\_UTF8).build(result);

}

}

对比[上篇](http://emacoo.cn/backend/spring5-reactive-tutorial/" \t "_blank)的RestaurantController，由于去除了路由信息，RestaurantHandler变得非常函数化，可以说就是一组相关的HandlerFunction的集合，同时各个方法的可复用性也大为提升。这就回答了上一小节提出的疑问，即以牺牲可维护性为代价，换取更好的函数特性。

### 单元测试

@RunWith(SpringRunner.class)

@SpringBootTest

public class RestaurantHandlerTests extends BaseUnitTests {

@Autowired

private RouterFunction<ServerResponse> restaurantRouter;

@Override

protected WebTestClient prepareClient() {

WebTestClient webClient = WebTestClient.bindToRouterFunction(restaurantRouter)

.configureClient().baseUrl("http://localhost:9090").responseTimeout(Duration.ofMinutes(1)).build();

return webClient;

}

}

和针对Controller的单元测试相比，编写Handler的单元测试的主要区别在于初始化WebTestClient方式的不同，测试方法的主体可以完全复用。