

# Spring WebFlux (讲师: Old Jia)

---

## 第一部分 响应式编程概述

---

### 1 响应式编程介绍

#### 1.1 为什么需要响应性?

传统的命令式编程在面对当前的需求时的一些限制。

当前的需求:

即使在应用负载较高时，应用需要有更高的可用性，并提供低的延迟时间。

#### Thread per Request模型

比如使用Servlet开发的单体应用，部署到tomcat。

tomcat有线程池，每个请求交给线程池中的一个线程来执行，如果执行过程中包括访问数据库，或者包括读取文件，则在调用数据库时或读取文件时，请求线程是阻塞的，即使是阻塞线程也是占用资源的，典型的每个线程要使用1MB的内存。

如果有并发请求，则会同时有多个线程处于阻塞状态，每个线程占据一份资源。

同时，Tomcat的线程池大小决定了可以同时处理多少个请求。

与传统方式使用Spring开发web应用做一个对比，就知道响应式编程是什么，以及它能提供了什么了。

传统的方式：使用Spring MVC开发web应用并部署到Servlet容器，如Tomcat。

Servlet容器有专门的线程池用于管理HTTP请求，每个请求对应一个线程，该线程负责该请求的整个生命周期（Thread per Request模型）。意味着应用仅能处理并发数为线程池大小的请求。可以配置更大的线程池，但是线程占用内存（一般一个线程1MB的样子），线程数越多，占用的内存越大。

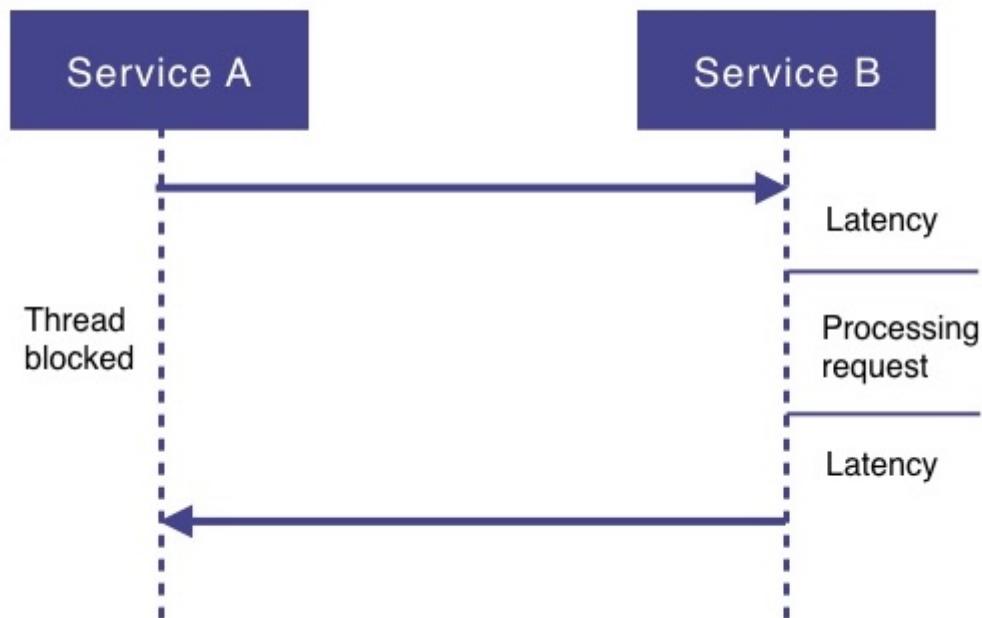
如果应用基于微服务架构，我们可以做横向扩展，但是也有内存高占用的问题。因此，当并发数很大的时候，Thread per Request模型很消耗资源。

微服务架构一个特性是分布式，运行很多分立的进程（很多服务器）。传统的命令式编程使用同步的请求/响应模式在服务之间通信，线程需要频繁的在服务调用的时候阻塞。浪费了资源。

## 等待I/O操作

在I/O操作中也存在大量的资源浪费：如调用数据库，读取文件等。

此时，发出I/O请求的线程会阻塞等待I/O操作的完成，即阻塞式I/O。这些线程的阻塞仅仅是为了等待一个响应，浪费了线程，浪费了内存。

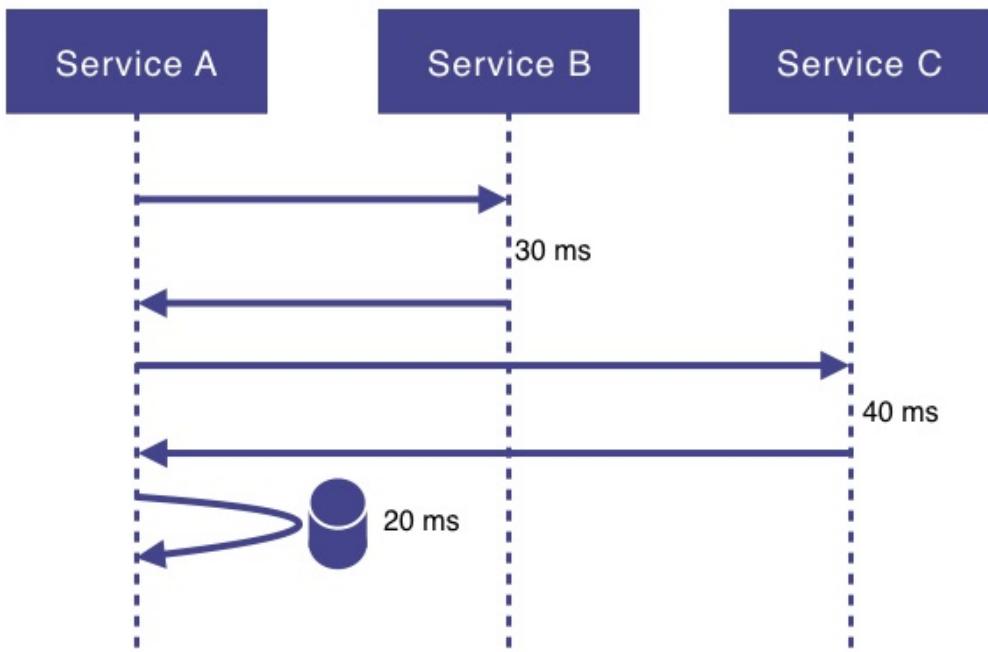


## 响应延迟

传统命令式编程另一个问题是：当一个服务需要做很多操作而不仅仅是I/O请求的时候，响应延迟相应的增大。

如服务A需要调用服务B和C，比如查询数据库，聚合结果并返回。意味着服务A的响应时间包括：

- 服务B的响应时间（网络延迟时间+处理时间）
- 服务C的响应时间（网络延迟时间+处理时间）
- 数据库请求响应时间（网络延迟时间+处理时间）



如果服务调用没有前后依赖关系，则可以并行调用服务。如果使用java的CompletableFuture异步调用并注册回调，开发会复杂很多，而且阅读和维护也会复杂很多。

### 压垮客户端

微服务的另一个问题是：服务A请求服务B的数据，如果数据量很大，超过了服务A能处理的程度，则导致服务OOM。

### 总结

上述的问题就是响应式编程要解决的。

响应式编程的优势：

- 不用Thread per Request模型，使用少量线程即可处理大量的请求。
- 在执行I/O操作时不让线程等待。
- 简化并行调用。
- 支持背压，让客户端告诉服务端它可以处理多少负载。

### 1.1.1 消息驱动通信

#### 响应式编程定义

响应式编程是使用异步、事件驱动构建非阻塞式应用的，此类应用仅需要少量的线程用于横向扩展。

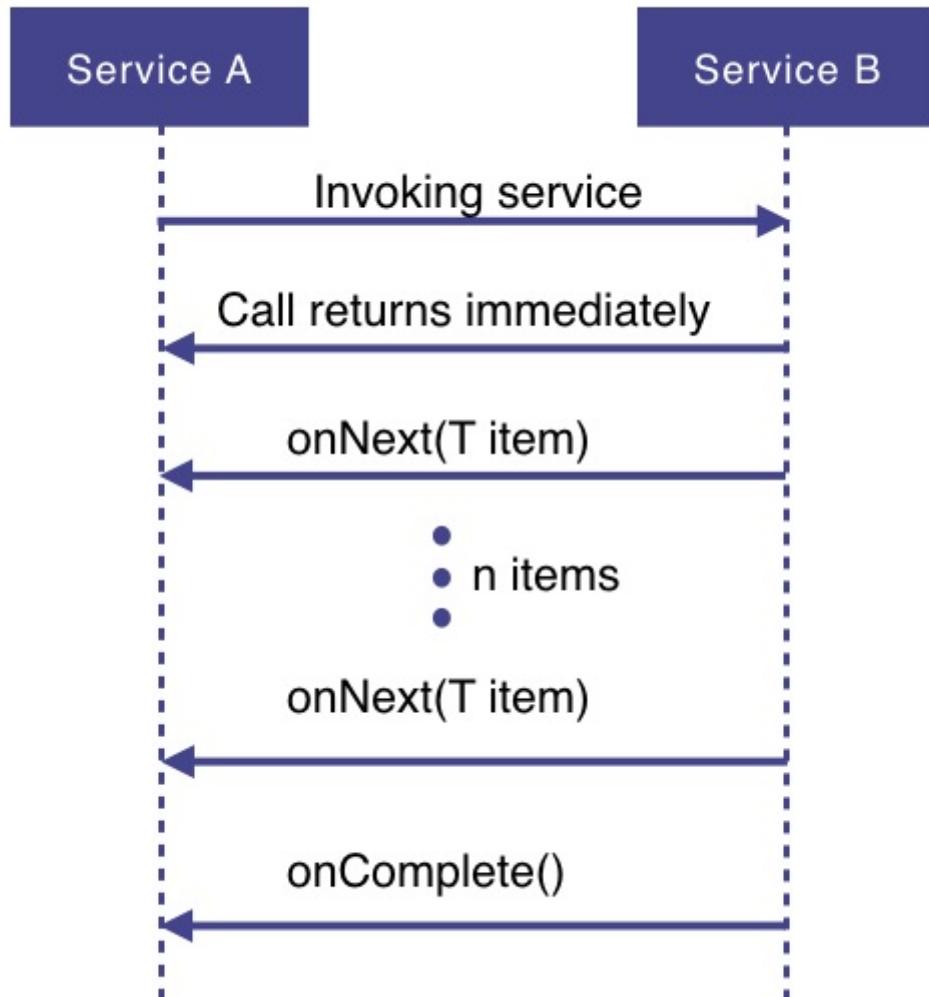
该定义的关键一点是：借助背压技术，防止生产者压垮消费者。

### 怎么做？

一言以蔽之，使用异步数据流编程。

如服务A需要从服务B获取数据。对于响应式编程，服务A向服务B发起请求，并立即返回（非阻塞、异步）。

之后，请求的数据以数据流的方式返回给服务A，服务B对每个数据项发布onNext事件。当所有的数据都发布了onNext事件，就发布onComplete事件结束。如果发生异常，服务B就发布onError事件，之后不再发布onNext事件。



响应式编程使用函数式编程风格，用于对数据流进行不同的转换。

流可以作为输入，也可以合并，映射和过滤等。

## 响应式系统

响应式系统应用的设计目标：

- 响应性（以时序的方式响应）
- 健壮（即使发生错误也可以保证响应性）

- 弹性（在不同的工作负载下保持响应性）
- 消息驱动（依赖异步消息传递机制）

响应式编程可以确保单个服务的异步非阻塞，整个系统的响应式需要整体考虑。

## 1.2 响应性应用案例

### 响应式编程是使用异步数据流进行编程

流是一个时序事件序列，可以发射三种不同的事件：（某种类型的）值、错误或者一个完成信号。分别为值、错误、完成定义事件处理函数，异步地处理事件。

监听一个流称为订阅，定义的函数是观察者，流是被观察者，即观察者模式。

每个流都会有多个方法，如 map, filter, scan, 等等。

当调用其中一个方法时，它会基于原来的流返回一个新的流。

它不会对原来的点击流作任何修改。这个特性称为不可变性，也可以对方法进行链式调用。

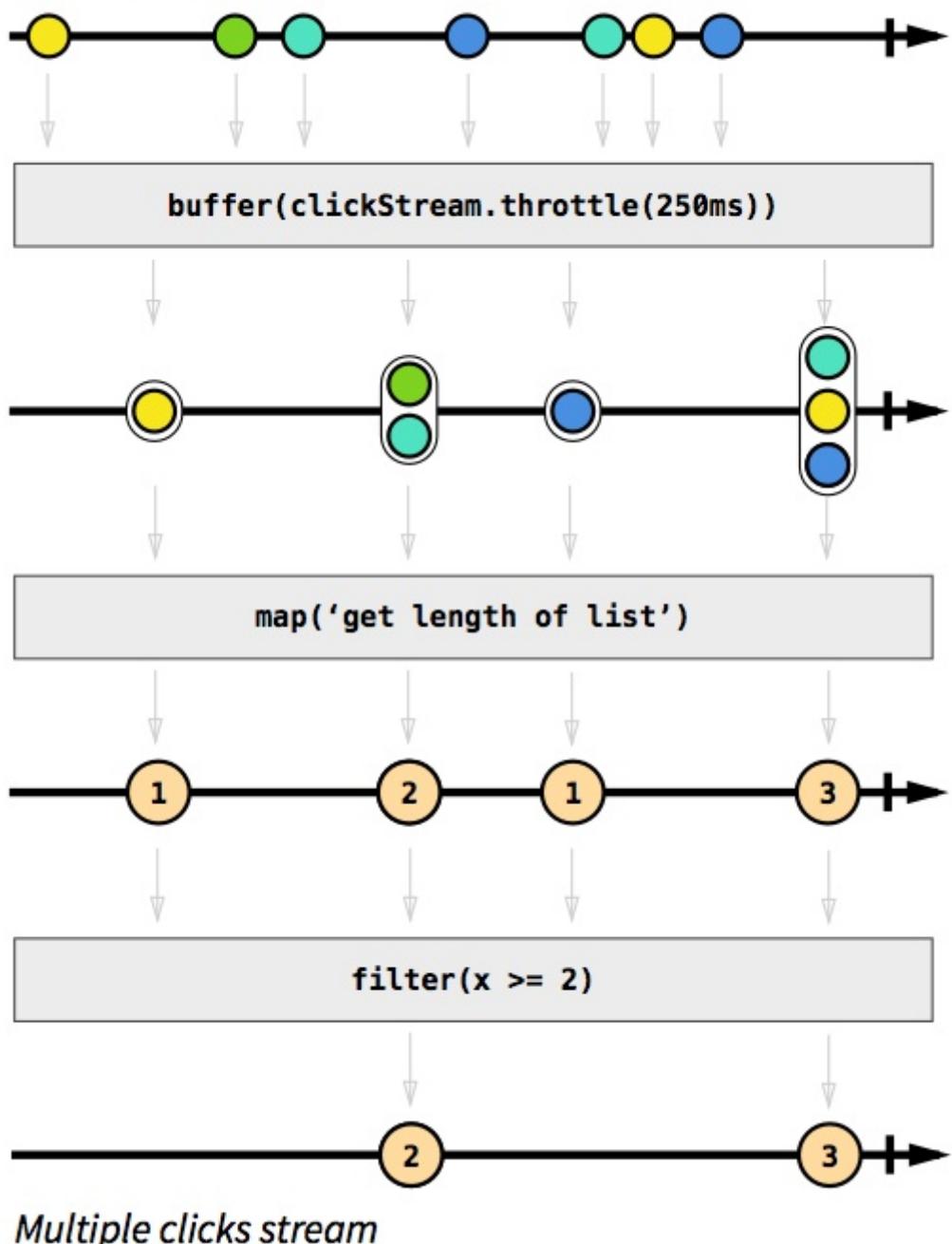
```
1 clickstream:   ---c----c--c----c-----c-->
2                      vvvvv map(c becomes 1) vvvv
3                      ---1----1--1----1-----1-->
4                      vvvvvvvvvv scan(+) vvvvvvvvvv
5 counterStream: ---1----2--3----4-----5-->
```

map(f) 会根据你提供的 f 函数把原 Stream 中的 Value 分别映射到新的 Stream 中。

scan(g) 会根据你提供的 g 函数把 Stream 中的所有 Value 聚合成一个 Value  $x = g(\text{accumulated}, \text{current})$

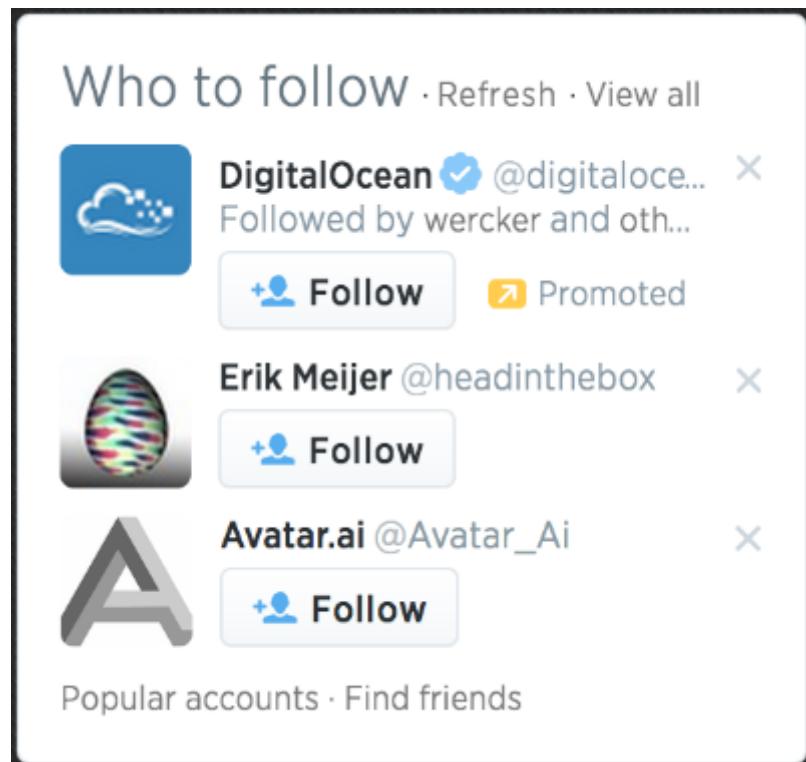
每 Click 一次， counterStream 就会把点击的总次数发给它的观察者。

### *Click stream*



1. 可以把同样的操作应用到不同种类的Stream上
2. 有很多其他可用的函数。

在 Twitter 上，这个表明其他账户的 UI 元素看起来是这样的：



如下核心功能：

- 启动时从 API 那里加载帐户数据，并显示 3 个推荐
- 点击"Refresh"时，加载另外 3 个推荐用户到这三行中
- 点击帐户所在行的'x'按钮时，只清除那一个推荐然后显示一个新的推荐
- 每行都会显示帐户的头像，以及他们主页的链接

请求和响应：

在 Rx 中你该怎么处理这个问题呢？

首先，所有的东西都可以转为一个流。

"在启动时，从API加载3个帐户的数据"。就只是简单的(1)发出一个请求，(2)收到一个响应，(3)渲染这个响应。

用流代表请求。

在启动的时候，只需要发出一个请求，如果把它转为一个数据流，就是一个只有一个值的流。

```
1 |     --a-----| ->
2 |
3 |     a表示字符串: 'https://api.github.com/users'
```

在 RX 中，创建只有一个值的流非常简单。官方把一个流称作“Observable”

```
1 |     var requestStream = Rx.Observable.just('https://api.github.com/users');
```

创建了包含一个**字符串**的流，并没有其他操作，需要以某种方式使那个值被映射。就是通过 `subscribing` 这个流。

```
1 requestStream.subscribe(function(requestUrl) {  
2     // 处理请求  
3     jQuery.getJSON(requestUrl, function(responseData) {  
4         // ...  
5     })  
6});
```

Rx 可以用来处理异步**数据流**。

请求的响应也可以当作一个包含了将会到达的数据的**流**。

```
1 requestStream.subscribe(function(requestUrl) {  
2     // 处理请求  
3     var responseStream = Rx.Observable.create(function (observer) {  
4         jQuerygetJSON(requestUrl)  
5             .done(function(response) { observer.onNext(response); })  
6             .fail(function(jqXHR, status, error) { observer.onError(error);  
7 })  
8             .always(function() { observer.onCompleted(); });  
9     });  
10    responseStream.subscribe(function(response) {  
11        // 对响应进行一些其他操作  
12    });  
13});
```

`Rx.Observable.create()` 创建了响应式流。该流通过显式的通知每一个 Observer (或者说是一个“Subscriber”) 数据事件( `onNext()` )或者错误事件 ( `onError()` )。

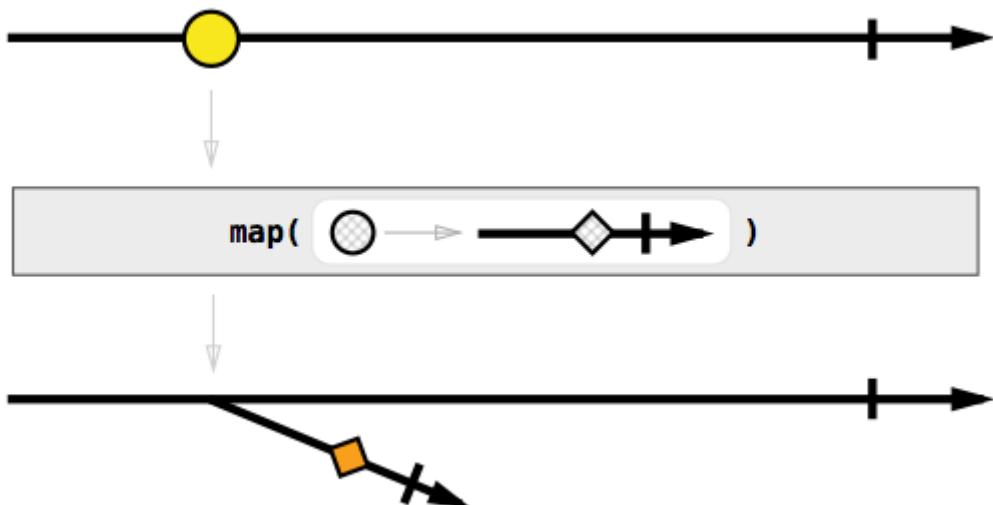
在 `subscribe()` 内又调用了另外一个 `subscribe()`，这类似于 Callback hell。

`responseStream` 是建立在 `requestStream` 之上的。

函数 `map(f)`，分别把 `f()` 应用到 流 A 中的每一个值，并把返回的值放进流 B 中。

```
1 var responseMetastream = requestStream  
2 .map(function(requestUrl) {  
3     // 从请求的URL获取结果，封装为响应流  
4     return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));  
5});
```

## *Request stream*



## *Response metastream*

`map`表示转换，将一个类型的数据转换为另一个类型的。

上述`map`转换是将流元素转换为流元素，流元素又可以订阅。但是我们只需要它输出一个流，流中的元素是结果数据即可。

因此需要使用`flatMap`，`flatMap`会打破元素之间的结构，将数据拉平：

```
1 var responseStream = requestStream
2 .flatMap(function(requestUrl) {
3   // 将从指定URL获取的结果转换为响应式流
4   return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
5 });
```

此时：

```
1 requestStream: --a----b--c-----| ->
2 responseStream: -----A-----B----C---| ->
3 (小写是请求，大写是响应)
```

渲染数据：

```
1 responseStream.subscribe(function(response) {
2   // 渲染数据的代码
3 });
```

目前为止所有的代码：

```
1 // 从指定URL创建请求流
2 var requestStream = Rx.Observable.just('https://api.github.com/users');
3 // 对请求流中的元素执行创建响应流
4 var responseStream = requestStream
5 .flatMap(function(requestUrl) {
6     return Rx.Observable.fromPromise(jQuery.getJSON(requestUrl));
7 });
8 // 响应流的订阅方法，在该方法中提供函数，对响应的结果进行渲染
9 responseStream.subscribe(function(response) {
10     // 渲染结果数据
11});
```

刷新按钮：

每点击一次刷新按钮，请求流就会映射一个新的 URL，同时我们也能得到一个新的响应。

两样东西：一个是刷新按钮上 点击事件组成的 流，同时需要根据刷新 点击流 改变请求流。

```
1 // 获取刷新按钮对象
2 var refreshButton = document.querySelector('.refresh');
3 // 对刷新按钮的点击事件封装为刷新点击事件流
4 var refreshClickStream = Rx.Observable.fromEvent(refreshButton, 'click');
```

把刷新点击事件流改为新的请求流，其中每一个 点击事件 都分别映射为带有随机偏移量的 API 端点。

```
1 // 刷新点击事件
2 var requestStream = refreshClickStream
3 .map(function() {
4     // 将刷新点击事件流中的每个元素按照如下方式进行转换
5     // 生成随机偏移量
6     var randomOffset = Math.floor(Math.random()*500);
7     // 生成新的URL，该URL包含了随机偏移量
8     return 'https://api.github.com/users?since=' + randomOffset;
9});
```

我们知道怎么分别为这两种情况生成 Stream：

```
1 // 刷新事件请求流
2 var requestOnRefreshStream = refreshClickStream
3 .map(function() {
4     var randomOffset = Math.floor(Math.random()*500);
5     return 'https://api.github.com/users?since=' + randomOffset;
6 });
7 // 启动事件请求流
8 var startupRequestStream =
Rx.Observable.just('https://api.github.com/users');
```

通过merge函数将上述两个流合并到一个流中：

```
1   stream A: ---a-----e----o---->
2   stream B: -----B---C-----D----->
3           vvvvvvvv merge vvvvvvvv
4           ---a-B--C--e--D--o---->
```

如此，则：

```
1 // 刷新点击事件请求流
2 var requestOnRefreshStream = refreshClickstream
3 .map(function() {
4     var randomOffset = Math.floor(Math.random()*500);
5     return 'https://api.github.com/users?since=' + randomOffset;
6 });
7 // 启动请求事件流
8 var startupRequestStream =
9 Rx.Observable.just('https://api.github.com/users');
10 // 将两个流进行合并
11 var requestStream = Rx.Observable.merge(
12     requestOnRefreshStream, startupRequestStream
13 );
```

更加简洁方式：

```
1 var requestStream = refreshClickstream
2 .map(function() {
3     var randomOffset = Math.floor(Math.random()*500);
4     return 'https://api.github.com/users?since=' + randomOffset;
5 })
6 // 调用merge操作符，合并另一个流
7 .merge(Rx.Observable.just('https://api.github.com/users'));
```

甚至可以更简短：

```
1 var requestStream = refreshClickstream
2 .map(function() {
3     var randomOffset = Math.floor(Math.random()*500);
4     return 'https://api.github.com/users?since=' + randomOffset;
5 })
6 // 调用startWith操作符，当启动的时候发送该事件
7 .startWith('https://api.github.com/users');
```

去掉 refreshClickStream 最后的 startWith()，并在一开始的时候“模拟”一次刷新 Click。

```
1 // 刷新点击事件流一开始刷新一次: startWith
2 var requestStream = refreshClickstream.startWith('startup click')
3 .map(function() {
4     var randomOffset = Math.floor(Math.random()*500);
5     return 'https://api.github.com/users?since=' + randomOffset;
6 });
```

为了让 UI 看起来舒服一些，当点击刷新时，我们需要清理掉当前的推荐。

```
1 refreshClickstream.subscribe(function() {
2     // 清除当前的推荐，然后等待显式新数据
3 });
```

所以让我们把显示的推荐设计成一个 stream，其中每一个映射的值都是包含了推荐内容的 JSON 对象。我们以此把三个推荐内容分开来。现在第一个推荐看起来是这样子的：

```
1 var suggestion1Stream = responseStream
2 .map(function(listUsers) {
3     // 从响应流中随机获取一个用户信息
4     return listUsers[Math.floor(Math.random()*listUsers.length)];
5 });
```

其他的，suggestion2Stream 和 suggestion3Stream 可以简单的拷贝 suggestion1Stream 的代码来使用。

responseStream 的 subscribe() 中处理渲染，直接在按钮订阅的函数中处理：

```
1 suggestion1Stream.subscribe(function(suggestion) {
2     // 渲染第一个结果
3 });
```

回到“当刷新时，清理掉当前的推荐”，我们可以很简单的把刷新点击映射为 null，并且在 suggestion1Stream 中包含进来，如下：

```
1 var suggestion1Stream = responseStream
2 .map(function(listUsers) {
3     // 从结果中随机获取一个用户信息
4     return listUsers[Math.floor(Math.random()*listUsers.length)];
5 })
6 .merge(
7     // 刷新点击流的元素映射到null。
8     refreshClickstream.map(function(){ return null; })
9 );
```

当渲染时，null 解释为“没有数据”，所以把 UI 元素隐藏起来。

```

1 suggestion1Stream.subscribe(function(suggestion) {
2     if (suggestion === null) {
3         // 隐藏第一个推荐的元素
4     }
5     else {
6         // 展示第一个推荐元素并渲染结果
7     }
8 });

```

现在的示意图：

```

1 refreshClickStream: -----o-----o--->
2     requestStream: -r-----r-----r--->
3     responseStream: ---R-----R-----R--->
4     suggestion1Stream: ---s----N---s----N-s-->
5     suggestion2Stream: ---q----N---q----N-q-->
6     suggestion3Stream: ---t----N---t----N-t-->

```

其中，N 代表了 null

作为一种补充，我们也可以在一开始的时候就渲染“空的”推荐内容。这通过把 `startWith(null)` 添加到 Suggestion stream 就完成了：

```

1 var suggestion1Stream = responseStream
2 .map(function(listUsers) {
3     // 从结果集中随机获取一个用户信息
4     return listUsers[Math.floor(Math.random()*listUsers.length)];
5 })
6 .merge(
7     refreshClickStream.map(function(){ return null; })
8 )
9 .startwith(null); // 以渲染“空”推荐内容开始

```

现在结果是：

```

1 refreshClickStream: -----o-----o--->
2     requestStream: -r-----r-----r--->
3     responseStream: ---R-----R-----R--->
4     suggestion1Stream: -N--s----N---s----N-s-->
5     suggestion2Stream: -N--q----N---q----N-q-->
6     suggestion3Stream: -N--t----N---t----N-t-->

```

关闭推荐并使用缓存的响应：

还有一个功能需要实现。每一个推荐，都该有自己的“X”按钮以关闭它，然后在该位置加载另一个推荐。最初的想法，点击任何关闭按钮时都需要发起一个新的请求：

用流的方式来思考。当点击'close1'时，我们想要用 `responseStream` 最近的映射从响应列表中获取一个随机的用户，如：

```
1 requestStream: --r----->
2 responseStream: -----R----->
3 close1clickStream: -----c---->
4 suggestion1Stream: -----s----s---->
```

在 Rx\* 中，叫做连接符函数的 combineLatest 实现了我们想要的功能。它接受两个流，A 和 B 作为输入，当其中一个流发射一个值时，combineLatest 把最近两个发射的值 a 和 b 从各自的流中取出并且返回一个  $c = f(x,y)$ ，其中  $f$  为你定义的函数。用图来表示更好：

```
1 stream A: --a-----e-----i----->
2 stream B: -----b---c-----d-----q---->
3           vvvvvvvv combineLatest(f) vvvvvvvv
4           ----AB--AC--EC--ED--ID--IQ---->
5
6 // f表示将字母转换为大写的函数
```

我们可以在 close1ClickStream 和 responseStream 上使用 combineLatest()，所以无论什么时候当一个按钮被点击时，我们可以获得最新的响应发射值，并且在 suggestion1Stream 上产生一个新的值。另一方面，combineLatest() 是对称的，当一个新的响应在 responseStream 发射时，它将会把最后的'关闭 1'的点击事件一起合并来产生一个新的推荐。这是有趣的，因为它允许我们把之前的 suggestion1Stream 代码简化成下边这个样子：

```
1 var suggestion1Stream = close1clickStream
2   .combineLatest(responseStream,
3     function(click, listUsers) {
4       return listUsers[Math.floor(Math.random()*listUsers.length)];
5     }
6   )
7   .merge(
8     refreshclickstream.map(function(){ return null; })
9   )
10  .startwith(null);
```

还有一个问题需要解决。combineLatest() 使用最近的两个数据源，但是当其中一个来源没发起任何事件时，combineLatest() 无法在 Output stream 中产生一个 Data event。从上边的 ASCII 图中，你可以看到，当第一个 Stream 发射值 a 时，这个值时并没有任何输出产生，只有当第二个 Stream 发射值 b 时才有值输出。

有多种方法可以解决这个问题，我们选择最简单的一种，一开始在'close 1'按钮上模拟一个点击事件：

```
1 var suggestion1stream = close1clickstream.startwith('startup click') // we
2     added this
3     .combineLatest(responseStream,
4         function(click, listUsers) {
5             return listUsers[Math.floor(Math.random()*listUsers.length)];
6         }
7     )
8     .merge(
9         refreshclickstream.map(function(){ return null; })
10    )
10 .startwith(null);
```

## 1.3 响应式现状

2011年，微软发布了.NET的响应式扩展（Reactive Extensions，即ReactiveX或Rx），以方便异步、事件驱动的程序。

ReactiveX混合了迭代器模式和观察者模式。不同之处在于一个是推模式，一个是基于迭代器的拉模式。

除了对变化事件的观察，完成事件和异常事件也会发布给订阅者。

ReactiveX的基本思想是**事件是数据，数据是事件**。

响应式扩展被移植到几种语言和平台上，包括JavaScript、Python、C++、Swift和java。ReactiveX很快成为一种跨语言的标准，将响应式编程引入到行业中。

RxJava，是Java的ReactiveX实现，很大程度上由Netflix的benchr istensen和david karnok创建的。

RxJava 1.0于2014年11月发布。

RxJava是其他reactivex jvm平台技术的主要技术，其他的如如RxScala、RxKotlin、RxGroovy。RxJava已经成为Android开发的核心技术，并且已经进入Java后端开发。

许多RxJava adapter库，例如RxAndroid，RxJava JDBC，RxNetty，和RxJavaFX调整了几个Java框架，使之成为响应式的，并且可以开箱即用地使用RxJava。

这表明RxJava不仅仅是一个库。它是更大的ReactiveX生态系统的一部分，代表了整个编程方法。

## 1.4 为什么采用响应式Spring

响应式系统非常复杂，在构建这类系统时困难比较多。

要轻松创建响应式系统，就必须首先分析能够构建这类系统的框架，然后选择其中一个。

选择框架最常用的方法之一是分析其可用功能、相关性以及社区。

在JVM领域，构建响应式系统的最知名框架是Akka和Vert.x生态系统。

一方面，Akka是一个受欢迎的框架，具有大量功能和大型社区。然而，Akka最初是作为Scala生态系统的一部分构建的，在很长一段时间内，它仅在基于Scala编写的解决方案中展示了它的强大功能。尽管Scala是一种基于JVM的语言，但它与Java明显不同。几年前，Akka直接开始支持Java，但由于某些原因，它在Java世界中不像在Scala世界中那么受欢迎。

另一方面，Vert.x框架也是构建高效响应式系统的强大解决方案。Vert.x的设计初衷是作为Node.js在Java虚拟机上的替代方法，它支持非阻塞和事件驱动。

然而，Vert.x仅在几年前才开始具备竞争力，在过去的15年中，Spring框架一直在构建灵活且健壮的应用程序的开发框架市场中占有主导地位。

Spring框架使用适合开发人员的编程模型，为构建Web应用程序提供了广泛的可能性。然而，长期以来，它在构建健壮的响应式系统方面存在一些局限性。

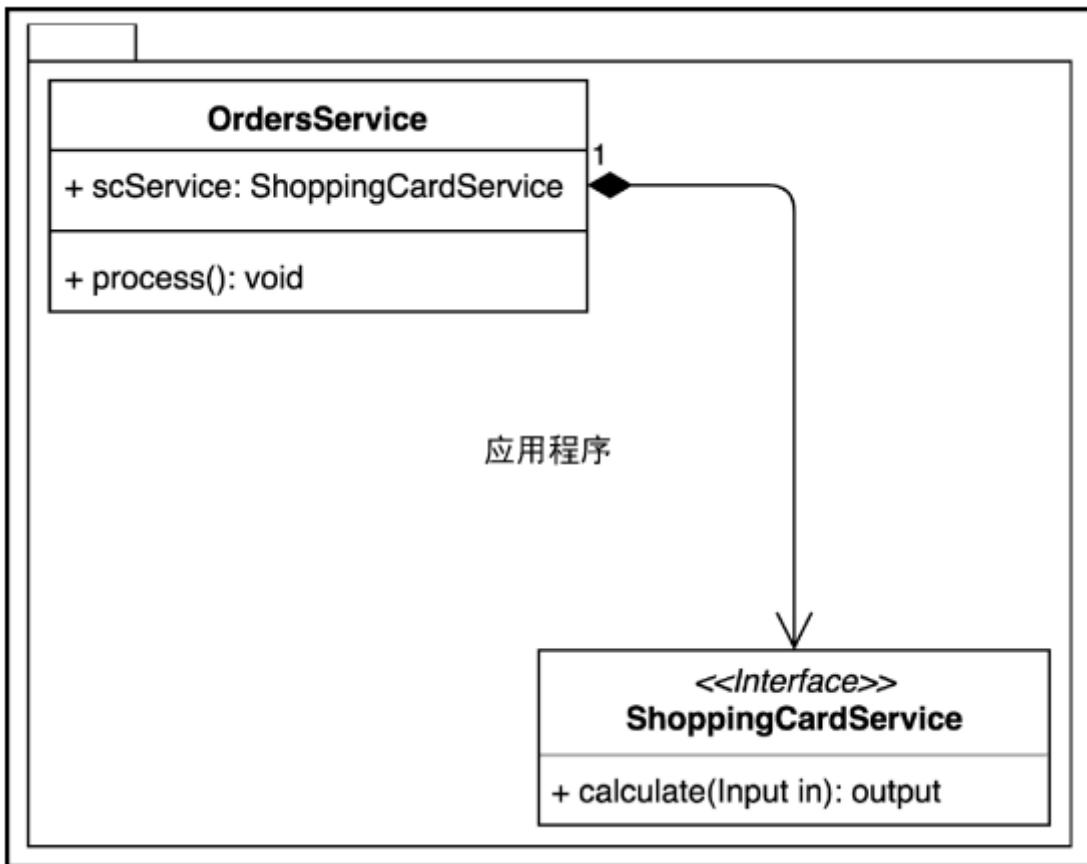
### 1.4.1 服务级别的响应性

现在大型系统由多个子系统组成，系统整体的响应性依赖于这些子系统的响应性。也就是说，响应式系统的设计原则适用于各个级别、各种规模的系统，有助于它们很好地组合在一起。

因此，在服务级别提供响应式设计和实现很重要。

在使用Java编写代码的过程中，最流行的传统技术是命令式编程(imperative programming)。

为了理解命令式编程是否遵循响应式系统设计原则，见下图：



在这里，Web 商店应用程序中有两个组件。在这种场景下，OrdersService 在处理用户请求时调用 ShoppingCardService。假设在 ShoppingCardService 内部执行长时间运行的 I/O 操作，例如 HTTP 请求或数据库查询，

```

1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Input;
4 import com.lagou.webflux.demo.entity.Output;
5
6 /**
7  * 接口
8  * 只有一个calculate方法接收一个参数，处理完之后返回结果
9 */
10 public interface ShoppingCardService {
11
12     Output calculate(Input value);
13
14 }
15
16
17 package com.lagou.webflux.demo.service;
18
19 import com.lagou.webflux.demo.entity.Input;
20 import com.lagou.webflux.demo.entity.Output;
21
22 public class OrdersService {
23
24     private ShoppingCardService service;
25
26     void process() {

```

```
27     Input input = ...;
28     // 同步调用 ShoppingCardService 并在执行后立即接收结果
29     Output output = service.calculate(input);
30 }
31 }
```

在这种场景下，OrdersService的执行过程与 ShoppingCardService 的执行过程紧密耦合。当 ShoppingCardService 处于处理阶段时，无法继续执行任何其他操作。

service.calculate(input)的执行过程阻塞了处理 OrdersService 逻辑的线程。想要在OrdersService 中运行单独的独立处理，**必须分配一个额外的线程**。

额外线程的分配可能是一种浪费。从响应式系统的角度来看，这种系统行为是不可接受的。

上述逻辑也可以通过应用回调（callback）技术来解决，以实现跨组件通信。

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Input;
4 import com.lagou.webflux.demo.entity.Output;
5
6 import java.util.function.Consumer;
7
8 public interface ShoppingCardService {
9     void calculate(Input value, Consumer<Output> c);
10 }
11
12
13 package com.lagou.webflux.demo.service;
14
15 import com.lagou.webflux.demo.entity.Input;
16
17 public class OrderService {
18
19     private final ShoppingCardService service;
20
21     public OrderService(ShoppingCardService service) {
22         this.service = service;
23     }
24
25     void process() {
26
27         Input input = null;
28
29         service.calculate(input, output -> {
30             // 需要执行的回调逻辑
31         });
32     }
33 }
34
35 package com.lagou.webflux.demo.entity;
36
37 public class Input {
```

```
38 }
39
40 package com.lagou.webflux.demo.entity;
41
42 public class Output {
43 }
```

OrdersService 传递回调函数以便在操作结束时做出响应。即OrdersService 现在与 ShoppingCardService 实现了解耦，OrdersService 可以通过ShoppingCardService#calculate 方法所实现（调用给定函数）的回调获取通知，这个过程可以是同步的也可以是异步的。

```
1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Input;
4 import com.lagou.webflux.demo.entity.Output;
5 import com.lagou.webflux.demo.service.ShoppingCardService;
6
7 import java.util.function.Consumer;
8
9 /**
10  * 该实现假定没有阻塞操作。由于我们没有执行 I/O 操作，将结果传递给回调函数来立即返回结果。
11 */
12 public class SyncShoppingCardService implements ShoppingCardService {
13     @Override
14     public void calculate(Input value, Consumer<Output> c) {
15         Output output = new Output();
16         c.accept(output);
17     }
18 }
```

```
1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Input;
4 import com.lagou.webflux.demo.entity.Output;
5 import com.lagou.webflux.demo.service.ShoppingCardService;
6 import org.springframework.web.client.RestTemplate;
7
8 import java.util.function.Consumer;
9
10 /**
11  * 当发生阻塞I/O 时，我们可以将它包装在单独的 Thread 中。
12  * 获取结果之后，处理并传递给回调函数。
13 */
14 public class AsyncShoppingCardService implements ShoppingCardService {
15
16     private final RestTemplate template = new RestTemplate();
17
18     @Override
19     public void calculate(Input value, Consumer<Output> c) {
20         new Thread(() -> {
```

```
21         outputResult = template.getForObject("", null, "");
22         c.accept(result);
23     }).start();
24 }
25 }
```

ShoppingCardService 的同步实现中，从 API 角度讲并不提供任何好处。

在异步的情况下，而请求将在单独的线程中执行。

OrdersService 与执行过程进行解耦，并通过执行回调获取完成的通知。

组件通过回调函数实时解耦，意味着在调用 service.calculate方法之后，能够立即继续执行其他操作，无须等待 ShoppingCardService 的阻塞式响应。

该技术的缺点是，回调要求开发人员对多线程有深入的理解，以避免共享数据修改陷阱和回调地狱（callback hell）。

回调技术不是唯一的选择。另一个选择是 java.util.concurrent.Future，它在某种程度上隐藏了执行行为并解耦了组件。

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Input;
4 import com.lagou.webflux.demo.entity.Output;
5
6 import java.util.concurrent.Future;
7
8 public interface ShoppingCardService {
9     // 返回值是可以阻塞方式获取结果的Future对象
10    Future<Output> calculate(Input value);
11 }
12
13 package com.lagou.webflux.demo.service;
14
15 import com.lagou.webflux.demo.entity.Input;
16 import com.lagou.webflux.demo.entity.Output;
17
18 import java.util.concurrent.ExecutionException;
19 import java.util.concurrent.Future;
20
21 public class OrdersService {
22
23     private final ShoppingCardService service;
24
25     public OrdersService(ShoppingCardService service) {
26         this.service = service;
27     }
28
29     void process() throws ExecutionException, InterruptedException {
30         Input input = null;
```

```

31     // 异步调用
32     Future<Output> result = service.calculate(input);
33     // 同步获取结果，也可以随后获取结果
34     Output output = result.get();
35
36 }
37
38 }
39
40 package com.lagou.webflux.demo.entity;
41
42 public class Input {
43 }
44
45
46 package com.lagou.webflux.demo.entity;
47
48 public class Output {
49 }

```

通过 Future 类，实现了对结果的延迟获取。

在 Future 类的支持下，避免了回调地狱，并将实现多线程的复杂性隐藏在了特定 Future 实现的背后。无论如何，为了获得需要的结果，必须阻塞当前的线程并与外部执行进行同步，这显著降低了可伸缩性。

作为改进，Java 8 提供了 CompletionStage 以及它的直接实现 CompletableFuture。同样，这些类提供了类似 promise 的 API，使构建如下代码成为可能。

```

1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Input;
4
5 public class OrderService {
6
7     private final ShoppingCardService service;
8
9     public OrderService(ShoppingCardService service) {
10         this.service = service;
11     }
12
13     void process() {
14         Input input = null;
15         // CompletionStage 是一个类似于 Future 的类包装器，但能以函数声明的方式处理返回的结果。
16         // CompletionStage 的整体行为类似于 Future,
17         // 但 CompletionStage 提供了一种流式 API，可以编写 thenAccept 和
18         // thenCombine 等方法。
19         // 这些方法定义了对结果的转换操作，然后 thenAccept 方法定义了最终消费者，用于处
理转换后的结果。
20         service.calculate(input).thenApply(output1 -> {return output1;})

```

```
20         .thenCombine(otherStage, (output2, otherOutput) -> {return
21             output2;})
22     }
23 }
24 }
25 }
```

在 CompletionStage 的支持下，可以编写函数式和声明式的代码，这些代码看起来很整洁，并且能够异步处理结果。还可以省略等待结果的过程，并提供在结果可用时对其进行处理的功能。

之前的所有技术都受到 Spring 团队的重视，并且已经在框架内的大多数项目中实现。尽管 CompletionStage 为编写高效且可读性强的代码提供了更好的可能性，但遗憾的是，其中存在一些遗漏点。例如，Spring 4 MVC 在很长时间内不支持 CompletionStage。

为了弥补这一点，Spring 提供了自己的 ListenableFuture。之所以发生这种情况，是因为 Spring 4 旨在与旧的 Java 版本兼容。

以下代码展示了如何结合使用 ListenableFuture 与 AsyncRestTemplate。

```
1 package com.lagou.webflux.demo.service;
2
3 import org.springframework.http.ResponseEntity;
4 import org.springframework.util.concurrent.FailureCallback;
5 import org.springframework.util.concurrent.ListenableFuture;
6 import org.springframework.util.concurrent.SuccessCallback;
7 import org.springframework.web.client.AsyncRestTemplate;
8
9 public class OrderService {
10
11     private final AsyncRestTemplate template = new AsyncRestTemplate();
12
13     void process() {
14         SuccessCallback success = r -> {};
15         FailureCallback fail = e -> {};
16         // 实现异步调用
17         ListenableFuture<ResponseEntity<Object>> response =
18         template.getEntity("urlstr", Object.class);
19         response.addCallback(success, fail);
20     }
21 }
```

上述代码展示了处理一个异步调用的回调风格。Spring 框架在一个单独的线程中包装了阻塞式网络调用。同时，Spring MVC 依赖于 Servlet API，这使所有实现都必须使用线程单次请求（thread-per-request）模型。

随着 Spring 5 框架和新的响应式 WebClient 的发布，许多事情发生了变化，在WebClient 的支持下，所有跨服务通信都不再是阻塞的。此外，Servlet 3.0 引入了异步客户端-服务器通信，Servlet 3.1 能对 I/O 进行非阻塞写入，并且 Servlet 3 的 API 的新异步非阻塞功能很好地集成到 Spring MVC 中。但是，Spring MVC 唯独没有提供一个开箱即用的异步非阻塞客户端，这一点使改进版 Servlet 带来的所有好处黯然失色。

这个模型非常不理想，效率低下。

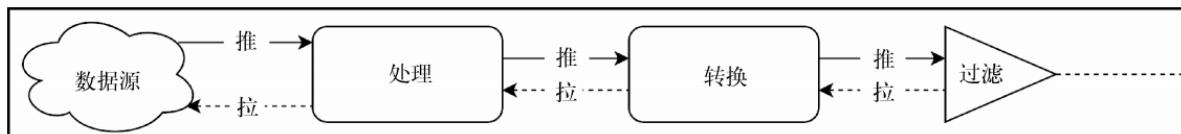
一方面，多线程本质上是一种复杂的技术。当我们使用多线程时，必须考虑很多事情，例如从不同线程访问共享内存、线程同步、错误处理等。

另一方面，Java 中的多线程设计假设一些线程可以共享一个 CPU，并同时运行它们的任务。CPU 时间能在多个线程之间共享的事实引入了上下文切换（context switching）的概念。这意味着后续要恢复线程，就需要保存和加载寄存器、存储器映射以及其他通常属于计算密集型操作的相关元素。因此，具有大量活动线程和少量 CPU 的应用方式将会效率低下。

同时，典型的 Java 线程在内存消耗方面有一定开销。在 64 位 Java 虚拟机上线程的典型栈大小为 1024 KB。一方面，尝试在单连接线程（thread per connection）模型中处理 64 000 个并发请求可能使用大约 64 GB 的内存。从业务角度来看，这可能代价高昂；从应用程序的角度来讲，这也是很严重的问题。另外，如果切换到具有有限大小的传统线程池和响应请求的预配置队列，那么客户端等待响应的时间会太长，不是太可靠，因为这增加了平均响应超时的现象，最后可能导致应用程序丧失即时响应性。

为此，响应式规范建议使用非阻塞操作，但这是 Spring 生态系统缺少的。此外，Spring 也没有与 Netty 等响应式服务器进行良好集成，而这些响应式服务器解决了上下文切换的问题。

异步处理不仅限于简单的请求-响应模式，对于包含无限元素的数据流，可以具有背压支持的对齐转换流的方式处理它，如下图所示：



处理此类情况的方法之一是使用响应式编程，它通过链式转换阶段来提供异步事件处理技术。

响应式编程是一种很好的技术，符合响应式系统的设计要求。

这对构建现代应用程序形成了另一个限制，同时降低了框架的竞争力。因此，围绕响应式系统和响应式编程的火热炒作中提到的所有差距，扩大了对框架进行大规模改进的需求。最后，这种需求极大地促进了 Spring 框架的改进，促使开发者在所有级别添加响应性支持并为开发人员提供强大的响应式系统开发工具。框架的关键开发人员决定实现新模块，这些模块将展示 Spring 框架作为响应式系统开发基础的全部能力。

## 2 无处不在的响应性

### 2.1 API不一致问题

大量的同类型响应式库可供选择（如RxJava，CompletableFuture，Vert.x，AKKA等）。

例如，我们可能依赖于使用 RxJava 的 API 来编写正在处理的数据项的流程。要构建一个简单的异步请求-响应交互，依赖 CompletableFuture 就足够了。

我们也可以使用特定于框架的类（如

`org.springframework.util.concurrent.ListenableFuture`）来构建组件之间的异步交互，并基于该框架简化开发工作。

另一方面，丰富的选择很容易使系统过于复杂。例如，若存在两个依赖于同一个异步非阻塞通信概念但具有不同 API 的库，会导致我们需要提供额外的工具类，以便将一个回调转换为另一个回调；反之亦然。代码如下：

```
1 // async 数据库客户端的接口声明，它是支持异步数据库访问的客户端接口的代表性示例
2 interface AsyncDatabaseClient {
3     <T> CompletionStage<T> store(CompletionStage<T> stage);
4 }
5
6 final class AsyncAdapters {
7
8     public static <T> CompletionStage<T> toCompletion(ListenableFuture<T>
9 future) {
10         CompletableFuture<T> completableFuture = new CompletableFuture<T>();
11
12         future.addCallback(completableFuture::complete,
13                             completableFuture::completeExceptionally);
14         return completableFuture;
15     }
16
17     public static <T> ListenableFuture<T> toListenable(CompletionStage<T>
18 stage) {
19         SettableListenableFuture<T> future = new SettableListenableFuture<T>();
20
21         stage.whenComplete((v, t) -> {
22             if (t == null) {
23                 future.set(v);
24             } else {
25                 future.setException(t);
26             }
27         });
28         return future;
29     }
30
31 @RestController
32 public class MyController {
```

```

32 // ...
33 @RequestMapping
34 public ListenableFuture<?> requestData() {
35     AsyncRestTemplate httpClient = ...;
36     AsyncDatabaseClient databaseClient = ...;
37     CompletionStage<String> completionStage = toCompletion(
38         httpClient.execute(...))
39     );
40     return toListenable(
41         databaseClient.store(completionStage)
42     );
43 }
44 }

```

从前面的示例中可以看出：

1. Spring 4.x 框架中的 ListenableFuture 和 CompletionStage 之间没有直接集成。
2. 该示例并没有脱离响应式编程的常见用法。
3. 许多库和框架为组件之间的异步通信提供了自己的接口和类，包括简单请求-响应通信以及流处理框架。
4. 在许多情况下，为了解决这个问题并使几个独立的库兼容，必须提供自己的适配并在几个地方重用。
5. 此外，自己写的适配可能有 bug，需要额外的维护。

Spring 5.x 框架扩展了 ListenableFuture 的 API 并且提供了一个 Completable 的方法来解决不兼容的问题。

这里的根本问题在于没有标准。

## 2.2 推拉

在整个响应式环境演变的早期阶段，所有库的设计思想都是把数据从源头**推送到**订阅者。

因为纯粹的拉模型在某些场景下效率不够高。

假设我们要过滤一大堆数据，但只取前 10 个元素。可以采用拉模型来解决这个问题：

```

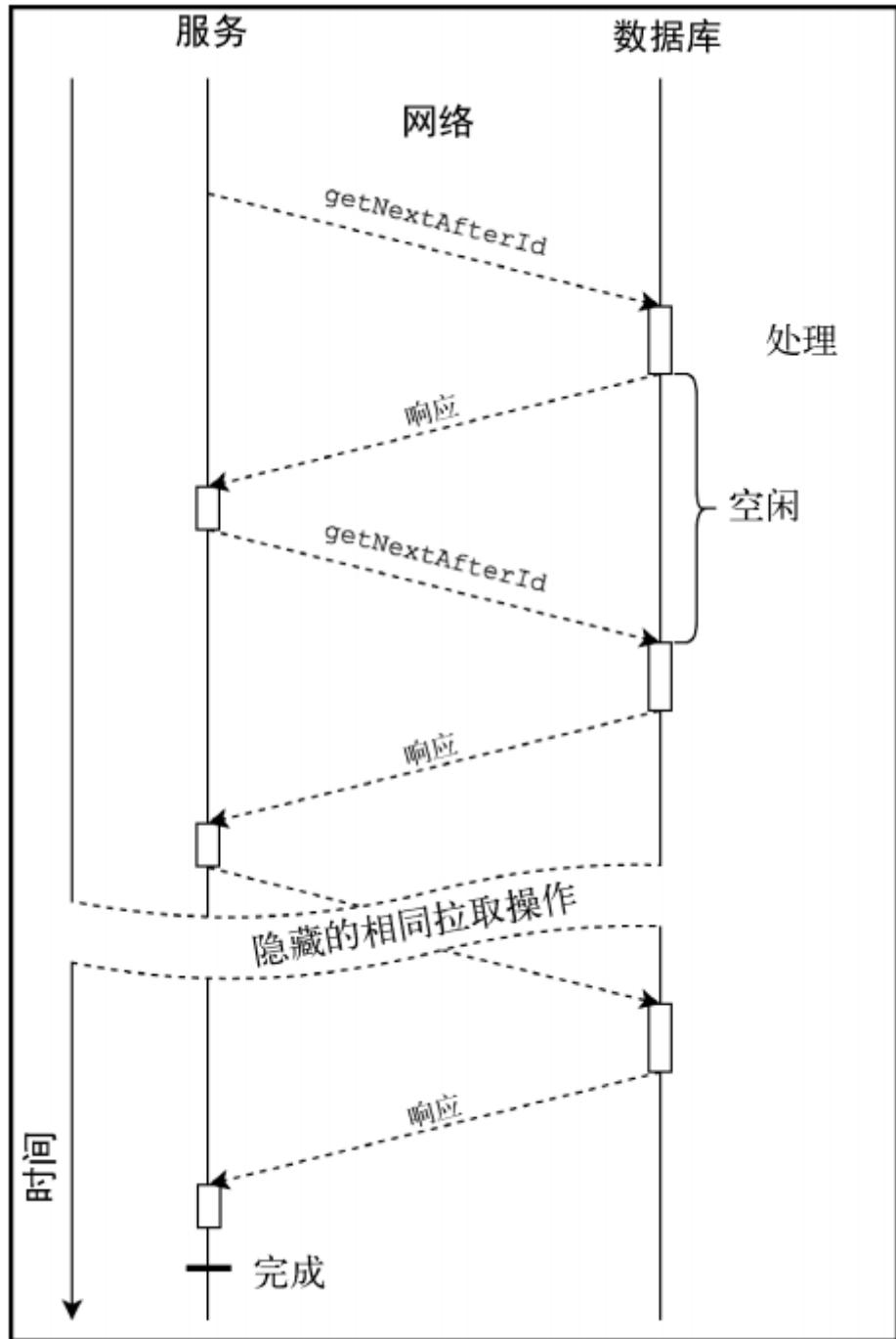
1 // AsyncDatabaseClient 字段声明。在这里，我们使用该客户端将异步、非阻塞通信与外部数据
2 库连接起来。
3 final AsyncDatabaseClient dbClient = ...;
4 /*
5 这是 list 方法声明。这里我们通过返回 CompletionStage 声明一个异步契约，并将其作为调用
6 list 方法的结果。
7 同时，为了聚合拉取结果并将其异步发送给调用者，我们声明 Queue 和 CompletableFuture 来存储
8 接收的值，并在稍后手动发送所收集的 Queue。
 */
9 public CompletionStage<Queue<Item>> list(int count) {

```

```
9     BlockingQueue<Item> storage = new ArrayBlockingQueue<>(count);
10    CompletableFuture<Queue<Item>> result = new CompletableFuture<>();
11
12    pull("1", storage, result, count);
13    return result;
14 }
15
16 /*
17 这是 pull 方法声明。在该方法中，我们调用 AsyncDatabaseClient#getNextAfterId 来执行查询并异步接收结果。然后，当收到结果时，我们在点(3.1)处对其进行过滤。如果是有效项，我们就将其聚合到队列中。另外，在点(3.2)，我们检查是否已经收集了足够的元素，将它们发送给调用者，然后退出拉操作。如果任何一个所涉及的 if 分支被绕过，就再次递归调用 pull 方法(3.3)。
18 */
19 void pull(String elementId, Queue<Item> queue, CompletableFuture<
20 resultFuture, int count) {
21     dbClient.getNextAfterId(elementId)
22         .thenAccept(item -> {
23             if (isValid(item)) {
24                 queue.offer(item);
25                 if (queue.size() == count) {
26                     resultFuture.complete(queue);
27                     return;
28                 }
29                 pull(item.getId(), queue, resultFuture, count);
30             });
31 }
```

从上述代码可以看出，在服务和数据库之间使用了异步、非阻塞交互。

但是，如果查看下图，就会看到其缺陷所在。



逐个请求下一个元素会导致在从服务传递请求到数据库上花费额外的时间。

从服务的角度来看，整体处理时间大部分浪费在空闲状态上。即使没有使用资源，由于额外的网络活动，整体处理时间也会是原来的两倍甚至三倍。

此外，数据库不知道未来请求的数量，这意味着数据库不能提前生成数据，并因此处于空闲状态。这意味着数据库正在等待新请求。

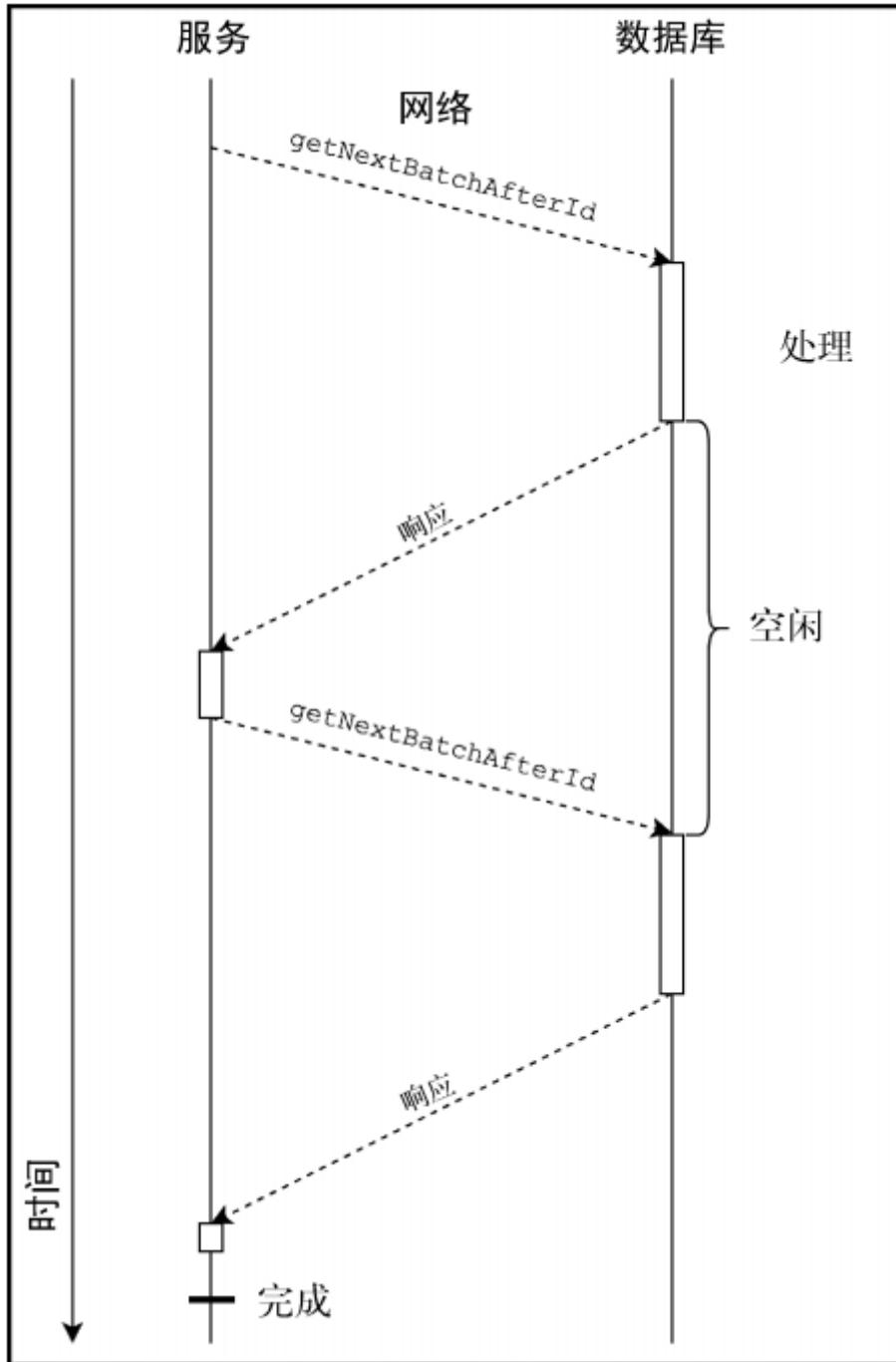
在响应被传递给服务、服务处理传入响应然后请求新数据的过程中，其效率会比较低下。

为了优化整体执行过程并将模型维持为一等公民，可以将拉取操作与批处理结合起来，示例代码的修改如下所示：

```
1 // 与上一个例子中的 pull 方法声明相同。
2 void pull(String elementId, Queue<Item> queue, CompletableFuture<
3   resultFuture, int count) {
4   /*
5    * 这是 getNextBatchAfterId 执行过程。可以注意到，AsyncDatabaseClient 方法可用于查
6    * 询特定数量的元素，这些元素作为 List <Item> 返回。反过来，当数据可用时，除了要创建额外的
7    * for 循环以分别处理该批的每个元素，我们对它们的处理方式几乎相同。
8   */
9   dbClient.getNextBatchAfterId(elementId, count)
10    .thenAccept(items -> {
11      for (Item item : items) {
12        if (isValid(item)) {
13          queue.offer(item);
14          if (queue.size() == count) {
15            resultFuture.complete(queue);
16            return;
17          }
18        }
19      }
20    })
21    // 这是递归 pull 方法的执行过程，在缺少来自上次拉取的数据项的情况下，这个设
22    // 计会被用于获取另外一批数据项。
23    .pull(items.get(items.size() - 1).getId(), queue, resultFuture,
24          count);
25  });
26 }
```

一方面，通过查询一批元素，可以显著提高 list 方法执行的性能并显著减少整体处理时间；

另一方面，交互模型中仍然存在一些缺陷，而该缺陷可以通过分析下图来检测。



该流程在处理时间方面仍然存在一些效率低下的情况。

例如，当数据库查询数据时，客户端仍处于空闲状态。

同时，发送一批元素比发送一个元素需要更多的时间。

最后，对整批元素的额外请求实际上可能是多余的。例如，如果只剩下一个元素就能完成处理，并且下一批中的第一个元素就满足验证条件，那么其余的数据项就完全是多余的且会被跳过。

为了提供最终的优化，只会请求一次数据，当数据变为可用时，该数据源异步推送数据。

以下代码修改展示了如何实现这一过程：

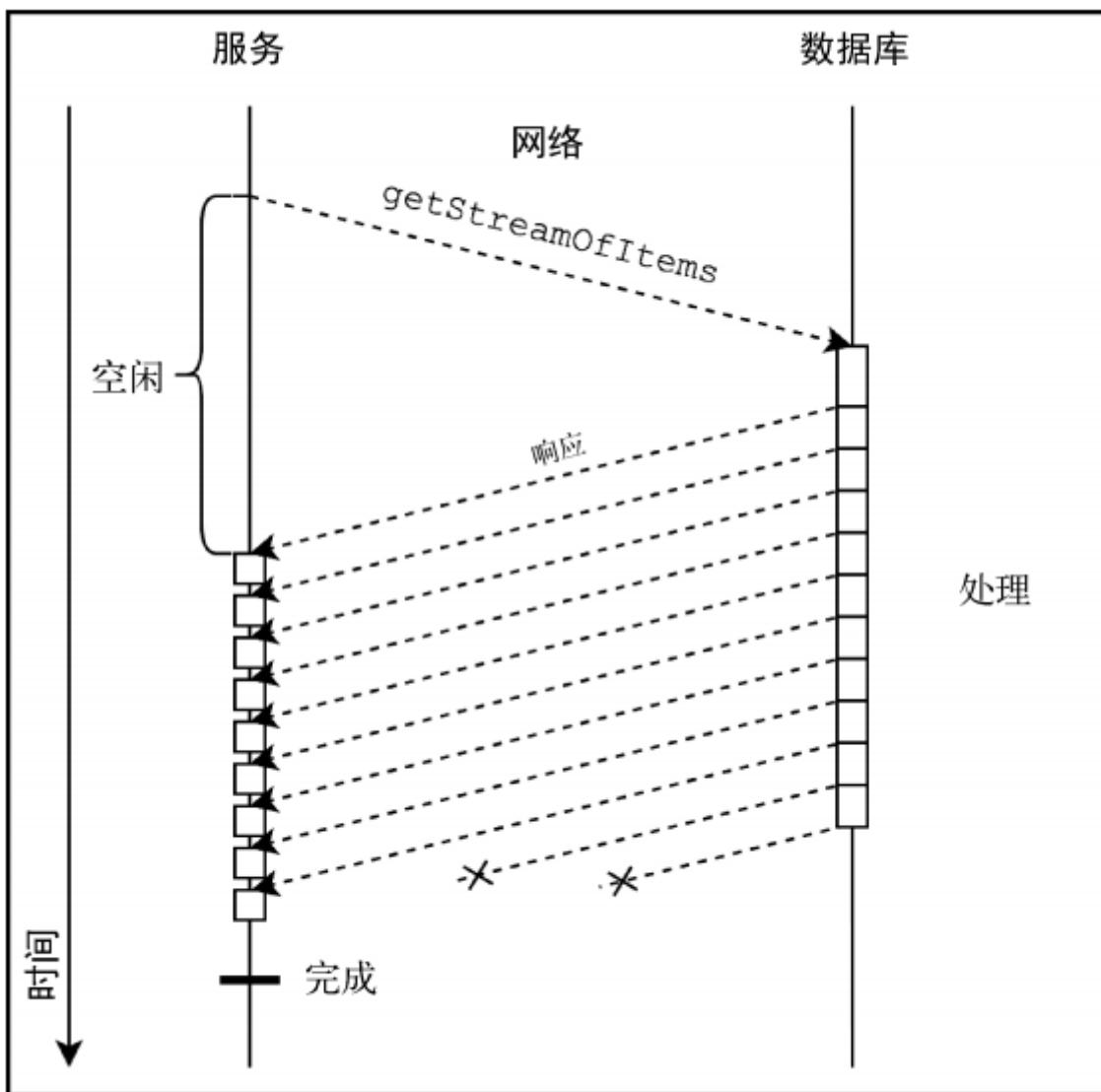
```

1 // 这是 list 方法声明。这里的 observable <Item>会返回一个类型，该类型标识正在被推送的元
素。
2 public Observable<Item> list(int count) {
3     /*
4      这是流查询阶段。通过调用 AsyncDatabaseClient#getStreamOfItems 方法，我们会对数据
5      库完成一次订阅。在点(2.1)，我们会过滤元素，并且通过使用.take()操作符根据调用者的请求获取特
6      定数量的数据(2.2)。
7      */
8     return dbClient.getStreamOfItems()
9         .filter(item -> isValid(item))
10        .take(count);
11    }

```

在这里，我们使用 RxJava 1.x 类作为一等公民来接收所推送的元素。反过来，一旦满足所有要求，就发送取消信号并关闭与数据库的连接。

当前的交互流程如下图所示。



如上图所示，再次对整体处理时间做了优化。在交互过程中，只有当服务等待第一个响应时会有一大段空闲时间。当第一个元素到达后，数据库会在数据到来时开始发送后续元素。

反过来，即使处理一个元素的过程可能比查询下一个元素快一点，服务的整体空闲时间也会很短。但是，在服务已经收集到所需数量的元素后，数据库仍可能生成多余元素，此时数据库会忽略它们。

## 2.3 流量控制问题

采用推模型主要是因为可以通过将请求量减少到最小以优化整体处理时间。

这就是为什么RxJava 1.x及类似的开发库以推送数据为目的进行设计，这也是为什么流技术能成为分布式系统中组件之间重要的通信技术。

如果仅仅与推模型进行组合，那么该技术有其局限性。消息驱动通信的本质是假设每个请求都会有一个响应，因此服务可能收到异步的、潜在的无限消息流。而这里存在陷阱，因为如果生产者不关注消费者的吞吐能力，它可能会以下面两节中描述的方式影响系统的整体稳定性。

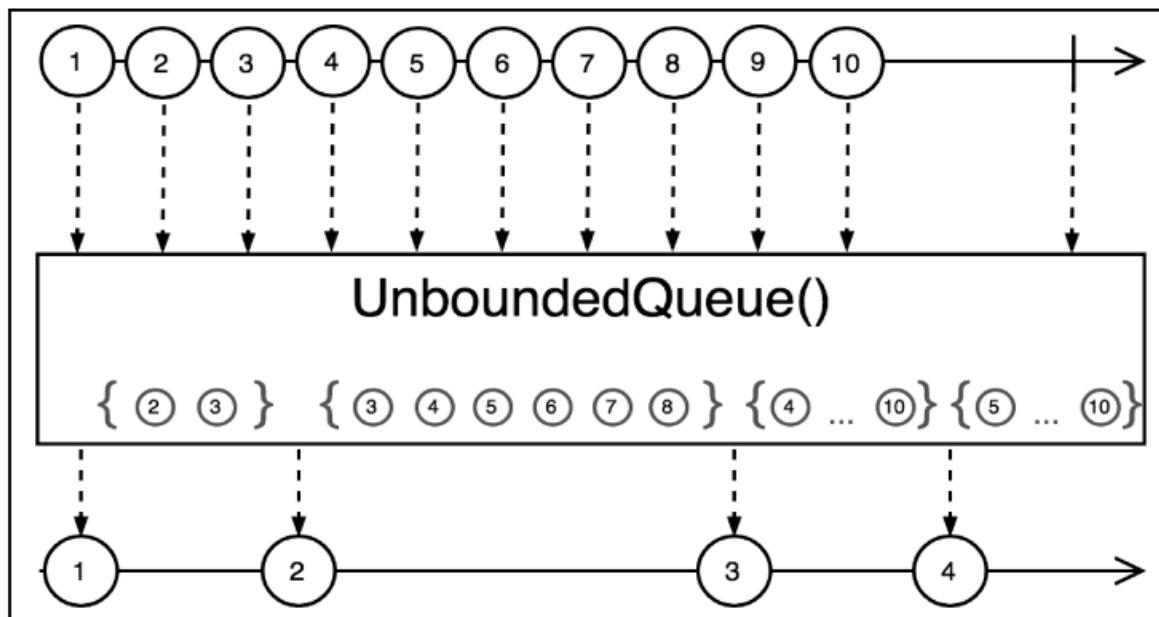
1. 慢生产者和快消费者
2. 快生产者和慢消费者

使用队列处理所推送数据的关键要素之一是选择具有合适特性的队列。

通常，有3种常见的队列类型：无界队列、有界丢弃队列、有界阻塞队列。

### 无界队列

最明显的解决方案是提供一个具有无限大小特性的队列，简单地说就是一个无界队列。在这种情况下，所有生成的元素首先存储在队列中，然后由实际订阅者进行消费。如下图所示：

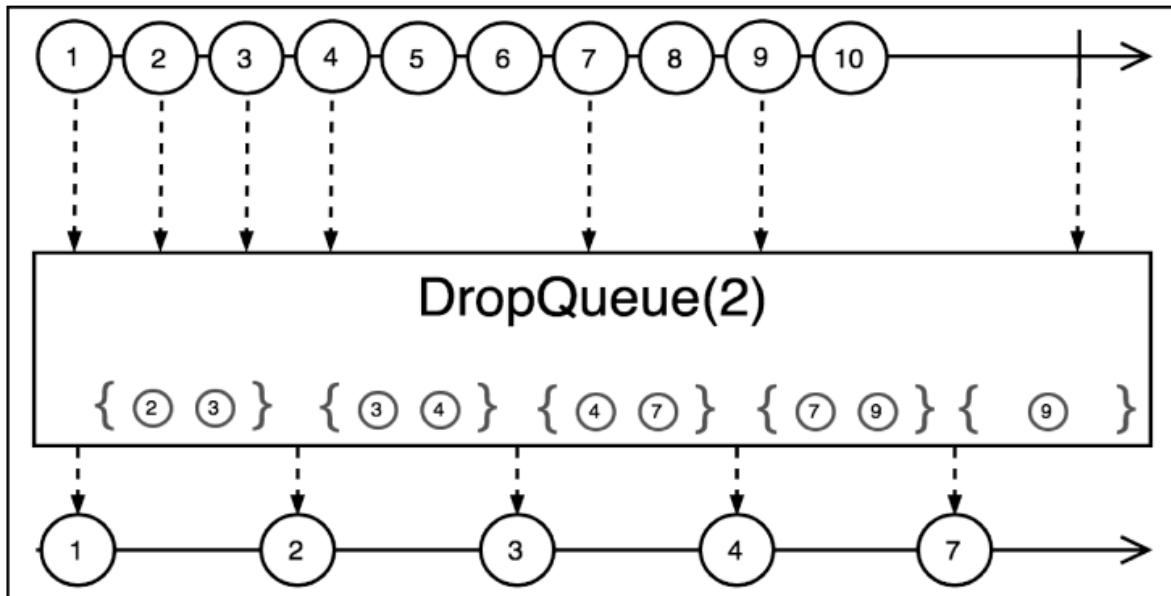


一方面，使用无界队列处理消息带来的核心好处是消息的可传递性，这意味着消费者将在某个时间点及时处理所有存储的元素。

另一方面，只要成功实现消息的可传递性，因为没有无限制的资源，应用程序的回弹性就会降低。例如，一旦内存达到上限，整个系统很容易崩溃。

## 有界丢弃队列

为了避免内存溢出，我们还可以使用在自身已满的情况下可以忽略传入的消息的队列。下图描绘了一个容量为 2 个元素的队列，其特性是在溢出时丢弃元素。



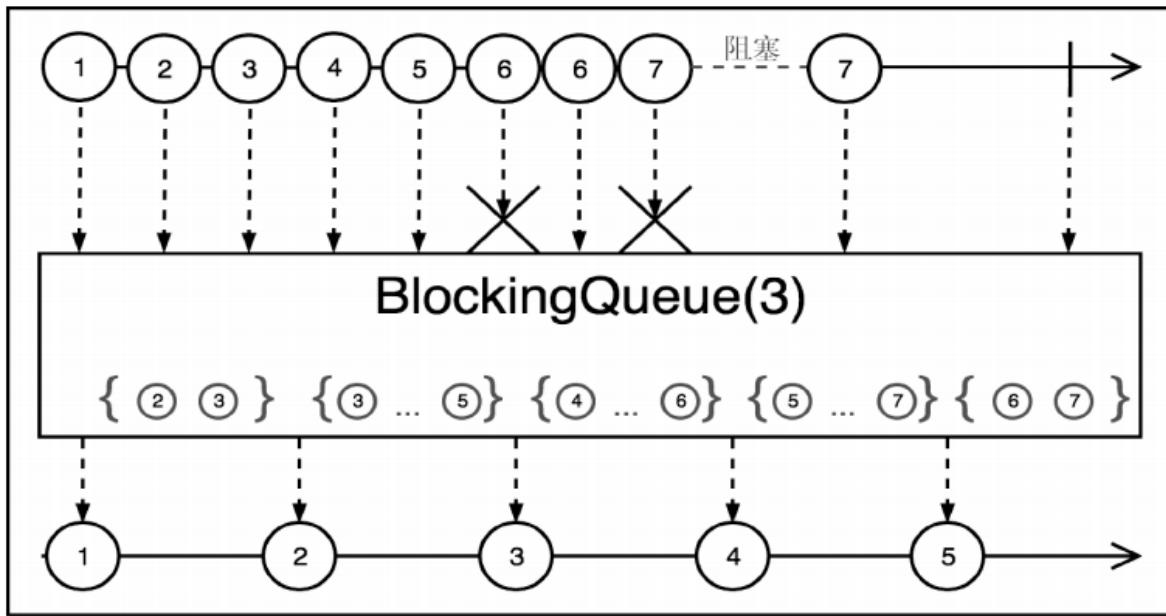
该技术考虑了资源的限制，并且可以根据资源的能力配置队列的容量。当消息的重要性很低时，采用这种队列是一种常见的做法。

一个业务场景的示例可以是一种代表数据集变更的事件流。同时，每个事件都会触发一些重新进行统计计算的操作，该操作使用整个数据集聚合，与传入事件数量相比会花费大量时间。在这种情况下，唯一重要的是数据集已发生变化这一事实，而哪些数据受到影响并不重要。

以上示例是最简单的策略，即丢弃最新元素。通常，有一些策略可以用于选择要丢弃的元素。例如，按优先级丢弃元素、删除最旧的数据等。

## 有界阻塞队列

然而，在每个消息都很重要的情况下，上述技术是不可接受的。例如，支付系统必须处理每个用户提交的支付请求，绝不允许丢弃一部分请求。因此，我们可以在达到上限后阻塞生产者，却不能丢弃消息并保留有界队列以处理被推送的数据。具备阻塞生产者能力的队列通常被称为阻塞队列。下图描述了使用阻塞队列进行交互的示例，该队列的容量为 3 个元素。



遗憾的是，这种技术否定了系统的所有异步行为。通常，一旦生产者达到队列的限制，它就会开始被阻塞并将一直处于该状态，直到消费者消费了一个元素，从而使队列中出现可用空间为止。由此我们可以得出结论，最慢的消费者的吞吐量限制了系统的总吞吐量。继而，除了否定异步行为，该技术还否定了有效的资源利用率。因此，如果我们想要实现回弹性、弹性和即时响应性所有这 3 个方面，那么这些场景全部不可接受。

此外，队列的存在不仅可能使系统的整体设计复杂化，还会增加在上述解决方案之间找到平衡的额外责任。

通常，纯推模型中不受控制的语义可能导致许多我们不希望出现的情况。这就是为什么响应式规范要提到使系统巧妙地响应负载的机制的重要性，即背压控制机制的重要性。

遗憾的是，类似 RxJava 1.x 这样的响应式库并没有提供这样的标准化功能。没有明确的 API 能用于控制开箱即用的背压机制。

应该提到的是，在纯粹的推模式中，我们可以使用批处理来稳定生产速率。RxJava 1.x 提供了诸如`window` 或 `buffer` 之类的操作符，它们可以在指定的时间段内将元素收集到子流或集合中。当对数据库进行例如批量插入或批量更新等操作时，此类技术的性能会出现爆发式提升。遗憾的是，并非所有服务都支持批量操作。因此，这种技术在应用中确实受到限制。

## 2.4 解决方案

2013 年年末，来自 Lightbend、Netflix 和 Pivotal 的一群天才工程师齐聚一堂，共同解决上述问题并为 JVM 社区提供标准。

经过长达一年的努力，[响应式流规范](#)的初稿公诸于世。

其概念就是响应式编程模式的标准化。

## 3 响应式流规范

### 3.1 响应式流规范基础

响应式流规范网址：

<http://www.reactive-streams.org>

响应式规范发布了一组接口，用于实现。

maven仓库地址：

```
1 <dependency>
2   <groupId>org.reactivestreams</groupId>
3   <artifactId>reactive-streams</artifactId>
4   <version>1.0.3</version>
5 </dependency>
6 <dependency>
7   <groupId>org.reactivestreams</groupId>
8   <artifactId>reactive-streams-tck</artifactId>
9   <version>1.0.3</version>
10 </dependency>
11 <dependency>
12   <groupId>org.reactivestreams</groupId>
13   <artifactId>reactive-streams-tck-flow</artifactId>
14   <version>1.0.3</version>
15 </dependency>
16 <dependency>
17   <groupId>org.reactivestreams</groupId>
18   <artifactId>reactive-streams-examples</artifactId>
19   <version>1.0.3</version>
20 </dependency>
```

响应式流的规范文档：

<https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.3/README.md>

#### 响应式流规范：

响应式流（Reactive Streams）规范，规定了异步组件之间使用**背压**进行交互。

响应式流在Java 9中使用Flow API适配。Flow API是互操作的规范，而不是具体的实现，它的语义跟响应式流规范一致。

响应式流规范包括如下接口：

#### Publisher

表示数据流的生产者或数据源，包含一个方法让订阅者注册到发布者，Publisher 代表了发布者和订阅者直接连接的标准化入口点。

```
1 public interface Publisher<T> {  
2     public void subscribe(Subscriber<? super T> s);  
3 }
```

### Subscriber:

表示消费者，onSubscribe 方法为我们提供了一种标准化的方式来通知 Subscriber 订阅成功。

```
1 public interface Subscriber<T> {  
2     public void onSubscribe(Subscription s);  
3     public void onNext(T t);  
4     public void onError(Throwable t);  
5     public void onComplete();  
6 }
```

- `onSubscribe` 发布者在开始处理之前调用，并向订阅者传递一个订阅票据对象（Subscription）。
- `onNext` 用于通知订阅者发布者发布了新的数据项。
- `onError` 用于通知订阅者，发布者遇到了异常，不再发布数据事件。
- `onComplete` 用于通知订阅者所有的数据事件都已发布完。

### Subscription:

同时，onSubscribe 方法的传入参数引入一个名为 Subscription（订阅）的订阅票据。

Subscription 为控制元素的生产提供了基础。

有如下方法：

```
1 public interface Subscription {  
2     public void request(long n);  
3     public void cancel();  
4 }
```

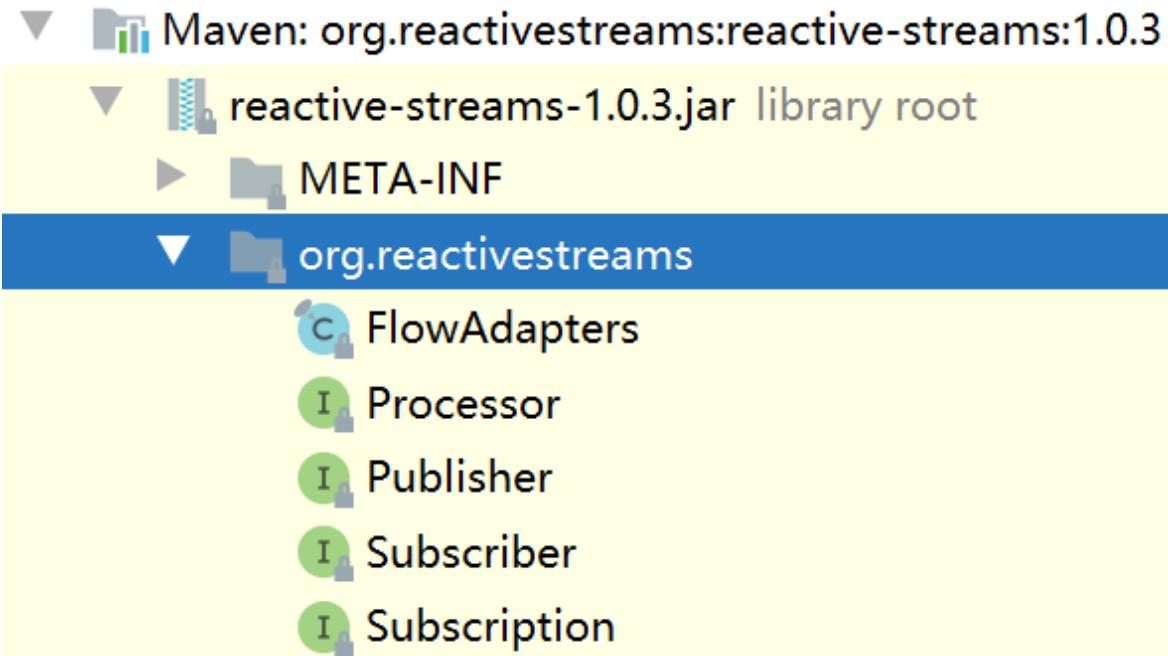
- `request` 用于让订阅者通知发布者随后需要发布的元素数量。
- `cancel` 用于让订阅者取消发布者随后的事件流。

### Processor:

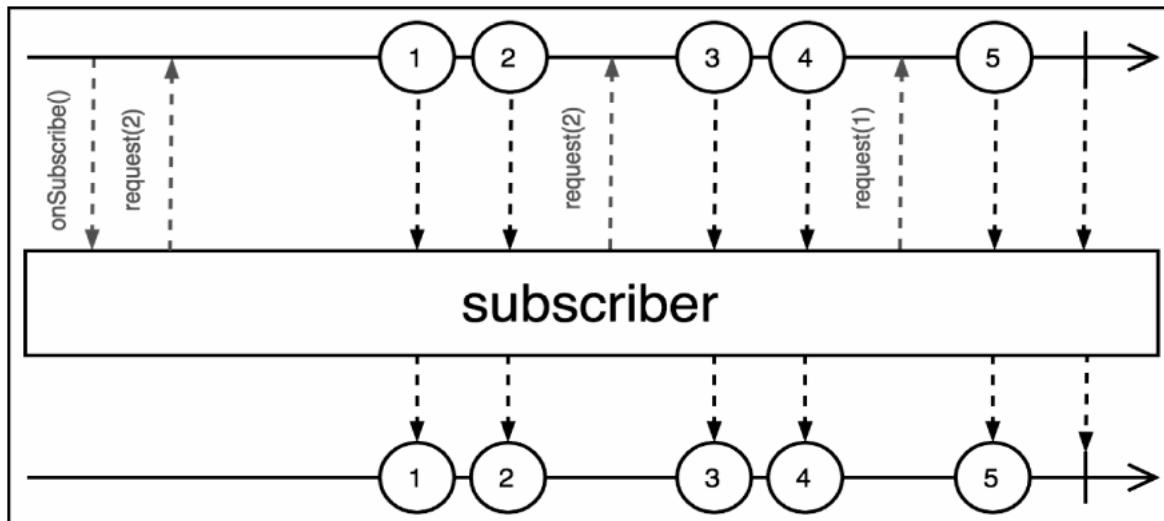
如果实体需要转换进来的项目，并将转换后的项目传递给另一个订阅者，此时需要Processor接口。该接口既是订阅者，又是发布者：

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
2 }
```

所有接口都存在于 org.reactivestreams 包中：



底层机制如下图：



Publisher 保证只有在 Subscriber 要求时才发送元素中新的部分。

Publisher 的整体实现既可以采用纯粹的阻塞等待，也可以采用仅在 Subscriber 请求下才生成数据的复杂机制。

该规范为我们提供了混合推拉模型，此模型可以对背压进行合理控制。

另外，在某些情况下，可以优先考虑纯推模型。响应式流非常灵活，除动态推拉模型外，该规范还提供了独立的推模型和拉模型。根据文档，为了实现纯推模型，我们可以考虑请求  $2^{63}-1$  (java.lang.Long.MAX\_VALUE) 个元素的需求。

这个数字可以被认为是无界的，因为对于当前或可预见的硬件而言，在合理的时间内满足  $2^{63}-1$  个元素的需求是不可能的（即使每纳秒 1 个元素也需要 292 年）。因此，发布者可以停止跟踪超出此要求的需求。

相反，要切换到纯拉模型，可以在 Subscriber.onNext 方法中请求新元素。

## 3.2 响应式流规范实战

首先，响应式规范地址：

<https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.3/README.md#specification>

添加依赖：

```
1 <dependencies>
2   <dependency>
3     <groupId>org.reactivestreams</groupId>
4     <artifactId>reactive-streams</artifactId>
5     <version>1.0.3</version>
6   </dependency>
7   <dependency>
8     <groupId>org.reactivestreams</groupId>
9     <artifactId>reactive-streams-tck</artifactId>
10    <version>1.0.3</version>
11    <scope>test</scope>
12  </dependency>
13 </dependencies>
```

异步发布者：

```
1 package com.lagou.webflux.demo;
2
3 import org.reactivestreams.Publisher;
4 import org.reactivestreams.Subscriber;
5 import org.reactivestreams.Subscription;
6
7 import java.util.Iterator;
8 import java.util.Collections;
9 import java.util.concurrent.Executor;
10 import java.util.concurrent.atomic.AtomicBoolean;
```

```
11 import java.util.concurrent.ConcurrentLinkedQueue;
12
13 /**
14 * Publisher的实现，使用指定的Executor异步执行，并为指定的Iterable生成元素。
15 * 以unicast的形式为Subscriber指定的Iterable生成元素。
16 * <p>
17 * 代码中使用了很多try-catch，用于展示什么时候可以抛异常，什么时候不能抛异常。
18 */
19 public class AsyncIterablePublisher<T> implements Publisher<T> {
20     // 默认的批次大小
21     private final static int DEFAULT_BATCHSIZE = 1024;
22
23     // 用于生成数据的数据源或生成器
24     private final Iterable<T> elements;
25
26     // 线程池，Publisher使用线程池为它的订阅者异步执行
27     private final Executor executor;
28
29     // 既然使用了线程池，就不要在一个线程中执行太多的任务。
30     // 此处使用批次大小调节单个线程的执行时长
31     private final int batchSize;
32
33 /**
34 * @param elements 元素生成器
35 * @param executor 线程池
36 */
37 public AsyncIterablePublisher(final Iterable<T> elements, final
Executor executor) {
38     // 调用重载的构造器，使用默认的批次大小，指定的数据源和指定的线程池
39     this(elements, DEFAULT_BATCHSIZE, executor);
40 }
41
42 /**
43 * 构造器，构造Publisher实例
44 *
45 * @param elements 元素发生器
46 * @param batchSize 批次大小
47 * @param executor 线程池
48 */
49 public AsyncIterablePublisher(final Iterable<T> elements, final int
batchSize, final Executor executor) {
50     // 如果不指定元素发生器则抛异常
51     if (elements == null) throw null;
52     // 如果不指定线程池，抛异常
53     if (executor == null) throw null;
54     // 如果批次大小小于1抛异常：批次大小必须是大于等于1的值
55     if (batchSize < 1) throw new IllegalArgumentException("batchsize
must be greater than zero!");
56     // 赋值元素发生器
57     this.elements = elements;
58     // 赋值线程池
59     this.executor = executor;
60     // 赋值批次大小
61     this.batchsize = batchSize;
62 }
63
64 /**
65 * 订阅者订阅的方法
```

```
66     * 规范2.13指出，该方法必须正常返回，不能抛异常等。。。
67     * 在当前实现中，我们使用unicast的方式支持多个订阅者
68     *
69     * @param s 订阅者
70     */
71    @Override
72    public void subscribe(final Subscriber<? super T> s) {
73        new SubscriptionImpl(s).init();
74    }
75
76    // These represent the protocol of the `AsyncIterablePublishers`
77    SubscriptionImpls
78    // 静态接口：信号
79    static interface Signal {
80    }
81
82    // 取消订阅的信号
83    enum Cancel implements Signal {Instance;}
84
85    // 订阅的信号
86    enum Subscribe implements Signal {Instance;}
87
88    // 发送的信号
89    enum Send implements Signal {Instance;}
90
91    // 静态类，表示请求信号
92    static final class Request implements Signal {
93        final long n;
94
95        Request(final long n) {
96            this.n = n;
97        }
98    }
99
100   // 订阅票据，实现了Subscription接口和Runnable接口
101   final class SubscriptionImpl implements Subscription, Runnable {
102       // 需要保有Subscriber的引用，以用于通信
103       final Subscriber<? super T> subscriber;
104       // 该订阅票据是否失效的标志
105       private boolean cancelled = false;
106
107       // 跟踪当前请求
108       // 记录订阅者的请求，这些请求还没有对订阅者回复
109       private long demand = 0;
110       // 需要发送给订阅者（Subscriber）的数据流指针
111       private Iterator<T> iterator;
112
113       SubscriptionImpl(final Subscriber<? super T> subscriber) {
114           // 根据规范，如果Subscriber为null，需要抛空指针异常，此处抛null。
115           if (subscriber == null) throw null;
116           this.subscriber = subscriber;
117       }
118
119       // 该队列记录发送给票据的信号（入站信号），如"request", "cancel"等。
120       // 通过该Queue，可以在Publisher端使用多线程异步处理。
121       private final ConcurrentLinkedQueue<Signal> inboundSignals = new
ConcurrentLinkedQueue<Signal>();
```

```
122 // 确保当前票据不会并发的标志
123 // 防止在调用订阅者的onXXX方法的时候并发调用。规范1.3规定的不能并发。
124 private final AtomicBoolean on = new AtomicBoolean(false);
125
126 // 注册订阅者发送来的请求
127 private void doRequest(final long n) {
128     // 规范规定，如果请求的元素个数小于1，则抛异常
129     // 并在异常信息中指明错误的原因：n必须是正整数。
130     if (n < 1)
131         terminateDueTo(new IllegalArgumentException(subscriber + "violated the Reactive Streams rule 3.9 by requesting a non-positive number of elements."));
132     // demand + n < 1表示long型数字越界，表示订阅者请求的元素数量大于Long.MAX_VALUE
133     else if (demand + n < 1) {
134         // 根据规范 3.17，当请求的元素数大于Long.MAX_VALUE的时候，将请求数设置为Long.MAX_VALUE即可。
135         // 此时数据流认为是无界流。
136         demand = Long.MAX_VALUE;
137         // 开始向下游发送数据元素。
138         doSend();
139     } else {
140         // 记录下游请求的元素个数
141         demand += n;
142         // 开始向下游发送数据元素。
143         doSend();
144     }
145 }
146
147 // 规范3.5指明，Subscription.cancel方法必须及时的返回，保持调用者的响应性，还必须是幂等的，必须是线程安全的。
148 // 因此该方法不能执行密集的计算。
149 private void doCancel() {
150     cancelled = true;
151 }
152
153 // 不是在`Publisher.subscribe`方法中同步地调用`subscriber.onSubscribe`方法，而是异步地执行`subscriber.onSubscribe`方法
154 // 这样可以避免在调用线程执行用户的代码。因为在订阅者的onSubscribe方法中要执行Iterable.iterator方法。
155 // 异步处理也无形中遵循了规范的1.9。
156 private void doSubscribe() {
157     try {
158         // 获取数据源的迭代器
159         iterator = elements.iterator();
160         if (iterator == null)
161             // 如果iterator是null，就重置为空集合的迭代器。我们假设
162             iterator永远不是null值。
163             iterator = Collections.<T>emptyList().iterator();
164     } catch (final Throwable t) {
165         // Publisher发生了异常，此时需要通知订阅者onError信号。
166         // 但是规范1.9指定了在通知订阅者其他信号之前，必须先通知订阅者
167         onSubscribe信号。
168         // 因此，此处通知订阅者onSubscribe信号，发送空的订阅票据
169         subscriber.onSubscribe(new Subscription() {
170             @Override
171             public void cancel() {
172                 // 空的
173             }
174         });
175     }
176 }
```

```

171 }
172
173     @Override
174     public void request(long n) {
175         // 空的
176     }
177 });
178 // 根据规范1.9, 通知订阅者onError信号
179 terminateDueTo(t);
180 }
181
182 if (!cancelled) {
183     // 为订阅者设置订阅票据。
184     try {
185         // 此处的this就是Subscription的实现类SubscriptionImpl的对象。
186         subscriber.onSubscribe(this);
187     } catch (final Throwable t) {
188         // Publisher方法抛异常, 此时需要通知订阅者onError信号。
189         // 但是根据规范2.13, 通知订阅者onError信号之前必须先取消该订阅者的订阅票据。
190         // Publisher记录下异常信息。
191         terminateDueTo(new IllegalStateException(subscriber + " violated the Reactive Streams rule 2.13 by throwing an exception from onSubscribe.", t));
192     }
193
194     // 立即处理已经完成的迭代器
195     boolean hasElements = false;
196     try {
197         // 判断是否还有未发送的数据, 如果没有, 则向订阅者发送onComplete信号
198         hasElements = iterator.hasNext();
199     } catch (final Throwable t) {
200         // 规范1.4规定
201         // 如果hasNext发生异常, 必须向订阅者发送onError信号, 发送信号之前先取消订阅
202         // 规范1.2规定, Publisher通过向订阅者通知onError或onComplete信号,
203         // 发送少于订阅者请求的onNext信号。
204         terminateDueTo(t);
205     }
206
207     // 如果没有数据发送了, 表示已经完成, 直接发送onComplete信号终止订阅票据。
208     // 规范1.3规定, 通知订阅者onXXX信号, 必须串行, 不能并发。
209     if (!hasElements) {
210         try {
211             // 规范1.6指明, 在通知订阅者onError或onComplete信号之前, 必须先取消订阅者的订阅票据。
212             // 在发送onComplete信号之前, 考虑一下, 有可能是Subscription取消了订阅。
213             docancel();
214             subscriber.onComplete();
215         } catch (final Throwable t) {
216             // 规范2.13指出, onComplete信号不允许抛异常, 因此此处只能记录下来日志

```

```
217                     (new IllegalStateException(subscriber + " violated  
the Reactive Streams rule 2.13 by throwing an exception from onComplete.",  
t)).printStackTrace(System.err);  
218                 }  
219             }  
220         }  
221     }  
222  
223     // 向下游发送元素的方法  
224     private void doSend() {  
225         try {  
226             // 为了充分利用Executor, 我们最多发送batchSize个元素, 然后放弃当前  
线程, 重新调度, 通知订阅者onNext信号。  
227             int leftInBatch = batchSize;  
228             do {  
229                 T next;  
230                 boolean hasNext;  
231                 try {  
232                     // 在订阅的时候已经调用过hasNext方法了, 直接获取元素  
233                     next = iterator.next();  
234                     // Need to keep track of End-of-Stream  
235                     // 检查还有没有数据, 如果没有, 表示流结束了  
236                     hasNext = iterator.hasNext();  
237                 } catch (final Throwable t) {  
238                     // If `next` or `hasNext` throws (they can, since  
it is user-provided), we need to treat the stream as errored as per rule  
1.4  
239                     // 如果next方法或hasNext方法抛异常(用户提供的), 认为流抛  
异常了发送onError信号  
240                     terminateDueTo(t);  
241                     return;  
242                 }  
243                 // Then we signal the next element downstream to the  
`Subscriber`  
244                 // 向下游的订阅者发送onNext信号  
245                 subscriber.onNext(next);  
246                 // 如果已经到达流的结尾  
247                 if (!hasNext) {  
248                     // We need to consider this `subscription` as  
cancelled as per rule 1.6  
249                     // 首先考虑是票据取消了订阅  
250                     doCancel();  
251                     // Then we signal `onComplete` as per rule 1.2 and  
1.5  
252                     // 发送onComplete信号给订阅者  
253                     subscriber.onComplete();  
254                 }  
255             } while (!cancelled) // 如果没有取消订阅。This makes  
sure that rule 1.8 is upheld, i.e. we need to stop signalling "eventually"  
256                     && --leftInBatch > 0 // 如果还有剩余批次的元素。This  
makes sure that we only send `batchSize` number of elements in one go (so  
we can yield to other Runnables)  
257                     && --demand > 0); // 如果还有订阅者的请求。This  
makes sure that rule 1.1 is upheld (sending more than was demanded)  
258  
259                     // If the `Subscription` is still alive and well, and we  
have demand to satisfy, we signal ourselves to send more data  
260                     // 如果订阅票据没有取消, 还有请求, 通知自己发送更多的数据
```

```
261             if (!cancelled && demand > 0)
262                 signal(Send.Instance);
263         } catch (final Throwable t) {
264             // We can only get here if `onNext` or `onComplete` threw,
265             // and they are not allowed to according to 2.13, so we can only cancel and
266             // log here.
267             // 如果到这里，只能是onNext或onComplete抛异常，只能取消。
268             // Make sure that we are cancelled, since we cannot do
269             anything else since the `Subscriber` is faulty.
270             // 确保已取消，因为是Subscriber的问题
271             docancel();
272             // 记录错误信息
273             (new IllegalStateException(subscriber + " violated the
274             Reactive Streams rule 2.13 by throwing an exception from onNext or
275             onComplete.", t)).printStackTrace(System.err);
276         }
277     }
278
279 /**
280 * 规范1.6指出，`Publisher`在通知订阅者`onError`或者`onComplete`信号之前，
281 * ***必须***先取消订阅者的订阅票据（`Subscription`）。
282 *
283 * 当发送onError信号之前先取消订阅
284 * @param t
285 */
286 private void terminateDueTo(final Throwable t) {
287     // 当发送onError之前，先取消订阅票据
288     cancelled = true;
289     try {
290         // 给下游Subscriber发送onError信号
291         subscriber.onError(t);
292     } catch (final Throwable t2) {
293         // 规范1.9指出，onError不能抛异常。
294         // 如果onError抛异常，只能记录信息。
295         (new IllegalStateException(subscriber + " violated the
296             Reactive Streams rule 2.13 by throwing an exception from onError.",
297             t2)).printStackTrace(System.err);
298     }
299 }
300
301 /**
302 * 该方法异步地给订阅票据发送指定信号
303 */
304 private void signal(final Signal signal) {
305     // 入站信号的队列，不需要检查是否为null，因为已经实例化过
306     ConcurrentLinkedQueue<Signal> inboundSignals = new ConcurrentLinkedQueue<Signal>();
307     // 将信号添加到入站信号队列中
308     if (inboundSignals.offer(signal))
309         // 信号入站成功，调度线程处理
310         tryScheduleToExecute();
311 }
312
313 /**
314 * 主事件循环
315 */
316 @Override
317 public final void run() {
318     // 与上次线程执行建立happens-before关系，防止并发执行
319     // 如果on.get()为false，则不执行，线程退出
320     // 如果on.get()为false，则表示没有线程在执行，当前线程可以执行
321     if (on.get())
322         tryScheduleToExecute();
323 }
```

```

310     try {
311         // 从队列取出一个入站信号
312         final Signal s = inboundSignals.poll();
313         // 规范1.8: 如果`Subscription`被取消了, 则必须最终停止向
314         // `Subscriber`发送通知。
315         // 规范3.6: 如果取消了`Subscription`, 则随后调用
316         // `Subscription.request(long n)`必须是无效的(NOPS)。
317         // 如果订阅票据没有取消
318         if (!cancelled) {
319             // 根据信号的类型调用对应的方法进行处理
320             if (s instanceof Request)
321                 // 请求
322                 doRequest(((Request) s).n);
323             else if (s == Send.Instance)
324                 // 发送
325                 doSend();
326             else if (s == Cancel.Instance)
327                 // 取消
328                 doCancel();
329             else if (s == Subscribe.Instance)
330                 // 订阅
331                 doSubscribe();
332         }
333     } finally {
334         // 保证与下一个线程调度的happens-before关系
335         on.set(false);
336         // 如果还有信号要处理
337         if (!inboundSignals.isEmpty())
338             // 调度当前线程进行处理
339             tryScheduleToExecute();
340     }
341 }

342     // 该方法确保订阅票据同一个时间在同一个线程运行
343     // 规范1.3规定, 调用`Subscriber`的`onSubscribe`, `onNext`, `onError`和
344     // `onComplete`方法必须串行, 不允许并发。
345     private final void tryScheduleToExecute() {
346         // CAS原子性地设置on的值为true, 表示已经有一个线程正在处理了
347         if (on.compareAndSet(false, true)) {
348             try {
349                 // 向线程池提交任务运行
350                 executor.execute(this);
351             } catch (Throwable t) {
352                 if (!cancelled) {
353                     // 首先, 错误不可恢复, 先取消订阅
354                     doCancel();
355                     try {
356                         // 停止
357                         terminateDueTo(new
358                             IllegalStateException("Publisher terminated due to unavailable Executor.",
359                             t));
360                     } finally {
361                         // 后续的入站信号不需要处理了, 清空信号
362                         inboundSignals.clear();
363                         // 取消当前订阅票据, 但是让该票据处于可调度状态, 以防清
364                         空入站信号之后又有入站信号加入。
365                     }
366                 }
367             }
368         }
369     }

```

```

362                     on.set(false);
363                 }
364             }
365         }
366     }
367 }
368
369 /**
370 * Subscription.request的实现，接收订阅者的请求给Subscription，等待处理。
371 *
372 * @param n 订阅者请求的元素数量
373 */
374 @Override
375 public void request(final long n) {
376     signal(new Request(n));
377 }
378
379 /**
380 * 订阅者取消订阅。
381 * Subscription.cancel的实现，用于通知Subscription，Subscriber不需要更多元素了。
382 */
383 @Override
384 public void cancel() {
385     signal(Cancel.INSTANCE);
386 }
387
388 // init方法的设置，用于确保SubscriptionImpl实例在暴露给线程池之前已经构造完成
389 // 因此，在构造器一完成，就调用该方法，仅调用一次。
390 // 先发个信号试一下
391 void init() {
392     signal(Subscribe.INSTANCE);
393 }
394 }
395
396 }

```

异步订阅者：

```

1 package com.tagou.webflux.demo;
2
3 import org.reactivestreams.Subscriber;
4 import org.reactivestreams.Subscription;
5
6 import java.util.concurrent.Executor;
7 import java.util.concurrent.atomic.AtomicBoolean;
8 import java.util.concurrent.ConcurrentLinkedQueue;
9
10 /**
11 * 基于Executor的异步运行的订阅者实现，一次请求一个元素，然后对每个元素调用用户定义的方法进行处理。
12 * 注意：该类中使用了很多try-catch用于说明什么时候可以抛异常，什么时候不可以抛异常
13 */

```

```
14 public abstract class AsyncSubscriber<T> implements Subscriber<T>, Runnable
15 {
16     // Signal表示发布者和订阅者之间的异步协议
17     private static interface Signal {
18     }
19
20     // 表示数据流发送完成，完成信号
21     private enum onComplete implements Signal {Instance;}
22
23     // 表示发布者给订阅者的异常信号
24     private static class OnError implements Signal {
25         public final Throwable error;
26
27         public OnError(final Throwable error) {
28             this.error = error;
29         }
30     }
31
32     // 表示下一个数据项信号
33     private static class OnNext<T> implements Signal {
34         public final T next;
35
36         public OnNext(final T next) {
37             this.next = next;
38         }
39     }
40
41     // 表示订阅者的订阅成功信号
42     private static class OnSubscribe implements Signal {
43         public final Subscription subscription;
44
45         public OnSubscribe(final Subscription subscription) {
46             this.subscription = subscription;
47         }
48     }
49
50     // 订阅单据，根据规范3.1，该引用是私有的
51     private Subscription subscription;
52
53     // 用于表示当前的订阅者是否处理完成
54     private boolean done;
55
56     // 根据规范的2.2条款，使用该线程池异步处理各个信号
57     private final Executor executor;
58
59 /**
60 * 仅有这一个构造器，只能被子类调用
61 * 传递一个线程池即可
62 * @param executor 线程池对象
63 */
64 protected AsyncSubscriber(Executor executor) {
65     if (executor == null) throw null;
66     this.executor = executor;
67 }
68
69 /**
70 * 隐等地标记当前订阅者已完成处理，不再处理更多的元素。
```

```
71     * 因此，需要取消订阅票据（Subscription）
72     */
73     private final void done() {
74         // 在此处，可以添加done，对订阅者的完成状态进行设置：
75         // 虽然规范3.7规定Subscription.cancel()是幂等的，我们不需要这么做。
76         // 当whenNext方法抛异常，认为订阅者已经处理完成（不再接收更多元素）
77         done = true;
78
79         //
80         if (subscription != null) { // If we are bailing out before we got
81             a `Subscription` there's little need for cancelling it.
82             try {
83                 // 取消订阅票据
84                 subscription.cancel();
85             } catch (final Throwable t) {
86                 // 根据规范条款3.15，此处不能抛异常，因此只是记录下来。
87                 (new IllegalStateException(subscription + " violated the
88                 Reactive Streams rule 3.15 by throwing an exception from cancel."),
89                 t).printStackTrace(System.err);
90             }
91         }
92     }
93
94     // This method is invoked when the OnNext signals arrive
95     // Returns whether more elements are desired or not, and if no more
96     // elements are desired,
97     // for convenience.
98
99     /**
100      *
101      * @param element
102      * @return
103      */
104     protected abstract boolean whenNext(final T element);
105
106     /**
107      *
108      */
109     protected void whenComplete() {
110
111     // This method is invoked if the OnError signal arrives
112     // override this method to implement your own custom onComplete logic.
113
114     /**
115      *
116      * @param error
117      */
118     protected void whenError(Throwable error) {
119
120     /**
121      *
122      * @param s
123      */
124 }
```

```
125     private final void handleOnSubscribe(final Subscription s) {
126         if (s == null) {
127             // Getting a null `Subscription` here is not valid so lets just
128             // ignore it.
129         } else if (subscription != null) { // If someone has made a mistake
130             and added this Subscriber multiple times, let's handle it gracefully
131             try {
132                 s.cancel(); // Cancel the additional subscription to follow
133                 rule 2.5
134             } catch (final Throwable t) {
135                 //Subscription.cancel is not allowed to throw an exception,
136                 according to rule 3.15
137                 (new IllegalStateException(s + " violated the Reactive
138                 Streams rule 3.15 by throwing an exception from cancel.",
139                 t)).printStackTrace(System.err);
140             }
141         } else {
142             // We have to assign it locally before we use it, if we want to
143             be a synchronous `Subscriber`
144             // Because according to rule 3.10, the Subscription is allowed
145             to call `onNext` synchronously from within `request`
146             subscription = s;
147             try {
148                 // If we want elements, according to rule 2.1 we need to
149                 call `request`
150                 // And, according to rule 3.2 we are allowed to call this
151                 synchronously from within the `onSubscribe` method
152                 s.request(1); // our Subscriber is unbuffered and modest,
153                 it requests one element at a time
154             } catch (final Throwable t) {
155                 // Subscription.request is not allowed to throw according
156                 to rule 3.16
157                 (new IllegalStateException(s + " violated the Reactive
158                 Streams rule 3.16 by throwing an exception from request.",
159                 t)).printStackTrace(System.err);
160             }
161         }
162     }
163     /**
164      *
165      * @param element
166      */
167     private final void handleOnNext(final T element) {
168         if (!done) { // If we aren't already done
169             if (subscription == null) { // Technically this check is not
170             needed, since we are expecting Publishers to conform to the spec
171                 // Check for spec violation of 2.1 and 1.09
172                 (new IllegalStateException("Someone violated the Reactive
173                 Streams rule 1.09 and 2.1 by signalling OnNext before
174                 `Subscription.request`. (no Subscription)").printStackTrace(System.err));
175             } else {
176                 try {
177                     if (whenNext(element)) {
178                         try {
179                             subscription.request(1); // our Subscriber is
180                             unbuffered and modest, it requests one element at a time
181                         } catch (final Throwable t) {
182                         }
183                     }
184                 }
185             }
186         }
187     }
188 }
```

```

165                                     // Subscription.request is not allowed to throw
166                                     according to rule 3.16
167                                     (new IllegalStateException(subscription + "
168                                     violated the Reactive Streams rule 3.16 by throwing an exception from
169                                     request.", t)).printStackTrace(System.err);
170                                     }
171                                     } else {
172                                         done(); // This is legal according to rule 2.6
173                                         }
174                                     } catch (final Throwable t) {
175                                         done();
176                                         try {
177                                             onError(t);
178                                         } catch (final Throwable t2) {
179                                             //Subscriber.onError is not allowed to throw an
180                                             //exception, according to rule 2.13
181                                             (new IllegalStateException(this + " violated the
182                                             Reactive Streams rule 2.13 by throwing an exception from onError.",
183                                             t2)).printStackTrace(System.err);
184                                         }
185                                     }
186                                     }
187                                     }
188                                     }
189                                     }
190                                     if (subscription == null) { // Technically this check is not
191                                     needed, since we are expecting Publishers to conform to the spec
192                                     // Publisher is not allowed to signal onComplete before
193                                     // onSubscribe according to rule 1.09
194                                     (new IllegalStateException("Publisher violated the Reactive
195                                     Streams rule 1.09 signalling onComplete prior to
196                                     onSubscribe.")).printStackTrace(System.err);
197                                     } else {
198                                         done = true; // Obey rule 2.4
199                                         whenComplete();
200                                         }
201                                         }
202                                         }
203                                         */
204                                         * @param error
205                                         */
206                                         private void handleOnError(final Throwable error) {
207                                         if (subscription == null) { // Technically this check is not
208                                         needed, since we are expecting Publishers to conform to the spec
209                                         // Publisher is not allowed to signal onError before
210                                         // onSubscribe according to rule 1.09

```

```
208             (new IllegalStateException("Publisher violated the Reactive
209             Streams rule 1.09 signalling onError prior to
210             onSubscribe.")).printStackTrace(System.err);
211         } else {
212             done = true; // Obey rule 2.4
213             whenError(error);
214         }
215     }
216
217     /**
218      *
219      * @param s
220      */
221     @Override
222     public final void onSubscribe(final Subscription s) {
223         // As per rule 2.13, we need to throw a
224         `java.lang.NullPointerException` if the `Subscription` is `null`
225         if (s == null) throw null;
226
227         signal(new OnSubscribe(s));
228     }
229
230     /**
231      *
232      * @param element
233      */
234     @Override
235     public final void onNext(final T element) {
236         // As per rule 2.13, we need to throw a
237         `java.lang.NullPointerException` if the `element` is `null`
238         if (element == null) throw null;
239
240         signal(new OnNext<T>(element));
241     }
242
243     /**
244      *
245      * @param t
246      */
247     @Override
248     public final void onError(final Throwable t) {
249         // As per rule 2.13, we need to throw a
250         `java.lang.NullPointerException` if the `Throwable` is `null`
251         if (t == null) throw null;
252
253         signal(new OnError(t));
254     }
255
256     /**
257      *
258      */
259     @Override
260     public final void onComplete() {
261         signal(OnComplete.Instance);
262     }
263 }
```

```
260
261     // This `ConcurrentLinkedQueue` will track signals that are sent to
262     // this `Subscriber`, like `OnComplete` and `OnNext` ,
263     // and obeying rule 2.11
264     /**
265      *
266      */
267     private final ConcurrentLinkedQueue<Signal> inboundSignals = new
268     ConcurrentLinkedQueue<Signal>();
269
270     /**
271      * 根据规范2.7和2.11，使用原子变量确保不会有多个订阅者线程并发执行。
272      */
273     private final AtomicBoolean on = new AtomicBoolean(false);
274
275     /**
276      *
277      */
278     @SuppressWarnings("unchecked")
279     @Override
280     public final void run() {
281         /**
282          * 跟上次线程执行建立happens-before关系，防止多个线程并发执行
283          */
284         if (on.get()) {
285             try {
286                 /**
287                   * 从入站队列取出信号
288                   */
289                 final Signal s = inboundSignals.poll();
290                 /**
291                   * 根据规范条款2.8，如果当前订阅者已完成，就不需要处理了。
292                   */
293                 if (!done) {
294                     /**
295                       * 根据信号类型调用对应的方法来处理
296                       */
297                     if (s instanceof OnNext<?>)
298                         handleOnNext(((OnNext<T>) s).next);
299                     else if (s instanceof OnSubscribe)
300                         handleOnSubscribe(((OnSubscribe) s).subscription);
301                     /**
302                       * 根据规范2.10，必须处理onError信号，不管有没有调用过
303                       */
304                     Subscription.request(long n)方法
305                     else if (s instanceof OnError)
306                         handleOnError(((OnError) s).error);
307                     /**
308                       * 根据规范2.9，必须处理onComplete信号，不管有没有调用过
309                       */
310                     Subscription.request(long n)方法
311                     else if (s == onComplete.Instance)
312                         handleOnComplete();
313                 }
314             } finally {
315                 /**
316                   * 保持happens-before关系，然后开始下一个线程调度执行
317                   */
318                 on.set(false);
319                 /**
320                   * 如果入站信号不是空的，调度线程处理入站信号
321                   */
322                 if (!inboundSignals.isEmpty())
323                     /**
324                       * 调度处理入站信号
325                       */
326                     tryScheduleToExecute();
327             }
328         }
329     }
330
331     /**
332      * what `signal` does is that it sends signals to the `Subscription`
333      * asynchronously
334      */
335     /**
336      * 该方法异步地向订阅票据发送信号
337      */
338 }
```

```

313     * @param signal
314     */
315     private void signal(final Signal signal) {
316         // 信号入站，线程池调度处理
317         // 不需要检查是否为null，因为已经实例化了。
318         if (inboundSignals.offer(signal))
319             // 线程调度处理
320             tryScheduleToExecute();
321     }
322
323 /**
324 * 确保订阅者一次仅在一个线程执行
325 * 调度执行
326 */
327 private final void tryScheduleToExecute() {
328     // 使用CAS原子性地修改变量on的值改为true。
329     if (on.compareAndSet(false, true)) {
330         try {
331             // 提交任务，多线程执行
332             executor.execute(this);
333         } catch (Throwable t) {
334             // 根据规范条款2.13，如果不能执行线程池的提交方法，需要优雅退出
335             if (!done) {
336                 try {
337                     // 由于错误不可恢复，因此取消订阅票据
338                     done();
339                 } finally {
340                     // 不再需要处理入站信号，清空之
341                     inboundSignals.clear();
342                     // 由于订阅票据已经取消，但是此处依然让订阅者处于可调度的状态，以防在清空入站信号之后又有信号发送过来
343                     // 因为信号的发送是异步的
344                     on.set(false);
345                 }
346             }
347         }
348     }
349 }
350 }
```

测试类：

```

1 package com.lagou.webflux.demo;
2
3 import org.reactivestreams.Subscriber;
4 import org.reactivestreams.Subscription;
5
6 import java.util.HashSet;
7 import java.util.Set;
8 import java.util.concurrent.ExecutorService;
9 import java.util.concurrent.Executors;
10
11 public class ReactiveTest {
12     public static void main(String[] args) {
```

```

14     Set<Integer> elements = new HashSet<>();
15     for (int i = 0; i < 10; i++) {
16         elements.add(i);
17     }
18
19     final ExecutorService executorService =
20         Executors.newFixedThreadPool(5);
21
22     AsyncIterablePublisher<Integer> publisher
23         = new AsyncIterablePublisher<>(elements, executorService);
24
25     //     publisher.subscribe(new Subscriber<Integer>() {
26     //         @Override
27     //         public void onSubscribe(Subscription s) {
28     //             //
29     //         }
30     //         //
31     //         @Override
32     //         public void onNext(Integer integer) {
33     //             //
34     //         }
35     //         //
36     //         @Override
37     //         public void onError(Throwable t) {
38     //             //
39     //         }
40     //         //
41     //         @Override
42     //         public void onComplete() {
43     //             //
44     //         }
45     //     });
46
47     final AsyncSubscriber<Integer> subscriber = new AsyncSubscriber<>
(Executors.newFixedThreadPool(2)) {
48         @Override
49         protected boolean whenNext(Integer element) {
50             System.out.println("接收到的流元素: " + element);
51             return true;
52         }
53     };
54
55     publisher.subscribe(subscriber);
56
57 }
58 }
```

### 3.3 响应式流技术兼容套件

### 3.3.1 TCK

响应式流看着比较简单，实际上包含许多隐藏的陷阱。

除 Java 接口之外，该规范还包含许多针对实现的文档化规则。

这些规则严格限制每个接口，同时，保留规范中提到的所有行为至关重要。

开发人员需要一个可以验证所有行为并确保响应式库标准化且相互兼容的通用工具。

Konrad Malawski 已经为此实现了一个工具包，其名称为响应式流技术兼容套件（Reactive Streams Technology Compatibility Kit），简称为 **TCK**。

TCK 是一组 TestNG 测试用例，需要对其进行扩展，并为相应的 Publisher 或 Subscriber 准备验证。

所有测试的命名都对应指定的规则。

例如，可以在 org.reactivestreams.tck.PublisherVerification 中找到如下所示的一个测试用例示例：

```
1  @Override @Test
2  public void
3      required_spec101_subscriptionRequestMustResultInTheCorrectNumberOfProducedEl
4      ements() throws Throwable {
5          activePublisherTest(5, false, new PublisherTestRun<T>() {
6              @Override
7              public void run(Publisher<T> pub) throws InterruptedException {
8                  // 对被测发布者的手动订阅。响应式流的 TCK 提供了自己的测试类，该测试类可以验证特
9                  定的行为。
10                 ManualSubscriber<T> sub = env.newManualSubscriber(pub);
11                 try {
12                     // 根据规则1.01对给定Publisher的行为进行了特定验证。在这种情况下，我们证实
13                     Publisher 发出的元素不能比 Subscriber 请求的更多。
14                     sub.expectNone(String.format("Publisher %s produced value before
15                     the first `request`:", pub));
16                     sub.request(1);
17                     sub.nextElement(String.format("Publisher %s produced no element
18                     after first `request`", pub));
19                     sub.expectNone(String.format("Publisher %s produced unrequested:
", pub));
20
21                     sub.request(1);
22                     sub.request(2);
23                     sub.nextElements(3, env.defaultTimeoutMillis(),
24                         String.format("Publisher %s produced less than 3 elements after two
25                         respective `request` calls", pub));
26                     sub.expectNone(String.format("Publisher %s produced unrequested ", pub));
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
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
307
308
309
309
310
311
311
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
13
```

```
19     } finally {
20         // subscription 的取消阶段。一旦测试通过或失败，我们将关闭打开的资源并完成交互，并使用 ManualSubscriber API 取消对 Publisher 的订阅。
21         sub.cancel();
22     }
23 }
24 });
25 }
```

上述测试的重要性隐藏在对交互基本保证的验证背后，任何一种 Publisher 的实现都应该提供这种交互。

PublisherVerification 中的所有测试用例都确保给定的 Publisher 在某种程度上符合响应式流规范。

这在某种程度上意味着不同大小的规则无法一一验证。此类规则的示例是规则 3.04，该规则声明，请求不应执行无法进行有效测试的繁重计算。

## 发布者验证

为了了解该套件如何工作，我们将验证新闻服务中的一个组件。由于 Publisher 是系统的重要组成部分，因此我们将从对它的分析开始。我们记得，TCK 提供了 org.reactivestreams.tck.PublisherVerification 来检查 Publisher 的基本行为。通常，PublisherVerification 是一个抽象类，它要求我们只扩展两个方法。

让我们看一下下面的例子，以了解如何编写对前面开发的 AsyncIterablePublisher 的验证：

### pom.xml

```
1 <dependencies>
2     <dependency>
3         <groupId>org.reactivestreams</groupId>
4         <artifactId>reactive-streams</artifactId>
5         <version>1.0.3</version>
6     </dependency>
7     <dependency>
8         <groupId>org.reactivestreams</groupId>
9         <artifactId>reactive-streams-tck</artifactId>
10        <version>1.0.3</version>
11        <!--          <scope>test</scope>-->
12    </dependency>
13    <dependency>
14        <groupId>org.testng</groupId>
15        <artifactId>testng</artifactId>
16        <version>6.9.10</version>
17        <!--          <scope>test</scope>-->
18    </dependency>
19    <dependency>
20        <groupId>org.projectlombok</groupId>
21        <artifactId>lombok</artifactId>
22        <version>1.18.16</version>
23    </dependency>
24 </dependencies>
```

```
1 import com.lagou.webflux.demo.unicast.AsyncIterablePublisher;
2 import org.reactivestreams.Publisher;
3 import org.reactivestreams.tck.PublisherVerification;
4 import org.reactivestreams.tck.TestEnvironment;
5
6 import java.util.HashSet;
7 import java.util.Set;
8 import java.util.concurrent.ExecutorService;
9 import java.util.concurrent.Executors;
10
11 public class TCKTest extends PublisherVerification<String> {
12
13     public TCKTest() {
14         super(new TestEnvironment());
15     }
16
17     @Override
18     public Publisher createPublisher(long elements) {
19
20         Set<String> set = new HashSet<>();
21         for (long i = 0; i < elements; i++) {
22             set.add("hello-" + i);
23         }
24
25         ExecutorService executorService = Executors.newFixedThreadPool(5);
26
27         AsyncIterablePublisher publisher = new AsyncIterablePublisher(set,
28                           executorService);
29
30         return publisher;
31     }
32
33     @Override
34     public Publisher createFailedPublisher() {
35         Set set = new HashSet<>();
36         set.add(new RuntimeException("手动异常"));
37         ExecutorService executorService = Executors.newFixedThreadPool(5);
38         return new AsyncIterablePublisher(set, executorService);
39     }
40 }
```

只遵循上述测试配置，无法检查该 Publisher 的准确性，因为许多测试用例假设流中存在多个元素。

响应式流 TCK 考虑到了这种极端情况，并支持设置一个名为 maxElementsFromPublisher()的附加方法，该方法返回一个值，用于指示生成元素的最大数量：

```
1 | @Override
2 | public long maxElementsFromPublisher() {
3 |     // 默认值为Long.MAX_VALUE - 1
4 |     //         return super.maxElementsFromPublisher();
5 |     return 10;
6 | }
```

一方面，重写该方法可以跳过需要多个元素的测试；另一方面，响应式流规则的覆盖范围将减小，可能需要实现自定义测试用例。

## 订阅者验证

上述配置是启动测试生产者行为所需的最小配置。但是，除了 Publisher 的实例，还需要测试 Subscriber 实例。

响应式流规范中针对 Subscriber 的那组规则没有针对 Publisher 的规则那么复杂，但仍然需要满足所有需求。

有两种不同的测试套件可以测试 AsyncSubscriber。

第一个名为 org.reactivestreams.tck.SubscriberBlackboxVerification，它可以在不知道内部的细节和修改的情况下验证 Subscriber。

当 Subscriber 来自外部代码库并且我们没有合法的途径来扩展行为时，黑盒（Blackbox）验证是一个有用的测试工具包。另外，黑盒验证仅涵盖一部分规则，并不能确保实现完全正确。

要了解如何检查 AsyncSubscriber，首先就要实现黑盒验证测试：

```
1 | import com.lagou.webflux.demo.unicast.AsyncSubscriber;
2 | import org.reactivestreams.Subscriber;
3 | import org.reactivestreams.tck.SubscriberBlackboxVerification;
4 | import org.reactivestreams.tck.TestEnvironment;
5 |
6 | import java.util.concurrent.Executors;
7 |
8 | public class TCKBlackBoxTest extends SubscriberBlackboxVerification<Integer>
9 | {
10 |
11 |     protected TCKBlackBoxTest() {
12 |         super(new TestEnvironment());
13 |     }
14 |
15 |     @Override
16 |     public Subscriber<Integer> createSubscriber() {
17 |
18 |         AsyncSubscriber subscriber = new
```

```

19         protected boolean whenNext(Object element) {
20             System.out.println("接收到的元素: " + element);
21             // 返回true, 接着请求下个元素, false表示不再请求了
22             return true;
23         }
24     };
25
26     return subscriber;
27 }
28
29 @Override
30 public Integer createElement(int element) {
31     return element;
32 }
33
34 // @Override
35 // public void triggerRequest(Subscriber<? super Integer> s) {
36 //     // 在该方法直接向订阅者发射信号。默认该方法什么都不做。
37 //     AsyncSubscriber<Integer> subscriber = (AsyncSubscriber<Integer>)
38 //     s;
39 //     subscriber.onNext(10000);
40 // }
41 }

```

上述示例展示了可用于 Subscriber 验证的 API。除了两个必需的方法 createSubscriber 和 createElement，还有一个额外的方法可以从外部处理 Subscription#request 方法。在该例中，能模拟真实的用户活动，是一个有用的补充。

第二个测试工具包被称为 org.reactivestreams.tck.SubscriberWhiteboxVerification。

为了通过验证，Subscriber 需要提供与 WhiteboxSubscriberProbe 的额外交互：

```

1 import com.lagou.webflux.demo.unicast.AsyncSubscriber;
2 import org.reactivestreams.Subscriber;
3 import org.reactivestreams.Subscription;
4 import org.reactivestreams.tck.Subscriberwhiteboxverification;
5 import org.reactivestreams.tck.TestEnvironment;
6
7 import java.util.concurrent.Executors;
8
9 public class TCKwhiteBoxTest extends Subscriberwhiteboxverification<Integer>
{
10
11     protected TCKwhiteBoxTest() {
12         super(new TestEnvironment());
13     }
14
15     @Override
16     public Subscriber<Integer>
createSubscriber(WhiteboxSubscriberprobe<Integer> probe) {

```

```

17     AsyncSubscriber subscriber = new
18     AsyncSubscriber(Executors.newFixedThreadPool(2)) {
19         @Override
20         protected boolean whenNext(Object element) {
21             System.out.println("接收到的元素: " + element);
22             // 返回true, 接着请求下个元素, false表示不再请求了
23             return true;
24         }
25
26         @Override
27         public void onSubscribe(Subscription subscription) {
28             super.onSubscribe(subscription);
29             probe.registerOnSubscribe(new SubscriberPuppet() {
30                 @Override
31                 public void triggerRequest(long elements) {
32                     subscription.request(elements);
33                 }
34
35                 @Override
36                 public void signalCancel() {
37                     subscription.cancel();
38                 }
39             });
40         }
41
42         @Override
43         public void onNext(Object element) {
44             super.onNext(element);
45             // 注册钩子
46             probe.registerOnNext((Integer) element);
47         }
48
49         @Override
50         public void onError(Throwable t) {
51             super.onError(t);
52             probe.registerOnError(t);
53         }
54
55         @Override
56         public void onComplete() {
57             super.onComplete();
58             probe.registerOnComplete();
59         }
60     };
61
62     return subscriber;
63 }
64
65     @Override
66     public Integer createElement(int element) {
67         return element;
68     }

```

关键点解释如下：

`createSubscriber` 方法实现与黑盒验证的工作方式相同，并返回 `Subscriber` 实例，但此处还有一个名为 `WhiteboxSubscriberProbe` 的附加参数。

在这种情况下，`WhiteboxSubscriberProbe` 代表了一种机制，该机制可以实现对需求的嵌入式控制和输入信号的捕获。

与黑盒验证相比，通过正确注册探针钩子，测试套件不仅能够发送需求，还能验证需求是否被满足以及所有元素是否被接受。同时，需求监管机制比以前更加透明。我们实现了 `SubscriberPuppet`，它会为直接访问收到的 `Subscription` 进行适配。

正如上例所述，与黑盒验证相反，白盒（whitebox）需要扩展 `Subscriber`，以在内部提供额外的钩子。虽然白盒测试覆盖了更多的规则，且这些规则确保了所测试 `Subscriber` 的行为正确，但是当我们想要避免一个类被扩展时，这可能是不可接受的。

同时TCK也提供了processor的测试，此处从略。

## 4 响应式流中的异步和并行

一方面，响应式流 API 中的规则 2.2 和 3.4 规定，对由 `Publisher` 生成并由 `Subscriber` 消费的所有信号的**处理过程**应该是非阻塞和非干扰的。

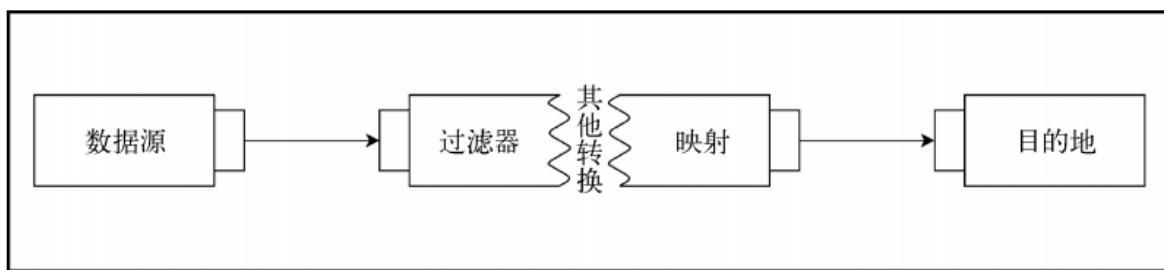
因此，基于具体的执行环境，可以高效地利用处理器的一个节点或一个内核。

另一方面，所有处理器或内核的高效利用需要并行化。对响应式流规范中的并行化概念的通常理解可以解释为对 `Subscriber#onNext` 方法的并行调用。

遗憾的是，规范中的规则 1.3 规定必须以线程安全的方式触发 `onXxx` 方法的调用，并且如果由多个线程执行，则使用外部同步。这一点假定对所有 `onXxx` 方法的串行化或简单顺序调用。反过来，这意味着无法创建类似 `ParallelPublisher` 的组件并在流中对元素执行并行处理。

因此，问题是**如何高效地利用资源**。

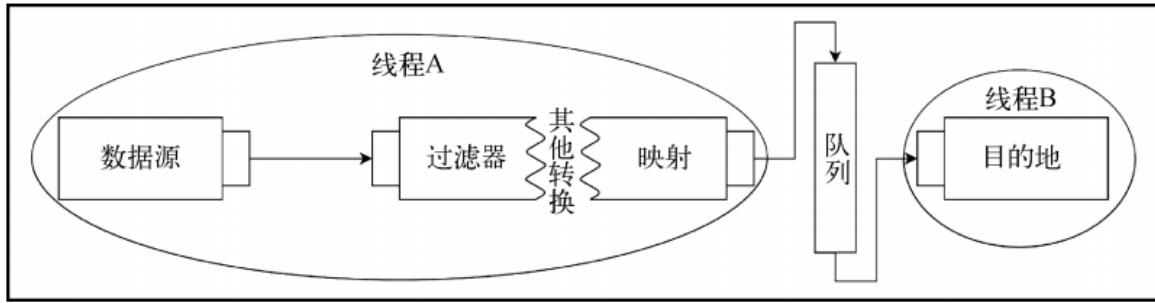
要找到答案，必须分析常见的流处理管道，见下图：



通常的管道处理（涉及数据源和最终目的地）包括一些处理或转换阶段。同时，每个处理阶段可能花费大量处理时间并延迟其他执行。

一种解决方案是在**阶段之间传递异步消息**。对基于内存的流处理而言，这意味着执行过程的一部分被绑定到一个线程而另一部分被绑定到另一个线程。

例如，最终元素消费可能是 CPU 密集型任务，而它将在单独的线程上进行合理处理，见下图：



通常的做法是：在两个独立的线程之间拆分处理过程，在阶段之间放置异步边界。

又因为两个线程可以彼此独立地工作，所以通过这样做，将元素的整体处理过程并行化。为了实现并行化，必须应用一种数据结构（例如 Queue）来正确地解耦处理过程。

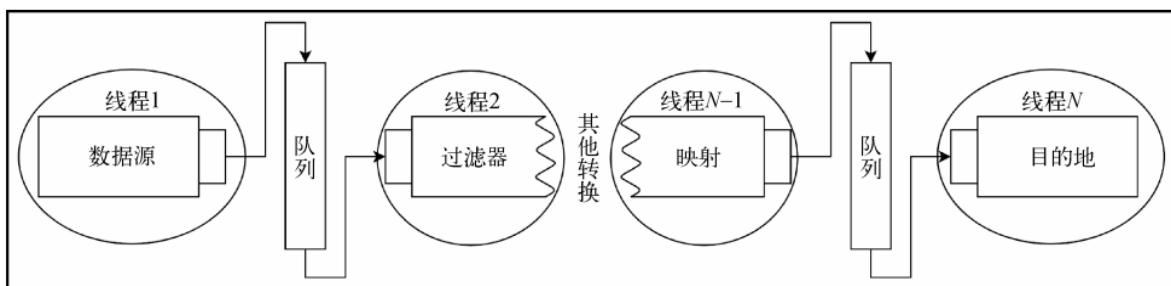
这样，线程 A 内的处理过程独立地提供数据项给 Queue，而在线程 B 内的 Subscriber 则独立地消费来自相同 Queue 的数据项。

拆分线程之间的处理过程会导致数据结构中的额外开销。当然，由于响应式流规范，这样的数据结构是有界的。数据结构中的数据项数量通常等于 Subscriber 从其 Publisher 请求的批处理的大小，而这取决于系统的一般容量。

除此之外，流处理部分应该连接到哪个异步边界？

三个简单的选项：

1. 将处理流附加到数据源资源，并且使所有操作都在与数据源相同的边界内执行。数据源这一侧的边界内，数据通过管道流式处理。
2. 处理过程连接到目的地或消费者线程，可以在元素生产过程为 CPU 密集型任务的场景下使用。
3. 发生在生产和消费是 CPU 密集型任务时。因此，在单独的线程对象上运行它，见下图



每个处理阶段可以绑定到一个单独的线程。

注意：在不同的线程之间对处理过程进行拆分不仅不是自由的，还应该在合理的资源消费之间进行平衡，以实现边界（线程和附加数据结构）和高效的元素处理。

同时，实现这种平衡是另一个挑战，如果响应式库没有提供有用的 API，就很难克服其实现和管理中的困难。

## 5 响应式环境的转变

JDK 9 包含响应式流规范这一事实强调了该规范的重要性，并且该规范已经开始改变这个行业。

开源软件行业的领导者（如 Netflix、Red Hat、Lightbend、MongoDB、亚马逊等）已经开始在他们的产品中采用这种出色的解决方案。

### 5.1 RxJava的转变

RxJava 提供了一个额外的模块，可以轻松地将一种响应式类型转换为另一种。

如何将 `Observable<T>` 转换为 `Publisher<T>` 并将 `rx.Subscriber<T>` 转换为 `org.reactivestreams.Subscriber<T>`？

假设有一个应用程序把 RxJava 1.x 和 Observable 作为组件之间的核心通信类型，如下例所示：

```
1 interface LogService {  
2     Observable<String> stream();  
3 }
```

遵循响应式流规范并从以下特定依赖项中抽象出我们的接口：

```
1 interface LogService {  
2     Publisher<String> stream();  
3 }
```

用 Publisher 替换了 Observable。

但是，实现所需的重构可能比仅更换返回类型花费的时间要多。

但是可以很容易地将现有的 Observable 调整为 Publisher，如下例所示：

```
1 // RxLogService 类声明。该类表示旧的基于 Rx 的实现  
2 class RxLogService implements LogService {  
3     // 使用 RxNettyHttpClient，它能使用封装在基于 RxJava API 中的 Netty Client 以  
4     // 异步、非阻塞方式与外部服务进行交互。  
5     final HttpClient<...> rxClient = HttpClient.newClient(...);  
6  
7     @Override  
8     public Publisher<String> stream() {
```

```
9     // 通过使用创建的 HttpClient 实例，从外部服务请求日志流，并将传入的元素转换为
10    String 实例。
11    Observable<String> rxStream = rxclient.createGet("/logs")
12        .flatMap(...)
13        .map(Utils::toString);
14    // 使用 RxReactiveStreams 库对 Publisher 进行的 rxStream 适配
15    return RxReactiveStreams.toPublisher(rxStream);
16 }
```

可以注意到，RxJava 的开发人员关心我们并提供了一个额外的 RxReactiveStreams 类，使我们可以将 Observable 转换为响应式流中的 Publisher。此外，随着响应式流规范的出现，RxJava 开发人员还提供了非标准化的背压支持，以使转换后的 Observable 兼容响应式流规范。

除将 Observable 转换为 Publisher 之外，我们还可以将 rx.Subscriber 转换为 org.reactivestreams.Subscriber。例如，日志流先前是存储在文件中。为此，我们实现了自定义 Subscriber 以负责 I/O 交互。如此，迁移到响应式流规范的代码变换如下所示：

```
1 class RxFileService implements FileService {
2     @Override
3     public void writeTo(String file, Publisher<String> content) {
4         AsyncFileSubscriber rxSubscriber = new AsyncFileSubscriber(file);
5         content.subscribe(RxReactiveStreams.toSubscriber(rxSubscriber));
6     }
7 }
```

关键点解释如下：

1. 这是 RxFileService 类声明。
2. 这是 writeTo 方法实现，它接受 Publisher 作为组件之间交互的核心类型。
3. 这是基于 RxJava 的 AsyncFileSubscriber 实例声明。
4. 这是 content 订阅。要重用基于 RxJava 的 Subscriber，我们需要使用相同的 RxReactiveStreams 实用工具类对其进行适配。

从前面的示例中可以看出，RxReactiveStreams 提供了一个丰富的转换器列表，使我们可以将 RxJava API 转换为响应式流 API。

同样，任何 Publisher<T>都可以转换回 RxJava Observable：

```
1 Publisher<String> publisher = ...;
2 RxReactiveStreams.toObservable(publisher)
3     .subscribe();
```

总体上讲，RxJava 会以某种方式开始遵循响应式流规范。遗憾的是，由于向后兼容性，实现该规范是不可能的，并且将来为 RxJava 1.x 实现响应式流规范扩展的计划也不存在。此外，从 2018 年 3 月 31 日开始，对 RxJava 1.x 的支持停止。

Dávid Karnok 是 RxJava 的第 2 版之父，他显著改进了整个库的设计，并引入了符合响应式流规范的其他类型。虽然由于向后兼容性，Observable 继续维持不变，但同时，RxJava 2 提供了名为 Flowable 的新响应式类型。

Flowable 响应式类型虽然提供与 Observable 相同的 API，但会从头开始扩展 org.reactive.streams.Publisher。

如下一个示例所示，Flowable 中嵌入了流式 API，可以转换为任何常见的 RxJava 类型并反向转化为响应式流兼容类型：

```
1 Flowable.just(1, 2, 3)
2   .map(String::valueOf)
3   .toObservable()
4   .toFlowable(BackpressureStrategy.ERROR)
5   .subscribe();
```

从 Flowable 到 Observable 的转换只是一个操作符的简单应用。但是，要将 Observable 转换回 Flowable，就必须提供一些可用的背压策略。在 RxJava 2 中，Observable 被设计为仅推送流。因此，保证转换后的 Observable 符合响应式流规范至关重要。

## 5.2 Vert.x 的调整

随着 RxJava 的转变，其他响应式库和框架提供商也开始采用响应式流规范。为了遵循规范，Vert.x 包含一个额外的模块，该模块为响应式流 API 提供支持。以下示例演示了该模块：

```
1 // ...
2 .requestHandler(request -> {
3     ReactiveReadStream<Buffer> rrs = ReactiveReadStream.readStream();
4     HttpServerResponse response = request.response();
5     Flowable<Buffer> logs =
6         Flowable.fromPublisher(logsservice.stream()).map(Buffer::buffer)
7             .doOnTerminate(response::end);
8     logs.subscribe(rrs);
9     response.setStatusCode(200);
10    response.setChunked(true);
11    response.putHeader("Content-Type", "text/plain");
12    response.putHeader("Connection", "keep-alive");
13    Pump.pump(rrs, response).start();
14});
```

关键点解释如下。

1. 这是请求处理器声明。这是一个通用请求处理器，能处理发送到服务器的任何请求。
2. 这是 Subscriber 和 HTTP 响应声明。这里的 ReactiveReadStream 同时实现了 org.reactivestreams.Subscriber 和 ReadStream，而二者能将任何 Publisher 转换为与 Vert.x API 兼容的数据源。
3. 这是处理流程声明。在该示例中，我们引用新的基于响应式流的 LogService 接口，并编写对流中元素的函数式转换，我们使用 RxJava 2.x 中的 Flowable API。
4. 这是订阅阶段。一旦声明了处理流程，我们就可以将 ReactiveReadStream 订阅到 Flowable。
5. 这是一个响应准备阶段。

6. 这是发送给客户端的最终响应。这里，Pump 类在一个复杂的背压控制机制中起着重要作用，以防止底层的 WriteStream 缓冲区过满。

我们可以看到，Vert.x 没有提供用于编写元素处理流的流式 API。但是，它提供了一个 API，能将任何 Publisher 转换为 Vert.x API，从而维持响应式流的复杂背压管理。

## 5.3 Ratpack的改进

除了 Vert.x，另一个名为 Ratpack 的著名 Web 框架也提供对响应式流的支持。与 Vert.x 相比，Ratpack 提供了对响应式流的直接支持。

例如，在使用 Ratpack 的场景下，发送日志流的代码如下所示：

```
1 RatpackServer.start(server -> server.handlers(chain -> chain.all(ctx -> {
2     Publisher<String> logs = logsService.stream();
3     ServerSentEvents events = serverSentEvents(
4         logs, event ->
5             event.id(objects::toString).event("log").data(function.identity());
6         );
7         ctx.render(events);
8     }));
9 })
```

关键点解释如下。

1. 这是服务器启动的操作和请求处理程序声明。
2. 这是日志流声明。
3. 这是 ServerSentEvents 的准备工作。这里，上述类在映射阶段起作用，该阶段将 Publisher 中的元素转换为服务器端发送事件的表示方式。如我们所见，ServerSentEvents 强制要求映射函数声明，而该声明描述了如何将元素映射到特定的 Event 字段。
4. 这是流到 I/O 的呈现。

如示例所示，Ratpack 在内核中提供对响应式流的支持。现在，我们可以重用相同的 LogService#stream 方法，而无须提供额外的类型转换或要求其他模块添加对特定响应式库的支持。

此外，与只提供对响应式流规范的简单支持的 Vert.x 相比，Ratpack 还提供了对规范接口的自身实现。此功能包含在 ratpack.stream.Streams 类中，类似于 RxJava API：

```
1 Publisher<String> logs = logsService.stream();
2 TransformablePublisher publisher = Streams.transformable(logs)
3     .filter(this::filterUserSensitiveLogs)
4     .map(this::escape);
```

在这里，Ratpack 提供了一个静态工厂，它可以将任何 Publisher 转换为 TransformablePublisher，使我们可以使用熟悉的操作符和转换阶段灵活地处理事件流。

# 6 Spring响应式编程

## 6.1 Spring的早期响应式解决方案

响应式编程是构建响应式系统的主要候选方案。

Spring 4.x 引入了 ListenableFuture 类，它扩展了 Java Future，并且可以基于 HTTP 请求实现异步执行操作。但是只有少数 Spring 4.x 组件支持新的 Java 8 CompletableFuture，后者引入了一些用于组合异步执行的简洁方法。

Spring 框架还提供了其他一些基础架构，它们对构建我们的响应式应用程序非常有用。

### 6.1.1 观察者模式

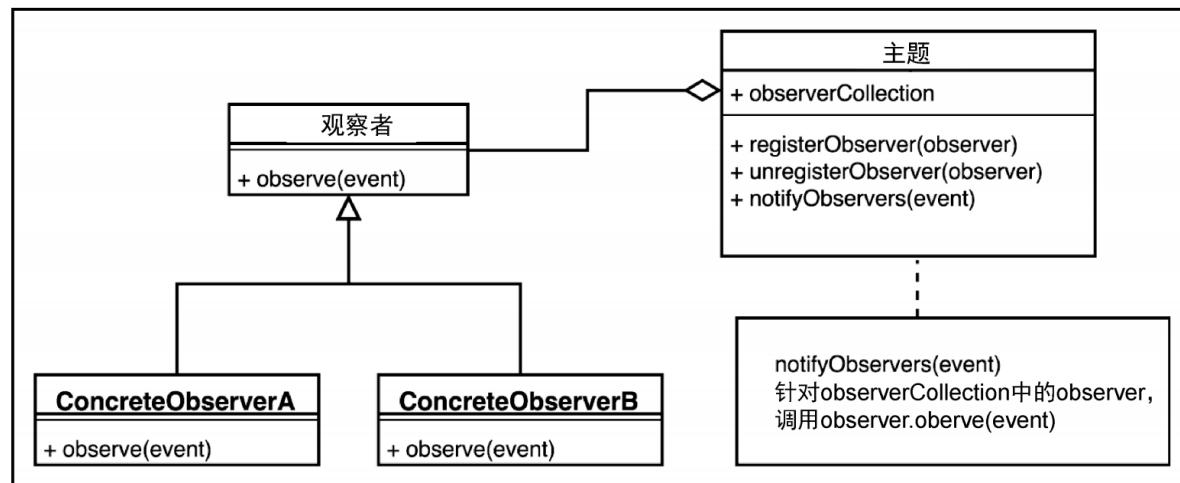
好像观察者模式似乎与响应式编程无关。但是，经过一些小修改，**它定义了响应式编程的基础**。

观察者模式拥有一个主题（subject），其中包含该模式的依赖者列表，这些依赖者被称为观察者（Observer）。

主题通常通过调用自身的一个方法将状态变化通知观察者。

在基于事件处理的系统中此模式至关重要。观察者模式是 MVC（模型-视图-控制器）模式的重要组成部分。

类图：



Subject接口：

```
1 package com.lagou.webflux.demo;
2
3 public interface Subject {
4
5     /**
6      * 注册观察者
7      * @param observer
```

```
8     */
9     void registerObserver(Observer observer);
10
11    /**
12     * 解绑观察者
13     * @param observer
14     */
15    void unregisterObserver(Observer observer);
16
17    /**
18     * 通知事件变更
19     * @param event
20     */
21    void notifyObservers(String event);
22
23 }
```

Observer接口:

```
1 package com.lagou.webflux.demo;
2
3 public interface Observer {
4
5     void observe(String event);
6
7 }
```

Subject的实现类:

```
1 package com.lagou.webflux.demo;
2
3 import java.util.Set;
4 import java.util.concurrent.CopyOnwriteArraySet;
5
6 public class ConcreteSubject implements Subject {
7
8     /**
9      * 保证Set是线程安全的
10     */
11     private Set<Observer> observers = new CopyOnwriteArraySet<>();
12
13     @Override
14     public void registerObserver(Observer observer) {
15         observers.add(observer);
16     }
17
18     @Override
19     public void unregisterObserver(Observer observer) {
20         observers.remove(observer);
21     }
22
23     @Override
24     public void notifyObservers(String event) {
```

```
25     observers.forEach(observer -> observer.observe(event));
26 }
27 }
```

Observer的实现类：

```
1 package com.lagou.webflux.demo;
2
3 public class ConcreteObserverA implements Observer {
4     @Override
5     public void observe(String event) {
6         System.out.println(getClass().getCanonicalName() + " --- " + event);
7     }
8 }
9
10 package com.lagou.webflux.demo;
11
12 public class ConcreteObserverB implements Observer {
13
14     @Override
15     public void observe(String event) {
16         System.out.println(getClass().getCanonicalName() + " --- " + event);
17     }
18 }
```

Main：

```
1 package com.lagou.webflux.demo;
2
3 public class Main {
4     public static void main(String[] args) {
5         Subject subject = new ConcreteSubject();
6         Observer observer1 = new ConcreteObserverA();
7         Observer observer2 = new ConcreteObserverB();
8
9         subject.registerObserver(observer1);
10        subject.registerObserver(observer2);
11
12        subject.notifyObservers("hello lagou");
13        System.out.println("=====");
14
15        subject.unregisterObserver(observer1);
16
17        subject.notifyObservers("great lagou");
18
19    }
20 }
```

为了在多线程场景中确保线程安全，使用 CopyOnWriteArrayList，这是一个线程安全的Set 实现，它在每次 update 操作发生时都会创建元素的新副本。

更新 CopyOnWriteArrayList 中的内容相对代价较高，当容器包含大量元素时尤为如此。但是，订阅者列表通常不会经常更改，因此对于线程安全的 Subject 实现来说，这是一个相当合理的选择。

### 6.1.2 观察者模式使用

在不需要取消订阅的情况下，我们可以活用 Java 8 特性，用 lambda 替换 Observer 实现类。下面编写相应的测试用例：

```
1  @Test
2  public void subjectLeveragesLambdas() {
3      Subject<String> subject = new ConcreteSubject();
4      subject.registerObserver(e -> System.out.println("A: " + e));
5      subject.registerObserver(e -> System.out.println("B: " + e));
6      subject.notifyObservers("This message will receive A & B");
7      // ...
8  }
```

在有很多观察者处理明显延迟的事件（由下游处理引入）时，我们可以使用其他线程或线程池（thread pool）并行传播消息。

基于这种处理方式可以得出 notifyObservers 方法的下一个实现：

```
1  private final ExecutorService executorService =
2  Executors.newCachedThreadPool();
3
4  public void notifyObservers(String event) {
5      observers.forEach(
6          observer -> executorService.submit(
7              () -> observer.observe(event)
8      )
9  );
10 }
```

这些方案通常不是最高效的，并且很可能隐藏着 bug。例如，我们可能忘记限制线程池大小，并最终导致 OutOfMemoryError。

因为每个线程在 Java 中消耗大约 1 MB，典型的 JVM 应用程序有可能创建几千个线程来耗尽所有可用内存。

为了防止资源滥用，我们可以限制线程池大小并将应用程序的活跃度（liveness）属性设置为 violate。当所有可用线程试图将某些事件推送到同一个缓慢的 Observer 时，就会出现这种情况。在这里，我们只是初步暴露了可能发生的潜在问题。

因此，当需要支持多线程的 Observer 模式时，最好使用经过实战验证的库。

java.util 包中的观察者模式从 JDK 1.0 发布的。如果查看源代码，会发现一个非常简单的实现，它与前面的实现非常相似。因为这些类是在 Java 泛型（Java generics）之前引入的，所以它们操作 Object 类型的事件，是类型不安全的。

此外，这种实现效率不高，尤其是在多线程环境中。这些类在 Java 9 中已被弃用。

在开发应用程序时，可能使用观察者模式的手工自定义实现。这能够对事件源和观察者进行解耦。但是，需要考虑许多对现代多线程应用程序至关重要的方面，包括错误处理、异步执行、线程安全、高性能需求等。

JDK 附带的事件实现只能满足演示用途。

### 6.1.3 基于@EventListener注解的发布和订阅模式

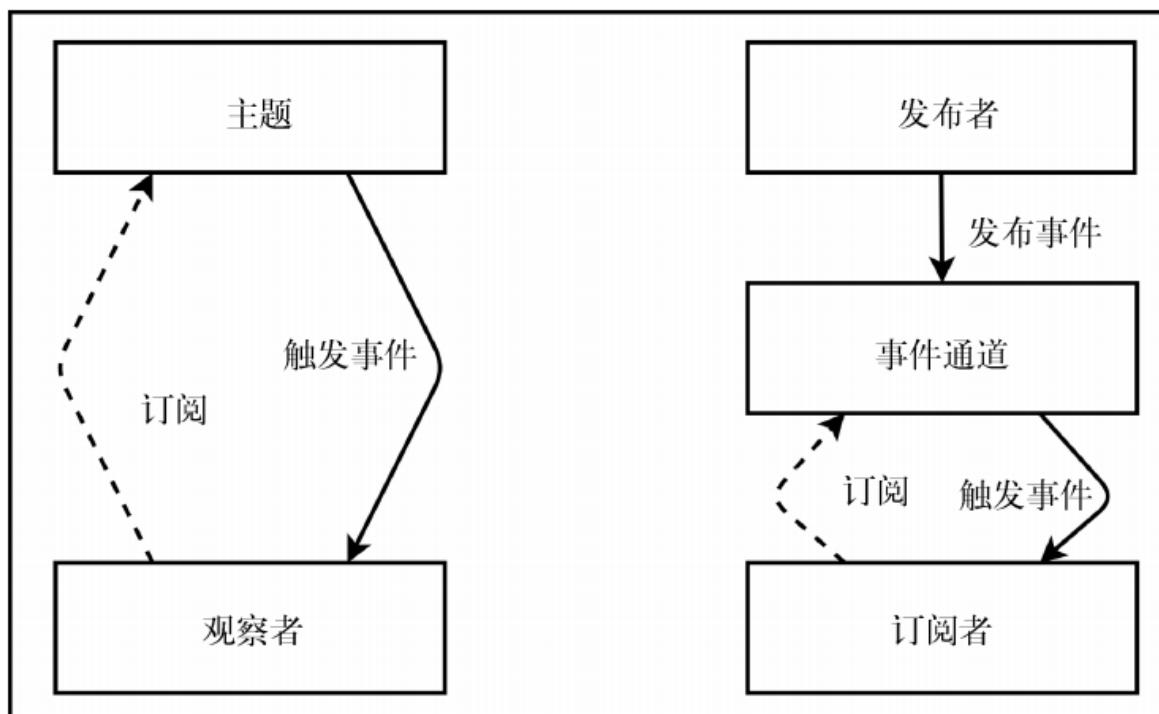
在很长一段时间内，Spring 框架有自己的观察者模式实现，这被广泛用于**跟踪应用程序的生命周期事件**。

从 Spring 4.2 开始，不仅用于处理应用程序事件，还用于处理业务逻辑事件。

Spring 的@EventListener 注解实现事件分发，ApplicationEventPublisher 类实现事件发布。

@EventListener 和 ApplicationEventPublisher 实现了发布-订阅模式（Publish-Subscribe pattern），它可以被视为观察者模式的变体。

在发布-订阅模式中，发布者和订阅者不需要彼此了解，如下图所示：



发布-订阅模式在发布者和订阅者之间提供了额外的**间接层**。

订阅者知道广播通知的事件通道，但通常不关心发布者的身份。此外，每个事件通道中可能同时存在几个发布者。

事件通道（event channel，也被称为消息代理或事件总线）可以额外过滤传入的消息并在订阅者之间分发它们。

过滤和路由的执行可以基于消息内容或消息主题，也可以同时基于这两者。因此，基于主题的系统中的订阅者将接收发布到自身感兴趣主题的所有消息。

@EventListener注解支持基于**主题**和基于**内容**的**路由**。

消息类型作为主题的角色；condition属性基于内容进行事件的路由，事件路由处理基于Spring表达式语言（SpEL）。

#### 6.1.4 使用@EventListener注解构建应用程序

实现一个简单的Web服务，用于显示房间当前的温度。

设置一个温度传感器，它不时地将当前的摄氏温度通过事件发送出来。使用随机数生成器模拟温度传感器。

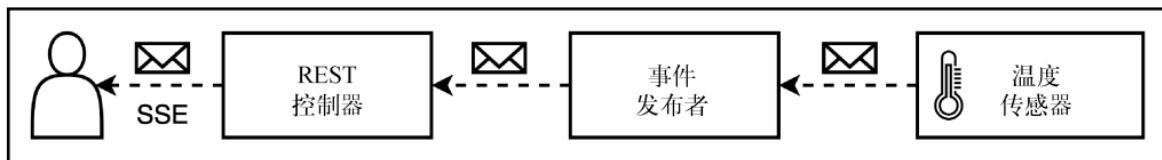
为了使应用程序遵循响应式设计，不使用旧的拉模型获取数据。

使用WebSocket和服务器发送事件（Server-Sent Events，SSE）。

SSE能使客户端从服务器接收自动更新，通常用于向浏览器发送消息更新或连续数据流。

使用EventSource的JavaScript API，请求特定URL并接收事件流。在通信发生问题时，EventSource默认自动重连。

#### 实现业务逻辑



pom.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     https://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>2.4.0</version>
11    <relativePath/> <!-- Lookup parent from repository -->
12  </parent>
13  <groupId>com.lagou.webflux.demo</groupId>
14  <artifactId>demo-listener</artifactId>
15  <version>0.0.1-SNAPSHOT</version>
16  <name>demo-listener</name>
17  <description>Demo project for Spring Boot</description>
18
19  <properties>
20    <java.version>11</java.version>
21  </properties>
22
23  <dependencies>
24    <dependency>
25      <groupId>org.springframework.boot</groupId>
26      <artifactId>spring-boot-starter-web</artifactId>
27    </dependency>
28
29    <dependency>
30      <groupId>org.springframework.boot</groupId>
31      <artifactId>spring-boot-starter-test</artifactId>
32      <scope>test</scope>
33    </dependency>
34
35    <dependency>
36      <groupId>org.json</groupId>
37      <artifactId>json</artifactId>
38      <version>20200518</version>
39    </dependency>
40  </dependencies>
41
42  <build>
43    <plugins>
44      <plugin>
45        <groupId>org.springframework.boot</groupId>
46        <artifactId>spring-boot-maven-plugin</artifactId>
47      </plugin>
48    </plugins>
49  </build>
50
51 </project>

```

实体类Temperature仅包含 double 值。它还被用作事件对象，代码如下所示：

```

1 package com.lagou.webflux.demo.entity;

```

```
2
3 public class Temperature {
4
5     private final double value;
6
7     public Temperature(double temperature) {
8         this.value = temperature;
9     }
10
11    public double getValue() {
12        return value;
13    }
14
15 }
```

TemperatureSensor类模拟传感器，并使用@Component注解，代码如下所示：

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Temperature;
4 import org.springframework.context.ApplicationEventPublisher;
5 import org.springframework.stereotype.Component;
6
7 import javax.annotation.PostConstruct;
8 import java.util.Random;
9 import java.util.concurrent.Executors;
10 import java.util.concurrent.ScheduledExecutorService;
11 import java.util.concurrent.TimeUnit;
12
13 @Component
14 public class TemperatureSensor {
15
16     private final ApplicationEventPublisher publisher;
17     private final Random random = new Random();
18     private final ScheduledExecutorService service =
19         Executors.newSingleThreadScheduledExecutor();
20
21     public TemperatureSensor(ApplicationEventPublisher publisher) {
22         this.publisher = publisher;
23     }
24
25     @PostConstruct
26     public void startProcessing() {
27         this.service.schedule(this::probe, 1, TimeUnit.SECONDS);
28     }
29
30     private void probe() {
31         double temperature = 16 + random.nextGaussian() * 10;
32         System.err.println("发送事件。。。");
33         // 通过ApplicationEventPublisher发布Temperature事件
34         publisher.publishEvent(new Temperature(temperature));
35         service.schedule(this::probe, random.nextInt(5000),
36             TimeUnit.MILLISECONDS);
37     }
38 }
```

模拟温度传感器仅依赖于 Spring 框架提供的 ApplicationEventPublisher类。该类可以将事件发布到系统。

## 基于 Spring Web MVC 的异步 HTTP

Servlet 3.0中引入的异步支持扩展了在**非容器线程**中处理 HTTP请求的能力。

基于Servlet 3.0， Spring Web MVC 可以返回 `Callable<T>` 或 `DeferredResult<T>`。

`Callable<T>` 可以在非容器线程内运行，但仍然是**阻塞调用**。

`DeferredResult<T>` 能通过调用 `setResult(T result)` 方法在**非容器线程上生成异步响应**，可以在事件循环中使用。

从 4.2 版开始，Spring Web MVC 可以返回 `ResponseBodyEmitter`，其行为类似于 `DeferredResult`，但可以用于发送多个对象。

`SseEmitter` 继承了 `ResponseBodyEmitter`，可以根据 SSE 的协议需求为一个请求发送多个响应。

## 暴露 SSE 端点

```
1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.Temperature;
4 import org.json.JSONObject;
5 import org.springframework.context.event.EventListener;
6 import org.springframework.scheduling.annotation.Async;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestMethod;
9 import org.springframework.web.bind.annotation.RestController;
10 import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;
11
12 import javax.servlet.http.HttpServletRequest;
13 import java.util.ArrayList;
14 import java.util.List;
15 import java.util.Set;
16 import java.util.concurrent.CopyOnwriteArraySet;
17
18 @RestController
19 public class TemperatureController {
20
21     private final Set<SseEmitter> clients = new CopyOnwriteArraySet<>();
22
23     @RequestMapping(value = "/temperature-stream", method =
24     RequestMethod.GET)
25     public SseEmitter events(HttpServletRequest request) {
```

```

25     // ResponseBodyEmitter的子类，用于发送SSE（Server-Send Event）：服务器发送
26     的事件
27     // SseEmitter emitter = new SseEmitter();
28     // 设置超时时间
29     SseEmitter emitter = new SseEmitter(10000L);
30
31     // 将当前发射器放到集合中
32     clients.add(emitter);
33     // 给当前发射器设置事件处理函数
34     /*
35      当异步请求超时的时候调用的代码。
36      该方法在异步请求超时的时候由容器线程调用。
37      */
38     emitter.onTimeout(() -> clients.remove(emitter));
39     /*
40      当异步请求结束的时候调用的代码。
41      当超时或网络错误而终止异步请求处理的时候，在容器线程调用该方法。
42      该方法一般用于检查一个ResponseBodyEmitter实例已经无用了。
43      */
44     emitter.onCompletion(() -> clients.remove(emitter));
45     return emitter;
46 }
47
48 @Async // 异步事件处理
49 @EventListener // 事件监听器，该监听器只接收Temperature事件
50 public void handleMessage(Temperature temperature) {
51     System.out.println("监听到web的调度事件了 -- " + temperature);
52     List<SseEmitter> deadEmitters = new ArrayList<>();
53     // 遍历发射器集合
54     clients.forEach(emitter -> {
55         try {
56             // 发射器发送温度对象，json类型
57             final JSONObject jsonObject = new JSONObject(temperature);
58             final String s1 = jsonObject.toString();
59             emitter.send(s1);
60         } catch (Exception ignore) {
61             // 如果抛异常，则将该发射器放到deadEmitters集合中
62             deadEmitters.add(emitter);
63         }
64     });
65     // 从clients中移除所有失效的发射器。
66     clients.removeAll(deadEmitters);
67 }

```

Spring Web MVC 提供 SseEmitter 的唯一目的是发送 SSE 事件。当控制器方法返回 SseEmitter 实例时，实际的请求处理过程将一直持续下去，直到 SseEmitter.complete() 方法被调用、发生错误或超时。

在客户端请求 /temperature-stream 时，创建并返回新的 SseEmitter 实例，同时将该实例注册到先前的活动 clients 列表中。此外，SseEmitter 构造函数可以使用 timeout 参数。

对于 clients 集合，我们可以使用 java.util.concurrent 包中的 CopyOnWriteArrayList 类。这样的实现使我们能在修改列表的同时执行迭代操作。当一个 Web 客户端打开新的 SSE 会话时，我们将新的发射器添加到 clients 集合中。SseEmitter 在完成处理或已达到超时时，会将自己从 clients 列表中删除。

handleMessage()方法使用@EventListener注解，以便从Spring接收事件。Spring框架仅在接收到Temperature事件时才会调用handleMessage()方法，因为该方法的参数是temperature对象。

@Async注解将方法标记为异步执行的候选方法，在手动配置的线程池中调用。

handleMessage()方法接收一个新的温度事件，并把每个事件并行地以JSON格式异步发送给所有客户端。此外，当发送到各个发射器时，跟踪所有发生故障的发射器并将其从活动clients列表中删除。这种方法使我们可以发现不运作的客户端。不幸的是，SseEmitter没有为处理错误提供任何回调，只能通过处理send()方法抛出的错误来完成错误处理。

## 配置异步支持

```
1 package com.lagou.webflux.demo.config;
2
3 import org.springframework.aop.interceptor.AsyncUncaughtExceptionHandler;
4 import
5     org.springframework.aop.interceptor.SimpleAsyncUncaughtExceptionHandler;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.scheduling.annotation.AsyncConfigurer;
8 import org.springframework.scheduling.annotation.EnableAsync;
9 import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
10
11 import java.util.concurrent.Executor;
12
13 @Configuration
14 @EnableAsync
15 public class MyAsyncConfig implements AsyncConfigurer {
16
17     // 为异步调用设置Executor
18     @Override
19     public Executor getAsyncExecutor() {
20         // 使用包含两个核心线程的 ThreadPoolTaskExecutor，可以将核心线程增加到一百
21         // 个。
22         ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
23         executor.setCorePoolSize(2);
24         executor.setMaxPoolSize(100);
25         // 如果没有正确配置队列容量，线程池就无法增长。
26         // 这是因为程序将转而使用 SynchronousQueue，而这限制了并发。
27         executor.setQueueCapacity(5);
28         executor.initialize();
29         return executor;
30     }
31
32     // 为异步执行引发的异常配置异常处理程序。
33     @Override
34     public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler()
35     {
36         // 此处仅记录异常
37         return new SimpleAsyncUncaughtExceptionHandler();
38     }
39 }
```

入口程序:

```
1 package com.lagou.webflux.demo;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Demo10Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(Demo10Application.class, args);
11     }
12
13 }
```

## 构建具有 SSE 支持的 UI

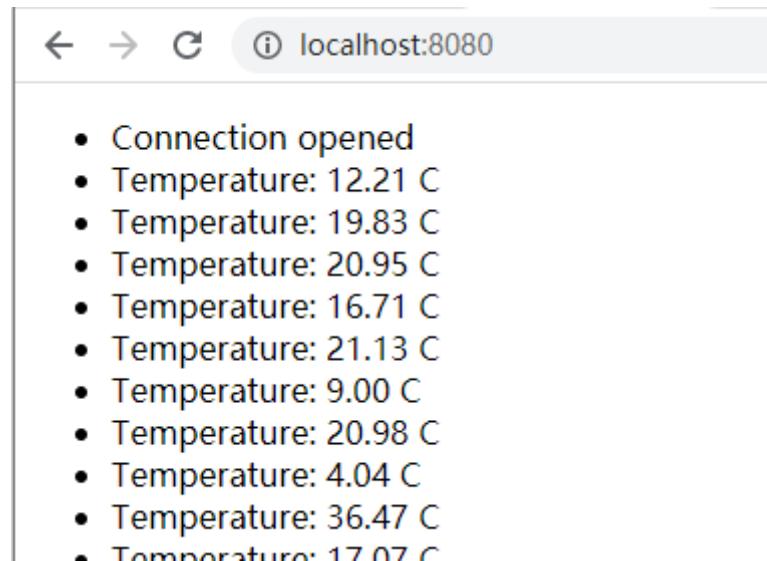
resources/static/index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Title</title>
6 </head>
7 <body>
8 <ul id="events"></ul>
9
10 <script type="application/javascript">
11     function add(message) {
12         const el = document.createElement("li");
13         el.innerHTML = message;
14         document.getElementById("events").appendChild(el);
15     }
16     var eventSource = new EventSource("/temperature-stream");
17     eventSource.onmessage = e => {
18         const t = JSON.parse(e.data);
19         const fixed = Number(t.value).toFixed(2);
20         add('Temperature: ' + fixed + ' C');
21     }
22     eventSource.onopen = e => add('Connection opened');
23     eventSource.onerror = e => add('Connection closed');
24 </script>
25
26 </body>
27 </html>
```

## 验证应用程序功能

在浏览器中打开以下地址访问网页：

<http://localhost:8080>



当前的解决方案不是 JavaScript 独有的，也可以使用curl访问：<http://localhost:8080/temperature-stream>：

```
root@DESKTOP-M0RJ5JJ:~# curl localhost:8080/temperature-stream
data:{"value":21.03459501301455}

data:{"value":21.739412677585808}

data:{"value":12.634994219563355}

data:{"value":36.00782492560518}

data:{"value":17.111058656451807}

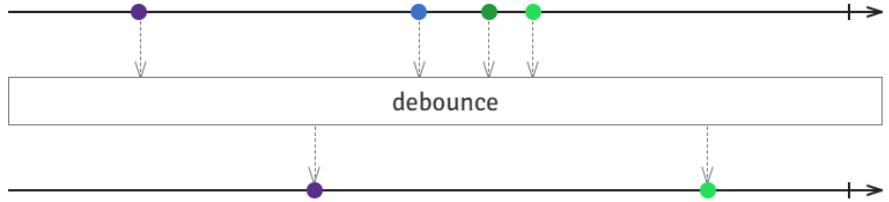
data:{"value":5.767181029865796}

data:{"value":4.4065092523360505}

data:{"value":14.3788245000077161}
```

## 6.2 使用RxJava作为响应式框架

<http://reactivex.io/>



The Observer pattern done right

ReactiveX is a combination of the best ideas from the **Observer** pattern, the **Iterator** pattern, and functional programming

ReactiveX 通常被定义为观察者模式、迭代器模式和函数式编程的组合。

Java平台上有一个用于响应式编程的标准库，即RxJava，是 Reactive Extensions（响应式扩展，也称为 ReactiveX）的 Java 实现。目前，它不是唯一的响应式库，还有Akka Streams和Project Reactor。

此外，随着 2.x 版的发布，RxJava 本身发生了很大的变化。

目前最新版本是 RxJava3。

RxJava 是迄今为止应用最广泛的响应式库。

这些 API 从该库的早期版本以来就没有改变。

虽然 RxJava 1.x 的生命周期结束于 2018 年 3 月，但它仍然被用于很多库和应用程序，这主要是因为该版本被长期而广泛地采用。

### 6.2.1 响应式流

观察者模式为我们提供了一张清晰分离的生产者（Producer）事件和消费者（Consumer）事件视图。代码如下所示：

```
1 package com.lagou.webflux.demo;
2
3 public interface Observer {
4     void observe(String event);
5 }
6
7 package com.lagou.webflux.demo;
8
9 public interface Subject {
10     /**
11      * 注册观察者
12      * @param observer
13 }
```

```
13     */
14     void registerObserver(Observer observer);
15
16     /**
17      * 解绑观察者
18      * @param observer
19      */
20     void unregisterObserver(Observer observer);
21
22     /**
23      * 通知事件变更
24      * @param event
25      */
26     void notifyObservers(String event);
27 }
```

如果不希望生产者在消费者出现之前生成事件，则可以使用迭代器（Iterator）模式。如下代码：

```
1 package java.util;
2
3 public interface Iterator<E> {
4     boolean hasNext();
5     E next();
6 }
```

将迭代器模式和观察者模式相结合，如下代码：

```
1 public interface RxObserver<T> {
2     void onNext(T next);
3     void onComplete();
4 }
```

虽然 RxObserver 非常类似于 Iterator，但它：

- 不是调用 Iterator 的 next()方法，而是通过 onNext()回调将一个新值通知给 RxObserver。
- 不是检查 hasNext()方法的结果是否为 true，而是通过调用 onComplete()方法通知 RxObserver 流的结束。

错误如何处理呢？

因为 Iterator 可能在处理 next()方法时抛出 Exception，所以应该有一个从生产者到 RxObserver 的错误传播机制。

为此添加一个特殊的回调，即 onError()。因此，最终解决方案如下所示：

```
1 public interface RxObserver<T> {  
2     void onNext(T next);  
3     void onComplete();  
4     void onError(Exception e);  
5 }
```

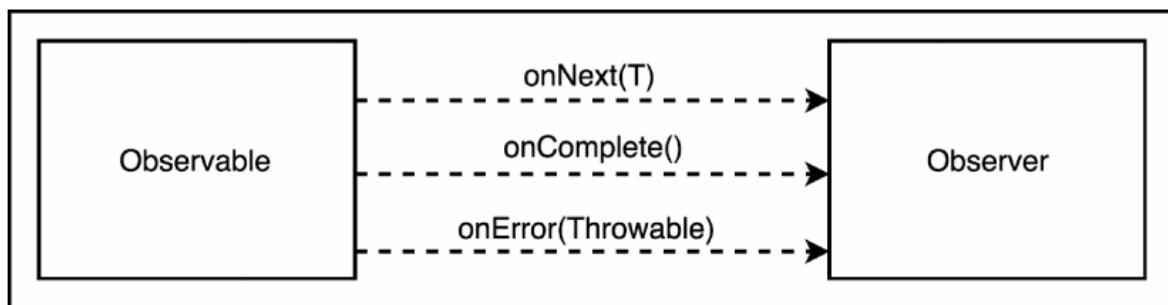
如此，则刚刚设计了一个 Observer 接口，这是 RxJava 的基本概念。

此接口定义了数据如何在响应式流的每个部分之间进行流动。作为库的最小组成部分，Observer 接口随处可见。RxObserver 类似于前面介绍的观察者模式中的 Observer。

**Observable** 响应式类是观察者模式中 Subject 的对应类。Observable 扮演事件源的角色，它会发出元素。它有数百种流转换方法以及几十种初始化响应式流的工厂方法。

**Subscriber** 抽象类不仅实现 Observer 接口并消费元素，还被用作 Subscriber 的实际实现的基础。

**Observable** 和 **Subscriber** 之间的运行时关系由 **Subscription** 控制，Subscription 可以检查订阅状态并在必要时取消订阅。如下图所示：



RxJava 定义了有关发送元素的规则，使 Observable 能发送任意数量的元素（包括零个）。然后它通过声明成功或引发错误来指示执行结束。

Observable 会为订阅它的每个 Subscriber 多次调用 onNext()，然后再调用 onComplete() 或 onError()（但不能同时调用两者）。所以在 onComplete() 或 onError() 之后调用 onNext() 是不可行的。

### 6.2.2 生产和消费数据

创建maven项目并在pom.xml添加依赖：

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <project xmlns="http://maven.apache.org/POM/4.0.0"  
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
5   http://maven.apache.org/xsd/maven-4.0.0.xsd">  
6     <modelVersion>4.0.0</modelVersion>  
7     <groupId>com.lagou.webflux.demo</groupId>  
8     <artifactId>demo-11</artifactId>  
9     <version>1.0-SNAPSHOT</version>
```

```

10
11 <dependencies>
12   <dependency>
13     <groupId>io.reactivex</groupId>
14     <artifactId>rxjava</artifactId>
15     <version>1.3.8</version>
16   </dependency>
17
18   <!--> <dependency>-->
19   <!-->   <groupId>io.reactivex.rxjava2</groupId>-->
20   <!-->   <artifactId>rxjava</artifactId>-->
21   <!-->   <version>2.2.20</version>-->
22   <!--> </dependency>-->
23
24   <!--> <dependency>-->
25   <!-->   <groupId>io.reactivex.rxjava3</groupId>-->
26   <!-->   <artifactId>rxjava</artifactId>-->
27   <!-->   <version>3.0.7</version>-->
28   <!--> </dependency>-->
29
30 </dependencies>
31
32 </project>

```

```

1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4 import rx.Observable.OnSubscribe;
5 import rx.Subscriber;
6
7 public class Main {
8     public static void main(String[] args) {
9         Observable<String> observable = Observable.create(
10             new Observable.OnSubscribe<String>() {
11                 @Override
12                 public void call(Subscriber<? super String> subscriber)
13                 {
14                     for (int i = 0; i < 10; i++) {
15                         subscriber.onNext("hello lagou " + i);
16                     }
17                     subscriber.onCompleted();
18                 }
19             });
20         observable.subscribe(new Subscriber<String>() {
21             @Override
22             public void onCompleted() {
23                 System.out.println("on completed");
24             }
25
26             @Override
27             public void onError(Throwable throwable) {
28                 System.out.println("on error :" + throwable.getMessage());
29             }
30         });
31     }
32 }

```

```
29         }
30
31     @Override
32     public void onNext(String s) {
33         System.out.println("on next: " + s);
34     }
35 );
36 }
```

创建 Observable 并使其带有一个回调，该回调将在订阅者出现时立即被触发。此时，Observer 将产生一个字符串值，并将流的结束信号发送给订阅者。

还可以使用 Java 8 lambda 改进此代码：

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4 import rx.Observable.OnSubscribe;
5 import rx.Subscriber;
6
7 public class Main {
8     public static void main(String[] args) {
9         Observable<String> observable = Observable.create(
10             subscriber -> {
11                 for (int i = 0; i < 10; i++) {
12                     subscriber.onNext("hello Lagou " + i);
13                 }
14                 subscriber.onCompleted();
15             });
16
17         observable.subscribe(new Subscriber<String>() {
18             @Override
19             public void onCompleted() {
20                 System.out.println("on completed");
21             }
22
23             @Override
24             public void onError(Throwable throwable) {
25                 System.out.println("on error :" + throwable.getMessage());
26             }
27
28             @Override
29             public void onNext(String s) {
30                 System.out.println("on next: " + s);
31             }
32         });
33     }
34 }
```

与 Java Stream API 相比，Observable 是可重用的，而且每个订阅者都将在订阅之后立即收到“Hello Lagou”事件。

注意，从 RxJava 1.2.7 开始，Observable 的创建已因不安全而被弃用。

这是因为它可能生成太多元素，导致订阅者超载。换句话说，这种方法不支持背压。

还可以使用 lambda 重写此示例，代码如下所示：

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 public class Main {
6     public static void main(String[] args) {
7         Observable.create(
8             subscriber -> {
9                 for (int i = 0; i < 10; i++) {
10                     subscriber.onNext("hello lagou " + i);
11                 }
12                 subscriber.onCompleted();
13             }
14         ).subscribe(
15             System.out::println,
16             System.err::println,
17             () -> System.out.println("结束")
18         );
19     }
20 }
```

可以使用 just 来引用元素、使用旧式数组，或者使用 from 通过 Iterable 集合来创建 Observable 实例，代码如下所示：

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 public class Main {
6     public static void main(String[] args) {
7         Observable<String> just = Observable.just("1", "2", "3", "4", "5");
8         just.subscribe(
9             item -> System.out.println("下一个元素是：" + item),
10             ex -> System.err.println("异常信息：" + ex.getMessage()),
11             () -> System.out.println("结束")
12         );
13     }
14 }
```

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 public class Main {
6     public static void main(String[] args) {
```

```
7     Observable<Integer> from = observable.from(new Integer[]{1, 2, 3, 4,
8         5});
9     from.subscribe(
10         item -> System.out.println("下一个元素是: " + item),
11         ex -> System.err.println("异常信息是: " + ex.getMessage()),
12         () -> System.out.println("结束")
13     );
14 }
```

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 import java.util.Collections;
6
7 public class Main {
8     public static void main(String[] args) {
9         Observable<Object> from = observable.from(Collections.emptyList());
10        from.subscribe(
11            item -> System.out.println("下一个元素是: " + item),
12            ex -> System.err.println("异常信息: " + ex.getMessage()),
13            () -> System.out.println("结束")
14        );
15    }
16 }
```

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 public class Main {
6     public static void main(String[] args) {
7         Observable<String> fromCallable = Observable.fromCallable(() ->
8             "hello lagou");
9         fromCallable.subscribe(
10             item -> System.out.println("下一个元素是: " + item),
11             ex -> System.err.println("错误信息是: " + ex.getMessage()),
12             () -> System.out.println("结束")
13         );
14     }
15 }
```

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 import java.util.concurrent.Executors;
6 import java.util.concurrent.Future;
7
```

```
8 public class Main {  
9     public static void main(String[] args) {  
10         Future<String> future = Executors.newCachedThreadPool().submit(() ->  
11             "hello lagou");  
12         Observable<String> from = Observable.from(future);  
13         from.subscribe(  
14             item -> System.out.println("下一个元素是: " + item),  
15             ex -> System.err.println("异常信息是: " + ex.getMessage()),  
16             () -> System.out.println("结束")  
17         );  
18     }  
}
```

除了简单的创建功能，还可以通过组合其他 Observable 实例来创建 Observable 流，这可以轻松实现非常复杂的工作流。

例如，每个传入流的 concat() 操作符会通过将每个数据项重新发送到下游观察者的方式来消费所有数据项。然后，传入流将被处理，直到发生终止操作 (onComplete(), onError())，并且其处理顺序会与 concat() 方法中参数的顺序保持一致。

以下代码展示了 concat() 用法的示例：

```
1 package com.lagou.webflux.demo;  
2  
3 import rx.Observable;  
4  
5 public class Main {  
6     public static void main(String[] args) {  
7         Observable.concat(  
8             Observable.just("hello"),  
9             Observable.from(new String[] {"lagou"}),  
10            Observable.just("!")  
11        ).forEach(  
12            item -> System.out.println("下一个元素是: " + item),  
13            ex -> System.err.println("异常信息是: " + ex.getMessage()),  
14            () -> System.out.println("结束")  
15        );  
16    }  
17 }
```

这里，作为几个 Observable 实例（使用不同来源）直接组合的一部分，我们还使用 Observable.forEach() 方法以类似于 Java 8 Stream API 的方式遍历结果。这样的程序生成以下输出：

下一个元素： hello

下一个元素： lagou

下一个元素： !

job done

请注意，虽然不为异常定义处理程序很方便，但在发生错误的情况下，默认的Subscriber 实现仍会抛出 rx.exceptions.OnErrorNotImplementedException。

### 6.2.3 生成异步序列

RxJava 不仅可以生成一个未来的事件，还可以基于时间间隔等生成一个异步事件序列，示例代码如下所示：

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 import java.util.concurrent.TimeUnit;
6
7 public class Main {
8     public static void main(String[] args) throws InterruptedException {
9         Observable.interval(1, TimeUnit.SECONDS)
10             .subscribe(System.out::println);
11         Thread.sleep(5000);
12     }
13 }
```

如果删除 Thread.sleep(...), 那么应用程序将在不输出任何内容的情况下退出。

因为生成事件并进行消费的过程发生在**一个单独的守护线程**中。因此，为了防止主线程完成执行，我们可以调用 sleep()方法或执行一些其他有用的任务。

Subscription可以控制观察者-订阅者协作，该接口声明如下：

```
1 public interface Subscription {
2     // 订阅取消
3     void unsubscribe();
4     // 检查 Subscriber是否仍在等待事件
5     boolean isUnsubscribed();
6 }
```

为了便于理解前面提到的取消订阅功能，请假设这种情况：订阅者是唯一对事件感兴趣的一方，并且订阅者会消费它们直到 CountDownLatch 发出一个外部信号。传入流每 100 毫秒生成一个新事件，而这些事件会产生无限序列，即 0, 1, 2, 3...。以下代码不仅演示了在定义响应式流时如何获取一个 Subscription，还展示了如何取消对流的订阅。

```
1 package com.lagou.webflux.demo;
2
```

```

3 import rx.Observable;
4 import rx.Subscription;
5
6 import java.util.concurrent.CountDownLatch;
7 import java.util.concurrent.TimeUnit;
8
9 public class Main {
10     public static void main(String[] args) throws InterruptedException {
11         CountDownLatch latch = new CountDownLatch(1);
12
13         Subscription subscription = Observable.interval(100,
14             TimeUnit.MILLISECONDS)
15             .subscribe(System.out::println);
16
17         // 启动新的线程，用于计数
18         new Thread(() -> {
19             try {
20                 Thread.sleep(3000);
21             } catch (InterruptedException e) {
22                 e.printStackTrace();
23             }
24             // 如果订阅票据还在订阅状态，则取消订阅
25             if (!subscription.isUnsubscribed()) subscription.unsubscribe();
26             latch.countDown();
27         }).start();
28
29         System.out.println("=====");
30         // 主线程等待
31         latch.await();
32         System.out.println("-----");
33     }
34 }
```

订阅者在此处接收事件 0, 1, 2, 3，之后，latch调用发生，这会导致订阅取消。

此时，响应式编程包含一个 Observable 流、一个 Subscriber，以及一个订阅票据：Subscription。

该 Subscription 会传达 Subscriber 从 Observable 生产者处接收事件的意图。

#### 6.2.4 操作符

响应式编程包含一个 Observable 流、一个 Subscriber，以及订阅票据 Subscription。

该 Subscription 会传达 Subscriber 从 Observable 生产者处接收事件的意图。

下面通过操作符对流过响应式流的数据进行转换。

RxJava 的整体功能仍隐藏在它的操作符中。操作符用于调整流的元素或更改流结构本身。

RxJava 为几乎所有可能的场景提供了大量的操作符，但是多数其他操作符只是这些基本操作符的组合。

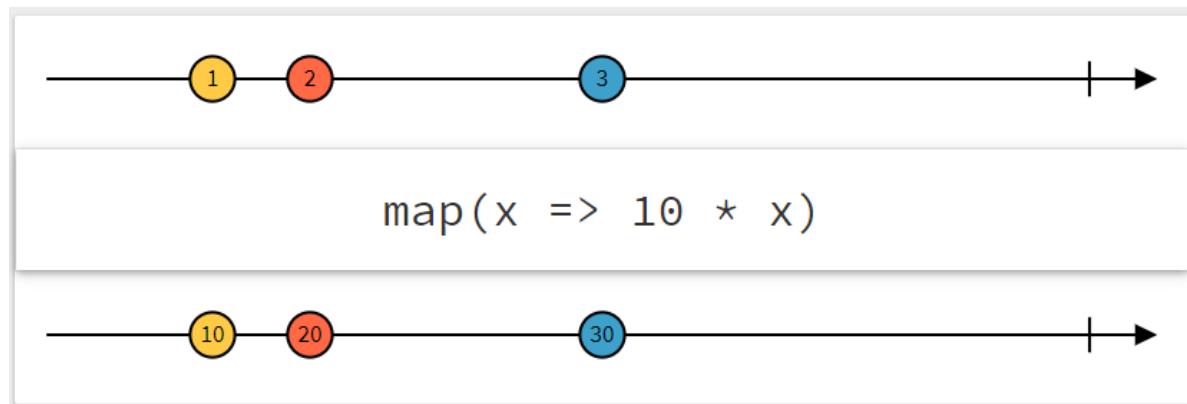
## map操作符

RxJava 中最常用的操作符是 map，它具有以下签名：

```
1 | public final <R> Observable<R> map(Func1<? super T, ? extends R> func)
```

func 函数可以将 T 对象类型转换为 R 对象类型，并且应用 map 将 Observable<T>转换为 Observable<R>。

使用弹珠图 (marble diagram) 描述操作符复杂的转换行为：



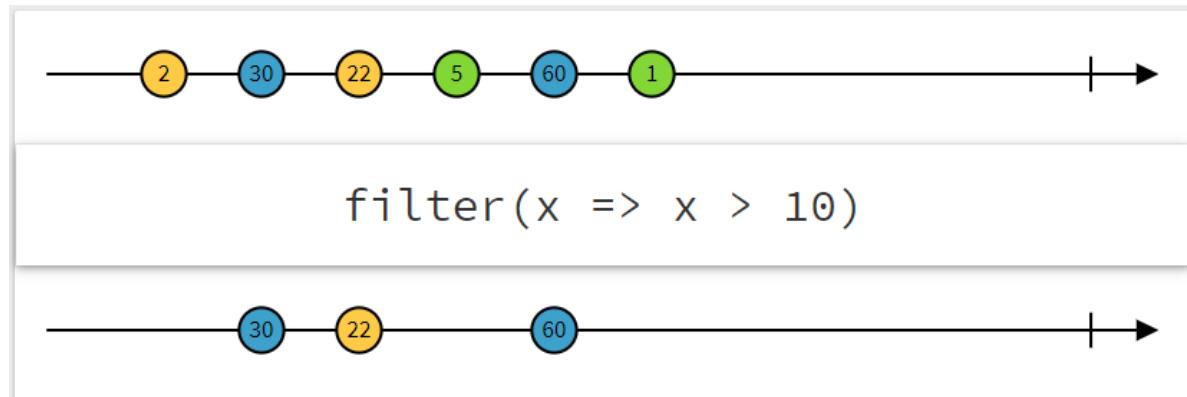
上图的map 操作符：通过对每个数据项应用函数来转换 Observable 发出的数据，map 执行一对一的转换。

此外，输出流具有与输入流相同数量的元素。

```
1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 import java.util.Arrays;
6
7 public class Main {
8     public static void main(String[] args) {
9         Observable<Integer> just = observable.just(1, 2, 3, 4, 5);
10        just.map(item -> {
11            var strings = new String[item];
12            Arrays.fill(strings, "a");
13            return strings;
14        }).forEach(item -> System.out.println(Arrays.toString(item)));
15    }
16 }
```

## filter操作符

与 map 操作符相比，filter 操作符所产生的元素可能少于它所接收的元素。它只发出那些已成功通过谓词测试的元素，如下图所示：

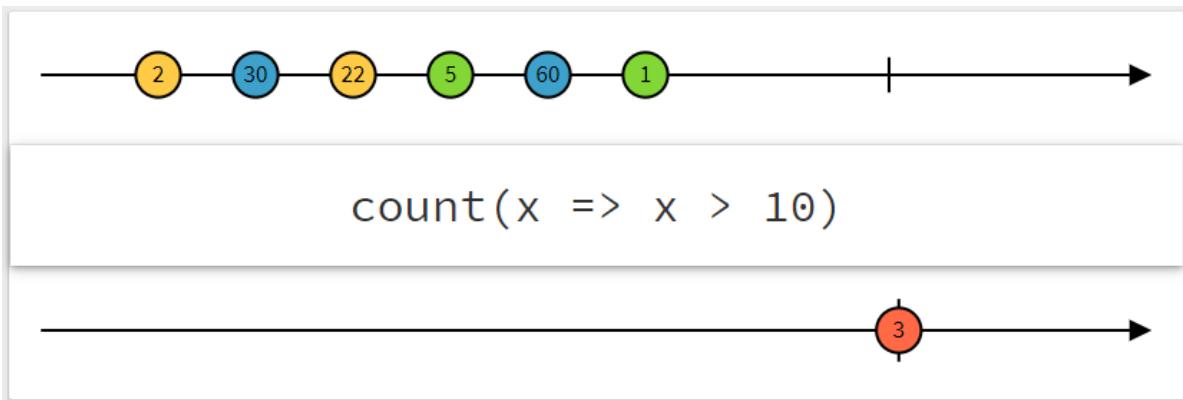


上图的filter 操作符：仅发出通过谓词测试的 Observable 中的数据项

```
1 package com.lagou.webflux.demo;  
2  
3 import rx.Observable;  
4  
5 import java.util.concurrent.TimeUnit;  
6  
7 public class Main {  
8     public static void main(String[] args) throws InterruptedException {  
9         Observable.interval(1, TimeUnit.SECONDS)  
10            .filter(item -> item % 2 == 0)  
11            .subscribe(System.out::println);  
12            Thread.sleep(10000);  
13    }  
14 }
```

## count操作符

count 操作符自描述性很强，它发出的唯一值代表输入流中的元素数量。但是，count 操作符只在原始流结束时发出结果，因此，在处理无限流时，count 操作符将不会完成或返回任何内容，如下图所示：



注意：该弹珠图的操作符不是java实现中的。

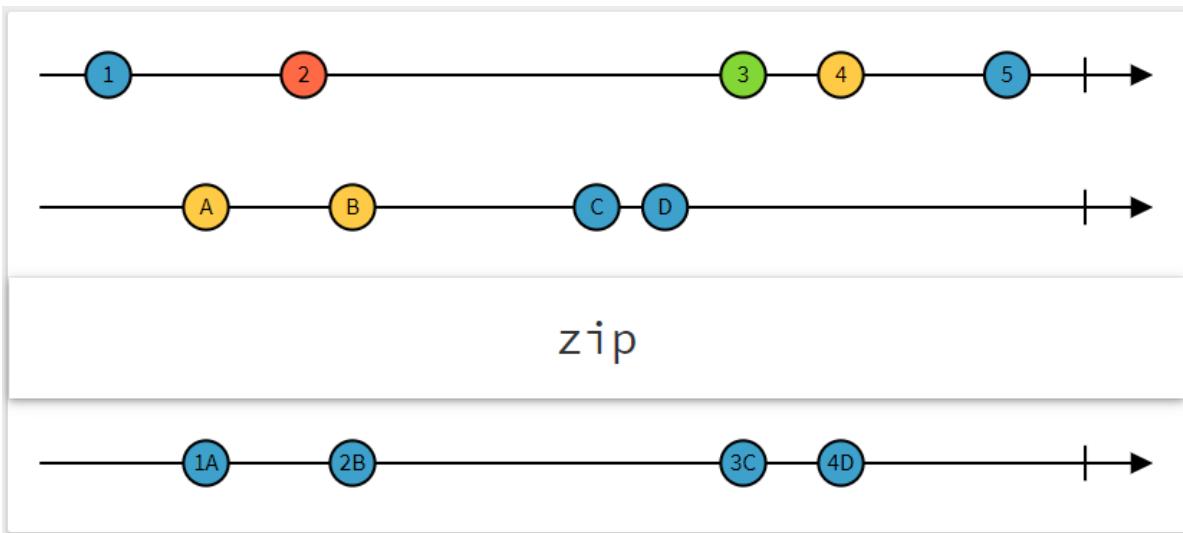
count 操作符：计算 Observable 源发出的数据项数，并仅发出该值。

```

1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 import java.util.ArrayList;
6
7 public class Main {
8     public static void main(String[] args) {
9
10         var list = new ArrayList<Integer>();
11
12         for (int i = 0; i < 1000; i++) {
13             list.add(i);
14         }
15
16         observable.from(list)
17             .filter(item -> item % 2 == 0)
18             .count()
19             .subscribe(item -> {
20                 System.out.println(Thread.currentThread().getName());
21                 System.out.println(item);
22             });
23     }
24 }
```

## zip操作符

该操作符具有更复杂的行为，因为它会通过应用 zip函数来组合来自两个并行流的值。它通常用于填充数据，且特别适用于部分预期结果从不同源获取的情况，如下图所示：



`zip` 操作符：通过指定的函数将多个 Observable 发送的元素组合在一起，并根据此函数的结果为每个组合发出单个数据项

简单起见，我们只 `zip` 两个字符串流，代码如下所示：

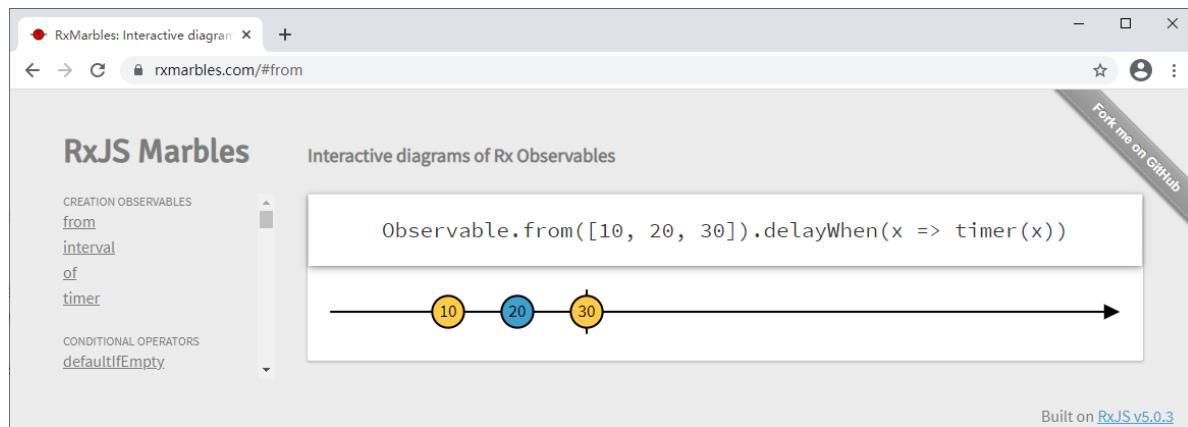
```

1 package com.lagou.webflux.demo;
2
3 import rx.Observable;
4
5 public class Main {
6     public static void main(String[] args) {
7         Observable.zip(
8             Observable.just(1, 2, 3, 4, 5),
9             Observable.just("a", "b", "c", "d", "e"),
10            (a, b) -> a + b)
11            .forEach(System.out::println);
12    }
13 }
```

访问<http://rxmarbles.com>查看响应式编程更多操作符的使用。该站点包含反映实际操作符行为的交互式图表。

交互式 UI 使我们能根据每个事件在流中出现的顺序和时间来将事件的转换可视化。

请注意，该站点本身是使用 RxJS 库构建的，而该库是 RxJava 在 JavaScript 中的对应。



我们也可以通过实现从 `Observable.Transformer<T,R>` 派生的类来编写自定义操作符。通过应用 `Observable.compose(transformer)` 操作符，我们可以将这样的操作符逻辑包括在工作流中。

RxJava 提供了一套强大的工具来构建复杂的异步工作流程。

### 6.2.5 RxJava的先决条件和优势

在生产者和订阅者之间通常存在一些订阅信息，这些信息使打破生产者-消费者关系成为可能。

这种方式非常灵活，并使我们可以控制生产和消费的事件数量，节省 CPU 时间（CPU 时间通常会浪费在创建永远不会用到的数据上）。

为了证明响应式编程提供了节省资源的能力，请假设我们需要实现一个简单的内存搜索引擎服务。该服务应该返回一个 URL 集合，其中的 URL 链接到包含了所需短语的文档。通常，客户端应用程序（Web 或移动应用程序）也会传入一些限制条件，例如有效结果的最大返回量。

如果没有响应式编程，我们可能使用以下 API 设计此类服务：

```

1 import java.net.URL;
2 import java.util.List;
3 public interface SearchEngine {
4     /**
5      * 将查询结果限定为limit条，返回结果
6      * @param query
7      * @param limit
8      * @return
9      */
10    List<URL> search(String query, int limit);
11 }
```

此时，即使有人在客户端结果界面只选择了第一个或第二个结果，服务的客户端也会收到整个结果集。

在这种情况下，虽然我们的服务做了很多工作，客户端也已经等了很长时间，客户端却忽略了大部分结果。这无疑是一种资源浪费。

我们可以通过遍历结果集来对搜索结果进行处理。因此，只要客户端继续消费它们，服务器就会搜索下一个结果项。

通常，服务器的搜索过程不是针对每一行，而是针对某些固定大小（比方说 100 项）。在客户端，结果以迭代器的形式表示。

改进后的服务 API：

```
1 import java.net.URL;
2 public interface IterableSearchEngine {
3     /**
4      * @param query
5      * @param limit
6      * @return
7      */
8     Iterable<URL> search(String query, int limit);
9 }
```

迭代器的唯一缺点是，客户端的线程在主动等待新的数据时会产生阻塞。

该交互方式效率不高，不足以构建高性能应用程序。

搜索引擎可以返回 CompletableFuture 以构建异步服务。

此时，客户端线程可以做一些有用的事情，而不会搅乱搜索请求，因为服务会在结果到达时立即执行回调。

但是在这里我们同样要么收到全部结果，要么收不到结果，因为 **CompletableFuture 只能包含一个值**，即使所包含的值是一个结果列表，也是如此。

代码如下所示：

```
1 import java.net.URL;
2 import java.util.List;
3 import java.util.concurrent.CompletableFuture;
4 public interface FutureSearchEngine {
5     /**
6      * 搜索
7      * @param query
8      * @param limit
9      * @return
10     */
11     CompletableFuture<List<URL>> search(String query, int limit);
12 }
```

通过使用 RxJava，返回一个流。

同时客户端可以随时取消订阅（即unsubscribe()），减少搜索服务处理过程中所需完成的工作量。

代码如下所示：

```
1 import rx.Observable;
2 import java.net.URL;
3 public interface RxSearchEngine {
4     /**
5      * 搜索
6      * @param query
7      * @return
8      */
9     Observable<URL> search(String query);
10 }
```

RxJava 使以更加通用和灵活的方式异步组合数据流成为可能。也可以将旧式同步代码包装到异步工作流中。

要调用速度比较慢的 Callable，可以使用 subscriberOn(Scheduler)操作符。该操作符定义启动流处理的 Scheduler（Java 中ExecutorService 的响应式对应类）。

应用场景：

```
1 String query = "";
2 Observable.fromCallable(() -> doSlowSyncRequest(query))
3     .subscribeOn(Schedulers.io())
4     .subscribe(this::processResult);
```

使用这种方法，不能依赖一个线程来处理整个请求，会阻塞。

**注意：**采用这种方法对可变对象有害，因为是多线程，唯一合理的策略是采用不变性(immutability)。

不变性不是一个新概念，它是函数式编程 (functional programming) 的核心原则之一。对象一旦被创建，就不会更改。这样一条简单的规则可以防止并行应用程序中可能出现的一大类问题。

在Java 8 引入lambda之前，没有lambda，就必须创建许多匿名类或内部类，这些类会污染应用程序代码，并且它们创建的样板代码多于有效代码。

在RxJava创建之初，尽管速度很慢，但Netflix仍广泛使用Groovy进行开发，这主要是因为Groovy 支持lambda。

## 6.2.6 RxJava使用

/src/main/resources/static/index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <title>Title</title>
6 </head>
7 <body>
8 <ul id="events"></ul>
9
10 <script type="application/javascript">
11   function add(message) {
12     const el = document.createElement("li");
13     el.innerHTML = message;
14     document.getElementById("events").appendChild(el);
15   }
16   var eventSource = new EventSource("/temperature-stream");
17   eventSource.onmessage = e => {
18     const t = JSON.parse(e.data);
19     const fixed = Number(t.value).toFixed(2);
20     add('Temperature: ' + fixed + ' C');
21   }
22   eventSource.onopen = e => add('Connection opened');
23   eventSource.onerror = e => add('Connection closed');
24 </script>
25
26 </body>
27 </html>
```

pom.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5 https://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>2.3.5.RELEASE</version>
11    <relativePath/> <!-- lookup parent from repository -->
12  </parent>
13  <groupId>com.lagou.webflux.demo</groupId>
14  <artifactId>demo</artifactId>
15  <version>0.0.1-SNAPSHOT</version>
16  <name>demo</name>
17  <description>Demo project for Spring Boot</description>
18
19  <properties>
```

```

18      <java.version>11</java.version>
19  </properties>
20
21  <dependencies>
22      <dependency>
23          <groupId>org.springframework.boot</groupId>
24          <artifactId>spring-boot-starter</artifactId>
25      </dependency>
26      <dependency>
27          <groupId>org.springframework.boot</groupId>
28          <artifactId>spring-boot-starter-web</artifactId>
29      </dependency>
30      <dependency>
31          <groupId>org.springframework.boot</groupId>
32          <artifactId>spring-boot-starter-thymeleaf</artifactId>
33      </dependency>
34
35      <dependency>
36          <groupId>io.reactivex</groupId>
37          <artifactId>rxjava</artifactId>
38          <version>1.3.8</version>
39      </dependency>
40
41      <dependency>
42          <groupId>org.json</groupId>
43          <artifactId>json</artifactId>
44          <version>20200518</version>
45      </dependency>
46
47      <dependency>
48          <groupId>org.springframework.boot</groupId>
49          <artifactId>spring-boot-starter-test</artifactId>
50          <scope>test</scope>
51          <exclusions>
52              <exclusion>
53                  <groupId>org.junit.vintage</groupId>
54                  <artifactId>junit-vintage-engine</artifactId>
55              </exclusion>
56          </exclusions>
57      </dependency>
58  </dependencies>
59
60  <build>
61      <plugins>
62          <plugin>
63              <groupId>org.springframework.boot</groupId>
64              <artifactId>spring-boot-maven-plugin</artifactId>
65          </plugin>
66      </plugins>
67  </build>
68
69  </project>

```

使用Temperature类来表示当前温度：

```
1 package com.lagou.webflux.demo.entity;
2
3 public class Temperature {
4     private final double value;
5
6     public Temperature(double value) {
7         this.value = value;
8     }
9
10    public double getValue() {
11        return value;
12    }
13 }
```

## 实现业务逻辑

TemperatureSensor 类先前已将事件发送到 Spring ApplicationEventPublisher，而现在它应该返回带有 Temperature 事件的响应式流。TemperatureSensor 的响应式实现如下所示：

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Temperature;
4 import org.springframework.stereotype.Component;
5 import rx.Observable;
6
7 import java.util.Random;
8 import java.util.concurrent.TimeUnit;
9
10 @Component
11 public class TemperatureSensor {
12     private final Random random = new Random();
13     private final Observable<Temperature> dataStream = Observable.range(0,
14     Integer.MAX_VALUE)
15         .concatMap(tick -> Observable.just(tick)
16             .delay(random.nextInt(5000), TimeUnit.MILLISECONDS)
17             .map(tickKey -> this.probe())
18             .publish()
19             .refCount()
20         );
21
22     private Temperature probe() {
23         return new Temperature(16 + random.nextGaussian() * 10);
24     }
25
26     public Observable<Temperature> temperatureStream() {
27         return dataStream;
28     }
29 }
```

## 自定义SseEmitter

TemperatureSensor 使用温度值暴露了一个流，通过使用 TemperatureSensor，可以将每个新的 SseEmitter 订阅到 Observable 流，并将收到的 onNext 信号发送给 SSE 客户端。

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Temperature;
4 import org.json.JSONObject;
5 import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;
6 import rx.Subscriber;
7
8 import java.io.IOException;
9
10 public class RxSseEmitter extends SseEmitter {
11
12     static final long SSE_SESSION_TIMEOUT = 30 * 60 * 1000L;
13
14     private final Subscriber<Temperature> subscriber;
15
16     public RxSseEmitter() {
17
18         super(SSE_SESSION_TIMEOUT);
19         this.subscriber = new Subscriber<Temperature>() {
20             @Override
21             public void onNext(Temperature temperature) {
22                 try {
23
24                     final JSONObject jsonObject = new
25                     JSONObject(temperature);
26                     final String temperatureJson = jsonObject.toString();
27
28                     System.out.println(temperatureJson);
29
30                     RxSseEmitter.this.send(temperatureJson);
31                 } catch (IOException e) {
32                     unsubscribe();
33                 }
34             }
35
36             @Override
37             public void onError(Throwable e) {
38                 System.err.println(e);
39             }
40
41             @Override
42             public void onCompleted() {
43                 System.out.println("job done");
44             }
45         };
46         onCompletion(subscriber::unsubscribe);
47         onTimeout(subscriber::unsubscribe);
48     }
}
```

```
49
50     public Subscriber<Temperature> getSubscriber() {
51         return subscriber;
52     }
53 }
54 }
```

## 暴露 SSE 端点

要暴露 SSE 端点，就需要一个与 TemperatureSensor 实例一起自动装配的 REST 控制器。以下代码展示使用了 RxSseEmitter 的控制器：

```
1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.service.RxSseEmitter;
4 import com.lagou.webflux.demo.service.TemperatureSensor;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RestController;
8 import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;
9
10 import javax.servlet.http.HttpServletRequest;
11
12 @RestController
13 public class TemperatureController {
14
15     private final TemperatureSensor temperatureSensor;
16
17     public TemperatureController(TemperatureSensor temperatureSensor) {
18         this.temperatureSensor = temperatureSensor;
19     }
20
21     @RequestMapping(value = "/temperature-stream", method =
22 RequestMethod.GET)
22     public SseEmitter events(HttpServletRequest request) {
23
24         RxSseEmitter emitter = new RxSseEmitter();
25         temperatureSensor.temperatureStream()
26             .subscribe(emitter.getSubscriber());
27         return emitter;
28     }
29
30 }
```

控制器既不管理无效的SseEmitter实例，也不关心线程同步。

响应式实现管理 TemperatureSensor 值、读取和发布的常规操作。

RxSseEmitter 将响应式流转换为传出的 SSE 消息，而 TemperatureController 仅将新的 SSE 会话绑定到订阅了温度读取流的新 RxSseEmitter。

此实现不使用 Spring 的事件总线，因此它更具可移植性，可以在不初始化 Spring 上下文的情况下进行测试。

## 应用程序配置

由于我们不使用发布-订阅方法以及Spring的@EventListener注解，就不依赖于异步支持，因此应用程序配置会变得更简单：

```
1 package com.lagou.webflux.demo;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class Demo12Application {
8
9     public static void main(String[] args) {
10         SpringApplication.run(Demo12Application.class, args);
11     }
12 }
13 }
```

注意：在基于 RxJava 的实现中，当没有订阅时，温度传感器不会探测温度。

## 6.3 Spring响应式编程的实现

### 6.3.1 Spring WebFlux

Spring Framework 5添加新模块 `spring-web-reactive`，使用响应式非阻塞引擎支持类似Spring MVC的@Controller编程模型。

下图是Spring MVC与Spring Web Reactive的关系对比：

## @Controller, @RequestMapping

Spring MVC

Spring Web Reactive

Servlet API

Reactive HTTP

Servlet Container

Servlet 3.1, Netty, Undertow

Spring Web Reactive 使用 Servlet 3.1 非阻塞特性。也可以运行于非 Servlet 运行时，如 Netty 和 Undertow 等。

对每个运行时适配了一组公共的响应式 `ServerHttpRequest` 和 `ServerHttpResponse` 抽象，以 `Flux<DataBuffer>` 的形式暴露请求和响应，读写完全支持 **背压**。

`spring-core` 模块提供了 `Encoder` 和 `Decoder` 契约，用于对 `Flux` 的数据进行序列化和反序列化。

`spring-web` 模块添加了 JSON 和 XML 的实现，用于 web 应用或其他的 SSE 流和零拷贝文件传输。

`spring-web-reactive` 模块包含了 Spring Web Reactive 框架以支持 `@Controller` 编程模型。

重新定义了很多 Spring MVC 的契约，如 `HandlerMapping` 和 `HandlerAdapter` 以支持异步和非阻塞，响应式地操作 HTTP 的请求和响应。

Spring MVC 和 Spring Web Reactive 不共享任何代码，处理逻辑有很多是共通的。

跟 Spring MVC 的编程模型一样，但是支持响应式类型并且以响应式的方式执行。

下述类型都可以作为控制器方法的 `@RequestBody` 参数来使用：

- `Account account` — `account` 在调用控制器之前非阻塞地反序列化。
- `Mono<Account> account` — 控制器使用 `Mono` 声明执行的逻辑，当 `account` 反序列化之后执行。
- `Single<Account> account` — 跟 `Mono` 一样，但是使用 RxJava 执行引擎。
- `Flux<Account> accounts` — 输入流场景
- `Observable<Account> accounts` — 使用 RxJava 的输入流

返回值类型：

- `Mono<Account>` — 当 `Mono` 结束，非阻塞地序列化给定的 `Account` 对象
- `Single<Account>` — 跟 `Mono` 的一样，但是使用 RxJava 执行引擎。
- `Flux<Account>` — 流场景，根据请求 `content type` 的不同，有可能是 SSE。
- `Flux<SseEvent>` — SSE 流。

- `observable<SseEvent>` — 使用RxJava执行引擎的SSE流。
- `Mono<Void>` — 当Mono结束，请求处理结束。
- `void` — 当方法返回，请求处理结束。表示同步、非阻塞的控制器方法。
- `Account` — 非阻塞地序列化给定的Account，表示同步、非阻塞控制器方法。

### 6.3.2 WebSocket

最知名的全双工客户端-服务器通信双工协议，即WebSocket。

WebSocket 协议的通信于2013 年初引入到Spring 框架中，旨在进行异步消息发送，但其实际的实现仍然有一些阻塞操作。

例如，将数据写入I/O 或从I/O 读取数据仍然是阻塞操作，因此这二者都会影响应用程序的性能。

WebFlux 模块为WebSocket 引入了改进版本的基础设施。WebFlux 同时提供客户端和服务器基础设施。

#### 服务器端WebSocket API

WebFlux 提供`websocketHandler` 作为处理WebSocket 连接的核心接口。

该接口有一个名为handle 的方法，它接收`WebSocketSession`。`WebSocketSession` 类表示客户端和服务器之间的成功握手，并提供对包括有关握手、会话属性和传入数据流的信息的访问。

使用echo 消息响应发送者的示例：

```

1 class EchowebsocketHandler implements websocketHandler {
2     @Override
3     public Mono<Void> handle(WebSocketSession session) {
4         return session.receive()
5             .map(WebSocketMessage::getPayloadAsText)
6             .map(tm -> "Echo: " + tm)
7             .map(session::textMessage)
8             .as(session::send);
9     }
10 }
```

上述代码表示接收入站消息，并转换，然后封装为`WebSocketMessage`对象，发送出去。其中发送返回`Mono<Void>`，当写出完成，该`Mono`完成。

`WebSocketMessage` 是`DataBuffer` 的包装器，它提供了额外功能，例如将以字节为单位的有效负载转换为文本。

一旦提取了传入消息，我们在该文本前面加上“Echo:”后缀，将新文本消息包装在`WebSocketMessage` 中，并使用`WebSocketSession#send` 方法将其发送回客户端。这里，`send` 方法接受`Publisher<WebSocketMessage>`并返回`Mono<Void>`作为结果。

因此，通过使用Reactor API 中的`as` 操作符，我们可以将`Flux` 视为`Mono<Void>`，并使用`session::send` 作为转换函数。

除了WebSocketHandler 接口实现，设置服务器端WebSocket API 还需要配置其他 HandlerMapping 实例和WebSocketHandlerAdapter 实例。

以下代码为此类配置的示例：

```
1 @Configuration
2 public class websocketConfiguration {
3     @Bean
4     public HandlerMapping handlerMapping() {
5         SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
6         mapping.setUrlMap(Collections.singletonMap("/ws/echo", new
7 EchowebsocketHandler()));
8         // 为了在其他HandlerMapping实例之前处理SimpleUrlHandlerMapping，它应该具有
9         // 更高的优先级
10        mapping.setOrder(-1);
11        return mapping;
12    }
13
14    @Bean
15    public HandlerAdapter handlerAdapter() {
16        // 将HTTP 连接升级到WebSocket，然后调用了websocketHandler#handle 方法
17        return new websocketHandlerAdapter();
18    }
19 }
```

WebSocket API 的配置其实很简单。

## 客户端WebSocket API

与WebSocket 模块（基于Web MVC）不同，WebFlux 还为我们提供了客户端支持。要发送 WebSocket 连接请求，可以使用WebSocketClient 类。

WebSocketClient 有两个执行WebSocket连接的核心方法，如下代码：

```
1 public interface websocketClient {
2     Mono<Void> execute(
3         URI url,
4         websocketHandler handler
5     );
6     Mono<Void> execute(
7         URI url,
8         HttpHeaders headers,
9         websocketHandler handler
10    );
11 }
```

WebSocketClient 使用相同的WebSocketHandler 接口来处理来自服务器的消息并发回消息。有一些WebSocketClient 实现与服务器引擎相关，例如TomcatWebSocketClient实现或JettyWebSocketClient 实现。

在下面的示例中，查看ReactorNettyWebSocketClient：

```
1 /**
2 * 需要添加VM选项:
3 * --add-opens java.base/jdk.internal.misc=ALL-UNNAMED
4 * -Dio.netty.tryReflectionSetAccessible=true
5 * --illegal-access=warn
6 * @param args
7 * @throws InterruptedException
8 */
9 public static void main(String[] args) throws InterruptedException {
10     WebSocketClient client = new ReactorNettyWebSocketClient();
11     client.execute(URI.create("http://localhost:8080/ws/echo"),
12         session -> {
13             session.receive()
14                 .map(WebSocketMessage::getPayloadAsText)
15                 .subscribe(System.out::println);
16             return Flux.interval(Duration.ofMillis(500))
17                 .map(string::valueOf)
18                 .map(session::textMessage)
19                 .as(session::send);
20         }
21     ).subscribe();
22     Thread.sleep(5000);
23 }
```

前面的示例展示了如何使用ReactorNettyWebSocketClient 连接WebSocket 并开始向服务器定期发送消息。

## 对比WebFlux WebSocket 与Spring WebSocket 模块

Spring WebSocket 模块的主要缺点是它阻塞了与I/O的交互，而Spring WebFlux 提供了完全无阻塞的写入和读取。

WebFlux 模块通过使用响应式流规范和Project Reactor 提供了更好的流抽象。旧WebSocket 模块中的WebSocketHandler 接口只允许一次处理一条消息。此外，WebSocketSession#sendMessage 方法仅允许以同步方式发送消息。

旧Spring WebSocket 模块的一个关键特性就是它与Spring Messaging 模块的良好集成，而这能用@MessageMapping 注解来声明WebSocket 端点。

以下代码展示了旧WebSocket API 的简单示例，这些API 基于Web MVC，且使用Spring Messaging 中的注解：

```
1  @Controller
2  public class GreetingController {
3
4      @MessageMapping("/hello")
5      @SendTo("/topic/greetings")
6      public Greeting greeting(HelloMessage message) {
7          return new Greeting("Hello, " + message.getName() + "!");
8      }
9 }
```

上述代码展示了我们如何使用Spring Messaging 模块声明WebSocket 端点。遗憾的是，WebFlux 模块的WebSocket 集成缺少此类支持，为了声明复杂的处理程序，必须提供自己的基础设施。

### 作为WebSocket轻量级替代品的响应式SSE

与重量级WebSocket 一起，HTML5 引入了一种创建静态（在本例中为半双工）连接的新方法，其中服务器能够推送事件。该技术解决了与WebSocket 类似的问题。

例如，可以使用相同的基于注解的编程模型声明服务器发送事件（Server-Sent Events， SSE）流，但是返回一个无限的ServerSentEvent 对象流，如以下示例所示：

```
1  @RestController
2  @RequestMapping("/sse/stocks")
3  class StocksController {
4      final Map<String, Stocksservice> stocksserviceMap;
5
6      ...
7      @GetMapping
8      public Flux<ServerSentEvent<?>> streamStocks() {
9          return Flux
10             .fromIterable(stocksserviceMap.values())
11             .flatMap(Stocksservice::stream)
12             .<ServerSentEvent<?>>map(item ->
13                 ServerSentEvent
14                 .builder(item)
15                 .event("StockItem")
16                 .id(item.getId())
17                 .build()
18             )
19             .startWith(
20                 ServerSentEvent
21                 .builder()
22                 .event("Stocks")
23                 .data(stocksserviceMap.keySet())
24                 .build()
25             );
26     }
```

上述代码中的数字可以解释如下。

(1) 这是@RestController 类的声明。为了简化代码，我们跳过了构造函数和字段初始化部分。

(2) 在这里，我们声明处理程序方法，该方法使用熟悉的@GetMapping 注解。streamStocks 方法返回ServerSentEvent 的Flux，这意味着当前处理程序启用了事件流。然后，我们合并所有可用的股票来源和流更改到客户端。之后，应用映射，将每个StockItem 映射到ServerSentEvent，这里使用了静态 builder 方法。为了正确设置ServerSentEvent 实例，我们在构建器参数中提供事件ID 和事件名称，它允许在客户端区分消息。此外，使用特定的ServerSentEvent 实例启动Flux，它向客户端声明可用的股票通道。

正如上述示例所示，Spring WebFlux 能映射Flux 响应式类型的流特性，并向客户端发送无限的股票事件流。此外，SSE 流不要求我们更改API 或使用其他抽象。它只需要我们声明一个特定的返回类型，以帮助框架找出处理响应的方法。我们不必声明ServerSentEvent 的Flux，我们可以直接提供内容类型，如下例所示：

```
1 @GetMapping(produces = "text/event-stream")
2 public Flux<StockItem> streamStocks() {
3     // ...
4 }
```

在这种情况下，WebFlux 框架在内部将流的每个元素包装到ServerSentEvent 中。

正如上述示例所示，ServerSentEvent 技术的核心优势在于这种流模型的配置不需要额外的样板代码，而在WebFlux 中采用WebSocket 时则需要这些样板代码。这是因为SSE 是一种基于HTTP 的简单抽象，既不需要协议切换，也不需要特定的服务器配置。如上述示例所示，我们可以使用@RestController 和@XXXMapping 注解的传统组合来配置SSE。但是，对于WebSocket而言，我们需要自定义消息转换配置，例如手动选择特定的消息传递协议。相比之下，Spring WebFlux 为SSE 提供的消息转换器配置与为典型REST 控制器提供的相同。

另外，SSE 不支持二进制编码并将事件限制为UTF-8 编码。这意味着WebSocket 可能对较小的消息有用，并且在客户端和服务器之间传输的流量较少，因此具有较低的延迟。

总而言之，SSE 通常是WebSocket 的一个很好的替代品。由于SSE 是HTTP 协议的抽象，因此 WebFlux 支持与典型REST 控制器相同的声明性函数式端点配置和消息转换。

### 6.3.3 RSocket

RSocket是一个应用通信协议，用在多路复用全双工通信中。可以在TCP、WebSocket或其他字节流传输中使用。提供了如下交互模型：

- Request-Response — 发送一个消息，接收一个消息
- Request-Stream — 发送一个消息，接收返回的消息流
- Channel — 双向发送消息流
- Fire-and-Forget — 发送单向消息

建立初始连接之后，就没有客户端服务端的概念了，因为双方地位对称，都可以初始化交互。因此，RSocket中只有请求者和响应者，而没有客户端和服务端的概念，交互称为“请求流”或简单地称为“请求们”。

RSocket协议的关键特性和优势：

- 跨网络边界的响应式流语义 — 对于诸如“请求流”和“通道”之类的流请求，背压信号在请求者和响应者之间传播，从而允许请求者放慢源处的响应者的速度，从而减少了对网络层拥塞控制的依赖以及在网络级别或任何级别缓冲。
- Request throttling — 可以从两端发送的“LEASE”帧，因此命名为“Leasing”，以限制给定时间内另一端允许的请求总数。租约定期更新。
- Session恢复 — 这是专为断开连接而设计的，用于维护会话的状态。状态管理对于应用程序是透明的，并且可以与背压结合使用，从而可以在可能的情况下停止生产者并减少所需的状态量。
- 对大消息的分割和再组装。
- Keepalive (心跳)

## The Protocol

RSocket的优点之一是，它在线路上具有定义明确的行为，并且易于阅读的规范以及某些协议扩展。

### 建立连接

最初，客户端通过一些低级流传输（例如TCP或WebSocket）连接到服务器，并向服务器发送“SETUP”帧以设置连接参数。

服务器可以拒绝“SETUP”帧，但是通常在发送（对于客户端）和接收（对于服务器）之后，双方都可以开始发出请求，除非“SETUP”指示使用租赁语义来限制数量。在这种情况下，双方都必须等待另一端的“租约”帧以允许发出请求。

### 发起请求

一旦建立连接，双方就可以通过帧“REQUEST\_RESPONSE”，“REQUEST\_STREAM”，“REQUEST\_CHANNEL”或“REQUEST\_FNF”之一发起请求。这些帧中的每一个都将一个消息从请求者传送到响应者。

响应者然后可以返回带有响应消息的“PAYLOAD”帧，并且在“REQUEST\_CHANNEL”的情况下，请求者还可以发送带有更多请求消息的“PAYLOAD”帧。

当请求涉及诸如“请求流”和“通道”之类的消息流时，响应者必须遵守来自请求者的需求信号。需求表示为许多消息。初始需求在“REQUEST\_STREAM”和“REQUEST\_CHANNEL”框架中指定。随后的需求通过“REQUEST\_N”帧发出信号。

每一端还可以通过“METADATA\_PUSH”帧发送元数据通知，该元数据通知与任何单独的请求无关，而与整个连接有关。

### 消息格式

RSocket消息包含数据和元数据。元数据可用于发送路由，安全令牌等。数据和元数据的格式可以不同。每个类的Mime类型都在“SETUP”框架中声明，并应用于给定连接上的所有请求。

尽管所有消息都可以具有元数据，但通常每个请求都包含诸如路由之类的元数据，因此仅包含在请求的第一条消息中，即带有帧“REQUEST\_RESPONSE”，“REQUEST\_STREAM”，“REQUEST\_CHANNEL”或“REQUEST\_FNF”之一。

协议扩展定义了用于应用程序的通用元数据格式：

- [Composite Metadata](#)— 多个独立格式化的元数据条目。
- [Routing](#) — 请求的路由

## Java实现

RSocket的Java实现基于Project Reactor构建。TCP和WebSocket的传输建立在Reactor Netty上。作为响应式库，Reactor简化了实现协议的工作。对于应用程序，自然要配合使用带有声明性运算符和透明背压支持的“Flux”和“Mono”。

RSocket Java中的API设计为最小且基本的。它着重于协议功能，而将应用程序编程模型（例如RPC代码生成与其他）作为一个更高级别的独立关注点。

主合同io.rsocket.RSocket对四种请求交互类型进行建模，其中“Mono”表示对单个消息的承诺，“Flux”表示消息流，而“io.rsocket.Payload”表示实际消息，可以访问数据和元数据作为字节缓冲区。RSocket合约是对称使用的。为了进行请求，应用程序被赋予了一个“RSocket”来执行请求。为了响应，应用程序实现了“RSocket”来处理请求。

在大多数情况下，Spring应用程序不直接使用其API。但是，独立于Spring查看或试验RSocket可能很重要。RSocket Java存储库包含许多示例应用程序，以演示其API和协议功能。

## Spring Support

`spring-messaging` 模块包含如下内容：

- [RSocketRequester](#)— 流式API，使用 `io.rsocket.RSocket` 对数据和元数据编码解码，发起请求。
- [Annotated Responders](#) — `@MessageMapping` 注解的用于处理请求的处理器方法。

`spring-web` 模块包含了 `Encoder` 和 `Decoder` 的实现，如Jackson的CBOR/JSON，以及Protobuf。也包含了 `PathPatternParser` 以可插拔的方式，高效处理路径匹配。

Spring Boot 2.2支持通过TCP或WebSocket建立RSocket服务器，包括在WebFlux服务器中通过WebSocket公开RSocket的选项。还为RSocketRequester.Builder和RSocketStrategies提供客户端支持和自动配置。

Spring Security 5.2提供了RSocket支持。

Spring Integration 5.2提供了入站出站网关用于RSocket客户端和服务端的交互。

Spring Cloud Gateway支持RSocket连接。

#### 6.3.4 WebClient

Spring Framework 5在 `RestTemplate` 之外添加了新的响应式 `WebClient`。

每个支持的HTTP客户端适配了一组公共的响应式 `clientHttpRequest` 和 `clientHttpResponse` 抽象，以 `Flux<DataBuffer>` 的形式对外暴露请求和响应，读写完全支持背压。

`spring-core` 提供了 `Encoder` 和 `Decoder` 抽象，用于客户端的 `Flux` 字节进行序列化和反序列化。

`webClient` 示例程序：

```
1 clientHttpConnector connector = new ReactorClientHttpConnector();
2
3 WebClient.builder().clientConnector(connector).build()
4     .get()
5     .uri(URI.create("https://edu.lagou.com"))
6     .accept(MediaType.TEXT_HTML)
7     .retrieve()
8     .bodyToMono(String.class)
9     .subscribe(System.out::println);
10
11 Thread.sleep(10000);
```

## 第二部分 Spring WebFlux核心原理

### 7 Project Reactor介绍

#### 7.1 Spring WebFlux与Project Reactor

Spring Framework从版本5开始，基于Project Reactor支持响应式编程。

Project Reactor是用于在JVM上构建非阻塞应用程序的Reactive库，基于Reactive Streams规范。

Project Reactor是Spring生态系统中响应式的基础，并且与Spring密切合作进行开发。

Spring WebFlux要求Project Reactor作为**核心依赖项**。

#### 模块

Project Reactor由Reactor文档中列出的一组模块组成。

主要组件是Reactor Core，其中包含响应式类型Flux和Mono，它们实现了Reactive Stream的Publisher接口以及一组可应用于这些类型的运算符。

其他一些模块是：

`Reactor Test`——提供一些实用程序来测试响应流。

`Reactor Extra`——提供一些额外的Flux运算符。

`Reactor Netty`——无阻塞且支持背压的TCP，HTTP和UDP的客户端和服务器。

`Reactor Adapter`——用于与其他响应式库（例如RxJava2和Akka Streams）的适配。

`Reactor Kafka`——用于Kafka的响应式API，作为Kafka的生产者和消费者。

## 并发模型

有两种在响应式链中切换执行某些的方式：`publishOn` 和 `subscribeOn`。

区别如下：

- `publishOn(Scheduler scheduler)`——影响所有后续运算符的执行（只要未指定其他任何内容）
- `subscribeOn(Scheduler scheduler)`——根据链中最早的subscribeOn调用，更改整个操作符链所订阅的线程。它不影响随后对publishOn的调用的行为。

Schedulers类包含用于提供执行上下文的静态方法：

- `parallel()`——为并行工作而调整的固定工作池，可创建与CPU内核数量一样多的工作线程池。
- `single()`——单个可重用线程。此方法为所有调用方重用同一线程，直到调度程序被释放为止。如果您希望使用按呼叫专用线程，则可以为每个呼叫使用Schedulers.newSingle()。
- `boundedElastic()`——动态创建一定数量的工作者。它限制了它可以创建的支持线程的数量，并且可以在线程可用时重新调度要排队的任务。这是包装同步阻塞调用的不错选择。
- `immediate()`——立即在执行线程上运行，而不切换执行上下文。
- `fromExecutorService(ExecutorService)`——可用于从任何现有ExecutorService中创建调度程序。

## 7.2 Project Reactor 1.x版本

案例1：

```
1 public static void main(String[] args) {
2     // 创建Environment实例。
3     // Environment实例是执行上下文，负责创建特定的Dispatcher。
4     // 可以提供不同类型的分派程序，范围包括进程间分派到分布式分派。
5     Environment env = new Environment();
6     /*
7      * 创建Reactor实例，它是Reactor模式的直接实现。
8      * 我们使用Reactors类创建Reactor实例。
9      * 使用基于RingBuffer结构的Dispatcher预定义实现。

```

```

10     */
11     Reactor reactor = Reactors.reactor()
12         .env(env)
13         .dispatcher(Environment.RING_BUFFER)
14         .get();
15     // 声明通道Selector和Event消费者声明。注册一个事件处理程序：打印
16     // 通过字符串选择器进行过滤，该字符串选择器指示事件通道的名称。
17     // Selectors.$提供了更全面的标准选择，因此事件选择的最终表达式可能更复杂。
18     reactor.on$("channel"), event -> System.out.println(event.getData());
19     /*
20     底层实现中，事件由Dispatcher进行处理，然后发送到目的地。
21     根据Dispatcher的实现，可以同步或异步处理事件。
22     这提供了一种功能分解，并且通常以与Spring框架事件处理方法类似的方式工作。
23     */
24     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
25         // 给通道发送通知
26         reactor.notify("channel", Event.wrap("test"));
27     }, 0, 100, TimeUnit.MICROSECONDS);
28 }

```

## 案例2：

```

1 public static void main(String[] args) {
2     Environment env = new Environment();
3     /*
4      * Reactor实例是一个事件网关，允许其他组件注册事件消费者，这些事件消费者随后会得到事件
5      * 的通知。
6      * 消费者一般通过Selector进行注册，通过匹配通知的key，消费事件。
7      * Reactor得到事件通知时，Reactor通过Dispatcher分发任务
8      * 任务在线程中执行。
9      * 根据Dispatcher实现的不同，线程的调度不同。
10     */
11     Reactor reactor = Reactors.reactor(env);
12     // on方法使用指定的Selector将Stream关联到Observable
13     Stream<String> stream = Streams.on(reactor, $("channel"));
14     stream.map(s->"hello lagou - " + s)
15         .distinct() // 对连续的相同值进行去重
16         .filter((Predicate<String>) s -> s.length() > 2)
17         .consume(System.out::println);
18     // 使用指定的环境创建一个延迟流
19     // 第一个泛型表示值类型
20     // 第二个泛型表示可以消费值的消费者类型
21     Deferred<String, Stream<String>> input = Streams.defer(env);
22     // 获取Composable的子类，用于消费异常和值
23     Stream<String> compose = input.compose();
24     compose.map(m -> m + " = hello lagou")
25         .filter((Function<String, Boolean>) s -> s.contains("123"))
26         .map(Event::wrap) // 将数据封装为事件
27         // reactor.prepare方法用于创建一个优化的路径，给指定的key广播事件通知
28         .consume(reactor.prepare("channel")); // 给当前Composable关联一个消
29         // 费者，消费composable的数据
30         for (int i = 0; i < 1000; i++) {
31             // 接收指定的值，让底层的Composable可以消费
32             input.accept(UUID.randomUUID().toString());
33         }

```

通过与Spring框架的完美集成以及与Netty的组合，非常适合开发具备异步和非阻塞消息处理的高性能系统。

Reactor 1.x的缺点：

1. 该库没有背压控制。

除了阻塞生产线程或跳过事件之外，事件驱动的Reactor 1.x并没有提供控制背压的方法。

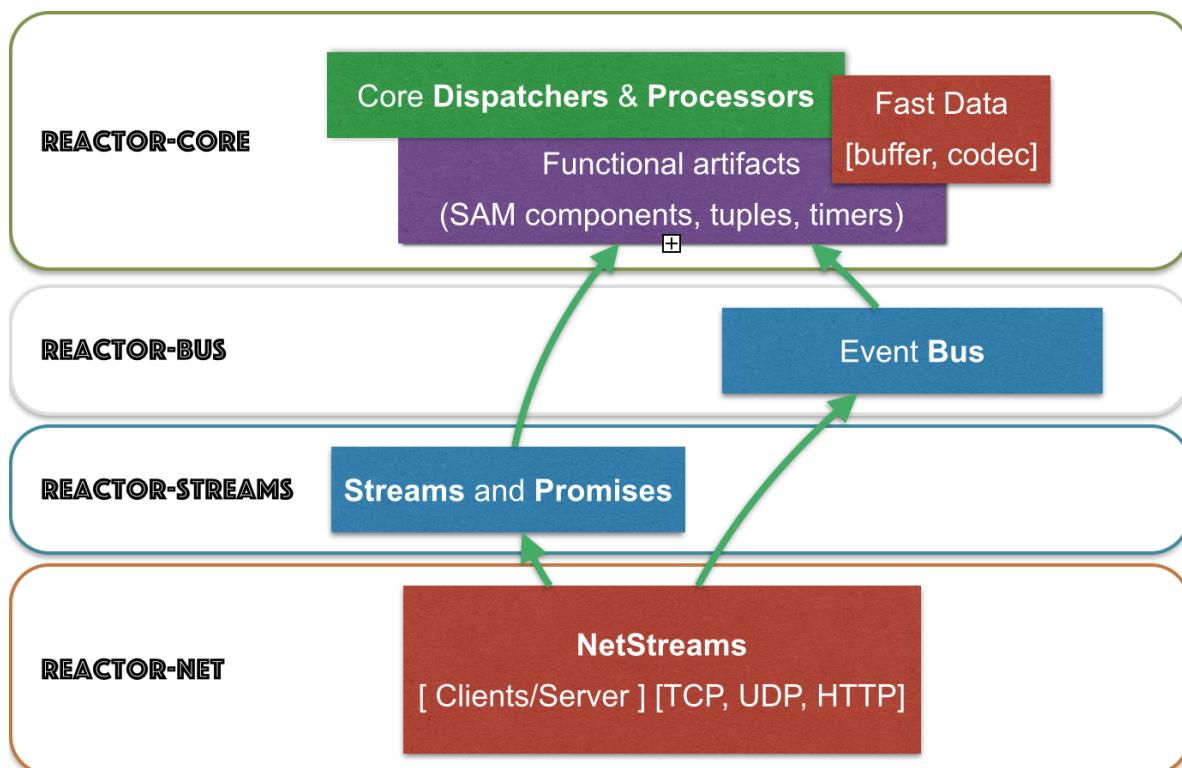
2. 错误处理非常复杂。

Reactor 1.x提供了几种处理错误和失败的方法，但是使用比较复杂。

### 7.3 Project Reactor 2.x版本

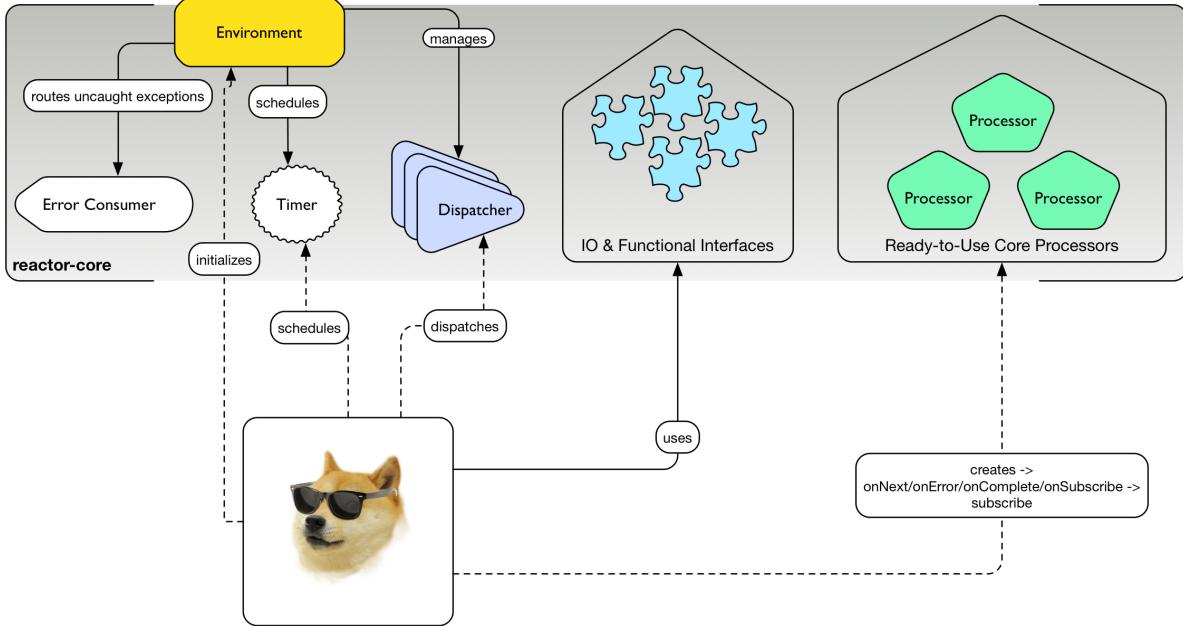
Stephane Maldini和Jon Brisbin在2015年初宣布Reactor 2.x版本。

Stephane Maldini形容Reactor 2.x： **Reactor 2首开响应式流的先河。**



在Reactor设计中，最重要的变化是将**事件总线**和**流功能**提取到单独的模块中。此外，深度的重新设计使新的Reactor Streams库完全符合响应式流规范。Reactor团队大大改进了Reactor的API。例如，新的Reactor API与Java Collections API具有更好的集成性。

在第二个版本中，Reactor的Streams API变得更加类似于RxJava API。除了用于创建和消费流的简单附加组件，它还在背压管理、线程调度和回弹性支持方面添加了许多有用的补充。



## 7.4 Project Reactor 3.x版本

Reactor事件总线在2中得到了改进。首先负责发送消息的Reactor对象被重命名为EventBus。该模块也经过重新设计以支持响应式流规范。

Maldini和Karnok将他们对RxJava和Project Reactor的想法和经验浓缩为一个名为reactive-stream-commons的库。后来该库成为Reactor 2.5的基础，并最终演变为Reactor 3.x。

经过一年的努力，Reactor 3.0发布。与此同时，一个完全相同的RxJava 2.0也浮出水面。RxJava与Reactor 3.x的相似性高于与其前身RxJava 1.x的相似性。这些最显著的区别是RxJava针对java6（包括安卓的支持），而Reactor 3选择java8作为基线。同时Reactor 3.x塑造了Spring 5框架的响应式变种。

该库支持所有常见的背压传播模式：

1. 仅推送：当订阅者通过`subscription.request(Long.MAX_VALUE)`请求有效无限数量的元素时。
2. 仅拉取：当订阅者通过`subscription.request(1)`仅在收到前一个元素后请求下一个元素时。
3. 拉-推（混合）：当订阅者有实时控制需求，且发布者可以适应所提出的数据消费速度时。

为适配不支持推-拉式操作模型的旧API，Reactor提供了许多老式背压机制，包括缓冲、开窗、消息丢弃、启动异常等。

某些情况下，上述策略甚至可以用于在实际需求出现之前预取数据，从而提高系统的响应性。

此外，Reactor API还提供了足够的工具用于消除用户活动的尖峰并防止系统过载。

Project Reactor在设计上旨在对并发透明，因此它不会强制执行任何并发模型。同时，它提供了一组有用的调度程序，它们几乎能以任何形式管理执行线程，如果所提出的所有调度程序都不符合要求，开发人员可以基于完全的低阶控制来创建自己的调度程序。

## 8 Project Reactor核心

### 8.1 在项目中添加Reactor

新建maven项目：

将Project Reactor作为依赖项添加到应用程序中：

```
1 <dependencies>
2   <dependency>
3     <groupId>io.projectreactor</groupId>
4     <artifactId>reactor-core</artifactId>
5     <version>3.4.0</version>
6   </dependency>
7   <dependency>
8     <groupId>io.projectreactor</groupId>
9     <artifactId>reactor-test</artifactId>
10    <version>3.4.0</version>
11    <scope>test</scope>
12  </dependency>
13 </dependencies>
```

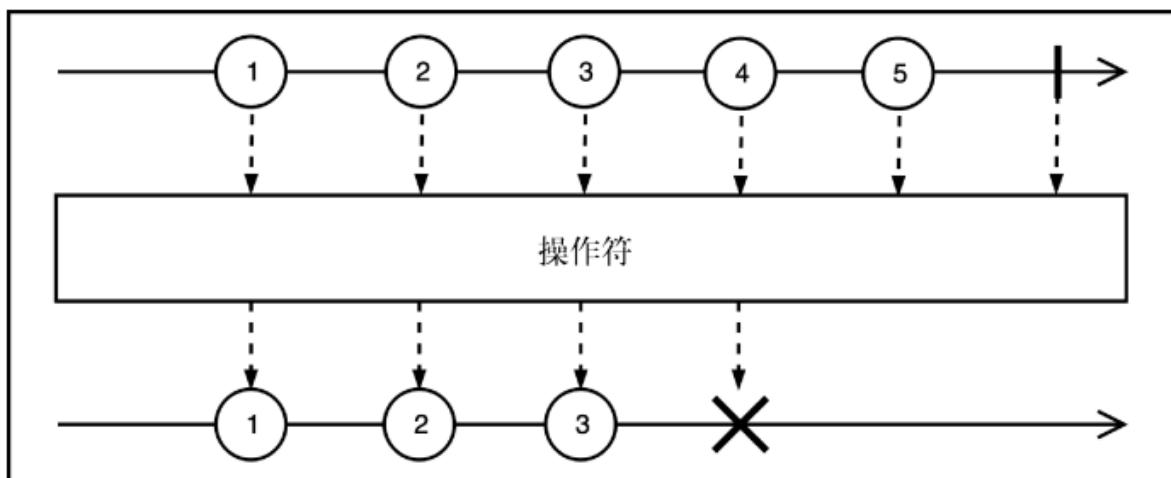
### 8.2 响应式类型——Flux和Mono

响应流规范只定义了4个接口，即

- `Publisher<T>`
- `Subscriber<T>`
- `Subscription`
- `Processor<T, R>`

Project Reactor提供了`Publisher<T>`接口的实现，即`Flux<T>`和`Mono<T>`。

#### 8.2.1. Flux



上图为将Flux流转换为另一个Flux流的示例。

Flux定义了一个通用的响应式流，它可以产生零个、一个或多个元素，乃至无限元素。有如下公式：

```
onNext x 0..N [onError | onComplete]
```

如下代码生成一个简单的无限响应流：

```
1 | Flux.range(1, 5).repeat();
```

该流重复产生1到5的数字（1,2,3,4,5,1,2。。。）。每个元素都无需完成整个流创建即可被转换和消费。

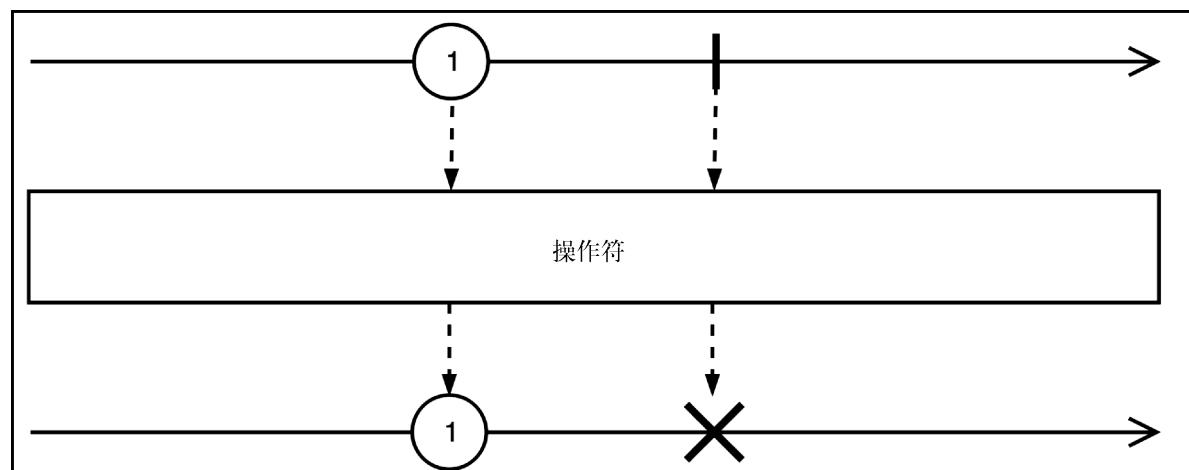
订阅者可以随时取消订阅从而将无限流转换为有限流。

收集无限流发出的所有元素会导致 `OutOfMemoryException`。

如下代码重现该问题：

```
1 | public static void main(String[] args) {
2 |     // range操作符创建从[1到100]的整数序列
3 |     Flux.range(1, 100)
4 |         // repeat操作符在源流完成之后一次又一次地订阅源响应式流。
5 |         // repeat操作符订阅流操作符的结果、接收从1到100的元素以及onComplete信号，
6 |         // 然后再次订阅、接收，不断重复该过程
7 |         .repeat()
8 |         // 使用collectList操作符尝试将所有生成的元素收集到一个集合中。
9 |         // 由于是无限流，最终它会消耗所有内存，导致OOM。
10 |        .collectList()
11 |        // block操作符触发实际订阅并阻塞正在运行的线程，直到最终结果到达
12 |        // 当前场景不会发生，因为响应流是无限的。
13 |        .block();
14 | }
```

### 8.2.2. Mono



上图为将Mono流转换为另一个Mono流的示例。

与Flux相比， Mono类型定义了一个最多可以生成一个元素的流，可以通过如下公式表示：

`onNext x 0..1 [onError | onComplete]`

当应用程序API最多返回一个元素时，可以使用 `Mono<T>`。

它可以轻松替换 `CompletableFuture<T>`，并提供相似的语义，只不过 `CompletableFuture` 在没有发出值的情况下无法正常完成。

`CompletableFuture` 会立即开始处理，而 `Mono` 在订阅者出现之前什么也不做。

`Mono`类型不仅提供了大量的响应式操作符，还能够整合到更大的响应式工作流程中。

当需要对已完成的操作通知客户端时，也可以使用 `Mono`。此时，可以返回 `Mono<Void>` 类型并在处理完成时发出 `onComplete()` 信号，或者在发生异常时返回 `onError()`。

此时，我们不返回任何数据，而是发出通知信号，而该信号可以用作进一步计算的触发器。

`Mono`和`Flux`可以容易地相互“转换”。

如：`Flux<T>.collectList()` 返回 `Mono<List<T>>`，而 `Mono<T>.flux()` 返回 `Flux<T>`。

### 8.2.3 RxJava 2响应式类型

即使 RxJava 2.x 库和 Project Reactor 具有相同的基础，RxJava 2 还是有一组不同的响应式发布者。

由于这两个库实现了相同的理念，包括响应式操作符、线程管理和错误处理，都非常相似。因此，或多或少熟悉其中一个库意味着同时熟悉了这两个库。

RxJava 1.x中最初只有 `observable` 这一个响应式类型，之后又添加了 `single` 和 `completable` 类型。

在版本 2 中，该库具有以下响应式类型：

- `Observable`
- `Flowable`
- `Single`
- `Maybe`
- `Completable`。

#### 1. Observable

1. 与 RxJava 1.x 的 Observable 语义几乎相同，但是，**不接收 null 值**。
2. Observable 既不支持背压，也不实现 Publisher 接口，所以它与响应式流规范**不直接兼容**。
3. Observable 类型的开销小于 Flowable 类型。
4. 它具有 toFlowable 方法，可以通过应用用户选择的背压策略将流转换为 Flowable。

## 2. Flowable

1. Flowable 类型是 **Reactor Flux** 类型的直接对应物。
2. 实现了响应式流的 Publisher，可以应用在由 Project Reactor 实现的响应式工作流中，因为 API 消费 Publisher 类型的参数，而不是针对特定库的 Flux 类型。

## 3. Single

1. Single 类型表示生成且仅生成一个元素的流。
2. 不继承 Publisher 接口。
3. 具有 toFlowable 方法。
4. 不需要背压策略。
5. 相较 Reactor 中的 Mono 类型，Single 更好地表示了 CompletableFuture 的语义，但是在订阅发生之前它仍然不会开始处理。

## 4. Maybe

1. 实现了与 Reactor 的 Mono 类型相同的语义，但是不兼容响应式流，因为 Maybe 不实现 Publisher 接口。
2. 具有 toFlowable 方法，以兼容响应式流规范。

## 5. Completable

1. 只能触发 onError 或 onComplete 信号，但不能产生 onNext 信号。
2. 不实现 Publisher 接口，但具有 toFlowable 方法。
3. 它对应不能生成 onNext 信号的 Mono<Void> 类型。

总而言之，要与其他兼容响应式流的代码集成，应将 RxJava 类型转换为 Flowable 类型。

## 8.3 创建 Flux 序列和 Mono 序列

Flux 和 Mono 提供了许多工厂方法，可以根据已有的数据创建响应流。

如，可以使用对象引用或集合创建 Flux，甚至可以简单地用数字范围来创建：

```
1 Flux<String> just = Flux.just("hello", "lagou");
2 just.subscribe(System.out::println);
3
4 Flux<String> stringFlux = Flux.fromArray(new String[]{"hello", "lagou",
5 "edu", "senior", "junior"});
6 stringFlux.subscribe(System.out::println);
7
8 Flux<Integer> integerFlux = Flux.fromIterable(Arrays.asList(1, 2, 3, 4, 5,
9 6, 7, 8));
10 integerFlux.subscribe(System.out::println);
11
12 // 第一个参数是起点, 第二个参数表示序列中元素的数量
13 Flux<Integer> range = Flux.range(1000, 5);
14 range.subscribe(System.out::println);
```

Mono 提供类似的工厂方法，但主要针对单个元素。它也经常与 nullable 类型和 Optional 类型一起使用：

```
1 Mono<String> just = Mono.just("yes");
2 just.subscribe(System.out::println);
3
4 Mono<Object> objectMono = Mono.justOrEmpty(null);
5 objectMono.subscribe(System.out::println);
6
7 // 避免空指针异常, 返回不包含任何值的optional对象。
8 Mono<Object> objectMono = Mono.justOrEmpty(Optional.empty());
9 objectMono.subscribe(System.out::println);
```

Mono 对于包装异步操作（如 HTTP 请求或数据库查询）非常有用。

Mono 提供了

- `fromCallable(Callable)`
- `fromRunnable(Runnable)`
- `fromSupplier(Supplier)`
- `fromFuture(CompletableFuture)`
- `fromCompletionStage(CompletionStage)`

等方法。

我们可以使用以下代码在 Mono 中包装长 HTTP 请求：

```
1 package com.lagou.webflux.demo;
2
3 import reactor.core.publisher.Mono;
4
5 import java.io.BufferedReader;
6 import java.io.IOException;
```

```

7 import java.io.InputStream;
8 import java.io.InputStreamReader;
9 import java.net.URL;
10 import java.netURLConnection;
11
12 public class Main {
13
14     public static String httpRequest() throws IOException,
15     InterruptedException {
16
17         URL url = new URL("https://edu.lagou.com");
18         URLConnection urlConnection = url.openConnection();
19         urlConnection.connect();
20         InputStream inputStream = urlConnection.getInputStream();
21         BufferedReader reader = new BufferedReader(new
22             InputStreamReader(inputStream));
23
24         String tmp = null;
25         StringBuffer stringBuffer = new StringBuffer();
26         while ((tmp = reader.readLine()) != null) {
27             stringBuffer.append(tmp).append("\r\n");
28         }
29         return stringBuffer.toString();
30     }
31
32     public static void main(String[] args) {
33         Mono.fromCallable(() -> httpRequest())
34             .subscribe(System.out::println);
35     }
36 }
```

或者，也可以使用 Java 8 方法引用语法将前面的代码重写为更短的代码：

```

1 public class Main {
2
3     // ...
4
5     public static void main(String[] args) {
6         Mono.fromCallable(Main::httpRequest)
7             .subscribe(System.out::println);
8     }
9 }
```

注意，上述代码不仅异步发出 HTTP 请求（由适当的 Scheduler 提供），还会处理onError信号传播的错误。

```

1 public class Main {
2
3     public static String httpRequest() throws IOException,
4     InterruptedException {
5
6         URL url = new URL("https://edu.mylagou.com");
7         URLConnection urlConnection = url.openConnection();
```

```

7     urlConnection.connect();
8     InputStream inputStream = urlConnection.getInputStream();
9     BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
10
11    String tmp = null;
12    StringBuffer stringBuffer = new StringBuffer();
13    while ((tmp = reader.readLine()) != null) {
14        stringBuffer.append(tmp).append("\r\n");
15    }
16    return stringBuffer.toString();
17}
18
19 public static void main(String[] args) {
20     Mono.fromCallable(Main::httpRequest)
21         .subscribe(
22             item -> System.out.println(item),
23             ex -> System.err.println("请求异常: " +
ex.toString()),
24             () -> System.out.println("请求结束")
25         );
26    }
27}

```

Flux 和 Mono 都可以使用 `from(Publisher<T> p)` 工厂方法适配任何其他 Publisher 实例。

```

1 package com.lagou.webflux.demo;
2
3 import org.reactivestreams.Publisher;
4 import org.reactivestreams.Subscriber;
5 import reactor.core.publisher.Flux;
6
7 public class Main {
8     public static void main(String[] args) {
9         Flux.from(new Publisher<String>() {
10             @Override
11             public void subscribe(Subscriber<? super String> subscriber) {
12                 for (int i = 0; i < 10; i++) {
13                     subscriber.onNext("hello" + i);
14                 }
15                 subscriber.onComplete();
16             }
17         }).subscribe(
18             System.out::println,
19             System.err::println,
20             () -> System.out.println("处理结束")
21         );
22     }
23 }

```

或者：

```

1 public class Main {

```

```

2
3     public static void main(String[] args) {
4         Flux.from((Publisher<String>) subscriber -> {
5             for (int i = 0; i < 10; i++) {
6                 subscriber.onNext("hello" + i);
7             }
8             subscriber.onComplete();
9         }).subscribe(
10            System.out::println,
11            System.err::println,
12            () -> System.out.println("处理结束")
13        );
14    }
15 }
```

两种响应式类型都提供了简便的方法来创建常用的空流以及只包含错误的流：

```

1 Flux<String> empty = Flux.empty();
2 Flux<String> never = Flux.never();
3 Mono<String> error = Mono.error(new RuntimeException("url不可达"));
```

1. empty()工厂方法，它们分别生成 Flux 或 Mono 的空实例。
2. never()方法会创建一个永远不会发出完成、数据或错误等信号的流。
3. error(Throwable)工厂方法创建一个序列，该序列在订阅时始终通过每个订阅者的onError(...)方法传播错误。由于错误是在 Flux 或 Mono 声明期间被创建的，因此，每个订阅者都会收到相同的 Throwable 实例。

defer 工厂方法创建一个序列，并在订阅时决定其行为，可以为不同的订阅者生成不同的数据：

```

1 package com.lagou.webflux.demo;
2
3 import reactor.core.publisher.Mono;
4
5 public class Main {
6
7     static boolean isvalidseed(String seed) {
8         System.out.println("调用了isValidSeed方法");
9         return true;
10    }
11
12    static String getData(String seed) {
13        try {
14            Thread.sleep(5000);
15        } catch (InterruptedException e) {
16            e.printStackTrace();
17        }
18        return "echo : " + seed;
19    }
20
21    static Mono<String> requestData(String seed) {
22        return isvalidseed(seed) ?
```

```

23             Mono.fromCallable(() -> getData(seed))
24             : Mono.error(new RuntimeException("Invalid seed
25             value"));
26
27     static Mono<String> requestDeferData(String seed) {
28         return Mono.defer(
29             () -> isvalidseed(seed) ?
30                 Mono.fromCallable(() -> getData(seed))
31                 : Mono.error(new RuntimeException("Invalid seed
32             value"))
33         );
34     }
35
36     public static void main(String[] args) {
37         requestData("zhangsan");
38         requestDeferData("zhangsan");
39         requestDeferData("zhangsan").subscribe();
40     }

```

总结：

1. Project Reactor 只需使用 just 方法枚举元素就可以创建 Flux 和 Mono 序列。
2. 可以使用 justOrEmpty 轻松地将 Optional 包装到 Mono 中，或者使用 fromSupplier 方法将 Supplier 包装到 Mono 中。
3. 可以使用 fromFuture 方法映射 Future，或使用 fromRunnable 工厂方法映射 Runnable。
4. 可以使用 fromArray 或 fromIterable 方法将数组或 Iterable 集合转换为 Flux 流。

## 8.4 订阅响应式流

Flux 和 Mono 提供了对 subscribe() 方法的基于 lambda 的重载，简化了订阅的开发。

subscribe 方法的所有重载都返回 Disposable 接口的实例，可以用于取消基础的订阅过程。

在重载方法1到4中，订阅发出对无界数据 (Long.MAX\_VALUE) 的请求。

**注意：**简单订阅请求无界数据 (Long.MAX\_VALUE) 的选项有时可能迫使生产者完成大量工作以满足需求。因此，如果生产者更适合处理有界数据请求，建议使用订阅对象或应用请求限制操作符来控制需求。

```

1 // 重载方法1
2 // 订阅流的最简单方法，忽略所有信号。通常用于触发具有副作用的流处理。
3 subscribe();
4 // 重载方法2
5 // 对每个值 (onNext 信号) 调用 dataConsumer，不处理 onError 和 onComplete 信号。
6 subscribe(Consumer<T> dataConsumer);
7 // 重载方法3
8 // 与重载方法2相同，处理 onError 信号，忽略 onComplete 信号。

```

```

9 subscribe(Consumer<T> dataConsumer, Consumer<Throwable> errorConsumer);
10 // 重载方法4
11 // 与重载方法3相同，处理 onComplete 信号。
12 subscribe(Consumer<T> dataConsumer, Consumer<Throwable> errorConsumer,
13   Runnable completeConsumer);
14 // 重载方法5
15 // 消费响应式流中的所有元素，包括错误处理和完成信号。重要的是，这种重载方法能通过请求足够数
16 // 量的数据来控制订阅，当然，请求数量仍然可以是 Long.MAX_VALUE。
17 subscribe(Consumer<T> dataConsumer, Consumer<Throwable> errorConsumer,
18           Runnable completeConsumer, Consumer<Subscription>
19             subscriptionConsumer);
17 // 重载方法6
18 // 订阅序列的最通用方式。在这里，可以为Subscriber的实现提供所需的行为。
19 subscribe(Subcriber<T> subscriber);

```

重载方法6非常通用，但很少被用到。

使用方式：

```

1 Flux.just("你好", "拉勾", "教育")
2   .subscribe(
3     data -> System.out.println("onNext:" + data),
4     ex -> System.err.println("异常信息: " + ex.getMessage()),
5     () -> System.out.println("响应流处理结束")
6   );

```

添加副作用：

```

1 public class Main {
2   public static void main(String[] args) {
3     Flux.range(1, 100)
4       .filter(item -> item % 7 == 0)
5       .map(num -> "hello lagou - " + num)
6       .doOnNext(System.out::println) // 添加副作用
7       .subscribe();
8   }
9 }

```

只处理正常情况：

```

1 public static void main(String[] args) {
2   Flux.range(1, 100)
3     .filter(item -> item % 9 == 0)
4     .subscribe(System.out::println);
5 }

```

添加异常的处理：

```
1 public static void main(String[] args) {
2     Flux.from(subscriber -> {
3         for (int i = 0; i < 5; i++) {
4             subscriber.onNext(i);
5         }
6         subscriber.onError(new Exception("异常测试"));
7     }).subscribe(
8         item -> System.out.println("onnext:" + item),
9         ex -> System.out.println("异常情况: " + ex)
10    );
11 }
```

添加完成事件的处理:

```
1 public static void main(String[] args) {
2     Flux.from(subscriber -> {
3         for (int i = 0; i < 5; i++) {
4             subscriber.onNext(i);
5         }
6         subscriber.onComplete();
7     }).subscribe(
8         item -> System.out.println("onnext:" + item),
9         ex -> System.out.println("异常情况: " + ex),
10        () -> System.out.println("处理完成")
11    );
12 }
```

添加订阅成功的处理:

```
1 public static void main(String[] args) {
2     Flux.range(1, 10).subscribe(
3         item -> System.out.println("onNext:" + item),
4         ex -> System.out.println("异常处理: " + ex),
5         () -> System.out.println("处理结束"),
6         subscription -> subscription.cancel() // 一订阅成功就取消订阅
7     );
8 }
```

如下响应流带有**手动订阅控制**功能:

```

1 Flux.range(1, 100)
2     .subscribe(
3         data -> System.out.println("onNext:" + data),
4         ex -> System.err.println("异常信息: " + ex.getMessage()),
5         () -> System.out.println("onComplete"),
6         subscription -> {
7             // 订阅响应式流5个元素
8             subscription.request(5);
9             // 取消订阅
10            subscription.cancel();
11        }
12    );

```

上述执行没有收到 onComplete 信号，因为订阅者在流完成之前取消了订阅。

### 注意：

1. 响应式流可以由生产者完成（使用 onError 或 onComplete 信号）；
2. 响应式流可以由订阅者通过 Subscription 实例进行取消。
3. Disposable 实例也可用于取消。

通常，Disposable实例不是由订阅者使用，而是由更上一级抽象的代码使用。

如在主线程通过调用 Disposable 来取消流处理：

```

1 Disposable disposable = Flux.interval(Duration.ofMillis(50))
2     .subscribe(
3         data -> System.out.println("onNext:" + data)
4     );
5 Thread.sleep(200);
6 // 主线程取消订阅
7 disposable.dispose();

```

### 实现自定义订阅者

如果默认的 subscribe(...)方法不提供所需的多种功能，则可以实现自己的Subscriber，直接从响应式流规范实现 Subscriber 接口，并将其订阅到流，如下所示：

```

1 Subscriber<String> subscriber = new Subscriber<String>() {
2     volatile Subscription subscription;
3     public void onSubscribe(Subscription s) {
4         subscription = s;
5         log.info("initial request for 1 element");
6         subscription.request(1);
7     }
8
9     public void onNext(String s) {
10        log.info("onNext: {}", s);
11        log.info("requesting 1 more element");
12    }
13 }

```

```

12     subscription.request(1);
13 }
14
15     public void onComplete() {
16         log.info("onComplete");
17     }
18
19     public void onError(Throwable t) {
20         log.warn("onError: {}", t.getMessage());
21     }
22 };
23
24 Flux<String> stream = Flux.just("Hello", "world", "!");
25 stream.subscribe(subscriber);

```

但是，上述定义订阅的方法是不对的。它打破了线性代码流，也容易出错。

最困难的部分是需要自己管理背压并正确实现订阅者的所有 TCK 要求。

在前面的示例中，打破了有关订阅验证和取消这几个 TCK 要求。

建议扩展 Project Reactor 提供的 BaseSubscriber 类。在这种情况下，订阅者如下所示：

```

1 class MySubscriber<T> extends BaseSubscriber<T> {
2     public void hookOnSubscribe(Subscription subscription) {
3         System.out.println("订阅成功，开始请求第一个元素");
4         request(1);
5     }
6
7     public void hookOnNext(T value) {
8         System.out.println("onNext: " + value);
9         System.out.println("再次请求一个元素");
10        request(1);
11    }
12 }

```

不仅可以重载 hookOnSubscribe(Subscription)方法，hookOnNext(T)方法，还可以重载 hookOnError(Throwable)方法、hookOnCancel()方法、hookOnComplete()方法以及其他方法。

BaseSubscriber 类提供了 request(long)和 requestUnbounded()这些方法来对响应式流需求进行粒度控制。

使用 BaseSubscriber 类，实现符合 TCK 的订阅者更为容易。

订阅者在本身拥有生命周期管理的宝贵资源时，会需要这种方法。例如，订阅者可能包装文件处理程序或连接到第三方服务的 WebSocket 链接。

## 8.5 用操作符转换响应式流

使用响应式流，除了需要能够创建和使用流，还必须能够完美地转换和操作。

Project Reactor 为几乎所有所需的响应式转换提供了工具，通常，可以对库的功能特性做如下分类：

1. 转换现有序列；
2. 查看序列处理的方法；
3. 拆分和聚合 Flux 序列；
4. 处理时间；
5. 同步返回数据。

下面讲解常用的操作符。

### 常规操作符

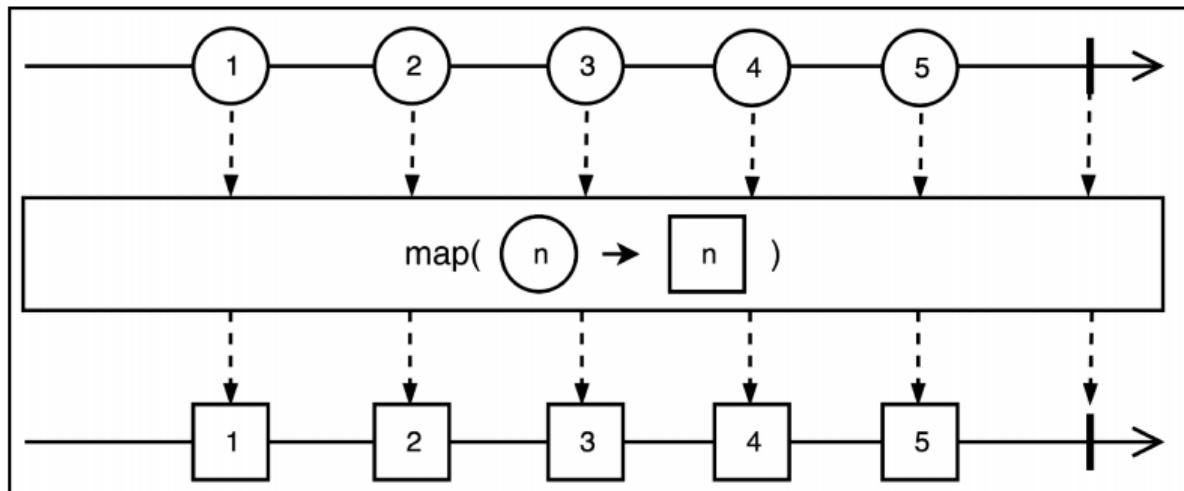
1. 映射响应式流元素

转换序列的最自然方式是将每个元素映射到一个新值。

Flux 和 Mono 给出了 `map` 操作符，具有 `map(Function<T, R>)` 签名的方法可用于逐个处理元素。

当操作符将元素的类型从 `T` 转变为 `R` 时，整个序列的类型将改变。

`Flux.map()` 的弹珠图



Mono 类的 `map` 操作符具有类似行为。

`cast(Class c)` 操作符将流的元素强制转换为目标类。

实现 `cast(Class c)` 操作符的最简单方法是使用 `map()` 操作符。

Flux类源码：

```
Flux.java x
3157     /**
3158     @ ...
3159     public final <E> Flux<E> cast(Class<E> clazz) {
3160         Objects.requireNonNull(clazz, "clazz");
3161         return map(clazz::cast);
3162     }
3163 }
```

`index` 操作符可用于枚举序列中的元素。该方法具有以下签名：`Flux<Tuple2<Long, T >> index()`。

```
1 | Flux.range(1, 10)
2 |     .map(item -> "hello lagou - " + item)
3 |     .index()
4 |     .subscribe(System.out::println);
```

`timestamp` 操作符的行为与 `index` 操作符类似，但会添加当前时间戳而不是索引。

```
1 | Flux.range(1, 10)
2 |     .map(item -> "hello lagou - " + item)
3 |     .timestamp()
4 |     .subscribe(System.out::println);
5 |
6 | Flux.range(1, 10)
7 |     .map(item -> "hello lagou - " + item)
8 |     .timestamp()
9 |     .subscribe(item -> System.out.println(item.getT1() + " <--> " +
item.getT2()));
```

## 2. 过滤响应式流

Project Reactor 包含用于过滤元素的各种操作符。

1. `filter` 操作符仅传递满足条件的元素。
2. `ignoreElements` 操作符返回 `Mono<T>` 并过滤所有元素。结果序列仅在原始序列结束后结束。
3. `take(n)` 操作符限制所获取的元素，该方法忽略除前  $n$  个元素之外的所有元素。
4. `takeLast` 仅返回流的最后一个元素。
5. `takeUntil(Predicate)` 传递一个元素直到满足某个条件。
6. `elementAt(n)` 只可用于获取序列的第  $n$  个元素。
7. `single` 操作符从数据源发出单个数据项，也为空数据源发出 `NoSuchElementException` 错误信号，或者为具有多个元素的数据源发出 `IndexOutOfBoundsException` 信号。它不仅可以基于一定数量来获取或跳过元素，还可以通过带有 `Duration` 的 `skip(Duration)` 或 `take(Duration)` 操作符。

8. `takeUntilOther(Publisher)` 或 `skipUntilOther(Publisher)` 操作符，可以跳过或获取一个元素，直到某些消息从另一个流到达。

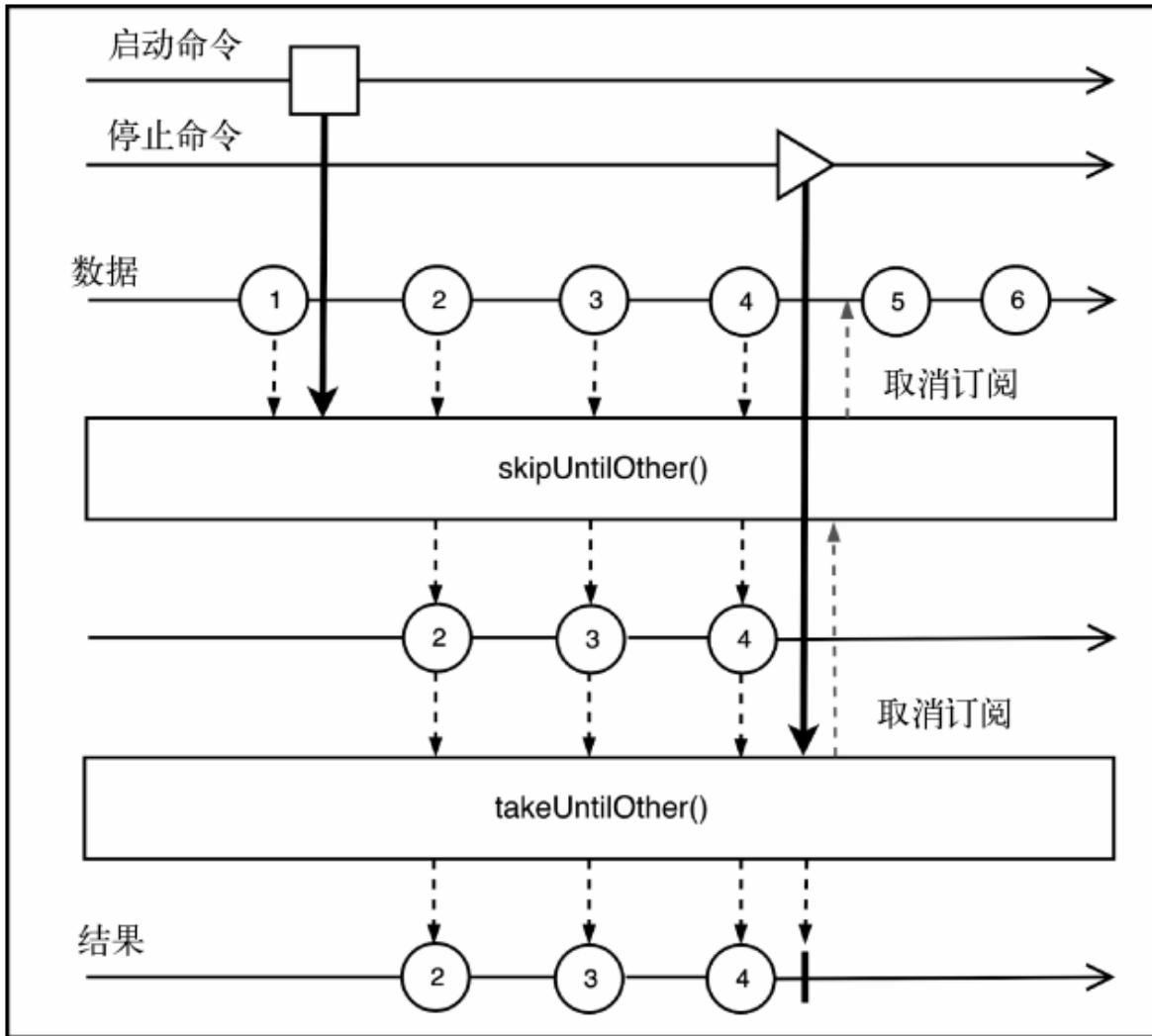
考虑如下工作流程，该工作流中，首先开始一个流的处理，然后从其他流收到特定事件之后，停止该流的处理。

代码如下所示：

```
1 System.out.println("开始");
2 Mono<String> start = Mono.just("start").delayElement(Duration.ofSeconds(3));
3 Mono<String> stop = Mono.just("stop").delayElement(Duration.ofSeconds(6));
4 Flux.interval(Duration.ofMillis(500))
5     .map(item -> "fluxelement" + item)
6     .skipUntilOther(start)
7     .takeUntilOther(stop)
8     .subscribe(System.out::println);
9 Thread.sleep(10000);
```

此时，可以启动然后停止元素处理，但只执行一次。

该场景的弹珠图



### 3. 收集响应式流

收集列表中的所有元素，并使用 `Flux.collectList()` 和 `Flux.collectSortedList()` 将结果集合处理为 Mono 流是可能的。`Flux.collectSortedList()` 不仅会收集元素，还会对它们进行排序。

请参考以下代码：

```

1 | Flux.just(1, 6, 2, 8, 3, 1, 5, 1)
2 |     .collectSortedList(Comparator.reverseOrder())
3 |     .subscribe(System.out::println);

```

请注意，收集集合中的序列元素可能耗费资源，当序列具有许多元素时这种现象尤为突出。

此外，尝试在无限流上收集数据可能消耗所有可用的内存。

Project Reactor 不仅可以将 Flux 元素收集到 List，还可以收集以下内容：

1. 使用 `collectMap` 操作符的映射 (`Map<K, T>`)；
2. 使用 `collectMultimap` 操作符的多映射 (`Map<K, Collection<T>>`)；
3. `Flux.collect(Collector)` 操作符收集到任何实现了 `java.util.stream.Collector` 的数据结构。

4. Flux 和 Mono 都有 `repeat()` 方法和 `repeat(times)` 方法，这两种方法可以针对传入序列进行循环操作。
5. `defaultIfEmpty(T)` 是另一个简洁的方法，它能为空的 Flux 或 Mono 提供默认值。
6. `Flux.distinct()` 仅传递之前未在流中遇到过的元素。但是，因为此方法会跟踪所有唯一性元素，所以（尤其是涉及高基数数据流时）请谨慎使用。distinct 方法具有重载方法，可以为重复跟踪提供自定义算法。因此，有时可以手动优化 distinct 操作符的资源使用。  
`Flux.distinctUntilChanged()` 操作符没有此限制，可用于无限流以删除出现在不间断行中的重复项。

高基数 (high-cardinality) 是指具有非常罕见元素或唯一性元素的数据。例如，身份编号和用户名就是典型的高基数数据，而枚举值或来自小型固定字典的值就不是。

`collectMap` 操作符的使用：

```

1  Flux.just(1,2,3,4,5,6)
2      .collectMap(new Function<Integer, String>() {
3          @Override
4              public String apply(Integer integer) {
5                  return "key:num-" + integer;
6              }
7      }).subscribe(System.out::println);
8
9  Flux.just(1, 2, 3, 4, 5)
10     .collectMap(item -> "key:num - " + item)
11     .subscribe(System.out::println);
12
13 Flux.just(1, 2, 3, 4, 5)
14     .collectMap(new Function<Integer, String>() {
15         @Override
16             public String apply(Integer integer) {
17                 return "key:" + integer;
18             }
19         , new Function<Integer, String>() {
20             @Override
21                 public String apply(Integer integer) {
22                     return "value:" + integer;
23                 }
24         }).subscribe(System.out::println);
25
26 Flux.just(1, 2, 3, 4, 5)
27     .collectMap(item1 -> "key:" + item1, item2 -> "value:" + item2)
28     .subscribe(System.out::println);
29
30 Flux.just(1, 2, 3, 4, 5)
31     .collectMap(integer -> "key:" + integer, integer -> "value:" +
32     integer, () -> {
33         Map<String, String> map = new HashMap<>();
34         for (int i = 0; i < 5; i++) {
35             map.put(i + "key", i + "value");
36         }
37         return map;
38     }).subscribe(System.out::println);

```

`collectMultimap` 的使用:

```
1 Flux.just(1, 2, 3, 4, 5)
2     .collectMultimap(item -> "key:" + item, item1 -> {
3         List<String> values = new ArrayList<>();
4         for (int i = 0; i < item1; i++) {
5             values.add("value" + i);
6         }
7         return values;
8     })
9     .subscribe(System.out::println);
10
11 Flux.just(1, 2, 3, 4, 5)
12     .collectMultimap(
13         item -> "key:" + item,
14         item1 -> {
15             List<String> values = new ArrayList<>();
16             for (int i = 0; i < item1; i++) {
17                 values.add("value" + i);
18             }
19             return values;
20         },
21         () -> {
22             Map map = new HashMap<String, List<String>>();
23             List<String> list = new ArrayList<>();
24             for (int i = 0; i < 3; i++) {
25                 list.clear();
26                 for (int j = 0; j < i; j++) {
27                     list.add("ele:" + j);
28                 }
29                 map.put(i + ":key", list);
30             }
31             return map;
32         }
33     )
34     .subscribe(System.out::println);
```

`repeat` 操作符的使用:

```
1 Flux.just(1, 2, 3)
2     .repeat(3) // 实际上是打印四次，1次原始的，3次重复的。
3     .subscribe(System.out::println);
```

`defaultIfEmpty` 操作符的使用:

```
1 Flux.empty()
2     .defaultIfEmpty("hello")
3     .subscribe(System.out::println);
```

`distinct` 操作符的使用:

```

1 Flux.just(1, 2, 3)
2     .repeat(3)
3     .distinct()
4     .subscribe(System.out::println);

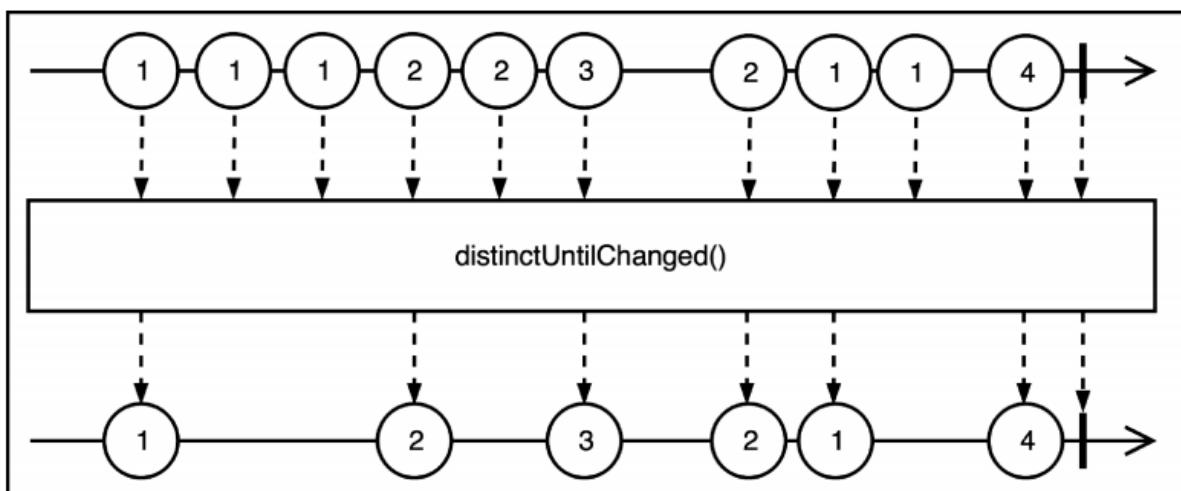
```

`distinctUntilChanged` 操作符的使用：

```

1 Flux.just(1, 2, 3)
2     .repeat(3)
3     .distinctUntilChanged()
4     .subscribe(System.out::println);
5
6 Flux.just(1, 1, 2, 2, 3, 3)
7     .distinctUntilChanged()
8     .subscribe(System.out::println);

```



#### 4. 裁剪流中元素

Project Reactor可以：

1. 统计流中元素的数量；
2. 检查所有元素是否具有 `Flux.all(Predicate)` 所需的属性；
3. 使用 `Flux.any(Predicate)` 操作符检查是否至少有一个元素具有所需属性；
4. 使用 `hasElements` 操作符检查流中是否包含多个元素；
5. 使用 `hasElement` 操作符检查流中是否包含某个所需的元素。短路逻辑，在元素与值匹配时立即返回true。
6. `any` 操作符不仅可以检查元素的相等性，还可以通过提供自定义 `Predicate` 实例来检查任何其他属性。

检查序列中是否包含偶数：

```
1 Flux.just(3, 5, 7, 9, 11, 15, 16, 17)
2     .any(item -> item % 2 == 0)
3     .subscribe(System.out::println);
```

`sort` 操作符在后台对元素进行排序，然后在原始序列完成后发出已排序的序列。

Flux 类能使用自定义逻辑来裁剪序列（也称为折叠）。`reduce` 操作符通常需要一个初始值和一个函数，而该函数会将前一步的结果与当前步的元素组合在一起。

将 1 到 5 之间的整数加起来：

```
1 Flux.range(1, 5)
2     .reduce(0, (item1, item2) -> item1 + item2) // 初始值，折叠操作
3     .subscribe(System.out::println);
```

`reduce` 操作符只生成一个具有最终结果的元素。

Flux.scan()操作符在进行聚合时，可以向下游发送中间结果。

`scan` 操作符对 1 到 5 之间的整数求和：

```
1 Flux.range(1, 5)
2     .scan(0, (num1, num2) -> num1 + num2)
3     .subscribe(System.out::println);
```

`scan` 操作符对于许多需要获取处理中事件的相关信息的应用程序有用。

例如，我们可以计算流上的移动平均值：

```
1 int arrLength = 5;
2 Flux.range(1, 500)
3     .index()
4     .scan(new int[arrLength], (arr, entry) -> { // scan第一个发射的元素是
它的初始值
5         arr[(int) (entry.getT1() % arrLength)] = entry.getT2();
6         return arr;
7     })
8     .skip(arrLength) // 当窗口数组被灌满之后开始计算平均值，因此跳过没有灌满的情况
9     .map(array -> Arrays.stream(array).sum() * 1.0 / arrLength)
10    .subscribe(System.out::println);
```

Mono 和 Flux 流有 `then`、`thenMany` 和 `thenEmpty` 操作符，它们在上游流完成时完成。

上游流完成处理后，这些操作符可用于触发新流，订阅是对于新流的。

```
1 Flux.just(1, 2, 3)
2     .doOnNext(item -> System.out.println("副作用: " + item))
3     .thenMany(Flux.just(4, 5, 6))
4     .subscribe(System.out::println);
```

即使 1、2 和 3 是由流生成和处理的，subscribe 方法中的 lambda 也只接收 4 和 5。

## 5. 组合响应式流

Project Reactor 可以将许多传入流组合成一个传出流。

指定的操作符虽然有许多重载方法，但是都会执行以下转换。

1. `concat` 操作符通过向下游转发接收的元素来连接所有数据源。当操作符连接两个流时，它首先消费并重新发送第一个流的所有元素，然后对第二个流执行相同的操作。
2. `merge` 操作符将来自上游序列的数据合并到一个下游序列中。与 `concat` 操作符不同，上游数据源是立即（同时）被订阅的。
3. `zip` 操作符订阅所有上游，等待所有数据源发出一个元素，然后将接收到的元素组合到一个输出元素中。
4. `combineLatest` 操作符与 `zip` 操作符的工作方式类似。但是，只要至少一个上游数据源发出一个值，它就会生成一个新值。

`concat` 操作符的使用：

```
1 Flux.concat(
2     Flux.range(1000, 20).delayElements(Duration.ofMillis(100)),
3     Flux.range(10, 20).delayElements(Duration.ofMillis(100))
4 )
5     .subscribe(System.out::println);
6 Thread.sleep(10000);
```

`merge` 操作符的使用：

```
1 Flux.merge(
2     Flux.range(10, 100).delayElements(Duration.ofMillis(100)),
3     Flux.range(1000, 100).delayElements(Duration.ofMillis(100))
4 )
5     .subscribe(System.out::println);
6 Thread.sleep(10000);
```

`zip` 操作符的使用：

```

1 Flux.zip(
2     Flux.range(1, 10),
3     Flux.range(100, 10)
4 )
5     .subscribe(System.out::println);

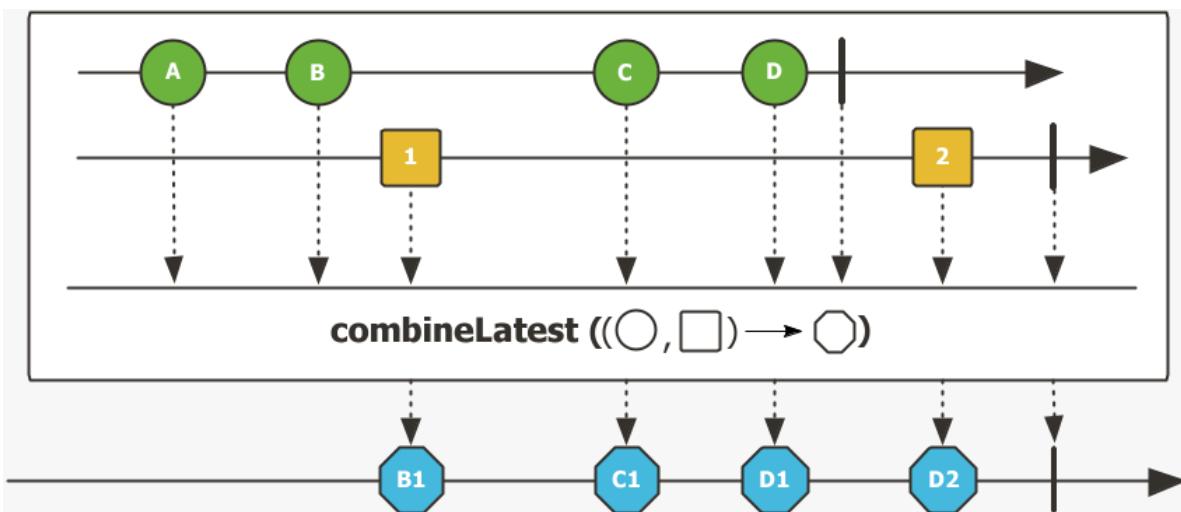
```

`combineLatest` 操作符的使用：

```

1 Flux.combineLatest(
2     Flux.range(1, 100).delayElements(Duration.ofMillis(100)),
3     Flux.range(1000, 100).delayElements(Duration.ofMillis(250)),
4     (integer, integer2) -> integer + "----" + integer2
5 )
6     .subscribe(System.out::println);
7 Thread.sleep(10000);

```



## 6. 流元素批处理

Project Reactor 支持以下几种方式对流元素 (`Flux<T>`) 执行批处理。

1. 将元素缓冲 (buffering) 到容器 (如 List) 中，结果流的类型为 `Flux<List<T>>`。
2. 通过开窗 (windowing) 方式将元素加入诸如 `Flux<Flux<T>>` 等流中。请注意，现在的流信号不是值，而是可以处理的子流。
3. 通过某些键将元素分组 (grouping) 到具有 `Flux<GroupedFlux<k, T>>` 类型的流中。

每个新键都会触发一个新的 `GroupedFlux` 实例，并且具有该键的所有元素都将被推送到 `GroupFlux` 类的该实例中。

可以基于以下场景进行缓冲和开窗操作：

1. 处理元素的数量，比方说每 10 个元素；

2. 一段时间，比方说每 5 分钟一次；
3. 基于一些谓语，比方说在每个新的偶数之前切割；
4. 基于来自其他 Flux 的一个事件，该事件控制着执行过程。

如，为列表（大小为 5）中的整数元素执行缓冲操作：

```
1 | Flux.range(1, 100)
2 |     .buffer(5)
3 |     .subscribe(System.out::println);
```

`buffer` 操作符将许多事件收集到一个事件集合中。该集合本身成为下游操作符的事件。当需要使用元素集合来生成一些请求，而不是使用仅包含一个元素的集合来生成许多小请求时，用缓冲操作符来实现批处理会比较方便。

如，可以将数据项缓冲几秒钟然后批量插入，而不是逐个将元素插入数据库。

`window` 操作符：

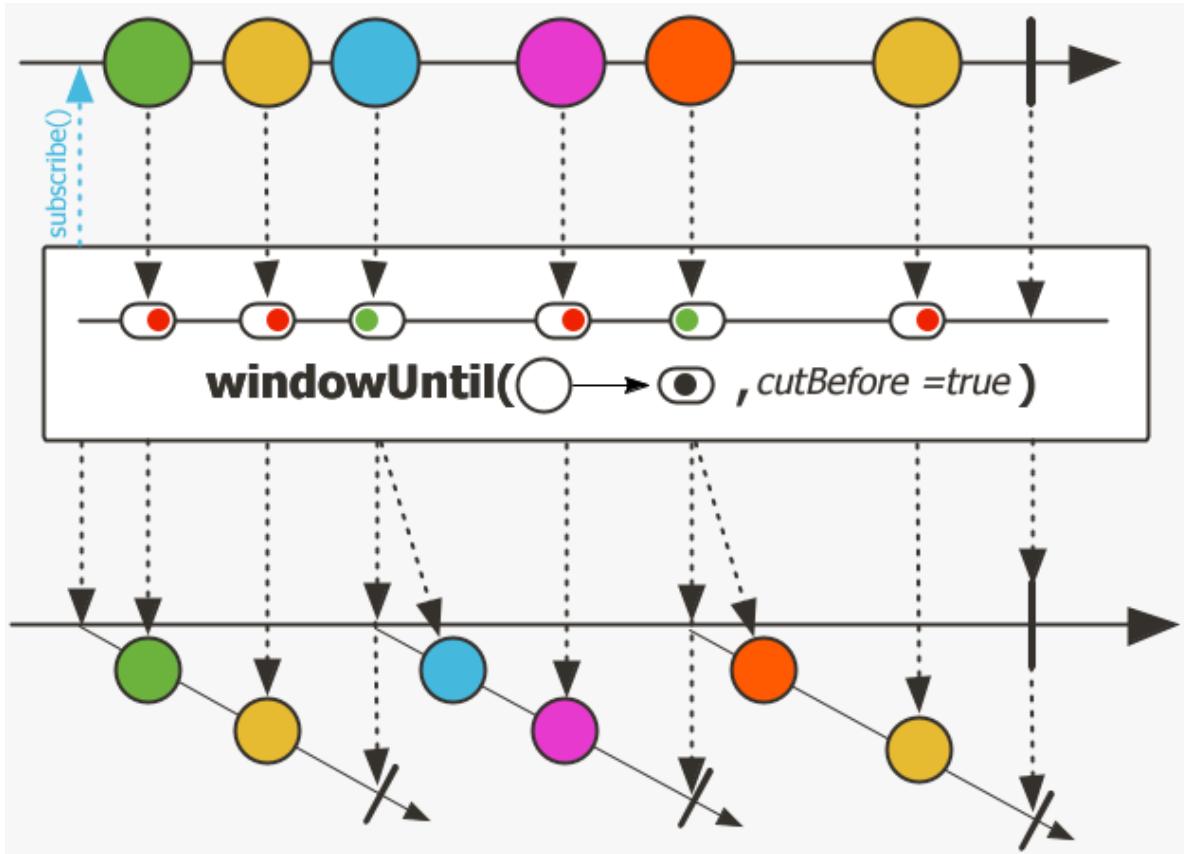
如果需要根据数字序列中的元素是否为素数进行开窗拆分，可以使用 `window` 操作符的变体 `windowUntil`。

它使用谓词来确定何时创建新切片。

代码如下所示：

```
1 | Flux.range(101, 20)
2 |     .windowUntil(Main::isPrime, true)
3 |     .subscribe(window ->
4 |         window.collectList()
5 |             .subscribe(
6 |                 item -> System.out.println("window:" + item)
7 |             )
8 |     );
9 |
10 // 判断是否为素数的方法
11 private static boolean isPrime(Integer integer) {
12     double sqrt = Math.sqrt(integer);
13     if (integer < 2) return false;
14     if (integer == 2 || integer == 3) return true;
15     if (integer % 2 == 0) return false;
16     for (int i = 3; i <= sqrt; i++) {
17         if (integer % i == 0) return false;
18     }
19     return true;
20 }
```

请注意第一个窗口为空。这是因为一旦启动原始流，就会生成一个初始窗口。然后，第一个元素会到达（数字 101），它是素数，会触发一个新窗口。因此，已经打开的窗口会在没有任何元素的情况下通过 `onComplete` 信号关闭。



window操作符和buffer操作符类似，后者仅在缓冲区关闭时才会发出集合，而 window 操作符会在事件到达时立即对其进行传播，以更快地做出响应并实现更复杂的工作流程。

groupBy 操作符通过某些条件对响应式流中的元素进行分组。通过对每个元素打一个标签(key)，按照标签将元素进行分组。

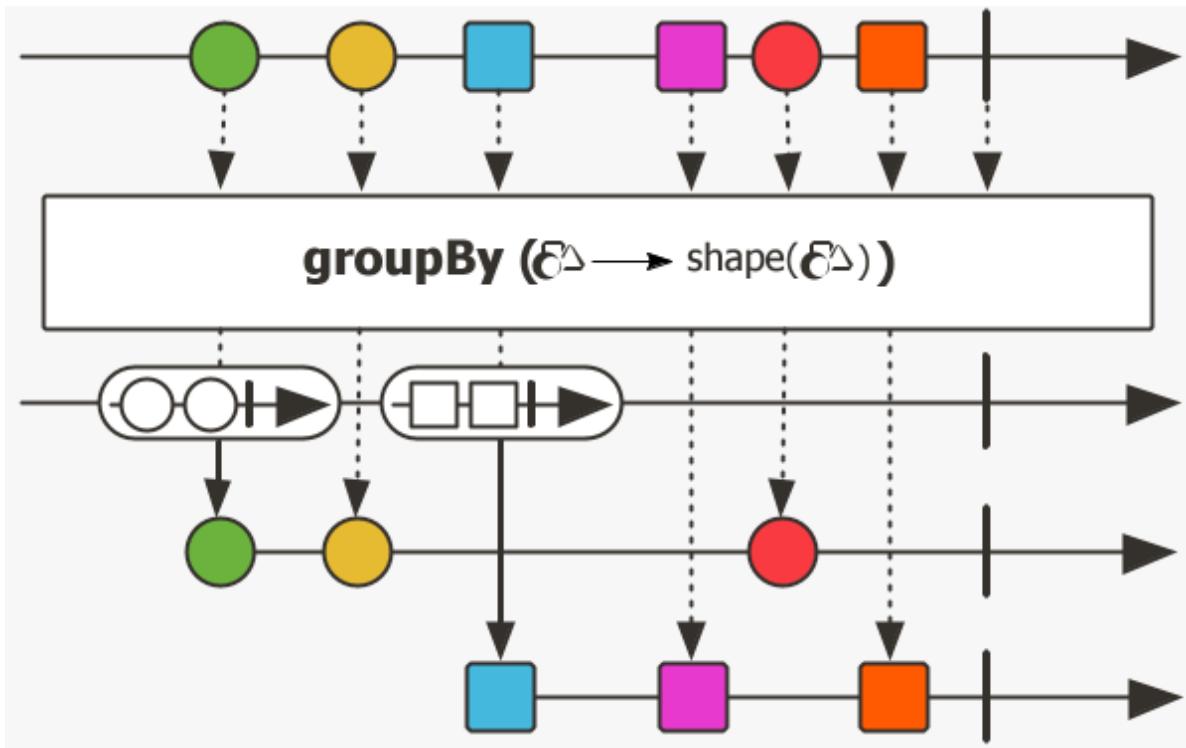
如：将整数序列按照奇数和偶数进行分组，并仅跟踪每组中的最后两个元素。

代码如下所示：

```

1 Flux.range(1, 7)
2     .groupBy(item -> item % 2 == 0 ? "偶数" : "奇数")
3     .subscribe(groupFlux -> groupFlux.scan(
4             new ArrayList<Integer>(),
5             (list, element) -> {
6                 list.add(element);
7                 if (list.size() > 2) {
8                     list.remove(0);
9                 }
10                return list;
11            }
12        ).filter(list -> !list.isEmpty())
13        .subscribe(item -> System.out.println(groupFlux.key() + " <-"
14          --> " + item))
15    );

```



此外，Project Reactor 库支持一些高级技术，例如在不同的时间窗口上对发出的元素进行分组。

## 7. flatMap、concatMap 和 flatMapSequential 操作符

flatMap 操作符在逻辑上由 map 和 flatten (就 Reactor 而言，flatten 类似于 merge 操作符) 这两个操作组成。

flatMap 操作符的 map 部分将传入的每个元素转换为响应式流 ( $T \rightarrow Flux<R>$ )；

flatten 部分将所有生成的响应式流合并为一个新的响应式流，通过该流可以传递 R 类型的元素。

Project Reactor 提供了 flatMap 操作符的一些不同变体。除了重载，该库还提供了 flatMapSequential 操作符和 concatMap 操作符。

这 3 个操作符在以下几个方面有所不同。

### 1. 操作符是否立即订阅其内部流；

flatMap 操作符和 flatMapSequential 操作符会立即订阅，而 concatMap 操作符则会在生成下一个子流并订阅它之前等待每个内部完成。

### 2. 操作符是否保留生成元素的顺序；

concatMap 天生保留与源元素相同的顺序，flatMapSequential 操作符通过对所接收的元素进行排序来保留顺序，而 flatMap 操作符不一定保留原始排序。

### 3. 操作符是否允许对来自不同子流的元素进行交错；

flatMap 操作符允许交错，而 concatMap 和 flatMapSequential 不允许交错。

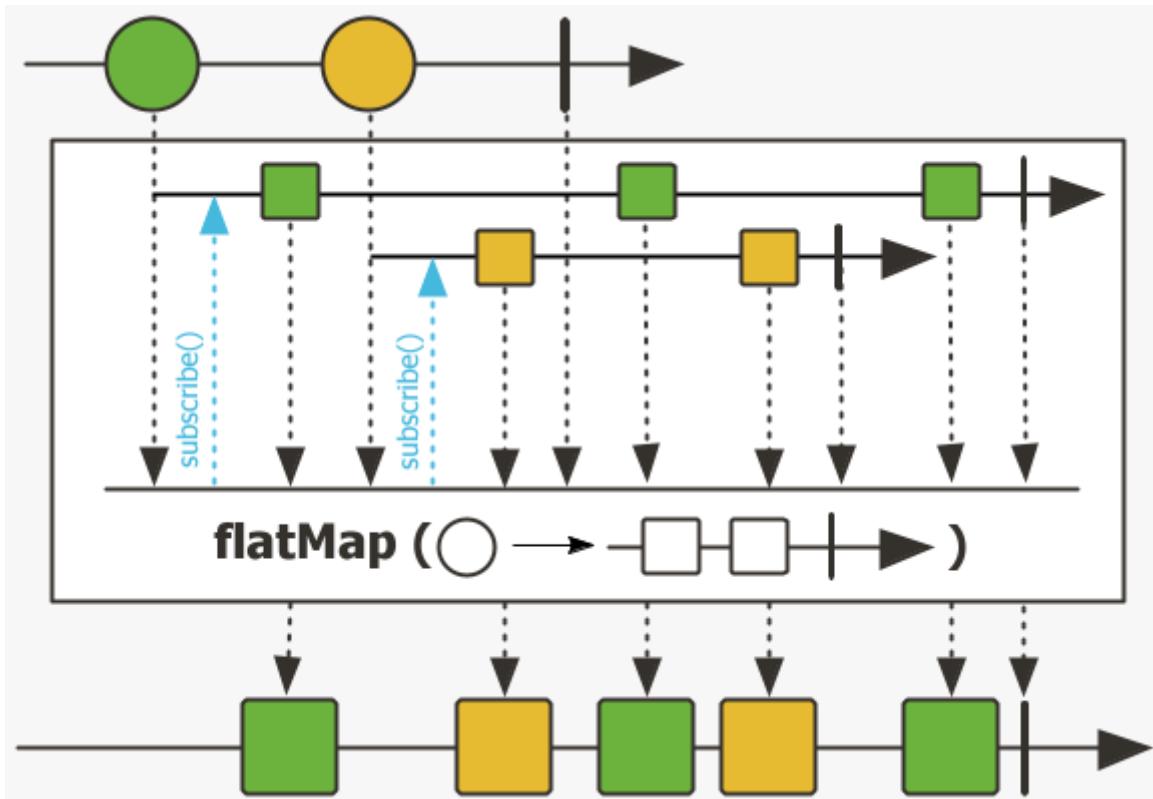
flatMap 操作符（及其变体）在函数式编程和响应式编程中都非常重要，因为它能使用一行代码实现复杂的工作流。

flatMap允许元素交错

flatMap 操作符:

```
1 Random random = new Random();
2 Flux.just(Arrays.asList(1, 2, 3), Arrays.asList("a", "b", "c", "d"),
3           Arrays.asList(7, 8, 9))
4         .doOnNext(System.out::println)
5         .flatMap(item -> Flux.fromIterable(item)
6                  .delayElements(Duration.ofMillis(random.nextInt(100) + 100))
7                  .doOnSubscribe(subscription -> {
8                     System.out.println("已订阅");
9                 }))
10        .subscribe(System.out::println);
```

flatMap 弹珠图:



concatMap不允许元素交错。

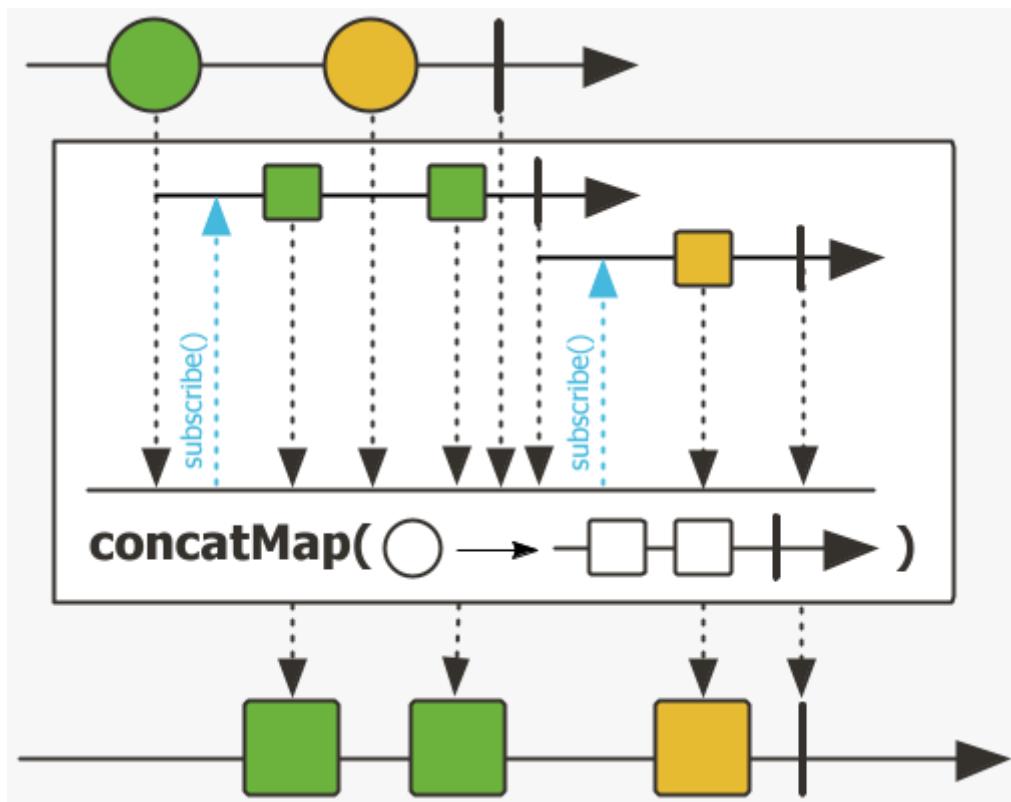
concatMap 操作符的使用:

```

1 Random random = new Random();
2 Flux.just(Arrays.asList(1, 2, 3), Arrays.asList("a", "b", "c", "d"),
3         Arrays.asList(7, 8, 9))
4     .doOnNext(System.out::println)
5     .concatMap(item -> Flux.fromIterable(item)
6                 .delayElements(Duration.ofMillis(random.nextInt(100) + 100)))
7     .doOnSubscribe(subscription -> {
8         System.out.println("已订阅");
9     })
10    .subscribe(System.out::println);
11 Thread.sleep(3000);

```

concatMap对每个上游的元素，在接收后都立即生成新的流，新流每个元素处理完之后，进行下一个新流的处理。



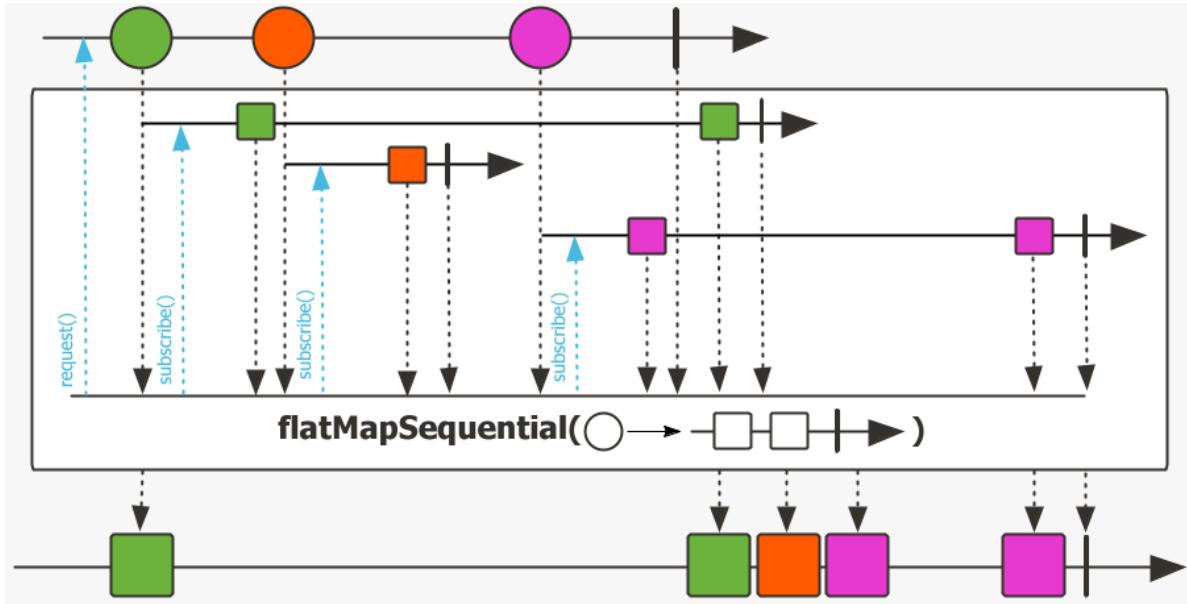
flatMapSequential不允许元素交错。

flatMapSequential 操作符的使用：

```

1 Random random = new Random();
2 Flux.just(Arrays.asList(1, 2, 3), Arrays.asList("a", "b", "c", "d"),
3         Arrays.asList(7, 8, 9))
4     .doOnNext(System.out::println)
5     .flatMapSequential(item -> Flux.fromIterable(item)
6                         .delayElements(Duration.ofMillis(random.nextInt(100) + 100)))
7     .doOnSubscribe(subscription -> {
8         System.out.println("已订阅");
9     })
10    .subscribe(System.out::println);

```



## 8. 元素采样

对于高吞吐量场景而言，通过应用采样技术处理一小部分事件是有意义的。

`sample` 操作符和 `sampleTimeout` 操作符可以让流周期性地发出与时间窗口内最近看到的值相对应的数据项。

我们假设使用以下代码：

```

1 // 每隔100ms就从流中取一个元素
2 Flux.range(1, 100)
3     .delayElements(Duration.ofMillis(10))
4     .sample(Duration.ofMillis(100))
5     .subscribe(System.out::println);
6 Thread.sleep(10000);

```

使我们每10毫秒都顺序生成数据项，订阅者也只会收到所指定的约束条件内的一小部分事件。通过这种方法，我们可以在不需要所有传入事件就能成功操作的场景下使用被动限速。流控。

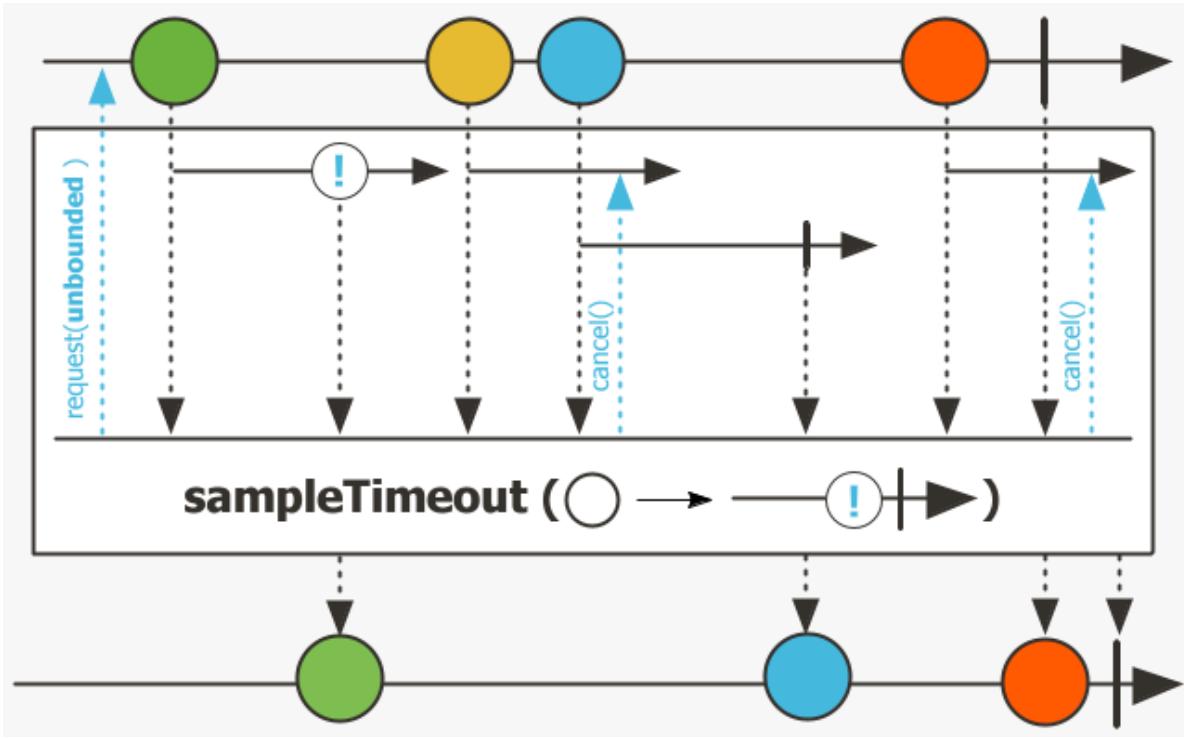
`sampleTimeout` 操作符：

```

1 Random random = new Random();
2 Flux.range(1, 20)
3     .delayElements(Duration.ofMillis(100))
4     .sampleTimeout(item -> Mono.delay(Duration.ofMillis(random.nextInt(100) +
5 50)), 20) // 并发计算超时时间，调节速度快慢。
6     .subscribe(System.out::println);
7 Thread.sleep(Long.MAX_VALUE);

```

流控。



## 9. 将响应式流转化为阻塞结构

Project Reactor 库提供了一个 API，用于将响应式流转换为阻塞结构。

有以下选项来阻塞流并同步生成结果：

1. `tolerable` 方法将响应式 Flux 转换为阻塞 Iterable。
2. `toStream` 方法将响应式 Flux 转换为阻塞 Stream API。从 Reactor 3.2 开始，在底层使用 `tolerable` 方法。
3. `blockFirst` 方法阻塞了当前线程，直到上游发出第一个值或完成流为止。
4. `blockLast` 方法阻塞当前线程，直到上游发出最后一个值或完成流为止。在 `onError` 的情况下，它会在被阻塞的线程中抛出异常。

`blockFirst` 操作符和 `blockLast` 操作符具有方法重载，可用于设置线程阻塞的持续时间。这应该可以防止线程被无限阻塞。

`tolerable` 和 `toStream` 方法能够使用 Queue 来存储事件，这些事件可能比客户端代码阻塞 Iterable 或 Stream 更快到达。微批处理。

```

1 | Flux.just(1, 2, 3).toIterable(3);
2 | Stream<Integer> integerStream = Flux.just(1, 2, 3).toStream(3);

```

```

1 | final Iterable<Integer> integers = Flux.just(1, 2, 3)
2 |         .delayElements(Duration.ofSeconds(1))
3 |         .toIterable();
4 | System.out.println("=====");
5 | for (Integer integer : integers) {
6 |     System.out.println(integer);

```

```

7 }
8 System.out.println("=====");
9
10
11
12 Stream<Integer> integerStream = Flux.just(1, 2, 3)
13     .delayElements(Duration.ofSeconds(1))
14     .toStream();
15 System.out.println("=====");
16 integerStream.forEach(System.out::println);
17 System.out.println("=====");
18
19
20
21 Integer integer = Flux.just(1, 2, 3)
22     .delayElements(Duration.ofSeconds(1))
23     .doOnNext(System.out::println)
24     .blockFirst();
25 System.out.println("=====");
26 System.out.println(integer);
27 System.out.println("=====");
28 Thread.sleep(5000);
29
30
31
32 // 该方法不会阻塞主线程
33 Flux.just(1, 2, 3)
34     .delayElements(Duration.ofSeconds(1))
35     .doOnEach(System.out::println)
36     .subscribe();
37 // 该方法阻塞，直到流处理到最后一个元素
38 Integer integer = Flux.just(1, 2, 3)
39     .delayElements(Duration.ofSeconds(1))
40     .doOnEach(System.out::println)
41     .blockLast();
42 System.out.println("=====");
43 System.out.println(integer);
44 System.out.println("=====");
45
46
47
48 Flux<Integer> integerFlux = Flux.just(1, 2,
49     3).delayElements(Duration.ofSeconds(1));
50 integerFlux.subscribe(item -> System.out.println("第一个订阅: " + item));
51 integerFlux.subscribe(item -> System.out.println("第二个订阅: " + item));
52 integerFlux.subscribe(item -> System.out.println("第三个订阅: " + item));
53 final Integer integer = integerFlux.blockLast();
54 System.out.println("阻塞等待最后一个元素" + integer);
55 System.out.println("=====");
56 Thread.sleep(5000);

```

## 10. 在序列处理时查看元素

有时，我们需要对处理管道中的每个元素或特定信号执行操作。为满足此类要求，Project Reactor 提供了以下方法。

1. `doOnNext(Consumer<T>)` 使我们能对 Flux 或 Mono 上的每个元素执行一些操作。
2. `doOnComplete` 和 `doOnError(Throwable)` 可以应用在相应的事件上。
3. `doOnSubscribe(Consumer<Subscription>)`、`doOnRequest(LongConsumer)` 和 `doOnCancel(Runnable)` 使我们能对订阅生命周期事件做出响应。
4. 无论是什么原因导致的流终止，`doOnTerminate(Runnable)` 都会在流终止时被调用。

此外，Flux 和 Mono 提供了 `doOnEach(Consumer<Signal>)` 方法，该方法处理表示响应式流领域的所有信号，包括 `onError`、`onSubscribe`、`onNext`、`onError` 和 `onComplete`。

考虑以下代码：

```
1 | Flux.just(1, 2, 3)
2 |     .concatWith(Flux.error(new RuntimeException("手动异常")))
3 |     .doOnEach(item -> System.out.println(item))
4 |     .subscribe();
```

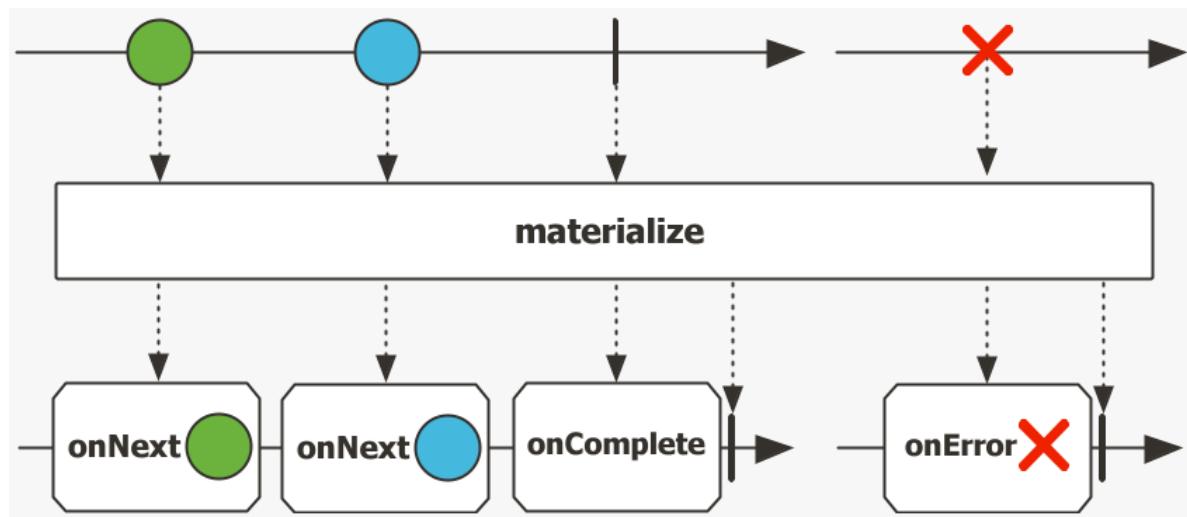
在这个例子中，我们不仅收到了所有的 `onNext` 信号，还收到了 `onError` 信号。

## 11. 物化和非物化信号

将流中的元素封装为 Signal 对象进行处理。

有时，采用信号进行流处理比采用数据进行流处理更有用。为了将数据流转换为信号流并再次返回，Flux 和 Mono 提供了 `materialize` 方法和 `dematerialize` 方法。示例如下：

```
1 | Flux.just(1, 2, 3).delayElements(Duration.ofMillis(1000))
2 |     .publishOn(Schedulers.parallel())
3 |     .concatWith(Flux.error(new Exception("手动异常")))
4 |     .materialize()
5 |     .doOnEach(item -> System.out.println(item.isIncomplete()))
6 |     .subscribe(System.out::println);
7 | Thread.sleep(10000);
```



这里，在处理信号流时，`doOnNext` 方法不仅接收带有数据的 `onNext` 事件，还接收包含在 `Signal` 类中的 `onComplete` 事件。此方法能采用一个类型层次结构来处理 `onNext`、`onError` 和 `onCompete` 事件。

如果我们只需要记录信号而不修改它们，那么 Reactor 提供了 `log` 方法，该方法使用可用的记录器记录所有处理过的信号。

## 8.6 以编程方式创建流

有时候需要一种更复杂的方法来在**流中生成信号**，或将对象的生命周期绑定到响应式流的生命周期。

### 1. `push` 和 `create` 工厂方法

`push` 工厂方法能通过适配一个单线程生产者来编程创建 `Flux` 实例。

此方法对于适配异步、单线程、多值 API 非常有用，而无须关注背压和取消，`push` 方法本身包含背压和取消。

如下代码：

```
1 // Java Stream API生成1000个整数元素并通过FluxSink的next方法将元素发送到下游响应式流
2 Flux.push(new Consumer<FluxSink<Integer>>() {
3     @Override
4     public void accept(FluxSink<Integer> fluxSink) {
5         IntStream.range(2000, 3000)
6             .forEach(fluxSink::next);
7     }
8 }).delayElements(Duration.ofMillis(1))
9 .subscribe(event -> System.out.println("onNext:" + event));
10 Thread.sleep(5000);
11
12 =====
13
14 Flux.push(fluxSink -> IntStream.range(1000, 2000).forEach(fluxSink::next))
15     .delayElements(Duration.ofMillis(100))
16     .subscribe(System.out::println);
17 Thread.sleep(10000);
```

`push` 工厂方法可以很方便地使用默认的背压和取消策略来适配异步 API。

`create` 工厂方法，与 `push` 工厂方法类似，起到桥接的作用。

该方法能从不同的线程发送事件。

如下代码所示：

```
1 static class MyEventProcessor {
2     private MyEventListener listener;
3     private Random random = new Random();
4     void register(MyEventListener listener) {
5         this.listener = listener;
6     }
7     public void process() {
8         while (random.nextInt(10) % 3 != 0) {
9             List<String> dataChunk = new ArrayList<>();
10            for (int i = 0; i < 10; i++) {
11                dataChunk.add("data-" + i);
12            }
13            listener.onDataChunk(dataChunk);
14        }
15        listener.processComplete();
16    }
17 }
18 interface MyEventListener<T> {
19     void onDataChunk(List<T> chunk);
20     void processComplete();
21 }
22
23 MyEventProcessor myEventProcessor = new MyEventProcessor();
24 Flux<String> bridge = Flux.create(sink -> {
25     myEventProcessor.register(
26         new MyEventListener<String>() {
27             public void onDataChunk(List<String> chunk) {
28                 for(String s : chunk) {
29                     sink.next(s);
30                 }
31             }
32             public void processComplete() {
33                 sink.complete();
34             }
35         });
36     });
37     bridge.subscribe(
38         System.out::println,
39         ex -> System.err.println(ex),
40         () -> System.out.println("处理完成")
41     );
42     myEventProcessor.process();
43     Thread.sleep(5000);
```

## 2. generate 工厂方法

generate 工厂方法旨在基于生成器的内部处理状态创建复杂序列。

它需要一个初始值和一个函数，该函数根据前一个内部状态计算下一个状态，并将 onNext 信号发送给下游订阅者。

例如，创建一个简单的响应式流来生成斐波那契 (Fibonacci) 数列 (1, 1, 2, 3, 5, 8, 13, ...)。

```
1 Flux.generate(
2     // 通过Callable提供初始状态实例
3     new Callable<ArrayList<Long>>() {
4         @Override
5         public ArrayList<Long> call() throws Exception {
6             final ArrayList<Long> longs = new ArrayList<>();
7             longs.add(0L);
8             longs.add(1L);
9             return longs;
10        }
11    },
12    // 负责生成斐波拉契数列
13    // 函数的第一个参数类型，函数第二个参数类型，函数计算结果类型
14    new BiFunction<ArrayList<Long>, SynchronousSink<Long>,
15    ArrayList<Long>>() {
16        @Override
17        public ArrayList<Long> apply(ArrayList<Long> longs,
18            SynchronousSink<Long> sink) {
19            final Long aLong = longs.get(longs.size() - 1);
20            final Long aLong1 = longs.get(longs.size() - 2);
21            // 向下游发射流元素
22            sink.next(aLong);
23            longs.add(aLong + aLong1);
24            return longs;
25        }
26    }
27 ).delayElements(Duration.ofMillis(500))
28     .take(10)
29     .subscribe(System.out::println);
30 Thread.sleep(5000);
```

Lambda形式：

```
1 Flux.generate(
2     () -> {
3         final ArrayList<Long> longs = new ArrayList<>();
4         longs.add(0L);
5         longs.add(1L);
6         return longs;
7    },
8    (state, sink) -> {
9        final Long aLong = state.get(state.size() - 1);
10       final Long aLong1 = state.get(state.size() - 2);
11       // 发射流元素
12       sink.next(aLong);
13       state.add(aLong + aLong1);
14       return state;
15 }
```

```
15     }
16 ).delayElements(Duration.ofMillis(500))
17     .take(10)
18     .subscribe(System.out::println);
19 Thread.sleep(5000);
```

状态还可以记录为二元组:

```
1 Flux.generate(
2     () -> Tuples.of(0L, 1L),
3     (state, sink) -> {
4         System.out.println("生成的数字: " + state.getT2());
5         sink.next(state.getT2());
6         long newValue = state.getT1() + state.getT2();
7         return Tuples.of(state.getT2(), newValue);
8     }
9 ).delayElements(Duration.ofMillis(500))
10    .take(10)
11    .subscribe(System.out::println);
12 Thread.sleep(5000);
```

在下一个值生成之前，每个新值都被同步传播给订阅者。

当生成不同的复杂响应式流，而该序列需要保持发射之间的中间状态时，该方法非常有用。

### 3. 将 disposable 资源包装到响应式流中

`using` 工厂方法能根据一个 `disposable` 资源创建流。它在响应式编程中实现了 `try-with-resources` 方法。

假设我们需要包装一个阻塞 API，而该 API 使用以下有如下表示:

```
1 static class Connection implements AutoCloseable {
2     private final Random rnd = new Random();
3     static Connection newConnection() {
4         System.out.println("创建Connection对象");
5         return new Connection();
6     }
7     public Iterable<String> getData() {
8         if (rnd.nextInt(10) < 3) {
9             throw new RuntimeException("通信异常");
10        }
11        return Arrays.asList("数据1", "数据2");
12    }
}
```

```
13     // close方法可以释放内部资源，并且应该始终被调用，即使在getData执行期间发生错误也是如此。
14     @Override
15     public void close() {
16         System.out.println("关闭Connection连接");
17     }
18 }
```

使用命令式方法，我们可以使用以下代码从连接接收数据：

```
1 try (Connection connection = Connection.newConnection()) {
2     connection.getData().forEach(data -> System.out.println("接收的数据: " +
3         data));
4 } catch (Exception e) {
5     System.err.println("错误信息: " + e);
6 }
```

有相同作用的响应式代码如下所示：

```
1 Flux.using(
2     Connection::newConnection,
3     connection -> Flux.fromIterable(connection.getData()),
4     Connection::close
5 ).subscribe(
6     data -> System.out.println("onNext接收到的数据: " + data),
7     ex -> System.err.println("onError接收到的异常信息: " + ex),
8     () -> System.out.println("处理完毕")
9 );
```

连接的生命周期与流的生命周期绑定。

操作符还可以在通知订阅者流终止之前或之后选择是否应该进行清除动作。

#### 4. 基于 usingWhen 工厂包装响应式事务

与 using 操作符类似，usingWhen 操作符使我们能以响应式方式管理资源。但是，using操作符会同步获取受托管资源（通过调用 Callable 实例）。同时，usingWhen 操作符响应式地获取受托管资源（通过订阅 Publisher 的实例）。此外，usingWhen 操作符接受不同的处理程序，以便应对主处理流终止的成功和失败。这些处理程序由发布者实现。

可以仅使用 usingwhen一个操作符实现完全无阻塞的响应式事务。

假设我们有一个完全响应式的事务。出于演示目的，代码做了简化处理。响应式事务实现如下所示：

```
1 static class Transaction {
2     private static final Random random = new Random();
3     private final int id;
4     /**
5      * 创建事务对象
6      * @param id
7      */
8     public Transaction(int id) {
9         this.id = id;
10        System.out.println("创建事务实例: " + id);
11    }
12    /**
13     * 开启响应式事务
14     * @return
15     */
16    public static Mono<Transaction> beginTransaction() {
17        return Mono.defer(() ->
18            Mono.just(new Transaction(random.nextInt(1000))));
19    }
20    /**
21     * 响应式插入数据
22     * @param rows
23     * @return
24     */
25    public Flux<String> insertRows(Publisher<String> rows) {
26        return Flux.from(rows)
27            .delayElements(Duration.ofMillis(100))
28            .flatMap(row -> {
29                if (random.nextInt(10) < 2) {
30                    return Mono.error(new RuntimeException("出错的条目: "
+ row));
31                } else {
32                    return Mono.just(row);
33                }
34            });
35    }
36    /**
37     * 响应式提交
38     * @return
39     */
40    public Mono<Void> commit() {
41        return Mono.defer(() -> {
42            System.out.println("[开始提交事务: " + id);
43            if (random.nextBoolean()) {
44                return Mono.empty();
45            } else {
46                return Mono.error(new RuntimeException("事务提交异常"));
47            }
48        });
49    }
50    /**
51     * 响应式回滚
52     * @return
53     */
54    public Mono<Void> rollback() {
55        return Mono.defer(() -> {
56            System.out.println("开始回滚事务: " + id);
57            if (random.nextBoolean()) {
```

```

58             return Mono.empty();
59         } else {
60             return Mono.error(new RuntimeException("连接异常"));
61         }
62     });
63 }
64 }
```

现在，可以使用 usingWhen 操作符实现一个更新的事务，代码如下：

```

1 Flux.usingWhen(
2     // 提供资源
3     Transaction.beginTransaction(),
4     new Function<Transaction, Publisher<?>>() {
5         @Override
6         public Publisher<?> apply(Transaction transaction) {
7             return transaction.insertRows(Flux.just("a", "b", "c"));
8         }
9     },
10    // 资源的使用
11    new Function<Transaction, Publisher<?>>() {
12        @Override
13        public Publisher<?> apply(Transaction transaction) {
14            return transaction.commit();
15        }
16    },
17    // 当资源正常使用结束，调用了onComplete，则使用该函数清理资源
18    new BiFunction<Transaction, Throwable, Publisher<?>>() {
19        @Override
20        public Publisher<?> apply(Transaction transaction, Throwable
throwable) {
21            return transaction.rollback();
22        }
23    },
24    // 如果取消资源的使用，则使用该函数清理资源。如果设置为null，则使用资源正常结束时
25    // 的清理函数
26    new Function<Transaction, Publisher<?>>() {
27        @Override
28        public Publisher<?> apply(Transaction transaction) {
29            return null;
30        }
31    }
32 ).subscribe(
33     event -> System.out.println("onNext:" + event),
34     ex -> System.err.println("onError:" + ex.getCause()),
35     () -> System.out.println("处理完成")
36 );
37 Thread.sleep(5000);
```

改为Lambda形式：

```

1 Flux.usingWhen(
2     Transaction.beginTransaction(),
3     transaction -> transaction.insertRows(Flux.just("A", "B", "C")),
4     Transaction::commit,
5     (transaction, throwable) -> transaction.rollback(),
6     Transaction::rollback
7 ).subscribe(
8     event -> System.out.println("onNext:" + event),
9     ex -> System.err.println("onError:" + ex.getCause()),
10    () -> System.out.println("处理完成")
11 );
12 Thread.sleep(5000);

```

使用 `usingWhen` 操作符，不仅可以更容易地以完全响应式的方式管理资源生命周期，还可以轻松实现响应式事务。

因此，与 `using` 操作符相比，`usingWhen` 操作符有巨大改进。

## 8.7 错误处理

`onError` 信号是响应式流规范的一个组成部分，一种将异常传播给可以处理它的用户。但是，如果最终订阅者没有为 `onError` 信号定义处理程序，那么 `onError` 抛异常。

```

1 Flux.from(new Publisher<String>() {
2     @Override
3     public void subscribe(Subscriber<? super String> s) {
4         s.onError(new RuntimeException("手动异常"));
5     }
6 // }).subscribe(System.out::println, System.err::println);
7 }).subscribe(System.out::println);

```

此外，响应式流的语义定义了 `onError` 是一个终止操作，该操作之后响应式流会停止执行。

此时，我们可能采取以下策略中的一种做出不同响应：

1. 为 `subscribe` 操作符中的 `onError` 信号定义处理程序。
2. 通过 `onErrorReturn` 操作符捕获一个错误，并用一个默认静态值或一个从异常中计算出的值替换它。
3. 通过 `onErrorResume` 操作符捕获异常并执行备用工作流。
4. 通过 `onErrorMap` 操作符捕获异常并将其转换为另一个异常来更好地表现当前场景。
5. 定义一个在发生错误时重新执行的响应式工作流。如果源响应序列发出错误信号，那么 `retry` 操作符会重新订阅该序列。

假设有如下推荐服务，该服务是不可靠的：

```

1 private static Random random = new Random();
2
3 public static Flux<String> recommendedBooks(String userId) {
4     return Flux.defer(() -> {
5         if (random.nextInt(10) < 7) {
6             return Flux.<String>error(new RuntimeException("Err"))
7                 // 整体向后推移指定时间，元素发射频率不变
8                 .delaySequence(Duration.ofMillis(100));
9         } else {
10            return Flux.just("Blue Mars", "The Expanse")
11                .delayElements(Duration.ofMillis(50));
12        }
13    }).doOnSubscribe(
14        item -> System.out.println("请求: " + userId)
15    );
16 }

```

客户端代码:

```

1 private static CountDownLatch latch = new CountDownLatch(1);
2
3 public static void main(String[] args) throws InterruptedException {
4     Flux.just("user-0010")
5         .flatMap(user -> recommendedBooks(user)).retry(3)
6         .subscribe(
7             System.out::println,
8             ex -> {
9                 System.err.println(ex);
10                latch.countDown();
11            },
12            () -> {
13                System.out.println("处理完成");
14                latch.countDown();
15            }
16        );
17    latch.await();
18 }
19
20 // =====
21 private static CountDownLatch latch = new CountDownLatch(1);
22
23 public static void main(String[] args) throws InterruptedException {
24     Flux.just("user-0010")
25         .flatMap(user -> recommendedBooks(user))
26         .onErrorResume(event -> Flux.just("java编程指南.pdf"))
27         .subscribe(
28             System.out::println,
29             ex -> {
30                 System.err.println(ex);
31                 latch.countDown();
32            },
33            () -> {
34                System.out.println("处理完成");
35                latch.countDown();
36            }

```

```

37     );
38     latch.await();
39 }
40 // =====
41 private static CountDownLatch latch = new CountDownLatch(1);
42
43 public static void main(String[] args) throws InterruptedException {
44     Flux.just("user-0010")
45         .flatMap(user -> recommendedBooks(user))
46         .onErrorReturn("程序员生存之道.pdf")
47         .subscribe(
48             System.out::println,
49             ex -> {
50                 System.err.println(ex);
51                 latch.countDown();
52             },
53             () -> {
54                 System.out.println("处理完成");
55                 latch.countDown();
56             }
57         );
58     latch.await();
59 }
60 // =====
61 private static CountDownLatch latch = new CountDownLatch(1);
62
63 public static void main(String[] args) throws InterruptedException {
64     Flux.just("user-0010")
65         .flatMap(user -> recommendedBooks(user))
66         .onErrorMap(Throwable -> {
67             if (throwable.getMessage().equals("Err")) {
68                 return new Exception("我的异常替换");
69             }
70             return new Exception("未知异常");
71         })
72         .subscribe(
73             System.out::println,
74             ex -> {
75                 System.err.println(ex);
76                 latch.countDown();
77             },
78             () -> {
79                 System.out.println("处理完成");
80                 latch.countDown();
81             }
82         );
83     latch.await();
84 }

```

总而言之，Project Reactor 提供了丰富的工具集，可以帮助处理异常情况，从而提高应用程序的弹性。

## 8.8 背压处理

尽管响应式流规范要求将背压构建到生产者和消费者之间的通信中，但这仍然可能使消费者溢出。

一些消费者可能无意识地请求无界需求，然后无法处理生成的负载。

另一些消费者则可能对传入消息的速率有严格的限制。例如，数据库客户端每秒不能插入超过 1000 条记录。在这种情况下，事件批处理技术可能有所帮助。

可以通过以下方式配置流以处理背压情况：

1. `onBackpressureBuffer` 操作符会请求无界需求并将返回的元素推送到下游。如果下游消费者无法跟上，那么元素将缓冲在队列中。
2. `onBackpressureDrop` 操作符也请求无界需求 (`Integer.MAX_VALUE`) 并向下游推送数据。如果下游请求数量不足，那么元素会被丢弃。自定义处理程序可以用来处理已丢弃的元素。
3. `onBackpressureLast` 操作符与 `onBackpressureDrop` 的工作方式类似。只是会记住最近收到的元素，并在需求出现时立即将其推向下游。
4. `onBackpressureError` 操作符在尝试向下游推送数据时请求无界需求。如果下游消费者无法跟上，则操作符会引发错误。

管理背压的另一种方法是使用速率限制技术。

`limitRate(n)` 操作符将下游需求拆分为不大于 n 的较小批次。可以保护脆弱的生产者免受来自下游消费者的不合理数据请求的破坏。`limitRate(n)` 操作符会限制来自下游消费者的需求（总请求值）。

如，`limitRequest(100)` 确保不会向生产者请求超过 100 个元素。发送 100 个事件后，操作符成功关闭流。

`onBackpressureBuffer` 操作符：

```
1 CountDownLatch latch = new CountDownLatch(1);
2 Flux.range(1, 1000)
3     .delayElements(Duration.ofMillis(10))
4     .onBackpressureBuffer(900)
5     .delayElements(Duration.ofMillis(100))
6     .subscribe(
7         System.out::println,
8         ex -> {
9             System.out.println(ex);
10            latch.countDown();
11        },
12        () -> {
13            System.out.println("处理完成");
14            latch.countDown();
15        }
16    );
17 latch.await();
18 System.out.println("main结束");
```

`onBackpressureDrop` 操作符：

```

1 CountDownLatch latch = new CountDownLatch(1);
2 Flux.range(1, 1000)
3     .delayElements(Duration.ofMillis(10))
4     .onBackpressureDrop()
5     .delayElements(Duration.ofMillis(100))
6     .subscribe(
7         System.out::println,
8         ex -> {
9             System.out.println(ex);
10            latch.countDown();
11        },
12        () -> {
13            System.out.println("处理完成");
14            latch.countDown();
15        }
16    );
17 latch.await();
18 System.out.println("main结束");

```

onBackpressureLast 操作符:

```

1 CountDownLatch latch = new CountDownLatch(1);
2 Flux.range(1, 1000)
3     .delayElements(Duration.ofMillis(10))
4     .onBackpressureLatest()
5     .delayElements(Duration.ofMillis(100))
6     .subscribe(
7         System.out::println,
8         ex -> {
9             System.out.println(ex);
10            latch.countDown();
11        },
12        () -> {
13            System.out.println("处理完成");
14            latch.countDown();
15        }
16    );
17 latch.await();
18 System.out.println("main结束");

```

onBackpressureError 操作符:

```

1 CountDownLatch latch = new CountDownLatch(1);
2 Flux.range(1, 1000)
3     .delayElements(Duration.ofMillis(10))
4     .onBackpressureError()
5     .delayElements(Duration.ofMillis(100))
6     .subscribe(
7         System.out::println,
8         ex -> {
9             System.out.println(ex);
10            latch.countDown();
11        },

```

```
12     } -> {
13         System.out.println("处理完成");
14         latch.countDown();
15     }
16 );
17 latch.await();
18 System.out.println("main结束");
```

## 8.9 热数据流和冷数据流

冷发布者行为方式：无论订阅者何时出现，都为该订阅者生成所有序列数据，**没有订阅者就不会生成数据**。

以下代码表示冷发布者的行为：

```
1 Flux<String> coldPublisher = Flux.defer(() -> {
2     System.out.println("生成新数据");
3     return Flux.just(UUID.randomUUID().toString());
4 });
5 System.out.println("尚未生成新数据");
6 coldPublisher.subscribe(e -> System.out.println("onNext: " + e));
7 coldPublisher.subscribe(e -> System.out.println("onNext: " + e));
8 System.out.println("为两个订阅者生成了两次数据");
```

每当订阅者出现时都会有一个新序列生成，而这些语义可以代表 HTTP 请求。

热发布者中的数据生成不依赖于订阅者而存在。因此，热发布者可能在第一个订阅者出现之前开始生成元素。

这种语义代表数据广播场景。例如，一旦股价发生变化，热发布者就可以向其订阅者广播有关当前股价的更新。

但是，当订阅者到达时，它仅接收未来的价格更新，而不接受先前价格历史。Reactor 库中的大多数热发布者扩展了 Processor 接口。但是，just 工厂方法会生成一个热发布者，因为它的值只在构建发布者时计算一次，并且在新订阅者到达时不会重新计算。

可以通过将 just 包装在 defer 中来将其转换为冷发行者。这样，即使 just 在初始化时生成值，这种初始化也只会在新订阅出现时发生。后一种行为由 defer 工厂方法决定。

### 多播流元素

通过响应式转换将冷发布者转变为热发布者。

如，一旦所有订阅者都准备好生成数据，希望在几个订阅者之间共享冷处理器的结果。同时，我们又不希望为每个订阅者重新生成数据。Project Reactor为此目的提供了 `ConnectableFlux`。

`ConnectableFlux`，不仅可以生成数据以满足最急迫的需求，还会缓存数据，以便所有其他订阅者可以按照自己的速度处理数据。队列和超时的大小可以通过类的 `publish` 方法和 `replay` 方法进行配置。

此外，`ConnectableFlux` 可以使用 `connect`、`autoConnect(n)`、`refCount(n)` 和 `refCount(int,Duration)` 等方法自动跟踪下游订阅者的数量，以便在达到所需阈值时触发执行操作。

如下案例：

```
1 Flux<Integer> source = Flux.range(0, 3)
2     .doOnSubscribe(
3         s -> System.out.println("对冷发布者的新订阅票据: " + s)
4     );
5 final ConnectableFlux<Integer> conn = source.publish();
6 conn.subscribe(item -> System.out.println("[Subscriber 1] onNext:" + item));
7 conn.subscribe(item -> System.out.println("[Subscriber 2] onNext:" + item));
8 System.out.println("所有订阅者都准备好建立连接了");
9 conn.connect();
```

可以看到，冷发布者收到了订阅，只生成了一次数据项。但是，两个订阅者都收到了整个事件集合。

## 缓存流元素

使用 `ConnectableFlux` 可以轻松实现不同的**数据缓存策略**。但是，Reactor 已经以 `cache` 操作符的形式提供了**用于事件缓存**的 API。

`cache` 操作符使用 `ConnectableFlux`，因此它的主要附加值是它所提供的一个流式而直接的 API。

可以调整缓存所能容纳的数据量以及每个缓存项的到期时间。

示例代码：

```
1 Flux<Integer> source = Flux.range(0, 5)
2     .doOnSubscribe(
3         s -> System.out.println("冷发布者的新订阅票据")
4     );
5 final Flux<Integer> cachedSource = source.cache(Duration.ofMillis(1000));
6 cachedSource.subscribe(item -> System.out.println("[S 1] onNext: " + item));
7 cachedSource.subscribe(item -> System.out.println("[S 2] onNext: " + item));
8 Thread.sleep(1200);
9 cachedSource.subscribe(item -> System.out.println("[S 3] onNext: " + item));
```

前两个订阅者共享第一个订阅的同一份缓存数据。然后，在一定延迟之后，由于第三个订阅者无法获取缓存数据，因此一个针对冷发布者的新订阅被触发了。最后，即使该数据不来自缓存，第三个订阅者也接收到了所需的数据。

## 共享流元素

我们可以使用 `ConnectableFlux` 向几个订阅者多播事件。但是需要等待订阅者出现才能开始处理。

`share` 操作符可以将冷发布者转变为热发布者。该操作符会为每个新订阅者传播订阅者尚未错过的事件。

示例代码：

```
1 Flux<Integer> source = Flux.range(0, 5)
2     .delayElements(Duration.ofMillis(100))
3     .doOnSubscribe(s -> System.out.println("冷发布者新的订阅票据"));
4 Flux<Integer> cachedSource = source.share();
5 cachedSource.subscribe(item -> System.out.println("[S 1] onNext: " + item));
6 Thread.sleep(400);
7 cachedSource.subscribe(item -> System.out.println("[S 2] onNext: " + item));
8 Thread.sleep(1000);
```

在前面的代码中，共享了一个冷发布流，该流以每 100 毫秒为间隔生成事件。然后，经过一些延迟，一些订阅者订阅了共享发布者。

第一个订阅者从第一个事件开始接收，而第二个订阅者错过了在其出现之前所产生的事件（S2 仅接收到事件 3 和事件 4）。

## 8.10 处理时间

响应式编程是异步的，因此它本身就假定存在时序。

基于 Project Reactor，可以使用 `interval` 操作符生成基于一定持续时间的事件，使用 `delayElements` 操作符生成延迟元素，并使用 `delaySequence` 操作符延迟所有信号。

Reactor 的 API 使你能对一些与时间相关的事件做出响应，`timestamp` 操作符用于输出元素的时间戳，`timeout` 操作符用于指定消息时间间隔的大小。与 `timestamp` 类似，`elapsed` 操作符测量与上一个事件的时间间隔。

`interval` 操作符：

```
1 Flux.interval(Duration.ofMillis(100))
```

```

2     .subscribe(
3         item -> {
4             System.out.println(Thread.currentThread().getName() + " "
5                 + item);
6         }
7     );
8 
9     Flux.interval(Duration.ofSeconds(3), Duration.ofMillis(100))
10    .subscribe(System.out::println);
11 
12    Flux.interval(Duration.ofMillis(100), Schedulers.parallel())
13        .subscribe(
14            item -> {
15                System.out.println(Thread.currentThread().getName() + " "
16                    + item);
17            }
18        );
19 
20    Flux.interval(Duration.ofMillis(100), Schedulers.newSingle("count"))
21        .subscribe(
22            item -> {
23                System.out.println(Thread.currentThread().getName() + " "
24                    + item);
25            }
26        );
27 
28    Thread.sleep(5000);
29    System.out.println("结束");

```

`delayElements` 操作符:

```

1 Flux.range(1, 1000)
2     .delayElements(Duration.ofSeconds(1))
3     .subscribe(item -> {
4         System.out.println(Thread.currentThread().getName() + " -- " +
5             item);
6     });
7 
8 Thread.sleep(5000);
9 System.out.println("结束");

```

`delaySequence` 操作符:

```

1 Flux.range(1, 1000)
2     .delaySequence(Duration.ofSeconds(3))
3     .subscribe(item -> {
4         System.out.println(Thread.currentThread().getName() + " -- " +
5             item);
6     });
7 Thread.sleep(5000);
8 System.out.println("结束");

```

`timeout` 操作符:

```
1 Random random = new Random();
2 CountDownLatch latch = new CountDownLatch(1);
3 Flux.interval(Duration.ofMillis(300))
4     .timeout(Duration.ofMillis(random.nextInt(20) + 290))
5     .subscribe(
6         System.out::println,
7         ex -> {
8             System.err.println(ex);
9             latch.countDown();
10        }
11    );
12 latch.await(10, TimeUnit.SECONDS);
```

`timestamp` 操作符:

```
1 Flux.interval(Duration.ofMillis(300))
2     .timestamp()
3     .subscribe(item -> {
4         final Long timestamp = item.getT1();
5         final Long element = item.getT2();
6         String result = element + "的时间戳: " + timestamp;
7         System.out.println(result);
8     });
9 Thread.sleep(5000);
```

`elapsed` 操作符:

```
1 Flux.interval(Duration.ofMillis(300))
2     .elapsed()
3     .subscribe(item -> {
4         final Long interval = item.getT1();
5         final Long element = item.getT2();
6         String result = element + " 与上一个元素的时间间隔: " + interval +
7             "ms";
8         System.out.println(result);
9     });
10 Thread.sleep(5000);
```

从前面的输出中可以明显看出，事件并未恰好在 300 毫秒的时间间隔内到达。

发生这种情况是因为 Reactor 使用 Java 的 ScheduledExecutorService 进行调度事件，而这些事件本身并不能保证精确的延迟。

因此，应该注意不要在 Reactor 库中要求太精确的时间（实时）间隔。

## 8.11 组合和转换响应式流

当我们构建复杂的响应式工作流时，通常需要在几个不同的地方使用相同的操作符序列。

`transform` 操作符，可以将这些常见的部分提取到单独的对象中，并在需要时重用它们。

`transform` 操作符，可以增强流结构本身。

示例代码：

```
1 Function<Flux<String>, Flux<String>> logUserInfo = stream ->
2     stream.index().doOnNext(
3         tp -> System.out.println("[" + tp.getT1() + "] User: " +
4             tp.getT2())
5         ).map(Tuple2::getT2);
6 Flux.range(1000, 3)
7     .map(i -> "user-" + i)
8     .transform(logUserInfo)
9     .subscribe(e -> System.out.println("onNext: " + e));
```

`transform` 操作符仅在流生命周期的**组装阶段**更新一次流行为，可以在响应式应用程序中实现代码重用。

## 8.12 处理器

响应式流规范定义了 Processor 接口。Processor 既是 Publisher 也是 Subscriber。

因此，既可以订阅 Processor 实例，也可以手动向它发送信号（onNext、onError 和 onComplete）。

Reactor 的作者建议忽略处理器，因为它们很难使用并且容易出错。

在大多数情况下，处理器可以被操作符的组合所取代。另外，生成器工厂方法（push、create 和 generate）可能更适合适配外部 API。

Reactor 提出以下几种处理器：

- **Direct** 处理器只能通过操作处理器的接收器来推送因用户手动操作而产生的数据。
  - `DirectProcessor` 和 `UnicastProcessor` 是这组处理器的代表。
  - `DirectProcessor` 不处理背压，可用于向多个订阅者发布事件。
  - `UnicastProcessor` 使用内部队列处理背压，最多只能为一个 `Subscriber` 服务。
- **Synchronous** 处理器
  - `EmitterProcessor` 和 `ReplayProcessor` 可以同时通过手动方式和订阅上游 Publisher 的方式来推送数据。
  - `EmitterProcessor` 可以为多个订阅者提供服务并满足它们的需求，但仅能以同步方式消费由单一 Publisher 产生的数据。

- `ReplayProcessor` 的行为类似于 `EmitterProcessor`，但是它能使用几种策略来缓存传入的数据。
- **Asynchronous 处理器**
  - `workQueueProcessor` 和 `TopicProcessor` 可以推送从多个上游发布者处获得的下游数据。
  - 为了处理多个上游发布者，这些处理器使用 `RingBuffer` 数据结构。这些处理器具有专用的构建器 API，因为配置选项的数量使它们很难初始化。
  - `TopicProcessor` 兼容响应式流，并可以为每个下游 `Subscriber` 关联一个 `Thread` 来处理交互。它可以服务的下游订阅者数量有限。
  - `workQueueProcessor` 具有与 `TopicProcessor` 类似的特性。但是，它放宽了一些响应式流要求，这使它在运行时所使用的资源更少。

## 8.13 测试和调试Project Reactor

Reactor 库附带了一个通用的测试框架。`io.projectreactor:reactor-test` 库提供了测试 Project Reactor 所实现的响应式工作流所需的所有必要工具。

虽然响应式代码不容易调试，但是 Project Reactor 提供了能在需要时简化调试过程的技术。与任何基于回调的框架一样，Project Reactor 中的栈跟踪信息量不大。它们没有在代码中给出发生异常情况的确切位置。Reactor 库具有面向调试的组装时检测功能，可以使用以下代码激活：

```
1 | Hooks.onOperatorDebug();
```

示例程序：

```
1 | Hooks.onOperatorDebug();
2 |
3 | Flux.range(1, 10)
4 |     .map(item -> "item-" + item)
5 |     .concatWith(Flux.error(new RuntimeException("手动异常")))
6 |     .subscribe(System.out::println);
```

启用后，此功能开始收集将要组装的所有流的栈跟踪，稍后此信息可以基于组装信息扩展栈跟踪信息，从而帮助我们更快地发现问题。但是，创建栈跟踪的过程成本很高。因此，作为最后的手段，它应该只以受控的方式进行激活。

此外，Project Reactor 的 `Flux` 和 `Mono` 类型提供了一个被称为 `log` 的便捷方法。它能记录使用操作符的所有信号。即使在调试情况下，许多方法的自定义实现也可以提供足够的自由度来跟踪所需的数据。

如下代码：

```
1 Flux.range(1, 10)
2     .map(item -> "item-" + item)
3     .concatWith(Flux.error(new RuntimeException("手动异常")))
4     .log()
5     .subscribe(System.out::println);
```

## 8.14 Reactor插件

Project Reactor 是一个通用且功能丰富的库。但是，它无法容纳所有有用的响应式工具。因此，有一些项目在一些领域扩展了 Reactor 的功能。官方的 Reactor 插件项目为 Reactor 项目提供了几个模块。

`reactor-adapter` 模块为 RxJava 2 响应式类型和调度程序提供桥接。此外，该模块还能与 Akka 进行集成。

```
1 <dependency>
2   <groupId>io.projectreactor.addons</groupId>
3   <artifactId>reactor-adapter</artifactId>
4   <version>3.4.0</version>
5 </dependency>
```

`reactor-logback` 模块提供高速异步日志记录功能。它以 Logback 的 AsyncAppender 和 LMAX Disruptor 的 RingBuffer 为基础，其中后者通过 Reactor 的 Processor 实现。

```
1 <dependency>
2   <groupId>io.projectreactor.addons</groupId>
3   <artifactId>reactor-logback</artifactId>
4   <version>3.2.6.RELEASE</version>
5 </dependency>
```

`reactor-extra` 模块包含用于高级需求的其他实用程序。例如，该模块包含 TupleUtils 类，该类简化了编写 Tuple 类的代码。此外，该模块具有 MathFlux 类，可以从数字源中计算最小值和最大值，并对它们求和或取平均。`ForkJoinPoolscheduler` 类使 Java 的 `ForkJoinPool` 适配 Reactor 的 Scheduler。

可以使用以下导入方式将模块添加到项目中：

```
1 <dependency>
2   <groupId>io.projectreactor.addons</groupId>
3   <artifactId>reactor-extra</artifactId>
4   <version>3.4.0</version>
5 </dependency>
```

此外，Project Reactor 生态系统还为流行的异步框架和消息代理服务器提供了响应式驱动程序。

Reactor RabbitMQ 模块使用熟悉的 Reactor API 为 RabbitMQ 提供了一个响应式 Java 客户端。

该模块不仅提供具有背压支持的异步非阻塞消息传递，还使应用程序能够通过使用 Flux 和 Mono 类型将 RabbitMQ 用作消息总线。

```
1 <dependency>
2   <groupId>io.projectreactor.rabbitmq</groupId>
3   <artifactId>reactor-rabbitmq</artifactId>
4   <version>1.5.0</version>
5 </dependency>
```

Reactor Kafka 模块为 Kafka 消息代理服务器提供了类似的功能。

```
1 <dependency>
2   <groupId>io.projectreactor.kafka</groupId>
3   <artifactId>reactor-kafka</artifactId>
4   <version>1.3.1</version>
5 </dependency>
```

另一个广受欢迎的 Reactor 扩展被称为 Reactor Netty。它使用 Reactor 的响应式类型来适配 Netty 的 TCP/HTTP/UDP 客户端和服务器。Spring WebFlux 模块在内部使用 Reactor Netty 来构建非阻塞式 Web 应用程序。

```
1 <dependency>
2   <groupId>io.projectreactor.netty</groupId>
3   <artifactId>reactor-netty</artifactId>
4   <version>1.0.2</version>
5 </dependency>
```

## 9 Project Reactor 高级

### 9.1 响应式流的生命周期

要理解多线程的工作原理以及 Reactor 中实现的各种内部优化，首先必须了解 Reactor 中响应式类型的生命周期。

## 组装时

流生命周期的第一部分是组装时 (assembly-time) 。

Reactor 提供了一个流式 API，用于构建复杂的元素处理流程。

Reactor 的 API 看起来像是一个组合流程中所选择的操作符的建造器。

建造器模式不仅是可变的，还假设像 build 这样的终端操作会执行另一个对象的构建。

Reactor API 提供了不变性，每个被使用的操作符都会生成一个新对象。

在响应式库中，构建执行流程的过程被称为组装 (assembling) 。

如果从头开发，不考虑Reactor API，流程组装的可能表现形式：

```
1 | Flux<Integer> sourceFlux = new FluxArray(1, 20, 300, 4000);
2 | Flux<String> mapFlux = new MapFlux(sourceFlux, String::valueOf);
3 | Flux<String> filterFlux = new FluxFilter(mapFlux, s -> s.length() > 1);
4 | // ...
```

在底层，Flux 对象是相互组合的。在组装过程之后，就获得了一个 Publishers 链，每个新的 Publisher 包装了前一个。

以下伪代码演示了这一点：

```
1 | FluxFilter(
2 |   FluxMap(
3 |     FluxArray(1, 2, 3, 40, 500, 6000)
4 |   )
5 | )
```

在流生命周期中，该阶段起着重要作用，因为在流组装期间，可以通过检查流的类型来一个接一个地替换操作符。

例如，`concatwith -> concatwith -> concatwith` 操作符序列可以被很容易地被压缩到一个串联结构中。

以下代码展示了它在 Reactor 中的实现过程：

```
1 public final Flux<T> concatWith(Publisher<? extends T> other) {  
2     if (this instanceof FluxConcatArray) {  
3         @SuppressWarnings({"unchecked"})  
4         FluxConcatArray<T> fluxConcatArray = (FluxConcatArray<T>)this;  
5         return fluxConcatArray.concatAdditionalSourceLast(other);  
6     }  
7     return concat(this, other);  
8 }
```

如果当前 Flux 是 FluxConcatArray 实例，则不创建

FluxConcatArray(FluxConcatArray(FluxA, FluxB), FluxC)，而是创建一个  
FluxConcatArray(FluxA, FluxB, FluxC)，并以这种方式改善整体流性能。

此外，在组装时，可以在组装过程中为流提供一些 Hooks，并启用一些额外的日志记录、跟踪、度量收集，以及其他在调试或流监控期间可能有用的重要补充。

在该阶段，可以操作流的构造过程并应用不同的技术来优化、监控或更好地进行流调试，这是构建响应式流必不可少的部分。

## 订阅时

流执行生命周期的第二个重要阶段是订阅时 (subscription-time)。

当调用指定的Publisher的subscribe方法时，就会发生订阅。

如下代码所示：

```
1 // ...  
2 filteredFlux.subscribe(...);
```

为了构建执行流程，对 Publishers 进行相互传递，因而产生了 Publishers 链。一旦调用了顶层包装器的subscribe方法，就开始了该链的订阅过程。

以下伪代码展示了一个 Subscriber 在订阅时如何通过 Subscriber 链进行传播：

```
1 filterFlux.subscribe(Subscriber) {  
2     mapFlux.subscribe(new FilterSubscriber(Subscriber)) {  
3         arrayFlux.subscribe(new MapSubscriber(FilterSubscriber(Subscriber)))  
4     }  
5     // 在这里开始推送真正的元素  
6 }  
7 }
```

最后，将获取如下相互包装的订阅者序列：

```
1 | ArraySubscriber(  
2 |     MapSubscriber(  
3 |         FilterSubscriber(  
4 |             Subscriber  
5 |         )  
6 |     )  
7 | )
```

订阅时阶段的重要性在于：

1. 在该阶段中，可以执行与组装时阶段相同的优化。
2. 其次，在 Reactor 中启用多线程的一些操作符能够更改订阅所发生的工作单元。

## 运行时

流执行的最后一步是运行时 (runtime) 阶段。

在该阶段，在 Publisher 和 Subscriber 之间进行实际信号交换。

响应式流规范规定，Publisher 和 Subscriber 交换的前两个信号是 onSubscribe 信号和 request 信号。

onSubscribe 方法由位于顶端的数据源调用，在本例中即 ArrayPublisher。这会将它的 Subscription 传递给给定的 Subscriber。

array -> map -> filter

描述通过Subscribers 传递 Subscription 过程的伪代码如下所示：

```

1 // 首先是Array调用MapSubscriber的onSubscribe方法，传参ArraySubscription对象
2 MapSubscriber(FilterSubscriber(Subscriber)).onSubscribe(new
3     ArraySubscription() {
4         // Map调用FilterSubscriber的onSubscribe方法，传参MapSubscription对象
5         FilterSubscriber(Subscriber).onSubscribe(MapSubscription(ArraySubscription(..))) {
6             // Filter调用真正的Subscriber的onSubscribe方法，传参
7             FilterSubscription对象
8             Subscriber.onSubscribe(FilterSubscription(MapSubscription(ArraySubscription(...)))) {
9                 // 真正的Subscriber对onSubscribe的处理逻辑。。
10            }
11        }
12    }

```

一旦 Subscription 完全通过 Subscriber 链，链中的每个 Subscriber 会将 Subscription 包装为特定表示。

如下面的代码所示：

```

1 FilterSubscription(
2     MapSubscription(
3         ArraySubscription()
4     )
5 )

```

最终，最后一个 Subscriber 接收 Subscription 链。并且，为了开始接收元素，应该调用 Subscription #request 方法。该方法会启动元素的发送。

如下代码所示：

```

1 // 真正的Subscriber调用FilterSubscription的request方法，参数为请求的元素个数
2 FilterSubscription(MapSubscription(ArraySubscription(..))).request(10) {
3     // Filter调用MapSubscription的request方法，传参为请求的元素个数
4     MapSubscription(ArraySubscription(..)).request(10) {
5         // Map调用ArraySubscription的request方法，传参请求的元素个数
6         ArraySubscription(..).request(10) {
7             // Array开始发送数据...
8         }
9     }
10 }

```

一旦所有订阅者传递了请求内容，并且 ArraySubscription 也接收到这些请求，ArrayFlux 就可以开始向 MapSubscriber(FilterSubscriber(Subscriber)) 链发送元素。

下面的伪代码，描述了通过所有 Subscriber 发送元素的过程：

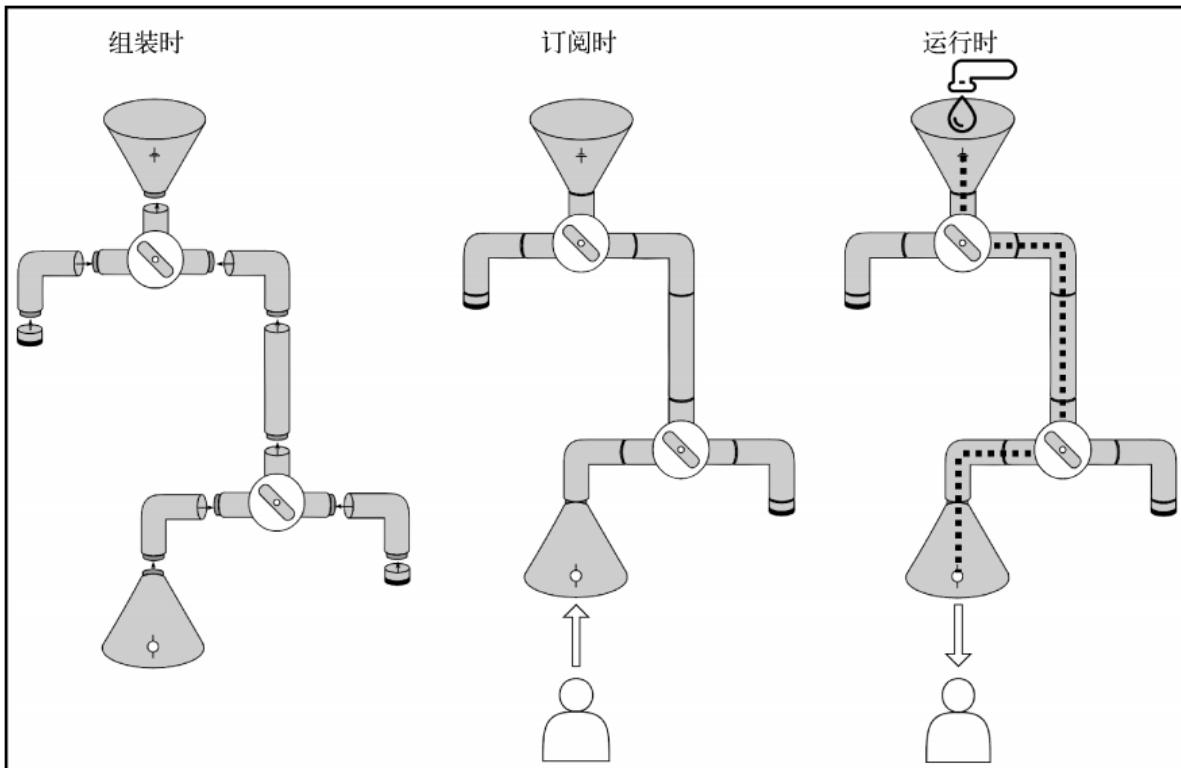
```
1 // Array接到请求之后
2 ArraySubscription.request(10) {
3     // 调用MapSubscriber的onNext方法，挨个儿传递元素
4     MapSubscriber(FilterSubscriber(Subscriber)).onNext(1) {
5         // Map接收到元素后，调用FilterSubscriber的onNext方法，传递处理完的元素
6         FilterSubscriber(Subscriber).onNext(1) {
7             // 最终Subscriber对传递来的元素进行处理
8             // Subscriber再次请求一个元素
9             MapSubscription(ArraySubscription(...)).request(1) {...}
10        }
11    }
12
13    // ...
14
15    // 调用MapSubscriber的onNext方法，传递第10个元素
16    MapSubscriber(FilterSubscriber(Subscriber)).onNext(10) {
17        // Map调用FilterSubscriber的onNext方法，传递第10个元素
18        FilterSubscriber(Subscriber).onNext(10) {
19            // Filter回调Subscriber的onNext方法，传递第10个元素
20            Subscriber.onNext(10) {
21                // 最终Subscriber的onNext方法中执行元素的处理逻辑
22            }
23        }
24    }
25 }
```

在运行时，数据源中的元素通过 Subscriber 链，并在每个阶段执行不同的功能。

在运行时我们可以应用优化，减少信号交换量。

如，可以减少 Subscription#request 调用的次数，从而提高流的性能。

下图总结了流的生命周期以及每个阶段的执行情况。



## 9.2 Reactor中的线程调度模型

### publishOn 操作符

publishOn 操作符能将部分运行时操作的执行移动到指定的**工作单元**。

为了指定应该在运行时处理元素的**工作单元**, Reactor 为此引入了一个特定的抽象, 叫作 Scheduler。Scheduler 是一个接口, 代表 Project Reactor 中的一个**工作单元**或**工作单元池**。

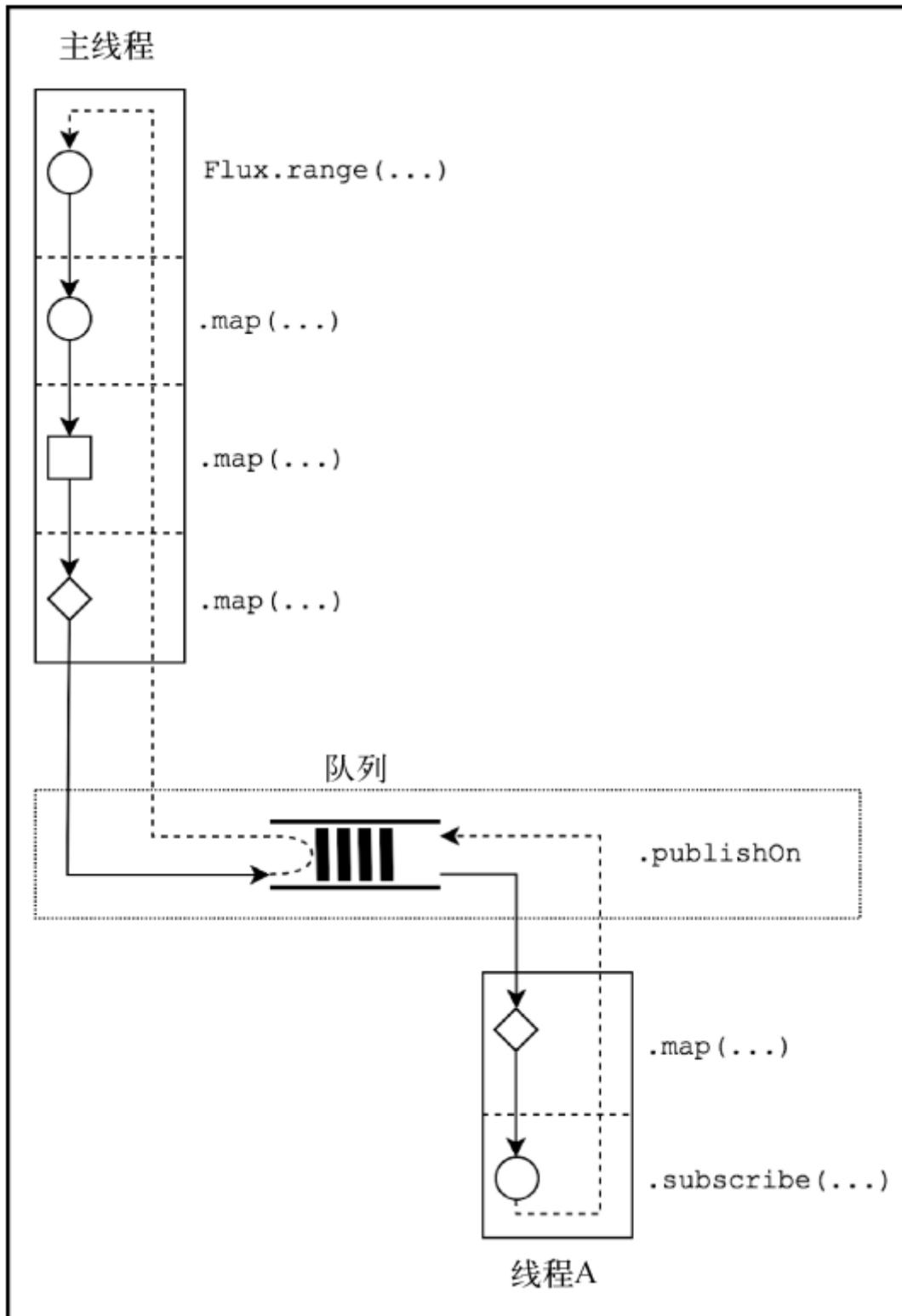
考虑以下代码:

```

1 Scheduler scheduler = ...;
2 Flux.range(0, 100)
3     .map(String::valueOf)
4     .filter(s -> s.length() > 1)
5     .publishOn(scheduler)
6     .map(this::calculateHash)
7     .map(this::doBusinessLogic)
8     .subscribe();

```

publishOn 操作符之后的执行位于不同的 Scheduler 工作单元上。这意味着对散列的计算发生在 Thread A 上，因此 calculateHash 和 doBusinessLogic 在与 Thread Main 不同的工作单元上执行。如果从执行模型角度来看 publishOn 操作符，可以得到下图所示流程。



publishOn 操作符的重点是**运行时执行**。在底层，publishOn 操作符会保留一个队列，并为该队列提供新元素，以便专用工作单元消费消息并逐个处理它们。

该示例表明工作正在单独的 Thread 上运行，因此其执行被一个异步边界所分割。所以，现在有两部分独立处理的流程。

注意：响应式流中的所有元素都是**逐个处理的**（而不是同时处理的），因此可以始终为所有事件**定义严格的顺序**。此属性也被称为**串行化**（serializability）。

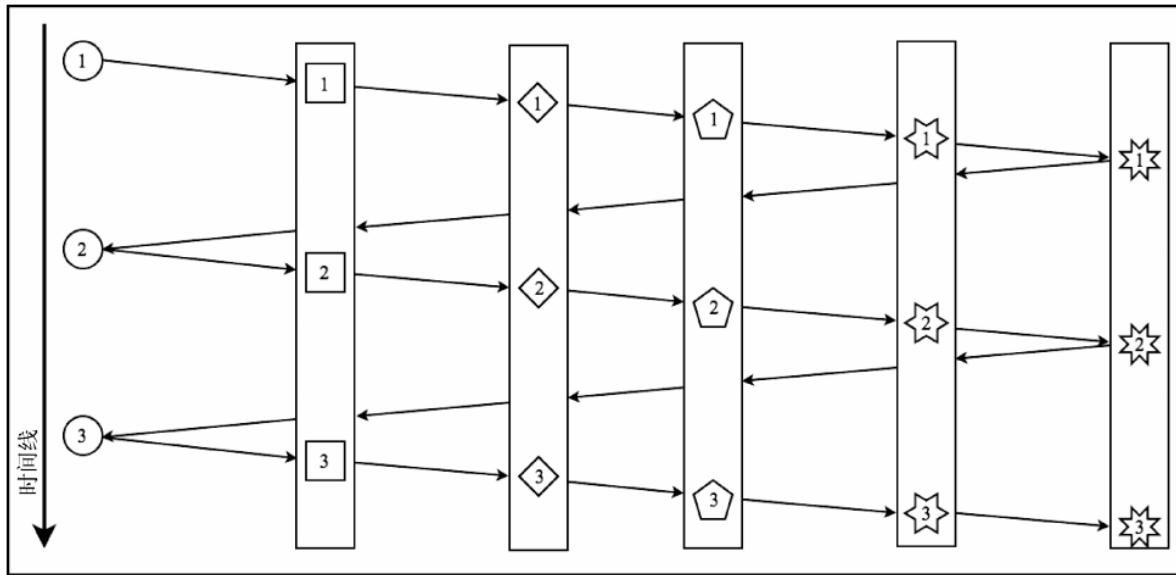
即，元素一旦进入 publishOn，就将被放入队列，并且一旦轮到它，它就将被移出队列进行处理。

注意，由于只有一个工作单元专门负责处理队列，因而元素的顺序始终是可预测的。

## 使用publishOn操作符实现并行化

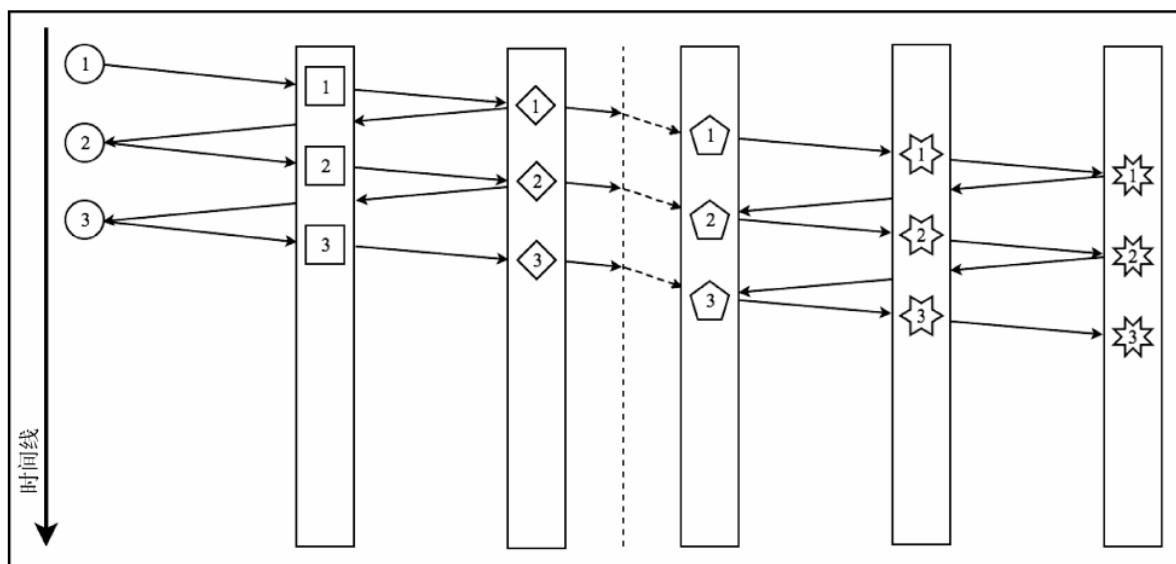
Project Reactor 提供的响应式编程范例可以使用 publishOn 操作符对处理流进行细粒度伸缩和并行化等处理。

考虑下图的过程：



如上图，有一个处理流程，其中包含 3 个元素。由于流中元素的同步处理特性，必须在所有转换阶段中逐个移动元素。但是，为了开始处理下一个元素，必须完全处理完前一个元素。

相反，如果在这个流程中放置一个 publishOn，就可能加快处理速度。如下图：



如上图，只要保持元素的处理时间相同，并在处理阶段之间提供异步边界（由publishOn 操作符表示），就可以实现并行处理。现在，处理流程的左侧不需要等待右侧处理完成。相反，它们可以独立工作，以正确地实现并行处理。

## subscribeOn操作符

Reactor 中多线程的另一个要点是名为 subscribeOn 的操作符。与 publishOn 相比，subscribeOn 使你能更改正在运行的订阅链的工作单元。

当从函数的执行过程中创建流的数据源时，此操作符很有用。

通常，此类执行在订阅时进行，它会调用一个函数，该函数会提供执行 subscribe 方法的数据源。

如下代码：

```
1 ObjectMapper objectMapper = ...
2
3 String json = "{ \"color\" : \"black\", \"type\" : \"BMW\" }";
4
5 Mono.fromCallable(() -> objectMapper.readValue(json, car.class))
6 // ...
```

这里，`Mono.fromCallable` 从 `callable<T>` 创建 `Mono`，并将其评估结果提供给每个 `Subscriber`。`Callable` 实例在调用 `subscribe` 方法时执行。

因此 `Mono.fromCallable` 在底层执行以下操作：

```
1 public void subscribe(Subscriber actual) {
2     // 准备Subscription对象
3     Subscription subscription = ...
4     try {
5         // 调用call方法，获取数据元素
6         T t = callable.call();
7         if (t == null) {
8             // 如果没有数据，直接调用onComplete方法，结束响应式流
9             subscription.onComplete();
10        } else {
11            //如果有数据，则调用订阅票据的onNext方法传递元素。
12            subscription.onNext(t);
13            //由于是Mono，传递一个元素之后，调用onComplete方法，结束响应式流
14            subscription.onComplete();
15        }
16    } catch (Throwable e) {
17        actual.onError(
18            Operators.onOperatorError(e, actual.currentContext())
19        );
20    }
21 }
```

`Callable` 的执行发生在 `subscribe` 方法中。

这意味着可以使用 `publishOn` 来更改执行 `Callable` 的工作单元。

可以使用subscribeOn 指定进行订阅的工作单元。

以下示例展示了具体方法：

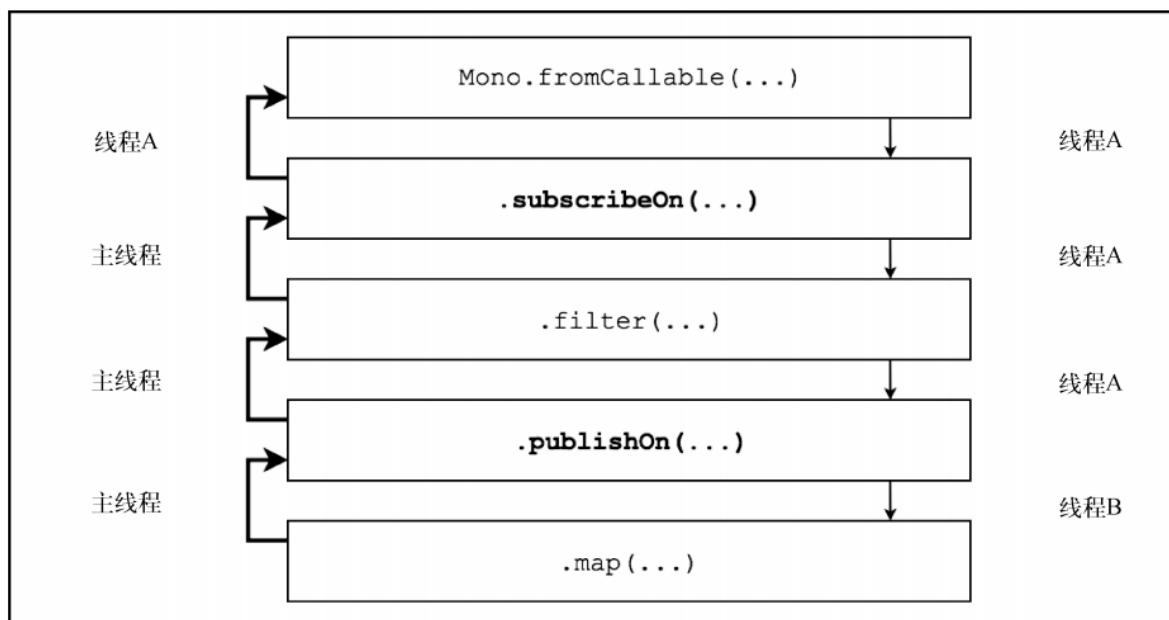
```
1 Scheduler scheduler = ...;
2
3 Mono.fromCallable(...)
4     .subscribeOn(scheduler)
5     .subscribe();
```

前面的示例展示了在单独的工作单元上执行给定的 `Mono.fromCallable` 的方法。

在底层，`subscribeOn` 将 ~~父 Publisher~~ 的订阅放在 `Runnable` 中执行（`Runnable` 是指定 `Scheduler` 的调度程序）。

如果比较 `subscribeOn` 和 `publishOn` 的执行模型，如下图：

左侧表示订阅，右侧表示消费。



由上图可知，`subscribeOn` 可以部分地指定运行时工作单元以及订阅时工作单元。

发生这种情况是因为除了对 `subscribe` 方法执行的调度，`subscribeOn` 还会把每次调用调度到 `Subscription.request()` 方法，以使调用发生在 `Scheduler` 实例指定的工作单元上。

根据响应式流规范，`Publisher` 可以开始在调用者 `Thread` 上发送数据，因此后续的 `Subscriber.onNext()` 将在与初始的 `Subscription.request()` 相同的 `Thread` 上被调用。

`publishOn` 只能为下游指定执行行为，而不能影响上游执行。

## 并行操作符

除了一些重要操作符（用于管理想要处理的执行流某些部分的线程），Reactor 还提供了一种熟悉的并行工作技术。为此，Reactor 有一个名为 `parallel` 的操作符，它能将流分割为并行子流并均衡它们之间的元素。

以下是此操作符的使用示例：

```
1 Random random = new Random();
2 CountDownLatch latch = new CountDownLatch(1);
3 Flux.range(1, 10000)
4     .parallel() // 轮询方式将元素交给各个处理器核心来处理，这里只是准备阶段，需要调用runOn真正调度执行。
5     .doOnNext(item -> {
6         System.out.println("parallel:" +
7             Thread.currentThread().getName());
7     })
8     .runOn(Schedulers.parallel()) // 每个处理器核心一个执行单元。4c8t，线程名称：runOn:parallel-{1-8}
9     .doOnNext(item -> {
10        System.out.println("runOn:" + Thread.currentThread().getName());
11    })
12     .map(num -> num + random.nextInt(10000)) // 4c8t，线程名称：
13     runOn:parallel-{1-8}
14     .doOnNext(item -> {
15         System.out.println("map:" + Thread.currentThread().getName());
16     })
17     .filter(num -> num % 2 == 0) // 某些核心执行单元的数字被过滤掉了
18     .doOnNext(item -> {
19         System.out.println("filter:" +
20             Thread.currentThread().getName());
21     })
22     .subscribe( // 执行在上述各自线程中
23         item -> System.out.println(Thread.currentThread().getName()
24             + ":" + item),
25         ex -> System.err.println(ex),
26         () -> latch.countDown()
27     );
28 latch.await();
```

`parallel()` 是 Flux API 的一部分。通过应用 `parallel` 操作符，开始在不同类型的 Flux 上执行操作，该 Flux 被称为 `ParallelFlux`。

`ParallelFlux` 是一组 Flux 的抽象，其中源 Flux 中的元素是**均衡的**。然后，通过应用 `runOn` 操作符，可以将 `publishOn` 应用于内部 Flux，并分配与元素（正在不同工作单元之间进行处理）相关的工作。

## 调度器

调度器是一个接口，具有两个核心方法，即 Scheduler.schedule 和 Scheduler.createWorker。

第一个方法可以调度 Runnable 任务；第二个方法不仅为我们提供了 Worker 接口的专用实例，还可以以相同的方式调度 Runnable 任务。Scheduler 接口和 Worker 接口之间的核心区别在于 **Scheduler 接口表示工作单元池，而 Worker 是 Thread 或资源的专用抽象。**

默认情况下，Reactor 提供 3 个核心调度程序接口实现。

1. **SingleScheduler** 能为一个专用工作单元安排所有可能的任务。它具有时间性，因此可以延迟安排定期事件。此调度程序可以使用 **Scheduler.single()** 调用进行引用。
2. **ParallelScheduler** 适用于固定大小的工作单元池（**默认情况下，其大小受 CPU 内核数限制**）。适合 **CPU密集型任务**。此外，默认情况下，它也处理与时间相关的调度事件，例如 Flux.interval(Duration.ofSeconds(1))。此调度程序可以使用 **Scheduler.parallel()** 调用进行引用。
3. **ElasticScheduler** 可以动态创建工作单元并缓存线程池。由于其所创建的线程池没有最大数量限制，因此此调度程序**非常适用于 I/O 密集型操作**的调度。此调度程序可以使用 **Scheduler.elastic()** 调用进行引用。

此外，还可以实现具有所期望特性的 Scheduler。

## 响应式上下文

Reactor 附带的另一个关键功能是 Context。Context 是沿数据流传递的接口。

Context 接口的核心思想是提供对某些上下文信息的访问，因为这些信息可能在稍后的运行时阶段有用。

既然已经有了可以做同样工作的 ThreadLocal，为什么还需要这个功能？例如，许多框架使用 ThreadLocal 来沿用户请求执行传递 SecurityContext，以便在任何处理点访问授权用户。

只是，这种概念只有在进行单线程处理时才能正常工作，因为执行是依附于同一个 Thread。如果开始在异步处理中使用该概念，那么 ThreadLocal 将会非常快速地释放。

例如，如果执行如下操作，将丢失可用的 ThreadLocal：

```
1 class ThreadLocalProblemShowcase {
2     public static void main(String[] args) {
3         ThreadLocal<Map<Object, Object>> threadLocal = new ThreadLocal<>();
4         threadLocal.set(new HashMap<>());
5         Flux
6             .range(0, 10)
7             .doOnNext(k -> threadLocal.get() // 将数据放到ThreadLocal中。
8                     .put(k, new Random(k).nextGaussian()))
9     )
10     .publishOn(Schedulers.parallel()) // 调度线程执行
```

```
11         .map(k -> threadLocal.get().get(k)) // 线程已经改变，无法访问到先前的  
12     数据  
13     }  
14 }
```

在多线程环境中使用 ThreadLocal 是非常危险的，并且可能导致意外行为。尽管 Java API 能将 ThreadLocal 数据从一个 Thread 传输到另一个 Thread，但它并不保证传输的完全一致性。

Reactor Context 通过以下方式解决了这个问题：

```
1 Flux.range(0, 10)  
2     .flatMap(  
3         k -> Mono.subscriberContext()  
4             .doOnNext(  
5                 context -> {  
6                     Map<Object, Object> map = context.get("randoms");  
7                     map.put(k, new Random(k).nextGaussian());  
8                 }  
9             ).thenReturn(k)  
10        )  
11        .publishOn(Schedulers.parallel())  
12        .flatMap(  
13            k -> Mono.subscriberContext()  
14                .map(  
15                    context -> {  
16                        Map<Object, Object> map = context.get("randoms");  
17                        return map.get(k);  
18                    }  
19                )  
20        )  
21        .subscriberContext(  
22            context -> context.put("randoms", new HashMap())  
23        )  
24        .blockLast();
```

```
806 */  
807 @Deprecated  
808 public static Mono<Context> subscriberContext() {  
809     return onAssembly(MonoCurrentContext.INSTANCE);  
810 }
```

```

26
27     @Deprecated
28     final class MonoCurrentContext extends Mono<Context>
29         implements Fuseable, Scannable {
30
31     @
32
33     /unchecked/
34     public void subscribe(CoreSubscriber<? super Context> actual) {
35         Context ctx = actual.currentContext();
36         actual.onSubscribe(Operators.scalarSubscription(actual, ctx));
37     }
38
39     @Override
40     public Object scanUnsafe(Attr key) {
41         if (key == Attr.RUN_STYLE) return Attr.RunStyle.SYNC;
42         return null;
43     }
44 }

```

- Reactor 使用静态操作符 subscriberContext 提供对当前流中 Context 实例的访问。一旦获取了 Context，就可以访问 Map 并将生成的值放在那里。最后，返回 flatMap 的初始参数。
- 在切换 Thread 后再次访问 Reactor 的 Context。尽管此示例与使用 ThreadLocal 的前一个示例相同，但将成功获取存储的映射并获得生成的随机高斯双精度数。
- 最后，在这里，为了生成 randoms 键（该键返回一个 Map），我们在上游填充一个新的 Context 实例，该实例包含所需键对应的 Map。

Context 可以通过无参数的 `Mono.subscriberContext` 操作符进行访问，并且可以通过单参数 `subscriberContext(Context)` 操作符提供给流。

既然 Context 接口具有与 Map 接口类似的方法，那为什么需要使用 Map 来传输数据？

**Context 是不可变对象**，一旦向它添加新元素，就实现了 Context 的新实例。这样的设计决策有利于多线程访问模型。

这意味着，这是向流提供 Context 并动态提供某些数据的唯一方法，这些数据将在组装时或订阅时的整个运行执行期间可用。如果在组装时提供了 Context，那么所有订阅者将共享相同的静态上下文，但这在每个 Subscriber（可能代表用户连接）具有其自身的 Context 的情况下可能没有用。因此，可以向每个 Subscriber 提供其自身上下文的唯一生命周期时段是 **订阅时阶段**。

在订阅时，Subscriber 通过 Publisher 链从流的底部上升到顶部，并在每个阶段中形成包装到本地 Subscriber 的表现形式，从而引入额外的运行时逻辑。为了保持该流程不变并通过流传递额外的 Context 对象，Reactor 使用名为 CoreSubscriber 的接口，该接口是 Subscriber 接口的特定扩展。CoreSubscriber 将 Context 作为其字段进行传递。CoreSubscriber 接口形式如下所示：

```
1 interface CoreSubscriber<T> extends Subscriber<T> {
2     default Context currentContext() {
3         return Context.empty();
4     }
5 }
```

CoreSubscriber 引入了一个名为 currentContext 的附加方法，该方法提供了对当前 Context 对象的访问。

Project Reactor 中的大多数操作符提供了对CoreSubscriber 接口的实现，并引用了下游 Context。

唯一能修改当前 Context 的操作符是 subscriberContext，它是 CoreSubscriber 的实现，持有被合并的下游 Context 并将其作为参数进行传递。

此外，这种行为意味着可访问的 Context 对象可能随流中的位置不同而不同。

例如，以下代码展示了上述行为：

```
1 void run() {
2     printCurrentContext("top")
3         .subscriberContext(Context.of("top", "context"))
4         .flatMap(__ -> printCurrentContext("middle"))
5         .subscriberContext(Context.of("middle", "context"))
6         .flatMap(__ -> printCurrentContext("bottom"))
7         .subscriberContext(Context.of("bottom", "context"))
8         .flatMap(__ -> printCurrentContext("initial"))
9         .block();
10 }
11
12 void print(String id, Context context) {
13 // ...
14 }
15
16 Mono<Context> printCurrentContext(String id) {
17     return Mono
18         .subscriberContext()
19         .doOnNext(context -> print(id, context));
20 }
```

上述代码展示了我们如何在流构造过程中使用 Context。如果我们运行上述代码，控制台将显示以下结果：

```
top {
    Context3{bottom=context, middle=context, top=context}
}
middle {
```

```
Context2{bottom=context, middle=context}  
}  
  
bottom {  
    Context1{bottom=context}  
}  
  
initial {  
    Context0{}  
}
```

如上述代码所示，流顶部的可用 Context 包含此流中可用的整个 Context，其中流的中间部分只能访问在下游中定义的 Context，而位于非常底部（具有 id 初始值）的上下文消费者上下文为空。

一般而言，Context 是一个杀手锏，它推动 Project Reactor 成为建立响应式系统的更高级别的工具。此外，这种特性对于我们需要访问上下文数据的许多场景很有用，例如，在流程中间处理用户请求的场景。此特性在 Spring 框架中得到了广泛使用，在响应式 Spring Security 中尤其如此。

尽管全面地讲解了 Context 特性，但这种 Reactor 技术仍有很大的可能性和应用场景。

## 9.3 Project Reactor内幕

Reactor 拥有丰富的有用操作符。整个 API 具有与 RxJava 类似操作符。

Project Reactor 3 最显著的改进是：

响应式流生命周期（Reactive Stream life-cycle）和操作符融合（operator fusion）。

### 宏融合

宏融合（macro-fusion）主要发生在组装时，其目的是用一个操作符替换另一个操作符。

同时，Flux 内部操作符的某些部分也应该处理一个或零个元素（例如，操作符 just(T)、empty() 和 error(Throwable)）。

在大多数情况下，这些简单的操作符会与其他转换流一起使用。因此，减少这种开销至关重要。

为此，Reactor 在组装时提供优化，如果它检测到上游 Publisher 实现了 Callable 或 ScalarCallable 等接口，那么上游 Publisher 将被经过优化的操作符所替换。应用此类优化的示例代码如下所示：

```
1 | Flux.just(1)  
2 |     .publishon(...)  
3 |     .map(...)
```

上面代码中，元素的执行应该在元素创建后立即移动到不同的工作单元。

如果没有应用优化，这样的执行会分配一个队列来保存来自不同工作单元的元素，而从这样一个队列中入队和出队的元素会导致一些不稳定的读写，因此这种普通 Flux 的执行开销过大。

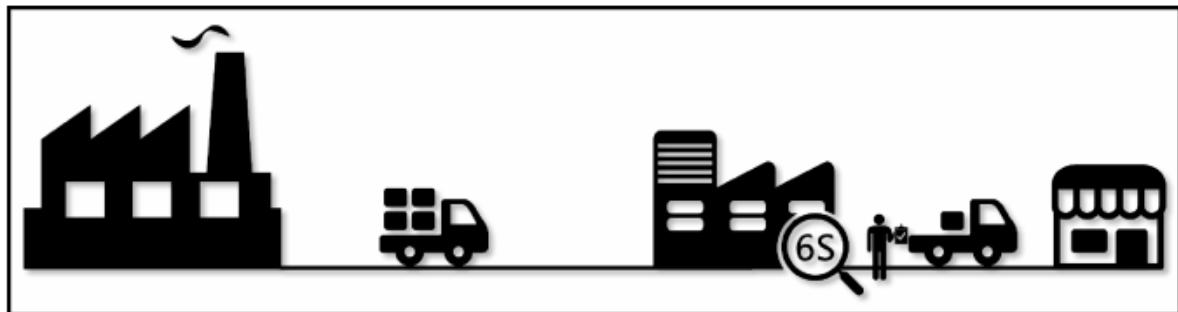
由于执行过程具体发生在哪个工作单元并不重要，并且提供一个元素可以被表示为 ScalarCallable#call，因而可以将 publishOn 操作符替换为不需要创建额外队列的 subscribeOn。此外，由于应用了优化，下游的执行不会改变，因此执行经过优化的流，将获得相同的结果。

前面的示例是隐藏在 Project Reactor 中的宏融合优化中的一种。

一般而言，在 Project Reactor 中应用宏融合的目的是优化组装流程，这样一来，就可以使用更原始、成本更低的解决方案，而不会把强大工具的宝贵资源浪费在简单任务上。

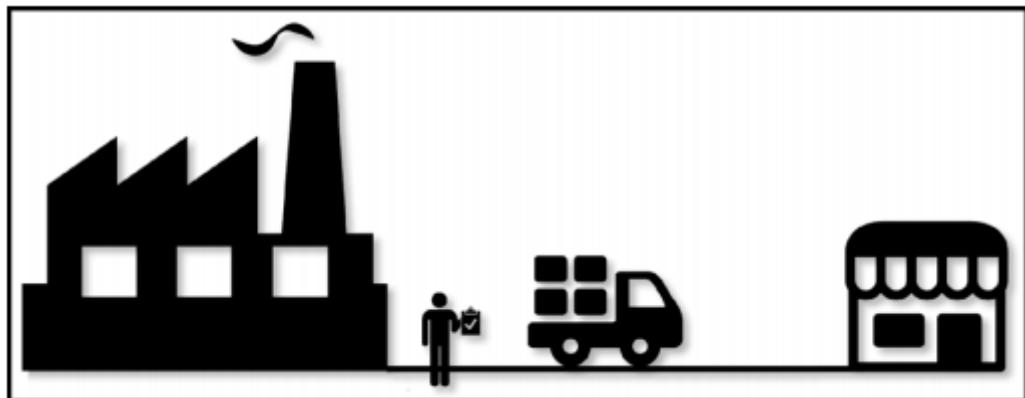
## 微融合

微融合（micro-fusion）是一种更复杂的优化，与运行时优化以及重用共享资源有关。微融合的一个很好的例子是条件操作符。见下图：



商店订购了  $n$  件商品。过了一段时间，工厂用卡车将物品送到商店。

但是，为了最终到达商店，卡车必须通过检验部门，以确保所有商品质量合格。由于有些物品没有仔细包装，因而只有部分订单到达了商店。在那之后，工厂准备了另一辆卡车，再次往商店送货。这种情况反复发生，直到所有订购的商品到达商店。幸好，工厂意识到他们在使商品通过单独的检验部门上花了太多的时间和金钱，并决定从检验部门雇用检验员到本地。



所有物品现在都可以在工厂进行检验后送到商店，而无须前往检验部门。

```
1 | Flux.from(factory)
2 |     .filter(inspectionDepartment)
3 |     .subscribe(store);
```

下游订阅者已从数据源请求了一定数量的元素。

在通过操作符链发出元素时，元素正在通过条件操作符，而这可能拒绝某些元素。为了满足下游的需求，每个被拒绝数据项的过滤器操作符必须执行附加的 request(1) 上游调用。

根据当前响应式库（例如 RxJava 或 Reactor 3）的设计，request 操作有自己的额外 CPU 开销。

根据 David Karnok 的研究，每个“对 request() 的调用通常最终都在一个原子 CAS 循环中，而每 21~45 个循环会掉落一个元素”。

这意味着条件操作符（如 filter 操作符）可能对整体性能产生重大影响！出于这个原因，出现了一种被称为 ConditionalSubscriber 的微融合类型。

这种类型的优化使我们能在数据源端验证条件，并发送所需数量的元素而无须额外的 request 调用。

第二种微融合是最复杂的一种。这种融合与操作符之间的异步边界有关。为了理解这个问题，假设一个具有一些异步边界的操作符链，如下例所示：

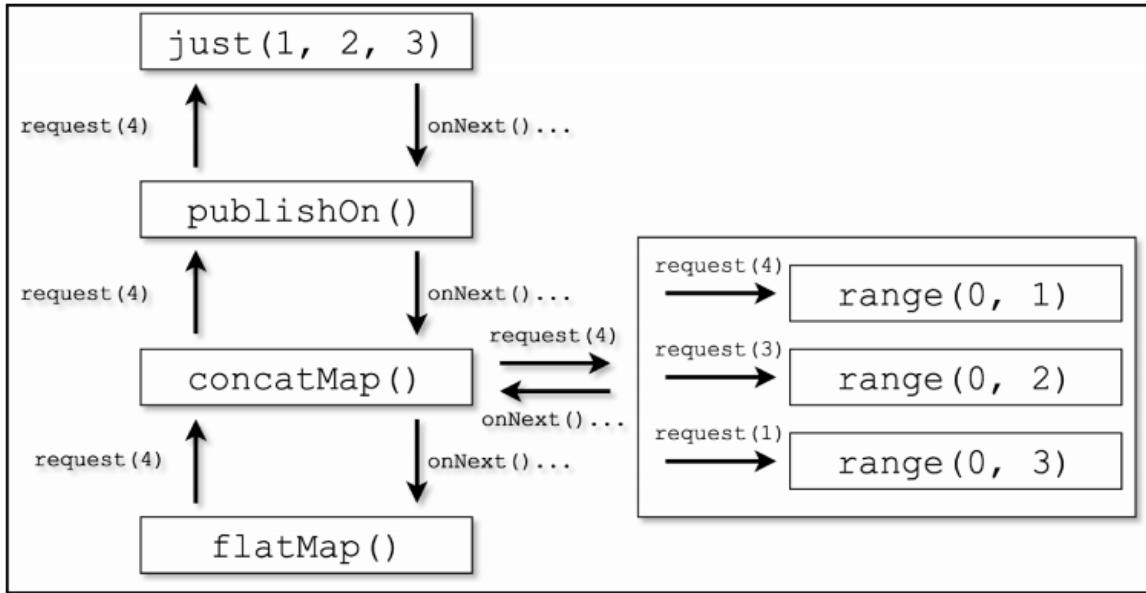
```
1 | Flux.just(1, 2, 3)
2 |     .publishOn(Schedulers.parallel())
3 |     .concatMap(i -> Flux.range(0, i).publishOn(Schedulers.parallel()))
4 |     .subscribe();
```

前面的例子展示了 Reactor 的操作符链。此链包含两个异步边界，这意味着这里会出现队列。

如，因为 concatMap 操作符的本质是它可能在来自上游的每个传入元素上产生  $n$  个元素，所以内部 Flux 将产生多少元素是无法预测的。

为了处理背压以避免压垮消费者，需要将结果放入队列中。而为了将响应式流中的元素从一个工作线程传输到另一个工作线程，publishOn 操作符也需要内部队列。

除了队列开销，还有更危险的跨越异步边界的 request() 调用。这些可能导致更大的内存开销。



上图展示了前面代码段的内部行为。在这里，concatMap 的内部有一个巨大的开销，这种情况下，需要为每个内部流发送一个 request 直到满足下游需求。

每个具有队列的操作符都有自己的 CAS 循环，这在不合理模型的请求事件中可能导致高额性能开销。例如，请求 1 个或任何（相比整个数据量）少得不合理的数量的元素，都可以被认为是不合理的请求模型。

CAS（比较和交换）是一个单独的操作，它会根据操作是否成功返回值 1 或值 0。由于希望操作成功，我们会重复 CAS 操作直到成功为止。这些重复的 CAS 操作被称为 CAS 循环。

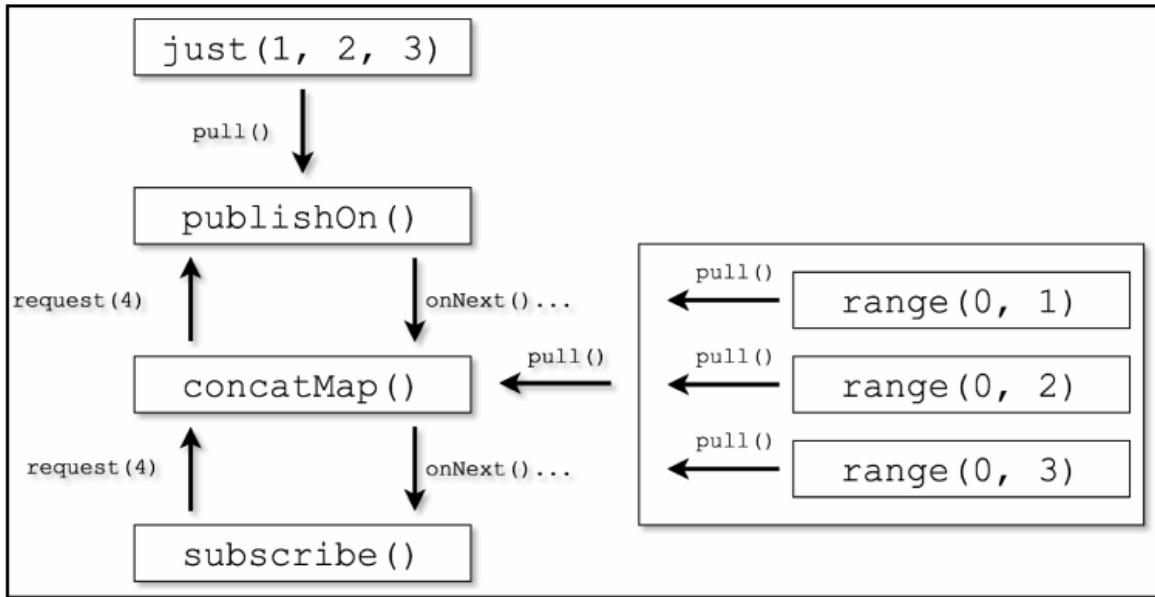
为了避免内存开销和性能开销，应该遵循响应式流规范的建议，切换通信协议。

**假设一个或多个边界内的元素链具有共享队列，那么可以切换整个操作符链以使用上游操作符作为无须额外 request 调用的队列，这样可以显著提高整体性能。**

因此，下游可以从上游排出值，如果该值不可用于指示流的结束，则返回 null。

为了通知下游元素可用，上游调用下游的 onNext，并使用null作为该协议的特例。

此外，错误情况或流的完成将照常通过onError或onComplete进行通知。因此，先前的示例可以通过以下方式优化，如图：



在该示例中，publishOn 和 concatMap 操作符可以被显著优化。在第一种情况下，因为没有必要在主线程中执行的中间操作符，所以我们可以直接使用 just 操作符作为队列，并在单独的线程上从该队列中执行 pull 操作。在 concatMap 的情况下，所有内部流也可以被视为队列，以在没有附加任何 request 调用的情况下排出每个流。

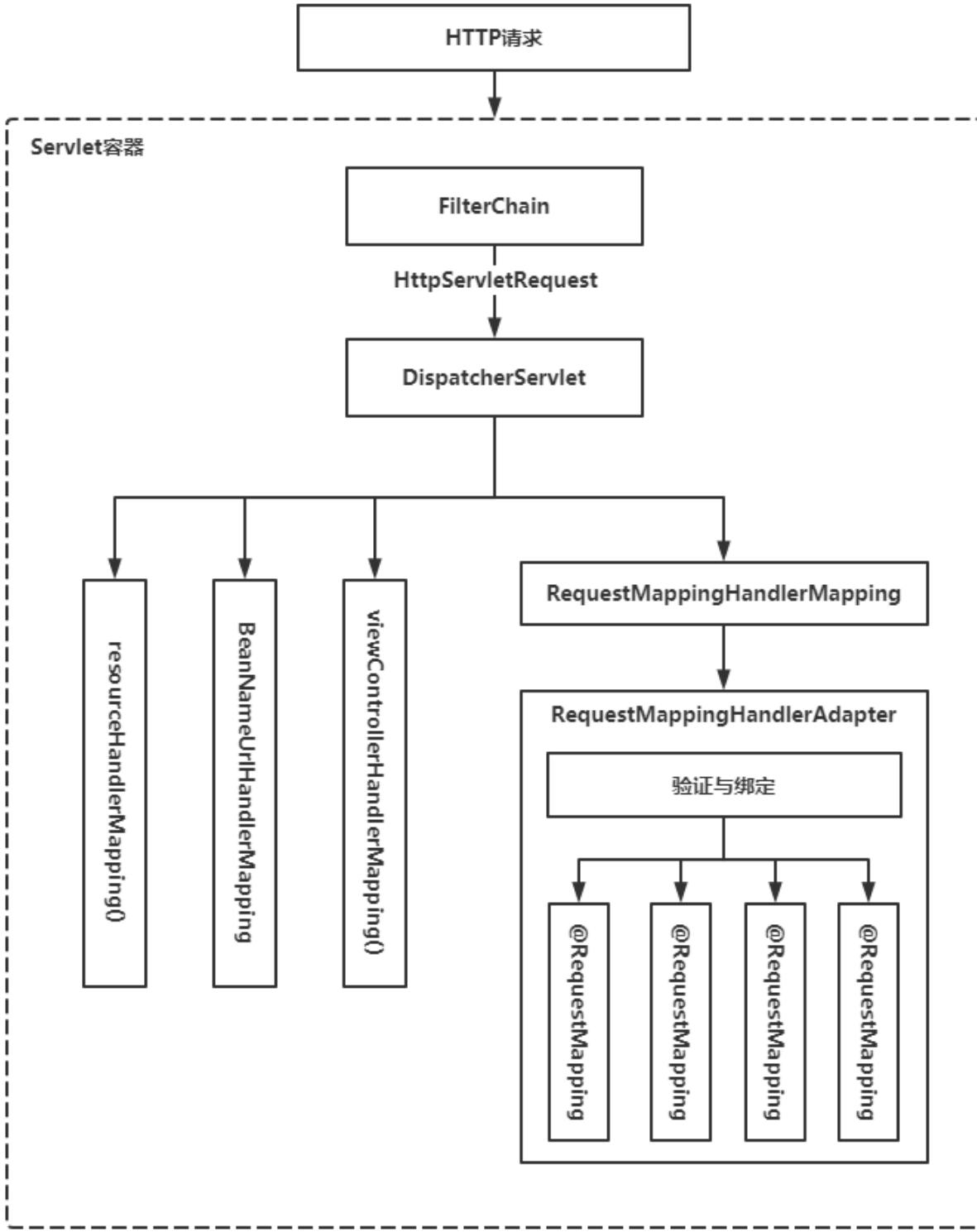
总之，Reactor 库的内部结构比看起来更复杂。通过强大的优化，Reactor 远远领先于 RxJava 1.x，从而提供了更好的性能。

## 第三部分 Spring WebFlux高级实战

### 10 WebFlux作为核心响应式服务器基础

Spring 框架的整个基础设施都是围绕 Servlet API 构建的，它们之间紧密耦合。

因此在开始深入响应式 Web 之前，先回顾一下 Web 模块的设计，看看它做了什么。



底层Servlet 容器负责处理容器内的所有映射Servlet。

DispatchServlet 作为一个集成点，用于集成灵活且高度可配置的Spring Web基础设施和繁重且复杂的Servlet API。

HandlerMapping将业务逻辑与Servlet API 分离。

Spring MVC的缺点：

1. 不允许在整个请求声明周期中出现非阻塞操作。没有开箱即用的非阻塞HTTP客户端。
2. WebMVC 抽象不能支持非阻塞 Servlet 3.1 的所有功能。
3. 对于非 Servlet 服务器，重用 Spring Web 功能或变成模块不灵活。

因此，Spring团队在过去几年中的核心挑战，就是如何构建一个新的解决方案，以在使用基于注解的编程模型的同时，提供异步非阻塞服务的所有优势。

## 10.1 响应式Web内核

响应式 Web 栈应该是什么样子？

响应式Web内核首先需要使用模拟接口和对请求进行处理的方法替换

`javax.servlet.Servlet.service`方法。

更改相关的类和接口

增强和定制 Servlet API 对客户端请求和服务器响应的交互方式。

```
1 /**
2  * 请求的封装。
3  * 获取请求报文体的类型是Flux，表示具备响应式能力。
4  * DataBuffer是针对字节缓冲区的抽象，便于对特定服务器实现数据交换。
5  * 除了请求报文体，还有消息头、请求路径、cookie、查询参数等信息，可以在该接口或子接口中提供。
6  */
7 interface ServerHttpRequest {
8     // ...
9     Flux<DataBuffer> getBody();
10    // ...
11 }
12
13 /**
14  * 响应的封装。
15  * writewith方法接收的参数是Publisher，提供了响应式，并与特定响应式库解耦。
16  * 返回值是Mono<Void>，表示向网络发送数据是一个异步的过程。
17  * 即，只有当订阅Mono时才会执行发送数据的过程。
18  * 接收服务器可以根据传输协议的流控支持背压。
19  */
20 interface ServerHttpResponse {
21     // ...
22     Mono<Void> writewith(Publisher<? extends DataBuffer> body);
23     // ...
24 }
25
26 /**
27  * HTTP请求-响应的容器。
28  * 这是高层接口，除了HTTP交互，还可以保存框架相关信息。
29  * 如请求的已恢复的webSession信息等。
30  *
31  */
32 /**
33 interface ServerWebExchange {
34     // ...
35     ServerHttpRequest getRequest();
36     ServerHttpResponse getResponse();
```

```
37     // ...
38     Mono<WebSession> getSession();
39     // ...
40 }
```

上述三个接口类似于Servlet API 中的接口。

响应式接口旨在从交互模型的角度提供几乎相同的方法，同时提供开箱即用的响应式。

请求的处理程序和过滤器 API:

```
1 /**
2  * 对应于webMVC中的DispatcherServlet
3  *   查找请求的处理程序，使用视图解析器渲染视图，因此handle方法不需要返回任何结果。
4  *
5  *   返回值Mono<Void>提供了异步处理。
6  *   如果在指定的时间内没有信号出现，可以取消执行。
7  */
8 interface WebHandler {
9     Mono<Void> handle(ServerWebExchange exchange);
10 }
11
12 /**
13  * 过滤器链
14  */
15 interface WebFilterChain {
16     Mono<Void> filter(ServerWebExchange exchange);
17 }
18
19 /**
20  * 过滤器
21  */
22 interface WebFilter {
23     Mono<Void> filter(ServerWebExchange exchange, WebFilterChain chain);
24 }
```

以上是响应式Web应该具备的基础API。

还需要为这些接口适配不同的服务器。

即与ServerHttpRequest和ServerHttpResponse进行直接交互的组件。

同时负责ServerWebExchange的构建，特定的会话存储、本地化解析器等信息的保存。

```
1 public interface HttpHandler {
2     Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse
3     response);
```

通过该适当的抽象，隐藏了服务器引擎的细节，具体服务器的工作方式对 Spring WebFlux 用户不重要。

## 10.2 响应式Web和MVC框架

Spring Web MVC 模块的关键特性 **基于注解**。因此，需要为响应式Web 栈提供相同的概念。

重用 WebMVC 的基础设施，用 `Flux`、`Mono` 和 `Publisher` 等响应式类型替换同步通信。

保留与 Spring Web MVC 相同的 `HandlerMapping` 和 `HandlerAdapter` 链，使用基于 Reactor 的响应式交互替换实时命令：

```
1 interface HandlerMapping {
2     /*
3      HandlerExecutionChain getHandler(HttpServletRequest request)
4      */
5     Mono<Object> getHandler(ServerWebExchange exchange);
6 }
7
8 interface HandlerAdapter {
9
10    boolean supports(Object handler);
11    /*
12     ModelAndView handle(HttpServletRequest request, HttpServletResponse
13     response, Object handler);
14     */
15    Mono<HandlerResult> handle(ServerWebExchange exchange, Object handler);
16 }
```

响应式 `HandlerMapping` 中，两个方法整体上类似，

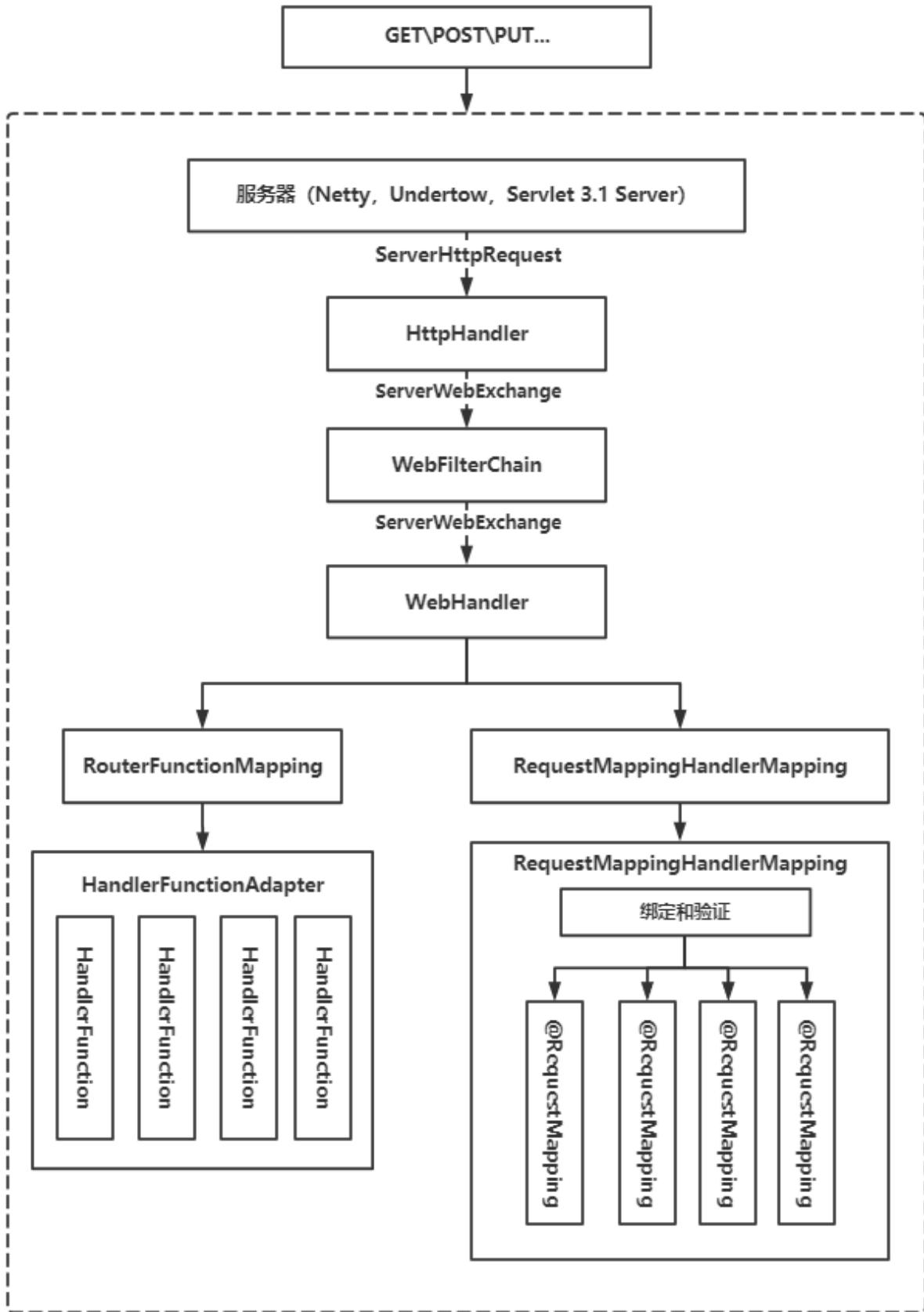
不同之处在于响应式返回 `Mono` 类型支持响应式。

响应式 `HandlerAdapter` 接口中，由于 `ServerWebExchange` 类同时组合了请求和响应，因此 `handle` 方法的响应式版本更简洁。

该方法返回 `HandlerResult` 的 `Mono` 而不是 `ModelAndView`。

遵循这些步骤，我们将得到一个响应式交互模型，而不会破坏整个执行层次结构，从而可以保留现有设计并能以最小的更改重用现有代码。

最后设计：



1. 传入请求，由底层服务器引擎处理。服务器引擎列表不限于基于ServletAPI 的服务器。每个服务器引擎都有自己的响应式适配器，将 HTTP 请求和 HTTP 响应的内部表示映射到 `ServerHttpRequest` 和 `ServerHttpResponse`。
2. `HttpHandler` 阶段，该阶段将给定的 `ServerHttpRequest`、`ServerHttpResponse`、用户 `Session` 和相关信息组合到 `ServerwebExchage` 实例中。
3. `webFilterChain` 阶段，它将定义的 `webFilter` 组合到链中。然后，`webFilterChain` 会负责执行此链中每个 `webFilter` 实例的 `webFilter#filter` 方法，以过滤传入的

- `ServerWebExchange`。
4. 如果满足所有过滤条件，`WebFilterChain` 将调用 `webHandler` 实例。
  5. 查找 `HandlerMapping` 实例并调用第一个合适的实例。可以是 `RouterFunctionMapping`、也可以是 `RequestMappingHandlerMapping` 和 `HandlerMapping` 资源。`RouterFunctionMapping`，引入到 `WebFlux` 之中，超越了纯粹的功能请求处理。
  6. `RequestMappingHandlerAdapter` 阶段，与以前功能相同，使用响应式流来构建响应式流。

在 `WebFlux` 模块中，默认服务器引擎是 `Netty`。

`Netty` 服务器很适合作为默认服务器，因为它广泛用于响应式领域。

该服务器引擎还同时提供客户端和服务器**异步非阻塞交互**。

同时，可以灵活地选择服务器引擎。

`WebFlux` 模块对应了 `Spring Web MVC` 模块的体系结构，很容易理解。

## 10.3 基于 `WebFlux` 的纯函数式 Web

纯函数式 Web 主要是函数式路由映射。

通过函数式映射，可以生成轻量级应用。

示例如下：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     https://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>2.4.0</version>
11    <relativePath/> <!-- lookup parent from repository -->
12  </parent>
13  <groupId>com.lagou.webflux.demo</groupId>
14  <artifactId>demo</artifactId>
15  <version>0.0.1-SNAPSHOT</version>
16  <name>demo</name>
17  <description>Demo project for Spring Boot</description>
18
19  <properties>
20    <java.version>11</java.version>
21  </properties>
```

```

21 <dependencies>
22     <dependency>
23         <groupId>org.springframework.boot</groupId>
24         <artifactId>spring-boot-starter-webflux</artifactId>
25     </dependency>
26
27     <dependency>
28         <groupId>org.projectlombok</groupId>
29         <artifactId>lombok</artifactId>
30     </dependency>
31
32     <dependency>
33         <groupId>org.springframework.security</groupId>
34         <artifactId>spring-security-crypto</artifactId>
35     </dependency>
36
37     <dependency>
38         <groupId>org.springframework.boot</groupId>
39         <artifactId>spring-boot-starter-test</artifactId>
40         <scope>test</scope>
41     </dependency>
42     <dependency>
43         <groupId>io.projectreactor</groupId>
44         <artifactId>reactor-test</artifactId>
45         <scope>test</scope>
46     </dependency>
47 </dependencies>
48
49 <build>
50     <plugins>
51         <plugin>
52             <groupId>org.springframework.boot</groupId>
53             <artifactId>spring-boot-maven-plugin</artifactId>
54         </plugin>
55     </plugins>
56 </build>
57
58 </project>

```

```

1 package com.lagou.webflux.demo;
2
3 import com.lagou.webflux.demo.handler.OrderHandler;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.http.HttpMethod;
8 import org.springframework.web.reactive.function.server.RouterFunction;
9 import org.springframework.web.reactive.function.server.ServerResponse;
10
11 import static org.springframework.http.MediaType.APPLICATION_JSON;
12 import static
13     org.springframework.web.reactive.function.server.RequestPredicates.*;
14 import static
15     org.springframework.web.reactive.function.server.RouterFunctions.nest;

```

```

14 import static
15     org.springframework.web.reactive.function.server.RouterFunctions.route;
16
17 @SpringBootApplication
18 public class WebfluxWebApplication {
19
20     public static void main(String[] args) {
21         SpringApplication.run(WebfluxWebApplication.class, args);
22     }
23
24     @Bean
25     public RouterFunction<ServerResponse> routes(OrderHandler handler) {
26         return
27             // 包含两个参数: 1, 测试条件是否通过, 如果通过, 则路由到第二个参数指定的
28             // 路由函数
29             nest(
30                 // 判断请求路径是否匹配指定的模式, (请求路径前缀)
31                 path("/orders"),
32                 // 如果匹配通过, 则路由到该路由函数
33                 nest(accept(APPLICATION_JSON), // 判断请求的报文头字段
34                     // accept是否匹配APPLICATION_JSON媒体类型
35                     // 如果匹配, 则路由到下面的路由函数: 将/orders/{id}路由
36                     // 给handler的get方法
37                     route(GET("/{id}"), handler::get)
38                         // 如果是get请求/orders, 则路由到handler的
39                     list
40                         .andRoute(method(HttpMethod.GET),
41                             handler::list))
42                     // 如果contentType匹配, 并且路径匹
43                     // 配/orders, 则路由到路由函数
44                     // 如果是POST请求/orders, 则路由到handler的
45                     create
46                         .andNest(contentType(APPLICATION_JSON),
47                             route(POST("/"), handler::create))
48                     );
49     }
50 }

```

以下代码展示了如何实现自定义RequestPredicate并将其应用于路由逻辑：

```

1 nest((serverRequest) -> serverRequest.cookies()
2     .containsKey("Redirect-Traffic"), route(all(), serverRedirectHandler)
3 )

```

```

1 package com.lagou.webflux.demo.handler;
2
3
4 import org.springframework.core.io.buffer.DataBuffer;
5 import org.springframework.web.reactive.function.BodyInserters;
6 import org.springframework.web.reactive.function.client.WebClient;

```

```

7 import org.springframework.web.reactive.function.server.HandlerFunction;
8 import org.springframework.web.reactive.function.server.ServerRequest;
9 import org.springframework.web.reactive.function.server.ServerResponse;
10 import reactor.core.publisher.Mono;
11
12 public class ServerRedirectHandler implements
13 HandlerFunction<ServerResponse> {
14
15     final WebClient webClient = WebClient.create();
16
17     @Override
18     public Mono<ServerResponse> handle(ServerRequest request) {
19         return webClient
20             .method(request.method())
21             .uri(request.headers()
22                 .header("Redirect-Traffic")
23                 .get(0))
24             .headers((h) -> h.addAll(request.headers().asHttpHeaders()))
25             .body(BodyInserters.fromDataBuffers(
26                 request.bodyToFlux(DataBuffer.class)
27             ))
28             .cookies(c -> request
29                 .cookies()
30                 .forEach((key, list) -> list.forEach(cookie -> c.add(key,
31                     cookie.getValue()))))
32             )
33             .exchange()
34             .flatMap(cr -> ServerResponse
35                 .status(cr.statusCode())
36                 .cookies(c -> c.addAll(cr.cookies()))
37                 .headers(hh -> hh.addAll(cr.headers().asHttpHeaders()))
38                 .body(BodyInserters.fromDataBuffers(
39                     cr.bodyToFlux(DataBuffer.class)
40                 )));
41     }

```

在以上示例中，如果出现“Redirect-Traffic”cookie，它会将流量重定向到另一台服务器。

新的函数式Web还引入了一种处理请求和响应的新方法。

以下示例是OrderHandler实现：

```

1 package com.lagou.webflux.demo.handler;
2
3 import com.lagou.webflux.demo.entity.Order;
4 import com.lagou.webflux.demo.repository.OrderRepository;
5 import org.springframework.stereotype.Service;
6 import org.springframework.web.reactive.function.server.ServerRequest;
7 import org.springframework.web.reactive.function.server.ServerResponse;

```

```

8 import reactor.core.publisher.Mono;
9
10 import java.net.URI;
11
12 @Service
13 public class OrderHandler {
14
15     final OrderRepository orderRepository;
16
17     public OrderHandler(OrderRepository repository) {
18         orderRepository = repository;
19     }
20
21     public Mono<ServerResponse> create(ServerRequest request) {
22         return request
23             .bodyToMono(Order.class)
24             .flatMap(orderRepository::save)
25             .flatMap(o ->
26                 ServerResponse.created(URI.create("/orders/" + o.getId()))
27                     .build()
28             );
29     }
30
31     public Mono<ServerResponse> get(ServerRequest request) {
32         return orderRepository
33             .findById(request.pathVariable("id"))
34             .flatMap(order ->
35                 ServerResponse
36                     .ok()
37                     .syncBody(order)
38             )
39             .switchIfEmpty(ServerResponse.notFound().build());
40     }
41
42     public Mono<ServerResponse> list(ServerRequest request) {
43         return null;
44     }
45 }
46

```

OrderRepository:

```

1 package com.lagou.webflux.demo.repository;
2
3 import com.lagou.webflux.demo.entity.Order;
4 import org.springframework.stereotype.Repository;
5 import reactor.core.publisher.Mono;
6
7 @Repository
8 public interface OrderRepository {
9
10     Mono<Order> findById(String id);
11
12     Mono<Order> save(Order order);
13

```

```
14     Mono<Void> deleteById(String id);  
15 }
```

Order:

```
1 package com.lagou.webflux.demo.entity;  
2  
3 import lombok.AllArgsConstructorConstructor;  
4 import lombok.Data;  
5 import lombok.NoArgsConstructorConstructor;  
6  
7 @Data  
8 @AllArgsConstructorConstructor  
9 @NoArgsConstructorConstructor  
10 public class Order {  
11     private String id;  
12 }
```

create 方法接收 ServerRequest (函数式路由请求类型)。

ServerRequest 可以将请求体手动映射到 Mono 或 Flux。该 API 还可以指定请求体应映射的类。

最后，WebFlux 中的函数式附加功能提供了一个 API，使用 ServerResponse 类的流式 API 构建响应。

我们可以看到，除函数式路由声明的 API 之外，我们还有一个用于请求和响应处理的函数式 API。

同时，函数式 Web 框架允许在不启动整个 Spring 基础设施的情况下构建 Web 应用程序。

如下案例：

```
1 package com.lagou.webflux.demo;  
2  
3 import com.lagou.webflux.demo.entity.PasswordDTO;  
4 import org.slf4j.Logger;  
5 import org.slf4j.LoggerFactory;  
6 import org.springframework.http.HttpStatus;  
7 import org.springframework.http.server.reactive.HttpHandler;  
8 import org.springframework.http.server.reactive.ReactorHttpHandlerAdapter;  
9 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
10 import org.springframework.security.crypto.password.PasswordEncoder;  
11 import org.springframework.web.reactive.function.server.RouterFunction;  
12 import org.springframework.web.reactive.function.server.RouterFunctions;  
13 import org.springframework.web.reactive.function.server.ServerResponse;  
14 import reactor.netty.DisposableServer;  
15 import reactor.netty.http.server.HttpServer;  
16  
17 import static  
18     org.springframework.web.reactive.function.server.RequestPredicates.POST;  
19 import static  
20     org.springframework.web.reactive.function.server.RouterFunctions.route;
```

```
19
20 public class StandaloneApplication {
21     static Logger LOGGER =
22         LoggerFactory.getLogger(StandaloneApplication.class);
23
24     public static void main(String... args) {
25         long start = System.currentTimeMillis();
26         // 调用routes 方法, 然后将RouterFunction 转换为HttpHandler。
27         HttpHandler httpHandler = RouterFunctions.toHttpHandler(routes(
28             // BCrypt算法进行18 轮散列, 这可能需要几秒钟来编码/匹配
29             new BCryptPasswordEncoder(18)
30         ));
31         // 内置HttpHandler适配器
32         ReactorHttpHandlerAdapter reactorHttpHandler = new
33         ReactorHttpHandlerAdapter(httpHandler);
34         // 创建HttpServer实例, 它是ReactorNetty API的一部分。
35         DisposableServer server = HttpServer.create()
36             .host("localhost") // 配置host
37             .port(8080) // 配置端口
38             .handle(reactorHttpHandler) // 指定handler
39             .bindNow(); // 调用bindNow启动服务
40     }
41
42     public static RouterFunction<ServerResponse> routes(PasswordEncoder
43     passwordEncoder) {
44         return
45             // 使用/ check 路径处理任何POST 方法的请求
46             route(POST("/password"),
47                 request -> request
48                     .bodyToMono(PasswordDTO.class)
49                     // 使用PasswordEncoder检查已加密密码的原始密码
50                     //
51                     .map(p -> passwordEncoder.matches(p.getRaw(),
52                         p.getSecured())))
52                     // 如果密码与存储的密码匹配
53                     // 则ServerResponse 将返回OK状态(200)
54                     // 否则, 返回EXPECTATION_FAILED(417)
55                     .flatMap(isMatched -> isMatched
56                         ? ServerResponse.ok()
57                             .build()
58                         :
59                     ServerResponse.status(HttpStatus.EXPECTATION_FAILED)
60                         .build()
61                     )
62             );
63     }
64 }
```

```

1 package com.lagou.webflux.demo.entity;
2
3 import com.fasterxml.jackson.annotation.JsonCreator;
4 import com.fasterxml.jackson.annotation.JsonProperty;
5
6 public class PasswordDTO {
7     private String raw;
8     private String secured;
9
10
11     @JsonCreator
12     public PasswordDTO(@JsonProperty("raw") String raw,
13                         @JsonProperty("secured") String secured) {
14         this.raw = raw;
15         this.secured = secured;
16     }
17
18     public String getRaw() {
19         return raw;
20     }
21
22     public String getSecured() {
23         return secured;
24     }
25
26     public void setRaw(String raw) {
27         this.raw = raw;
28     }
29
30     public void setSecured(String secured) {
31         this.secured = secured;
32     }
33 }

```

这种Web 应用程序的好处是它的启动时间要短得多。

非Standalone服务：

pom.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5   https://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7     <parent>
8       <groupId>org.springframework.boot</groupId>
9       <artifactId>spring-boot-starter-parent</artifactId>
10      <version>2.4.0</version>
11      <relativePath/> <!-- lookup parent from repository -->
12    </parent>
13    <groupId>com.lagou.webflux.demo</groupId>
14    <artifactId>demo</artifactId>

```

```

13 <version>0.0.1-SNAPSHOT</version>
14 <name>demo</name>
15 <description>Demo project for Spring Boot</description>
16
17 <properties>
18   <java.version>11</java.version>
19 </properties>
20
21 <dependencies>
22   <dependency>
23     <groupId>org.springframework.boot</groupId>
24     <artifactId>spring-boot-starter-security</artifactId>
25   </dependency>
26   <dependency>
27     <groupId>org.springframework.boot</groupId>
28     <artifactId>spring-boot-starter-webflux</artifactId>
29   </dependency>
30
31   <dependency>
32     <groupId>org.projectlombok</groupId>
33     <artifactId>lombok</artifactId>
34     <optional>true</optional>
35   </dependency>
36   <dependency>
37     <groupId>org.springframework.boot</groupId>
38     <artifactId>spring-boot-starter-test</artifactId>
39     <scope>test</scope>
40   </dependency>
41   <dependency>
42     <groupId>io.projectreactor</groupId>
43     <artifactId>reactor-test</artifactId>
44     <scope>test</scope>
45   </dependency>
46   <dependency>
47     <groupId>org.springframework.security</groupId>
48     <artifactId>spring-security-test</artifactId>
49     <scope>test</scope>
50   </dependency>
51 </dependencies>
52
53 <build>
54   <plugins>
55     <plugin>
56       <groupId>org.springframework.boot</groupId>
57       <artifactId>spring-boot-maven-plugin</artifactId>
58     </plugin>
59   </plugins>
60 </build>
61
62 </project>

```

## PasswordEncoder.java

```

1 package com.lagou.webflux.demo.entity;
2

```

```
3 import com.fasterxml.jackson.annotation.JsonCreator;
4 import com.fasterxml.jackson.annotation.JsonProperty;
5
6 public class PasswordDTO {
7     private String raw;
8     private String secured;
9
10
11     @JsonCreator
12     public PasswordDTO(@JsonProperty("raw") String raw,
13                         @JsonProperty("secured") String secured) {
14         this.raw = raw;
15         this.secured = secured;
16     }
17
18     public String getRaw() {
19         return raw;
20     }
21
22     public String getSecured() {
23         return secured;
24     }
25
26     public void setRaw(String raw) {
27         this.raw = raw;
28     }
29
30     public void setSecured(String secured) {
31         this.secured = secured;
32     }
33 }
```

### PasswordEncoder.java

```
1 package com.lagou.webflux.demo.handler;
2
3 import com.lagou.webflux.demo.entity.PasswordDTO;
4 import org.springframework.http.HttpStatus;
5 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
6 import org.springframework.stereotype.Component;
7 import org.springframework.web.reactive.function.server.HandlerFunction;
8 import org.springframework.web.reactive.function.server.ServerRequest;
9 import org.springframework.web.reactive.function.server.ServerResponse;
10 import reactor.core.publisher.Mono;
11
12 @Component
13 public class PasswordHandlerFunction implements HandlerFunction {
14
15     private BCryptPasswordEncoder passwordEncoder = new
16     BCryptPasswordEncoder(18);
17
18     @Override
19     public Mono<Void> handle(ServerRequest request) {
20         return request.bodyToMono(PasswordDTO.class)
```

```
20             .map(p -> passwordEncoder.matches(p.getRaw(),  
21         p.getSecured()))  
22                 .flatMap(isMatched -> isMatched  
23                     ? ServerResponse.ok()  
24                         .build()  
25                         :  
26             ServerResponse.status(HttpStatus.EXPECTATION_FAILED)  
27                         .build()).then(Mono.empty());  
28     }  
29 }
```

### WebfluxStandaloneApplication.java

```
1 package com.lagou.webflux.demo;  
2  
3 import com.lagou.webflux.demo.handler.PasswordHandlerFunction;  
4 import org.springframework.boot.SpringApplication;  
5 import org.springframework.boot.autoconfigure.SpringBootApplication;  
6 import org.springframework.context.annotation.Bean;  
7 import org.springframework.security.config.web.server.ServerHttpSecurity;  
8 import org.springframework.security.web.server.SecurityWebFilterChain;  
9 import org.springframework.web.reactive.function.server.RouterFunction;  
10 import org.springframework.web.reactive.function.server.ServerResponse;  
11  
12 import static  
13     org.springframework.web.reactive.function.server.RequestPredicates.POST;  
14 import static  
15     org.springframework.web.reactive.function.server.RouterFunctions.route;  
16  
17 @SpringBootApplication  
18 public class webfluxStandaloneApplication {  
19  
20     public static void main(String[] args) {  
21         SpringApplication.run(webfluxStandaloneApplication.class, args);  
22     }  
23  
24     @Bean  
25     public static RouterFunction<ServerResponse>  
26     routes(PasswordHandlerFunction handlerFunction) {  
27         return route(POST("/password"), handlerFunction);  
28     }  
29  
30     @Bean  
31     public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity  
32     serverHttpSecurity) {  
33         return serverHttpSecurity.csrf()  
34             .disable()  
35             .build();  
36     }  
37 }
```

通过切换到函数式路由声明，

1. 可以在一个位置维护所有路由配置，并使用响应式方法对传入请求进行处理。
2. 在访问传入的请求参数、路径变量和请求的其他重要组件方面，函数式路由的灵活性与基于注解的常规方法几乎相同。
3. 函数式路由不但能避免运行整个Spring 框架基础设施，并且在路由设置方面同样灵活，让应用程序的启动更快。

## 10.4 基于WebClient的非阻塞跨服务通信

从本质上讲，`webClient` 是旧 `RestTemplate` 的响应式替代品。

WebClient 中有一个函数式API，并提供内置的到 Project Reactor 类型（如 `Flux` 或 `Mono`）的映射。

以下示例：

```
1 // 创建的时候指定基础URI
2 webclient.create("http://localhost/api")
3     // 指定请求方法: GET
4     .get()
5     // 指定相对URI，并对URI变量进行扩展
6     // 还可以指定消息头、cookie 和请求主体。
7     .uri("/users/{id}", userId)
8     // 指定结果的处理方式
9     .retrieve()
10    // 将响应体进行反序列化
11    .bodyToMono(User.class)
12    // 进行其他操作
13    .map(...)
14    // 订阅触发异步执行，发起远程调用。这里只使用了订阅的副作用。
15    .subscribe();
```

WebClient 遵循响应式流规范中描述的行为。

只有通过 `subscribe` 方法，`webclient` 才会建立连接并开始发送数据到远程服务器。

最常见的响应处理是处理消息主体，在某些情况下需要处理响应状态、消息头或cookie。

构建一个对密码检查服务的调用，并使用WebClient API以自定义方式处理响应状态：

```
1 package com.taobao.webflux.demo;
2
```

```
3 import com.lagou.webflux.demo.client.DefaultPasswordVerificationService;
4 import org.junit.jupiter.api.Test;
5 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
6 import org.springframework.web.reactive.function.client.WebClient;
7 import reactor.test.StepVerifier;
8
9 import java.time.Duration;
10
11 public class StandaloneTest {
12
13     @Test
14     public void checkApplicationRunning() {
15         BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(18);
16         DefaultPasswordVerificationService service =
17             new DefaultPasswordVerificationService(webClient.builder());
18
19         StepVerifier.create(service.check("test", encoder.encode("test")))
20             .expectSubscription()
21             .expectComplete()
22             .verify(Duration.ofSeconds(30));
23     }
24 }
```

```
1 package com.lagou.webflux.demo.client;
2
3 import reactor.core.publisher.Mono;
4
5 public interface PasswordVerificationService {
6
7     Mono<Void> check(String raw, String encoded);
8 }
```

```
1 package com.lagou.webflux.demo.client;
2
3 import com.lagou.webflux.demo.entity.PasswordDTO;
4 import org.springframework.http.ResponseEntity;
5 import org.springframework.security.authentication.BadCredentialsException;
6 import org.springframework.web.reactive.function.BodyInserters;
7 import org.springframework.web.reactive.function.client.WebClient;
8 import reactor.core.publisher.Mono;
9
10 import static org.springframework.http.HttpStatus.EXPECTATION_FAILED;
11
12 public class DefaultPasswordVerificationService
13     implements PasswordVerificationService {
14
15     final WebClient webClient;
16
17     public DefaultPasswordVerificationService(WebClient.Builder
18 webClientBuilder) {
19         webClient = webClientBuilder
20             .baseUrl("http://localhost:8080")
```

```

20         .build();
21     }
22
23     @Override
24     public Mono<Void> check(String raw, String encoded) {
25
26         return webClient.create("http://localhost:8080").post()
27             .uri("/password")
28             .body(BodyInserters.fromPublisher(
29                 Mono.just(new PasswordDTO(raw, encoded)),
30                 PasswordDTO.class
31             ))
32             .retrieve()
33             .toEntityFlux(ResponseEntity.class)
34             .flatMap(response -> {
35                 if (response.getStatusCode().is2xxSuccessful()) {
36                     return Mono.empty();
37                 } else if (response.getStatusCode() ==
38 EXPECTATION_FAILED) {
39                     return Mono.error(new
40                         BadCredentialsException("Invalid credentials"));
41                 }
42             });
43     }

```

在需要处理公共HTTP响应的状态码、消息头、cookie和其他内部数据的情况下，最合适的是exchange方法，该方法返回ClientResponse。

如前文所述，DefaultWebClient 使用ReactorNetty HttpClient 来提供与远程服务器的异步和非阻塞交互。

但是，DefaultWebClient 旨在轻松更改底层HTTP客户端。为此，出现了一个名为org.springframework.http.client.reactive.ClientHttpConnector 的针对HTTP连接的低级别响应式抽象。

默认情况下，DefaultWebClient 预先配置为使用ReactorClientHttpConnector，而这是ClientHttpConnector 接口的实现。

从SpringWebFlux 5.1 开始，JettyClientHttpConnector 实现出现，它使用Jetty 中的响应式HttpClient。

为了更改底层HTTP客户端引擎，我们可以使用WebClient.Builder#clientConnector 方法并传递所需的实例，该实例既可以是自定义实现，也可以是现有实例。

```

1 | WebClient.builder().clientConnector(new JettyClientHttpConnector())
2 |     .build()
3 |     .get()
4 |     .uri("http://localhost:8080/password/{id}", "ABOP8UOFDSA")
5 |     .retrieve()
6 |     .toEntityFlux(ClientResponse.class)
7 |     .subscribe();

```

除了有用的抽象层，ClientHttpConnector 还可以以原始格式的方式使用。

例如，它可用于下载大文件、即时处理或简单的字节扫描。

## 10.5 响应式模板引擎

Spring 5.x 和 WebFlux 模块**已经放弃**支持包括Apache Velocity 在内的许多技术。

Spring WebFlux 与 Web MVC 拥有相同的视图渲染技术。

以下示例展示了一种指定渲染视图的常用方法：

```
1 @RequestMapping("/")
2 public String index() {
3     return "index";
4 }
```

模板渲染过程中如何支持响应式方法？

考虑一个涉及渲染大型音乐播放列表的案例：

```
1 @RequestMapping("/play-list-view-ftl")
2 public Mono<String> getPlaylist(final Model model) {
3     List<Song> songs = new ArrayList<>();
4     Song song = null;
5     for (int i = 0; i < 5; i++) {
6         song = new Song("曲目" + i, "张三" + i, "1001" + i, "专辑1" + (i %
3));
7         songs.add(song);
8     }
9     final Flux<Song> playlistStream = Flux.fromIterable(songs);
10    return playlistStream
11        .collectList()
12        .doOnNext(list -> model.addAttribute("playList", list))
13        .then(Mono.just("/freemarker/play-list-view"));
14 }
```

正如上述示例中所示，使用了一个响应式类型 `Mono<String>`，以便异步返回视图名称。另外，我们的模板有一个占位符 `dataSource`，它应该由给定 `Song` 的列表填充。

提供特定于上下文数据的常用方法是定义 `Model`，并在其中放置所需的属性。

FreeMarker 不支持数据的响应式呈现和非阻塞呈现，必须将所有歌曲收集到列表中并将收集的数据全部放入Model 中。

src/main/resources/templates/freemarker/play-list-view.ftl:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4 <head>
5     <meta charset="UTF-8"/>
6     <title>曲目列表-freemarker</title>
7 </head>
8 <table border="1">
9     <thead>
10    </thead>
11    <tbody>
12    <#list playList as e>
13    <tr>
14        <td>${e.id}</td>
15        <td>${e.name}</td>
16        <td>${e.artist}</td>
17        <td>${e.album}</td>
18    </tr>
19    </#list>
20    </tbody>
21 </table>
22 </body>
23 </html>
```

src/main/java/com/lagou/webflux/demo/config/WebConfig.java:

```
1 package com.lagou.webflux.demo.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.web.reactive.config.EnableWebFlux;
6 import org.springframework.web.reactive.config.ViewResolverRegistry;
7 import org.springframework.web.reactive.config.WebFluxConfigurer;
8 import
9 import org.springframework.web.reactive.result.view.freemarker.FreeMarkerConfigurer;
10 ;
11
12 @Configuration
13 @EnableWebFlux
14 public class WebConfig implements WebFluxConfigurer {
15
16     @Override
17     public void configureViewResolvers(ViewResolverRegistry registry) {
18         // 使用".ftl"后缀注册一个FreeMarkerViewResolver
19         registry.freeMarker();
20     }
21
22     @Bean
```

```
21     public FreeMarkerConfigurer freeMarkerConfigurer() {  
22  
23         FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();  
24         // 设置模板路径  
25         configurer.setTemplateLoaderPath("classpath:/templates");  
26         return configurer;  
27     }  
28 }  
29 }
```

渲染这些模板是一项CPU密集型操作。

如果我们有一个庞大的数据集，执行该操作可能需要一些时间和内存。

Thymeleaf 支持响应式WebFlux，并为异步和流模板渲染提供更多可能性。

Thymeleaf 提供与FreeMarker类似的功能，并允许编写相同的代码来呈现UI。

Thymeleaf 能够将响应式类型用作模板内的数据源，并在流中的新元素可用时呈现模板的一部分。

以下示例展示了在处理请求期间如何将响应式流与Thymeleaf一起使用：

```
1 @RequestMapping("/play-list-view-thy")  
2 public String view(final Model model) {  
3     List<Song> songs = new ArrayList<>();  
4     Song song = null;  
5     for (int i = 0; i < 10; i++) {  
6         song = new Song("曲目" + i, "张三" + i, "1001" + i, "专辑1" + (i %  
3));  
7         songs.add(song);  
8     }  
9     final Flux<Song> playlistStream = Flux.fromIterable(songs);  
10    model.addAttribute(  
11        "playlist",  
12        new ReactiveDataContextVariable(playlistStream, 1, 1)  
13    );  
14    return "thymeleaf/play-list-view";  
15 }
```

`ReactiveDataContextVariable` 用于创建懒加载上下文变量，封装异步响应式数据流，并指定缓冲区大小（单位是消息个数），第一个SSE事件ID。

接收的响应式类型，如Publisher、Flux、Mono、Observable 和由ReactiveAdapterRegistry类支持的其他响应式类型。

响应式支持需要流使用额外的类包装器，模板端不需要任何更改。

以下示例展示了如何使用与处理普通集合类似的方式处理响应式流：

src/main/resources/templates/thymeleaf/play-list-view.html:

```

1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <body>
4 <head>
5   <meta charset="UTF-8"/>
6   <title>曲目列表</title>
7 </head>
8 <table border="1">
9   <thead>
10  </thead>
11  <tbody>
12    <tr th:each="e : ${playList}">
13      <td th:text="${e.id}">...</td>
14      <td th:text="${e.name}">...</td>
15      <td th:text="${e.artist}">...</td>
16      <td th:text="${e.album}">...</td>
17    </tr>
18  </tbody>
19 </table>
20 </body>
21 </html>

```

生成一个表，带有一些表头和一个正文。该表是由 Song 条目的 playList 和它们的信息构成的行所填充的。

Thymeleaf 的渲染引擎开始将数据流传输到客户端，而不必等待最后一个元素被发射。

它支持渲染无限的元素流。这可以通过添加对 Transfer-Encoding:chunked 的支持来实现。

Thymeleaf 不会渲染内存中的整个模板，而会首先渲染可用的部分，然后在新元素可用时以块的形式异步发送模板的其余部分。

## 10.6 响应式Web安全

Spring Web的SpringSecurity 模块通过在任何控制器和Web 处理程序调用之前**提供Filter 来设置安全的Web应用程序**。

为了支持响应式和非阻塞交互，Spring Security的**响应式栈**，使用**WebFilter并依赖Project Reactor上下文**。

pom.xml文件：

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-security</artifactId>
5   </dependency>
6   <dependency>

```

```

7      <groupId>org.springframework.boot</groupId>
8      <artifactId>spring-boot-starter-thymeleaf</artifactId>
9  </dependency>
10 <dependency>
11     <groupId>org.springframework.boot</groupId>
12     <artifactId>spring-boot-starter-webflux</artifactId>
13 </dependency>
14 <dependency>
15     <groupId>org.projectlombok</groupId>
16     <artifactId>lombok</artifactId>
17 </dependency>
18 <dependency>
19     <groupId>org.springframework.boot</groupId>
20     <artifactId>spring-boot-starter-test</artifactId>
21     <scope>test</scope>
22 </dependency>
23 <dependency>
24     <groupId>io.projectreactor</groupId>
25     <artifactId>reactor-test</artifactId>
26     <scope>test</scope>
27 </dependency>
28 <dependency>
29     <groupId>org.springframework.security</groupId>
30     <artifactId>spring-security-test</artifactId>
31     <scope>test</scope>
32 </dependency>
33 </dependencies>
```

## SecurityProfileController.java

```

1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.Profile;
4 import com.lagou.webflux.demo.service.ProfileService;
5 import
6     org.springframework.security.core.context.ReactiveSecurityContextHolder;
7 import org.springframework.security.core.context.SecurityContext;
8 import org.springframework.web.bind.annotation.GetMapping;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11 import reactor.core.publisher.Mono;
12
13 @RestController
14 @RequestMapping("/api/v1")
15 public class SecurityProfileController {
16
17     private final ProfileService profileService;
18
19     public SecurityProfileController(ProfileService profileService) {
20         this.profileService = profileService;
21     }
22
23     @GetMapping("/profiles")
24     public Mono<Profile> getProfile() {
25         return ReactiveSecurityContextHolder
```

```
25         .getContext() // 访问当前的SecurityContext
26         .map(SecurityContext::getAuthentication) // 从
27         SecurityContext获取认证信息
28         .flatMap(auth -> profileService.getByUser(auth.getName()));
29     // 访问用户个人信息
30 }
```

### Profile.java

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Getter;
5 import lombok.NoArgsConstructor;
6
7 @AllArgsConstructor
8 @NoArgsConstructor
9 @Getter
10 public class Profile {
11
12     private String name;
13     private String desc;
14 }
15 }
```

### ProfileService.java

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Profile;
4 import reactor.core.publisher.Mono;
5
6 public interface ProfileService {
7
8     Mono<Profile> getByUser(String name);
9 }
```

### DefaultProfileService.java

```
1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Profile;
4 import com.lagou.webflux.demo.service.ProfileService;
5 import org.springframework.stereotype.Service;
6 import reactor.core.publisher.Mono;
7
8 @Service
9 public class DefaultProfileService implements ProfileService {
10     @Override
```

```
11     public Mono<Profile> getByUser(String name) {
12         return Mono.just(new Profile(name, "这是【" + name + "】的描述信
息。"));
13     }
14 }
```

## SecurityConfiguration.java

```
1 package com.lagou.webflux.demo.config;
2
3 import org.springframework.boot.SpringBootConfiguration;
4 import org.springframework.context.annotation.Bean;
5 import
6 org.springframework.security.config.annotation.method.configuration.EnableRe
activeMethodSecurity;
7 import
8 org.springframework.security.config.web.server.ServerHttpSecurity;
9 import
10 org.springframework.security.core.userdetails.MapReactiveUserDetailsService;
11 import
12 org.springframework.security.core.userdetails.ReactiveUserDetailsService;
13 import
14 org.springframework.security.core.userdetails.User;
15 import org.springframework.security.core.userdetails.UserDetails;
16 import org.springframework.security.web.server.SecurityWebFilterChain;
17
18 @SpringBootConfiguration
19 // 该注解导入所需的配置，以启用特定的带有注解MethodInterceptor
20 @EnableReactiveMethodSecurity
21 public class SecurityConfiguration {
22
23     @Bean
24     public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity
httpSecurity) {
25         return httpSecurity
26             .formLogin() // 配置基于表单的认证
27             .and() // 接着向下配置
28             .authorizeExchange() // 配置授权
29             .anyExchange() // 禁用授权
30             .authenticated() // 需要有认证的用户
31             .and() // 接着向下配置
32             .build(); // 创建web安全过滤器链对象
33     }
34
35     @Bean
36     public ReactiveUserDetailsService userDetailsService() {
37
38         User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
39         UserDetails user = userBuilder.username("zhangsan")
40             .password("zs123")
41             .roles("emp")
42             .build();
43         UserDetails user1 = userBuilder.username("lisi")
44             .password("ls123")
45             .roles("mgr")
46             .build();
47     }
48 }
```

```
43     return new MapReactiveUserDetailsService(user, user1);
44 }
45 }
```

## 对SecurityContext 的响应式访问

使用Spring Security中的 `ReactiveSecurityContextHolder` 访问响应式 `SecurityContext`。

`ReactiveSecurityContextHolder` 的 `getContext` 方法返回 `Mono<SecurityContext>`。

```
1 @GetMapping("/profiles")
2 @PreAuthorize("hasRole('emp')")
3 public Mono<Profile> getProfile() {
4     return ReactiveSecurityContextHolder
5         .getContext() // 访问当前的SecurityContext
6         .map(SecurityContext::getAuthentication) // 从SecurityContext获取
7             认证信息
8             .flatMap(auth -> profileService.getByUser(auth.getName())); // 访
9                 问用户个人信息
10 }
```

`@PreAuthorize` 注解用于访问控制，检查 `Authentication` 是否具有所需的角色。

如果是响应式返回类型，则方法调用推迟，直到所需的 `Authentication` 解析完并存在所需的权限。

新的响应式API 类似于同步API。

基于新一代的Spring Security，可以使用相同的注解来检查所需的权限。

在内部，`ReactiveSecurityContextHolder` 依赖于Reactor Context API。

有关登录用户的当前信息保存在Context 接口的实例中。

如下源码：

```
42     */
43     public static Mono<SecurityContext> getContext() {
44         // @formatter:off ProjectReactor的subscriberContext
45         return Mono.subscriberContext()
46             .filter(ReactiveSecurityContextHolder::hasSecurityContext)
47             .flatMap(ReactiveSecurityContextHolder::getSecurityContext);
48     } // @formatter:on
49 }
50 // 判断是否包含安全上下文
51 @ private static boolean hasSecurityContext(Context context) {
52     return context.hasKey(SECURITY_CONTEXT_KEY);
53 }
54 // 获取对应的安全上下文
55 @ private static Mono<SecurityContext> getSecurityContext(Context context) {
56     return context.<Mono<SecurityContext>>get(SECURITY_CONTEXT_KEY);
57 }
```

可以使用 subscriberContext 访问内部 Reactor Context。

执行过程与获取存储（如数据库）的 SecurityContext 有关，该过程仅在某人订阅给定的 Mono 时执行。

尽管 ReactiveSecurityContextHolder 的 API 看起来很熟悉，但它隐藏了许多陷阱。

例如，可能错误地遵循使用 SecurityContextHolder 时的惯用法。这样一来，可能盲目地实现以下示例代码中描述的常见交互：

```
1 | ReactiveSecurityContextHolder
2 |     .getContext()
3 |     .map(SecurityContext::getAuthentication)
4 |     .block();
```

就像以前从 ThreadLocal 中获取 SecurityContext 一样，可能尝试使用 ReactiveSecurityContextHolder 执行相同的操作。但是，当调用 getContext 并使用 block 方法订阅流时，我们在流中配置的是一个空的上下文。

因此，一旦 ReactiveSecurityContextHolder 类尝试访问内部 Context，就不会在那里找到可用的 SecurityContext。

当我们正确连接流时，如何设置 Context 并使其可访问？Spring Security 中的 `ReactorContextWebFilter`。

在调用期间，`ReactorContextWebFilter` 使用 subscriberContext 方法提供一个 Reactor Context。

此外，`SecurityContext` 的解析是通过 `ServerSecurityContextRepository` 来执行的。

ServerSecurityContextRepository 有两个方法，分别是 save 和 load：

```
1 package org.springframework.security.web.server.context;
2
3 public interface ServerSecurityContextRepository {
4
5     Mono<Void> save(ServerWebExchange exchange, SecurityContext context);
6
7     Mono<SecurityContext> load(ServerWebExchange exchange);
8 }
```

如上代码，save 方法将 `SecurityContext` 与特定的 `ServerWebExchange` 关联，然后使用来自用户请求的 `load` 方法将其还原。

响应式访问意味着实际的 `SecurityContext` 可以存储在数据库中，因此解析存储的 `SecurityContext` 不需要阻塞操作。因为上下文解析的策略是惰性的，所以只有在订阅 `ReactiveSecurityContextHolder.getContext()` 时才会执行对底层存储的实际调用。

基于 `SecurityContext` 的传输机制，可以轻松构建复杂的流处理，而不必关注 Thread 实例之间常见的 ThreadLocal 传播问题。

## 启用响应式安全性

基于 WebFlux 的安全性配置只需要声明少量 bean。

以下是我们如何执行此操作的参考示例：

```
1 @SpringBootConfiguration
2 // 该注解导入所需的配置，以启用特定的带注解MethodInterceptor
3 @EnableReactiveMethodSecurity
4 public class SecurityConfiguration {
5
6     @Bean
7     public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity
8 httpSecurity) {
9         return httpSecurity
10            .formLogin() // 配置基于表单的认证
11            .and() // 接着向下配置
12            .authorizeExchange() // 配置授权
13            .anyExchange() // 禁用授权
14            .authenticated() // 需要有认证的用户
15            .and() // 接着向下配置
16            .build(); // 创建web安全过滤器链对象
17
18     @Bean
19     public ReactiveUserDetailsService userDetailsService() {
```

```
20     User.UserBuilder userBuilder = User.withDefaultPasswordEncoder();
21
22     UserDetails user = userBuilder.username("zhangsan")
23         .password("zs123")
24         .roles("emp")
25         .build();
26
27     UserDetails user1 = userBuilder.username("lisi")
28         .password("ls123")
29         .roles("mgr")
30         .build();
31
32
33     return new MapReactiveUserDetailsService(user, user1);
34 }
35 }
```

为了启用特定的带注解的 `MethodInterceptor`，必须添加 `@EnableReactiveMethodSecurity` 注解，导入所需的配置。

Spring Security为我们提供了`ServerHttpSecurity`，它是一个带有流式API 的构建器。

为了在默认的Spring Security 设置中对用户进行身份验证，必须提供`ReactiveUserDetailsService` 的实现。

出于演示目的，提供了接口的内存实现，并配置一个测试用户以登录系统。

如上述代码所示，Spring Security 的整体配置与之前的类似。

## 10.7 与其他响应式库的交互

WebFlux 使用Project Reactor 3 作为核心构建块，同时，WebFlux 也允许使用其他响应式库。

为了实现跨库互操作性，WebFlux 中的大多数操作基于**响应式流规范**中的接口。

通过这种方式，我们可以轻松地用 RxJava 2 或 Akka Streams 替换Reactor 3 所编写的代码。

```
1 | curl -H "Content-Type: application/json" -d '张三' --request POST
  | http://localhost:8080/songs
```

核心代码如下：

```
1 @PostMapping("/songs")
2 public Observable<Song> findAlbumByArtist(@RequestBody Mono<String>
3 nameMono) {
4     // 将Mono转换为Observable类型
5     Observable<String> observable = (Observable<String>)
6     adapterRegistry.getAdapter(Observable.class).fromPublisher(nameMono);
7     return observable.flatMap(new Function<String, ObservableSource<Song>>()
8     {
9         @Override
10        public ObservableSource<Song> apply(String s) throws Exception {
11            return Observable.just(new Song("五环之歌", s));
12        }
13    });
14 }
```

首先，导入 RxJava 2 的 `observable`。

`findAlbumByArtists` 接受一个 `Mono<String>` 类型的 `Publisher` 并返回 `observable<Song>`。

声明将 `artistsFlux` 映射到 `observable<string>`，执行业务逻辑，并将结果返回给调用者。

响应式类型转换是 Spring Core 模块的一部分，并受 `org.springframework.core.ReactiveAdapterRegistry` 和 `org.springframework.core.ReactiveAdapter` 的支持。

因此，通过使用该支持库，可以使用几乎任何响应式库，与 Project Reactor 解耦。

具体案例：

pom.xml

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-webflux</artifactId>
5     </dependency>
6     <dependency>
7         <groupId>io.reactivex.rxjava2</groupId>
8         <artifactId>rxjava</artifactId>
9         <version>2.2.20</version>
10    </dependency>
11    <dependency>
12        <groupId>org.projectlombok</groupId>
13        <artifactId>lombok</artifactId>
14    </dependency>
15    <dependency>
16        <groupId>org.springframework.boot</groupId>
17        <artifactId>spring-boot-starter-test</artifactId>
18        <scope>test</scope>
19    </dependency>
20    <dependency>
21        <groupId>io.projectreactor</groupId>
```

```
22     <artifactId>reactor-test</artifactId>
23     <scope>test</scope>
24   </dependency>
25 </dependencies>
```

### AlbumsController.java

```
1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.Song;
4 import io.reactivex.Observable;
5 import io.reactivex.ObservableSource;
6 import io.reactivex.functions.Function;
7 import org.springframework.core.ReactiveAdapterRegistry;
8 import org.springframework.web.bind.annotation.PostMapping;
9 import org.springframework.web.bind.annotation.RequestBody;
10 import org.springframework.web.bind.annotation.RestController;
11 import reactor.core.publisher.Mono;
12
13 @RestController
14 public class AlbumsController {
15
16     final ReactiveAdapterRegistry adapterRegistry;
17
18     public AlbumsController(ReactiveAdapterRegistry adapterRegistry) {
19         this.adapterRegistry = adapterRegistry;
20     }
21
22     @PostMapping("/songs")
23     public Observable<Song> findAlbumByArtist(@RequestBody Mono<String>
nameMono) {
24
25         // 将Mono转换为Observable类型
26         Observable<String> observable = (Observable<String>)
adapterRegistry.getAdapter(Observable.class).fromPublisher(nameMono);
27
28         return observable.flatMap(new Function<String,
29             ObservableSource<Song>>() {
30             @Override
31             public ObservableSource<Song> apply(String s) throws Exception {
32                 return Observable.just(new Song("五环之歌", s));
33             }
34         });
35     }
36 }
37 }
```

### Song.java

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Getter;
5
6 @AllArgsConstructor
7 @Getter
8 public class Song {
9
10     private String name;
11     private String artistName;
12 }
```

index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Title</title>
6     <script type="text/javascript" src="jquery-1.9.1.js"></script>
7     <script type="text/javascript">
8
9         $(function () {
10             $("#btn").click(function () {
11
12                 if ($("#artistName").val() == '') {
13                     alert("请填写歌手名称")
14                     $("#artistName").focus()
15                     return
16                 }
17
18                 $.ajax({
19                     url: "/songs",
20                     data: $("#artistName").val(),
21                     contentType: "application/json; charset=utf-8",
22                     cache: false,
23                     method: "post",
24                     success: function (data) {
25                         alert(JSON.stringify(data))
26                     },
27                     error: function () {
28                         alert("通信异常")
29                     }
30                 })
31             })
32         })
33     </script>
34 </head>
35 <body>
36 <input type="text" id="artistName" placeholder="请输入歌手名称" >
37 <input id="btn" type="button" value="获取数据">
38
39 </body>
40 </html>
```

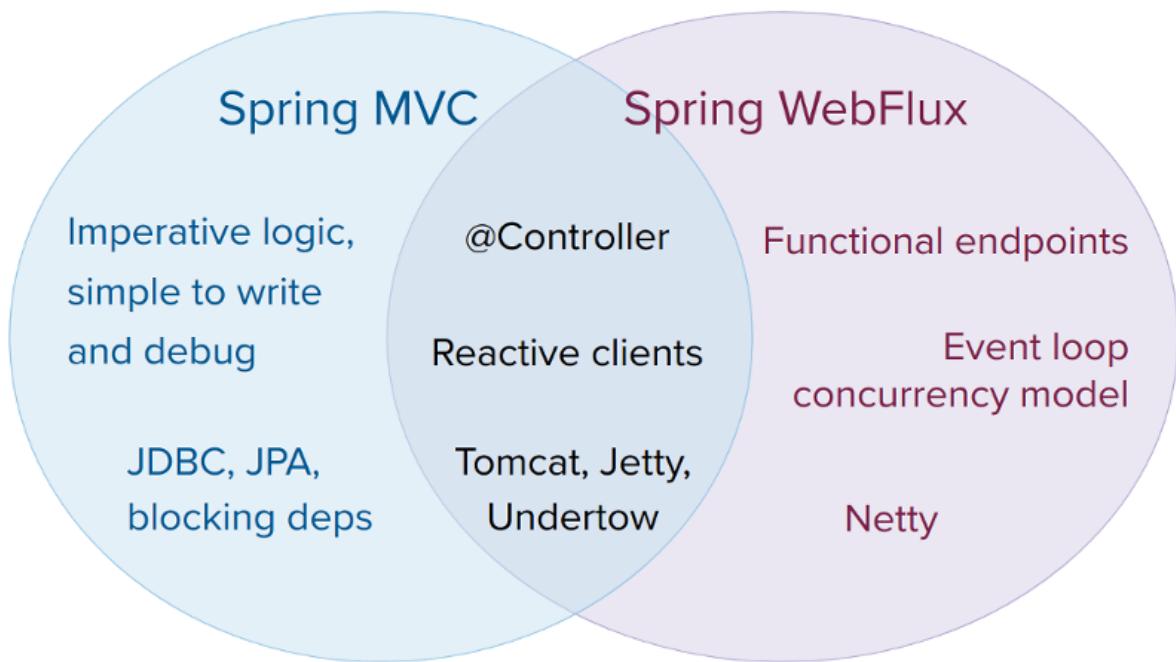
## 11 SpringWebFlux和SpringWebMVC对比

使用Spring MVC还是WebFlux?

Spring MVC和Spring WebFlux并不是分立的。它们都扩展了开发的可用选项。

两者设计的目标就是彼此的连续性和一致性，可以一起使用，发挥各自优势。

下图展示了两者的联系和区别：



具体如何使用，考虑如下：

- 如果现存的项目是基于Spring MVC的并且没有问题，就别更改了。命令式编程开发、阅读、debug都是最简单的。同时可选择的库也很多，只不过大多数都是阻塞式的。
- 如果项目的技术栈是非阻塞的，则使用WebFlux可以使用与环境相同的模型来执行，WebFlux也提供了服务器的选项（Netty、Tomcat、Jetty、Undertow以及Servlet 3.1及以上的容器），提供了编程模型的选项（基于注解的控制器和函数式web端点），以及响应式库的选项（Reactor、RxJava以及其他）。
- 如果希望发挥java8 lambda或Kotlin的优势，使用轻量级、函数式的web框架，则可以使用Spring WebFlux函数式web端点的编程模型。Spring WebFlux非常适合小型的应用或没有复杂需求的微服务。
- 在微服务架构中，可以同时使用Spring WebFlux和Spring MVC，或者将Spring WebFlux作为函数式端点来使用。由于它们基于相同的注解编程模型，可以很方便的做到在正确的场合使用正确的工具。
- 一个简单的评估应用的方式是检查应用的依赖。如果使用的是阻塞式的持久化API（JPA, JDBC）或者阻塞式的网络API，Spring MVC基本上是最好的选择。技术上Reactor和RxJava也可以使用分立的线程支持阻塞式的操作，但无法发挥非阻塞web技术栈的全部优势。
- 如果Spring MVC应用需要调用远程服务，可以使用响应式的 webClient。可以让Spring MVC控制器的方法直接返回响应式类型（Reactor、RxJava或其他的）数据。每个远程调用的延迟

越大，各个远程调用之间的依赖越大，响应式模型的优势发挥的越明显。当然，Spring MVC 的控制器也可以调用其他响应式组件。

- 如果开发团队很大，就要考虑到转向非阻塞、函数式、声明式编程模型的陡峭的学习曲线。最佳实践是先使用响应式 `webClient` 做部分转向。然后在小的模块中使用并评估响应式模型带来的优势。一般对于整个项目，没必要全部转向响应式模型。如果不確定响应式编程带来的优势，可以先学习一下非阻塞I/O的工作流程（例如单线程Node.js的并发）以及效果。

## 12 使用SpringBoot

### 12.1 Spring Core中的响应式

Spring 生态系统的核心模块是Spring Core 模块。

Spring 5.x 引入对响应式流和响应式库的原生支持，其中，响应式库包含RxJava 1/2 和Project Reactor 3。

#### 12.1.1 响应式类型转换支持

为了支持响应式流规范所进行的最全面的改进之一是引入了 `ReactiveAdapter` 和 `ReactiveAdapterRegistry`。

`ReactiveAdapter` 类为响应式类型转换提供了两种基本方法，用于将任何类型转换为 `Publisher<T>` 并将其转换回 `Object`。

如以下源码所示：

```
org.springframework.core.ReactiveAdapter#toPublisher

107      @SuppressWarnings("unchecked")
108  ⏺ @     public <T> Publisher<T> toPublisher(@Nullable Object source) {
109      ⏺     if (source == null) {
110          ⏺         source = getDescriptor().getEmptyValue();
111      ⏺     }
112      ⏺     return (Publisher<T>) this.toPublisherFunction.apply(source);
113  ⏺ }
```

```
org.springframework.core.ReactiveAdapter#fromPublisher

120      public Object fromPublisher(Publisher<?> publisher) {
121          ⏺     return this.fromPublisherFunction.apply(publisher);
122  ⏺ }
```

如，为了提供对RxJava 2 中的 `Maybe` 响应式类型的转换，我们可以通过以下方式创建自己的 `ReactiveAdapter`：

```

2  * 适配器的构造器，包含了异步类型或响应式类型与Reactive Streams Publisher之间的相互转换。
3  *
4  * @param descriptor          响应式类型描述符
5  * @param toPublisherFunction 转换到Publisher的适配器
6  * @param fromPublisherFunction 从Publisher转换的适配器
7  */
8  public MaybeReactiveAdapter(ReactiveTypeDescriptor descriptor,
9                  Function<Object, Publisher<?>>
10                 toPublisherFunction,
11                 Function<Publisher<?>, Object>
12                 fromPublisherFunction) {
13     super(descriptor, toPublisherFunction, fromPublisherFunction);
14 }
15
16 public MaybeReactiveAdapter() {
17     super(
18         ReactiveTypeDescriptor.singleOptionalValue(Maybe.class,
19             Maybe::empty),
20         rawMaybe -> ((Maybe<?>) rawMaybe).toFlowable(),
21         publisher -> Flowable.fromPublisher(publisher).singleElement()
22     );
23 }
```

上面实例中，扩展了默认的 `ReactiveAdapter` 并提供了一个自定义实现。

父构造函数的第一个参数是 `ReactiveTypeDescriptor` 实例的定义。

`ReactiveTypeDescriptor` 提供了有关 `ReactiveAdapter` 中使用的响应式类型的信息。

父构造函数需要定义转换函数，而该函数将原始对象（`Maybe`）转换为 `Publisher` 并将任何 `Publisher` 转换回 `Maybe`。

为简化交互，`ReactiveAdapterRegistry` 使我们能将 `ReactiveAdapter` 的实例保存在一个位置，并提供对它们的通用访问。

如以下代码所示：

```

1  ReactiveAdapterRegistry.getSharedInstance().registerReactiveType(
2      ReactiveTypeDescriptor.singleOptionalValue(Maybe.class,
3          Maybe::empty),
4          rawMaybe -> ((Maybe<?>) rawMaybe).toFlowable(),
5          publisher -> Flowable.fromPublisher(publisher).singleElement()
6      );
7  // 后续使用的时候直接通过共享实例访问
8  ReactiveAdapter maybeAdapter = ReactiveAdapterRegistry.getSharedInstance()
9      .getAdapter(Maybe.class);
```

如代码所示，`ReactiveAdapterRegistry` 表示针对不同响应式类型的`ReactiveAdapter`实例的公共池。同时，`ReactiveAdapterRegistry` 提供了一个单例实例，该实例既可以在框架内的许多地方使用，也可以在开发的应用程序中使用。

## 12.1.2 响应式IO

Spring Core 模块在 byte 缓冲区实例上引入了一个称为 `DataBuffer` 的抽象。

之所以避免使用 `java.nio.ByteBuffer`，主要是为了提供一个既可以支持不同字节缓冲区，又不需要在它们之间进行任何额外的转换的抽象。

例如，为了将 `io.netty.buffer.ByteBuf` 转换为 `ByteBuffer`，必须访问所存储的字节，而这些字节可能需要从堆外空间被拉入到堆中。这可能破坏Netty 提供的高效内存使用和缓冲区回收（重用相同的字节缓冲区）。

Spring DataBuffer 提供特定实现的抽象，能以通用方式使用底层实现。

`DataBuffer` 的 `PooledDataBuffer` 子接口，还启用了引用计数功能，并支持开箱即用的高效内存管理。

此外，Spring Core 的第五版引入了 `DataBufferUtils` 类，能以响应式流的形式与 I/O 进行交互（与网络、资源、文件等交互）。

例如，可以基于背压支持并通过以下响应式的方式读取文件内容：

```
1 Flux<DataBuffer> reactiveHamlet = DataBufferUtils.read(  
2     new DefaultResourceLoader().getResource("lagou.txt"),  
3     new DefaultDataBufferFactory(),  
4     1024  
5 );
```

`DataBufferUtils.read` 返回一个 `DataBuffer` 实例的 Flux。因此，可以使用 Reactor 的所有功能读取文件内容。

最后，与Spring Core 中响应式相关的最后一个意义重大且不可或缺的特性是响应式编解码器 (reactive codecs)。

响应式编解码器提供了一种将 `DataBuffer` 实例流 和 对象流 进行相互转换的简便方式。

`Encoder` 和 `Decoder` 接口即用于此目的，并提供以下用于编码/解码数据流的 API：

```
5             ResolvableType elementType,
6             @Nullable MediaType mimeType,
7             @Nullable Map<String, Object> hints);
8 }
9
10 interface Decoder<T> {
11
12     Flux<T> decode(Publisher<DataBuffer> inputStream,
13                     ResolvableType elementType,
14                     @Nullable MediaType mimeType,
15                     @Nullable Map<String, Object> hints);
16
17     Mono<T> decodeToMono(Publisher<DataBuffer> inputStream,
18                           ResolvableType elementType,
19                           @Nullable MediaType mimeType,
20                           @Nullable Map<String, Object> hints);
21 }
```

两个接口都与响应式流中的 Publisher 一起运行，并能将 DataBuffer 实例流 编码/解码 为对象。

它以非阻塞的方式，将序列化数据转换为java对象，将java对象序列化为序列化数据。

这种编码/解码数据的方式可以减少处理延迟，这是因为响应式流在本质上支持独立的元素处理，而不必等到最后一个字节才开始解码整个数据集。

## 12.2 响应式Web

Spring Boot 2 引入了 WebFlux，支持高吞吐量、低延迟。

Spring WebFlux建立在 响应式流适配器 之上，可以与 Netty 和 Undertow 以及基于 Servlet 3.1 的传统服务器等集成。

Spring WebFlux 作为非阻塞的基础，将响应式流作为业务逻辑代码和服务器交互的中心抽象。

注意，Servlet API 3.1 的适配器提供了与Web MVC 适配器不同的纯异步和非阻塞集成。

Spring Web MVC 模块也支持Servlet API 4.0，后者支持HTTP/2。

Spring WebFlux 将 Project Reactor 3 作为一等公民并广泛使用。

响应式编程可以开箱即用，还可以在 Netty 上运行Web 应用程序。

Spring WebFlux 模块提供内置的背压支持，可以确保 I/O 不会变得不堪重负。

Spring WebFlux 的 WebClient 类，实现非阻塞的客户端交互。

旧Web MVC 模块还获得了对响应式流的一些支持。

从框架的第五版开始，Servlet API 3.1 成为 Web MVC 模块的基线。意味着 Web MVC 现在支持 Servlet 规范提出的非阻塞I/O。Web MVC 模块的设计在适当级别的非阻塞 I/O 方面没有太大变化。

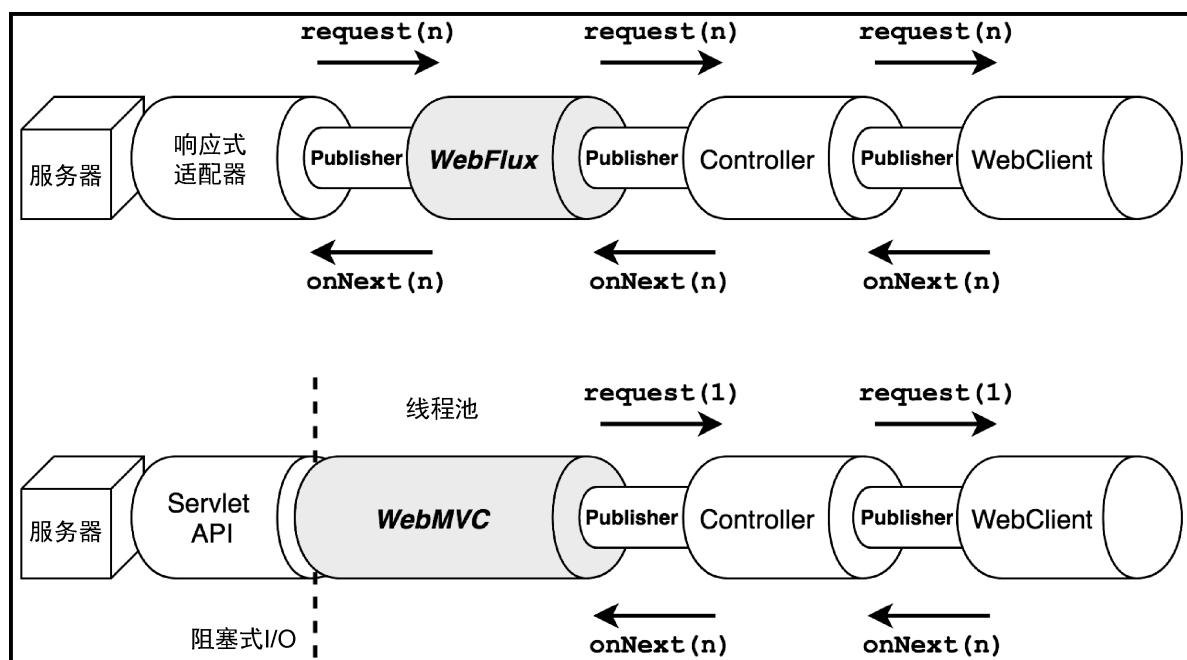
尽管如此，Servlet 3.0 的异步行为已经正确实现了一段时间。Spring Web MVC 为 `ResponseBodyEmitterReturnValueHandler` 类提供了升级。

由于Publisher 类可能被视为无限的事件流，因此在不破坏 Web MVC 模块的整体基础结构的情况下，Emitter 处理程序是放置响应式处理逻辑的适当位置。为此，Web MVC 模块引入了 `ReactiveTypeHandler` 类，它负责正确处理 Flux 和 Mono 等响应式类型。

为了在客户端获得非阻塞行为，除了支持服务器端响应式类型的变更，还可以使用 WebFlux 模块所提供的 `webClient`。

Spring Boot 可以提供基于类路径中可用类的复杂环境管理行为。因此，通过提供 Web MVC (spring-boot-starter-web) 模块以及 WebFlux，我们可以从 WebFlux 模块获得 Web MVC 环境和非阻塞响应式 WebClient。

最后，当将这两个模块作为响应式管道进行比较时，得到的结构如下图所示：



在 Web MVC 或 WebFlux 这两种用法中，得到了几乎相同的基于响应式流的编程模型。

这两个模块之间的显著差异之一是 Web MVC 需要在与源自旧模块设计的 Servlet API 集成时进行**阻塞式写入或阻塞式读取**。该缺陷导致响应式流内的相互作用模型退化，使其降级为普通的拉模型。

同时，Web MVC 现在使用内部专用于所有阻塞读/写的线程池。因此，应该合理配置它以避免意外行为。

WebFlux 通信模型取决于网络吞吐量以及可定义其自身控制流的底层传输协议。

总而言之，Spring 5 引入了一个强大的工具，用于使用响应式流规范和 Project Reactor 构建响应式非阻塞应用程序。

此外，Spring Boot 支持强大的依赖管理和自动配置，可以保护我们免受依赖地狱的侵害。

## 12.3 响应式Spring Data

Spring Data 主要提供对底层存储区域的同步阻塞访问。

现在，Spring Data 框架提供了 `ReactiveCrudRepository` 接口，该接口暴露了Project Reactor 的响应式类型。

Spring Data 还提供了几个通过扩展 `ReactiveCrudRepository` 接口而与存储方法集成的模块。

- 基于 Spring Data Mongo 响应式模块的 MongoDB：与NoSQL 数据库之间的完全响应式非阻塞交互，同时也包含背压控制。
- 基于 Spring Data Cassandra 响应式模块的 Cassandra：与Cassandra 数据存储的异步非阻塞交互，支持基于TCP 流控制的背压。
- 基于 Spring Data Redis 响应式模块的 Redis：通过Lettuce Java 客户端实现的与 Redis 之间的响应式集成。
- 基于 Spring Data Couchbase 响应式模块的 Couchbase：通过基于 RxJava 的驱动程序实现的与 Couchbase 数据库之间的响应式Spring Data 集成。

此外，Spring Boot 提供了额外的启动器模块，可以与所选的存储方法实现平滑集成。

除了NoSQL 数据库，Spring Data 还引入了 Spring Data JDBC，与 JDBC 轻量级集成，快速提供响应式 JDBC 连接。

其他 Spring 框架模块的大多数改进以 WebFlux 的响应式能力或响应式 Spring Data 模块为基础。

## 12.4 响应式Spring Session

Spring 框架中与Spring Web 模块相关的另一个重要更新是 Spring Session 模块中的响应式支持。

Spring Session 引入了 `ReactiveSessionRepository`，可以使用 Reactor 的 Mono 类型对存储的会话进行异步非阻塞访问。

除此之外，作为响应式 Spring Data 的会话存储，Spring Session 还提供与 Redis 的响应式集成。

可以通过包含以下依赖项来实现分布式 WebSession：

```
1 <dependency>
2   <groupId>org.springframework.session</groupId>
3   <artifactId>spring-session-data-redis</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-webflux</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.springframework.boot</groupId>
11  <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
12 </dependency>
```

为了实现响应式 Redis WebSession 管理，必须将这3个依赖项组合在一个地方。

同时，Spring Boot 负责提供 bean 的精确组合并生成合适的自动配置，以便顺利地运行 Web 应用程序。

## 12.5 响应式 Spring Security

旧的 Spring Security 使用 **ThreadLocal** 作为 SecurityContext 实例的存储方法。在单个 Thread 内执行时，该技术很有效，在任何时候，都可以访问存储在 ThreadLocal 中的 SecurityContext。

但是，在执行异步通信时，该技术就会出现问题。这时，必须提供额外的工作来将 ThreadLocal 内容传输到另一个 Thread，并为 Thread 实例之间的每个切换实例执行此操作。

尽管 Spring 框架通过使用一个额外的 ThreadLocal 扩展简化了 Threads 之间的 SecurityContext 传输，但在基于 Project Reactor 或类似的响应式库应用响应式编程范例时，仍然会有问题。

**新一代 Spring Security 采用了 Reactor 上下文功能**，以便在 Flux 或 Mono 流中传输安全上下文。通过这种方式，即使在运作着不同执行线程的复杂响应式流中，我们也可以安全地访问安全上下文。

## 12.6 响应式 Spring Cloud

首先，响应式影响了分布式的入口点，即网关（gateway）。

很长一段时间，唯一能够将应用程序作为网关运行的 Spring 模块是 Spring Cloud Netflix Zuul 模块。Netflix Zuul 基于使用阻塞同步请求路由的 Servlet API。使处理请求获得更好性能的唯一方法是调整底层服务器线程池。

这种模型的伸缩性无法与响应式方法相比。

Spring Cloud 引入了新的 Spring Cloud Gateway 模块，该模块构建于 Spring WebFlux 之上，并在 Project Reactor 3 的支持下提供异步和非阻塞路由。

除了新的网关模块，Spring Cloud Streams 还获得了Project Reactor 的支持，并且引入了更加细粒度的流模型。

为了简化响应式系统的开发，Spring Cloud 引入了一个名为 Spring Cloud Function 的新模块，该模块旨在为构建我们自己的函数即服务（function as a service，FaaS）解决方案提供必要的组件。

如果没有适当的附加基础设施，Spring Cloud Function 模块将无法应用在普通开发中。Spring Cloud Data Flow 不仅提供了这种可能性，还包含了 Spring Cloud Function 的部分功能。

## 12.7 响应式Spring Test

Spring 生态系统提供了改进后的 Spring Test 和 Spring Boot Test 模块，它们扩展了一系列用于测试响应式 Spring 应用程序的附加功能。

Spring Test 提供了一个 `webTestClient` 来测试基于 WebFlux 的 Web 应用程序，同时，Spring Boot Test 使用普通的注解来处理测试套件的自动配置。

同时，为了测试响应式流的 Publisher，Project Reactor 提供了Reactor-Test 模块，它与 Spring Test 和 Spring Boot Test 模块相结合，可以为使用响应式 Spring 实现的业务逻辑编写完整的验证套件。

## 12.8 响应式监控

基于 Project Reactor 和响应式 Spring 框架构建的面向生产的响应式系统**应该暴露所有重要的运维指标**。

首先，Project Reactor 本身具有内置指标。它提供 `Flux#metrics()` 方法，可以跟踪响应式流中的不同事件。

Spring 框架生态系统提供了更新后的 Spring Boot Actuator 模块，该模块支持应用程序监控和故障排除的主要指标。

新一代SpringActuator 提供与 WebFlux 的完全集成，并使用其异步、非阻塞编程模型，以便有效地暴露指标端点。

Spring Cloud Sleuth 模块提供了监控和跟踪应用程序的最终选项。该模块提供开箱即用的分布式跟踪，它的一个显著优点是支持 Project Reactor 的响应式编程，因此应用程序中的所有响应式工作流都可以被正确跟踪。

Spring 生态系统不仅改进了内核框架的响应性，还负责面向生产的功能，而且支持详细的应用程序监控（这种监控甚至包括这些功能的响应式解决方案）。

## 13 WebFlux的应用

### 13.1 基于微服务的系统

WebFlux 的第一个应用是微服务系统。

微服务系统最显著的特点是**大量的I/O 通信**。

I/O 的存在，尤其是阻塞式I/O，会降低整体系统延迟和吞吐量。

#### 13.1.1 微服务网关

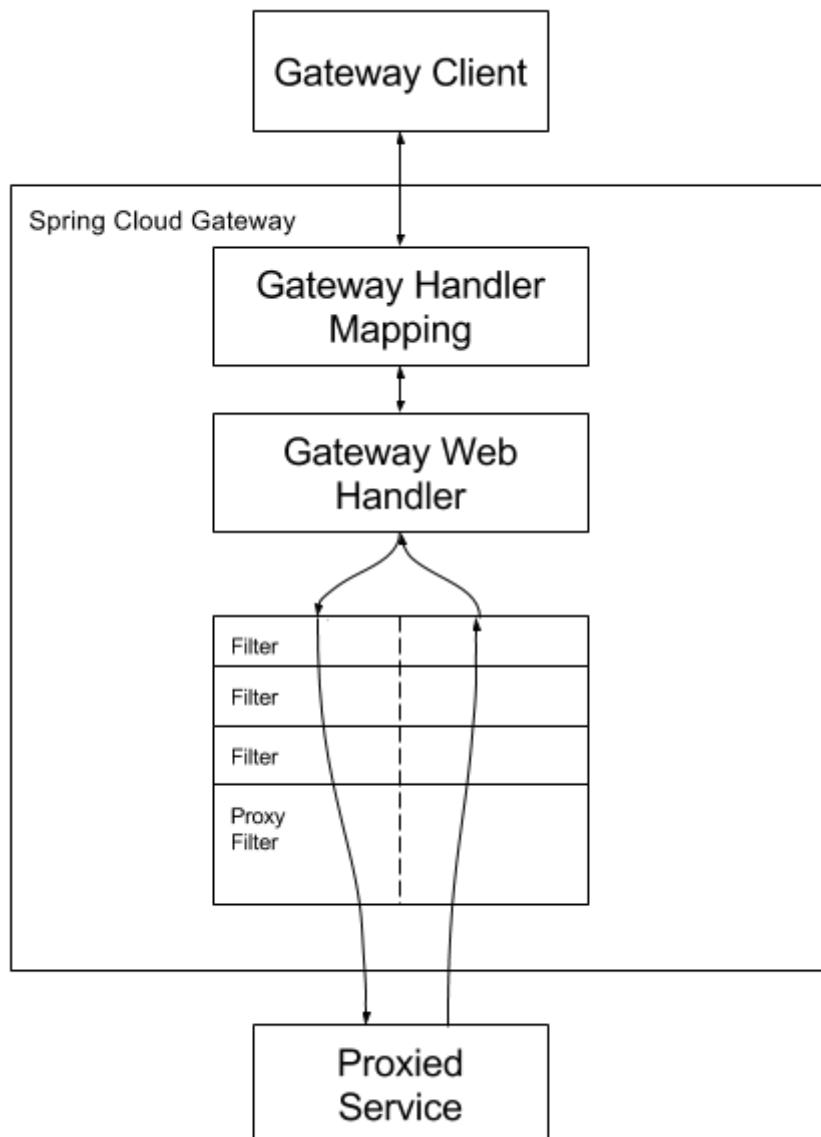
##### Spring Cloud Gateway

Spring Cloud Gateway 是 Spring 官方基于 Spring 5.0, Spring Boot 2.0 和 Project Reactor 等技术开发的网关，Spring Cloud Gateway 旨在为微服务架构提供一种简单而有效的统一的 API 路由管理方式。**Spring Cloud Gateway 作为 Spring Cloud 生态系中的网关，目标是替代 Netflix ZUUL**，其不仅提供统一的路由方式，并且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控/埋点，和限流等。

##### Spring Cloud Gateway 功能特征

- 基于 Spring Framework 5, Project Reactor 和 Spring Boot 2.0
- 动态路由
- Predicates 和 Filters 作用于特定路由
- 集成 Hystrix 断路器
- 集成 Spring Cloud DiscoveryClient
- 易于编写的 Predicates 和 Filters
- 限流
- 路径重写

## Spring Cloud Gateway 工程流程



客户端向 Spring Cloud Gateway 发出请求。然后在 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler。Handler 再通过指定的过滤器链来将请求发送到实际服务执行业务逻辑，然后返回。

过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前（pre）或之后（post）执行业务逻辑。

## 13.2 大文件上传

pom.xml:

```
1 <properties>
2   <java.version>11</java.version>
3 </properties>
4 <dependencies>
5   <dependency>
6     <groupId>org.springframework.boot</groupId>
```

```
7      <artifactId>spring-boot-starter-webflux</artifactId>
8  </dependency>
9  <dependency>
10     <groupId>org.springframework.boot</groupId>
11     <artifactId>spring-boot-starter-test</artifactId>
12     <scope>test</scope>
13   </dependency>
14   <dependency>
15     <groupId>io.projectreactor</groupId>
16     <artifactId>reactor-test</artifactId>
17     <scope>test</scope>
18   </dependency>
19 </dependencies>
20 <build>
21   <plugins>
22     <plugin>
23       <groupId>org.springframework.boot</groupId>
24       <artifactId>spring-boot-maven-plugin</artifactId>
25     </plugin>
26   </plugins>
27 </build>
```

src/main/java/com/lagou/webflux/demo/controller/FileController.java

```
1 package com.lagou.webflux.demo.controller;
2
3 import org.springframework.core.io.buffer.DataBufferUtils;
4 import org.springframework.http.codec.multipart.FilePart;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RestController;
8 import reactor.core.publisher(Flux);
9 import reactor.core.publisher.Mono;
10
11 import java.io.IOException;
12 import java.nio.channels.AsynchronousFileChannel;
13 import java.nio.file.Files;
14 import java.nio.file.Path;
15 import java.nio.file.StandardOpenOption;
16 import java.util.List;
17
18 @RestController
19 public class FileController {
20
21     @RequestMapping("/single")
22     public Mono<String> singleFile(@RequestPart("file")Mono<FilePart> file)
23     {
24         return file.map(filepart -> {
25             Path tmpFile = null;
26             try {
27                 // 创建临时文件
28                 tmpFile = Files.createTempFile("file-", filepart.filename());
29             } catch (IOException e) {
30                 e.printStackTrace();
31             }
32             DataBufferUtils.write(tmpFile, filepart).block();
33             return tmpFile.getFileName().toString();
34         });
35     }
36 }
```

```

30 }
31
32     System.out.println("文件路径: " + tmpFile.toAbsolutePath());
33     // 异步文件channel
34     AsynchronousFileChannel channel = null;
35     try {
36         // 打开指定文件写操作的channel
37         channel = AsynchronousFileChannel.open(tmpFile,
38             StandardOpenOption.WRITE);
39         } catch (IOException e) {
40             e.printStackTrace();
41         }
42
43         DataBufferUtils.write(filepart.content(), channel, 0)
44             .doOnNext(System.out::println)
45             .doOnComplete(() -> {
46                 System.out.println("文件拷贝完成");
47             }).subscribe();
48     return tmpFile;
49 }.map(tmp -> tmp.toFile())
50         .flatMap(fileSingle -> file.map(FilePart::filename));
51
52
53 @RequestMapping(value = "/multi")
54 public Mono<List<String>> multiFiles(@RequestPart("file") Flux<FilePart>
55 filePartFlux) {
56     return filePartFlux.map(filePart -> {
57         Path tmpFile = null;
58         try {
59             tmpFile = Files.createTempFile("mfile-", filePart.filename());
60             } catch (IOException e) {
61                 e.printStackTrace();
62             }
63             System.out.println(tmpFile.toAbsolutePath());
64             // 对每个filePart执行写文件的方法
65             filePart.transferTo(tmpFile.toFile());
66             // 返回Path对象
67             return tmpFile;
68         }.map(tfile -> tfile.toFile()) // 将每个Path对象映射为文件
69             .flatMap(fileSingle ->
70                 filePartFlux.map(FilePart::filename)).collectList();
71     }
72 }

```

src/main/resources/static/index.html:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>文件上传</title>

```

```
6  </head>
7  <body>
8      <form action="/single" method="post" enctype="multipart/form-data">
9          <input type="file" name="file">
10         <input type="submit" value="上传单个文件">
11     </form>
12     <hr>
13     <form action="/multi" method="post" enctype="multipart/form-data">
14         <input type="file" name="file">
15         <input type="file" name="file">
16         <input type="file" name="file">
17         <input type="file" name="file">
18         <input type="submit" value="上传多个文件">
19     </form>
20
21 </body>
22 </html>
```

### 13.3 处理客户端连接速度慢的系统

WebFlux 的第二个应用是构建系统，而这些系统的目地是在缓慢或不稳定网络连接条件下适用于移动设备客户端。要理解为什么 WebFlux 在这个领域有用，就要回想一下在处理一个慢速连接时会发生什么。问题在于，将数据从客户端传输到服务器可能花费大量时间，并且相应的响应也可能花费大量时间。在使用单连接单线程模型的情况下，已连接客户端数量越多，系统崩溃的可能性越大。例如，黑客能很容易的通过使用拒绝服务（Denial-of-Service，DoS）攻击使我们的服务器不可用。

相比之下，WebFlux 使我们能在不阻塞工作线程的情况下接受连接。这样，慢速连接不会导致任何问题。在等待传入请求体时，WebFlux 将继续接收其他连接而不会阻塞。响应式流抽象使我们能在需要时消费数据。这意味着服务器可以根据网络的就绪情况控制事件消费。

### 13.4 流系统或实时系统

WebFlux 的另一个有用的应用是实时流系统。要了解 WebFlux 为什么能在这一点上提供帮助，就要回想实时流系统是什么。

首先，这些系统的特点是低延迟和高吞吐量。在流系统中，大多数数据是从服务器端传出的，因此客户端扮演消费者的角色。通常来自于客户端的事件少于来自于服务器端的事件。但是，在在线游戏等实时系统中，传入数据量等于传出数据量。

使用非阻塞通信可以实现低延迟和高吞吐量。正如前文所述，非阻塞异步通信可以实现高效的资源利用，而基于 Netty 或类似框架的系统可以实现最高的吞吐量和最低的延迟。然而，这种响应式框架有其自身的缺点，即使用通道和回调的复杂交互模型。

尽管如此，响应式编程仍然可以巧妙地解决这两个问题。正如第4章所述，响应式编程，尤其是响应式库（如Reactor 3）可以帮助我们构建一个异步的非阻塞流而只需要很少的开销。这些开销来自基础代码复杂性和可接受的学习曲线。这两种解决方案都包含在WebFlux中。使用Spring框架可以让我们轻松构建这样的系统。

## 14 Spring WebFlux数据库访问

### 14.1 响应式持久化库的工作原理

Spring Data 中的**响应式存储库**通过适配底层数据库驱动来工作。

`ReactiveMongoRepository` 继承了 `ReactiveSortingRepository` 和 `ReactiveQueryByExampleExecutor` 等更多通用接口。

`ReactiveQueryByExampleExecutor` 接口可以使用 QBE 语言执行查询。

`ReactiveSortingRepository` 接口扩展了 `ReactiveCrudRepository` 接口；并添加了 `findAll` 方法，该方法能对请求查询结果进行排序。

`ReactiveCrudRepository` 声明了用于**保存、查找和删除**实体的方法。

1. `Mono<T> save(T entity)` 方法保存 `entity`，然后返回所保存的实体。保存操作可能更改整个实体对象。
2. `Mono<T> findById(ID id)` 操作实体的 id 并返回包装在 `Mono` 中的结果。
3. `findAllById` 方法有两个重载方法，其中一个重载方法以 `Iterable<ID>` 集合的形式消费 ID，另一个则采用 `Publisher<ID>` 的形式。
4. `ReactiveCrudRepository` 和 `CrudRepository` 之间唯一值得注意的区别在于 `ReactiveCrudRepository` **没有分页且不能进行事务操作**。

#### 分页支持

Spring Data 故意省略分页，因为同步存储库中使用的实现方案不适合响应式。

因为：

1. 要计算下一页的参数，需要知道前一个结果的返回记录数。
2. 要计算分页总数，需要查询记录总数。

这两个方面都不符合响应式非阻塞范式。

另外，通过查询数据库计算总行数不仅相当消耗资源，还在实际数据处理之前增加了延迟。

但是，通过将 `Pageable` 对象传递到存储库，仍然可以获取数据块，如下所示：

```
1 public interface ReactiveBookRepository extends  
2     ReactiveSortingRepository<Book, Long> {  
3     Flux<Book> findByAuthor(String author, Pageable pageable);  
4 }
```

所以，现在请求结果的第二页（索引从 0 开始），其中每页包含 5 个元素：

```
1 Flux<Book> result = reactiveBookRepository.findByAuthor("Andy Weir",  
2 PageRequest.of(1, 5));
```

## ReactiveMongoRepository实现细节

Spring Data MongoDB Reactive 模块只有一个针对 `ReactiveMongoRepository` 接口的实现，即

`SimpleReactiveMongoRepository` 类：

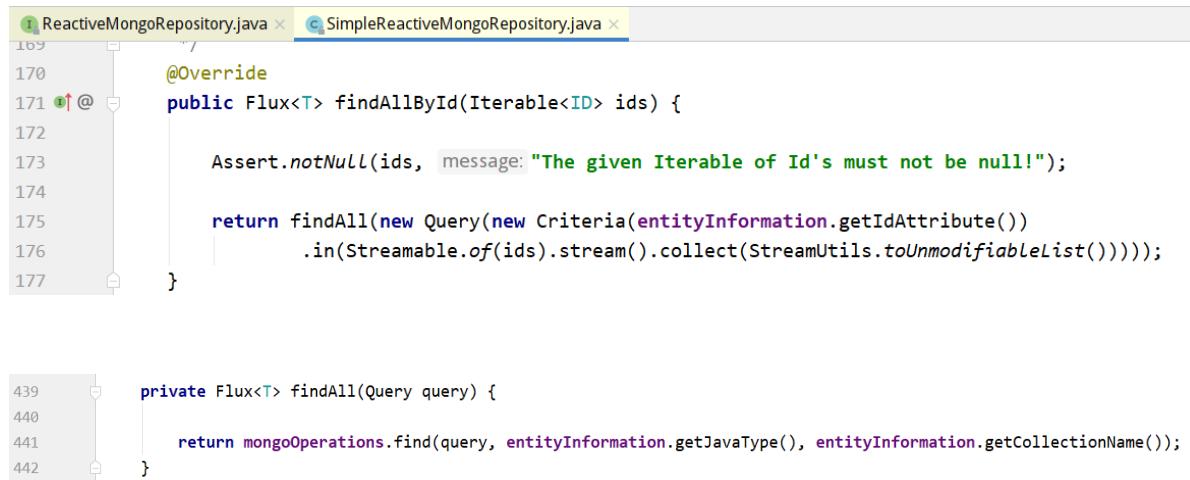
1. 它为 `ReactiveMongoRepository` 的所有方法提供实现；
2. 并使用 `ReactiveMongoOperations` 接口处理所有较低级别的操作。

如 `findAllById(Publisher<ID> ids)` 方法的实现：



```
182 */  
183 @Override  
184 public Flux<T> findAllById(Publisher<ID> ids) {  
185     Assert.notNull(ids, message: "The given Publisher of Id's must not be null!");  
186     return Flux.from(ids).buffer().flatMap(this::findAllById);  
187 }  
188 }
```

此方法使用 `buffer` 操作收集所有 `ids`，然后使用 `findAllById(Iterable<ID> ids)` 方法创建一个请求。该方法构建 `Query` 对象并调用 `findAll(Query query)` 方法，该方法触发 `ReactiveMongoOperations` 实例的 `mongoOperations.find(query, ...)`。



```
169 */  
170 @Override  
171 public Flux<T> findAllById(Iterable<ID> ids) {  
172     Assert.notNull(ids, message: "The given Iterable of Id's must not be null!");  
173     return findAll(new Query(new Criteria(entityInformation.getIdAttribute()))  
174         .in(Streamable.of(ids).stream().collect(StreamUtils.toUnmodifiableList())));  
175 }  
176 }  
177 }  
439 private Flux<T> findAll(Query query) {  
440     return mongoOperations.find(query, entityInformation.getJavaType(), entityInformation.getCollectionName());  
441 }  
442 }
```

`insert(Iterable<S> entities)` 方法在一个批处理中插入实体。

```
ReactiveMongoRepository.java | SimpleReactiveMongoRepository.java
271     */
272     @Override
273     public <S extends T> Flux<S> insert(Iterable<S> entities) {
274
275         Assert.notNull(entities, message: "The given Iterable of entities must not be null!");
276
277         List<S> source = Streamable.of(entities).stream().collect(StreamUtils.toUnmodifiableList());
278
279         return source.isEmpty() ? Flux.empty() : Flux.from(mongoOperations.insertAll(source));
280     }
```

`insert(Publisher<S> entities)` 方法在 `flatMap` 操作符内生成许多查询。

```
ReactiveMongoRepository.java | SimpleReactiveMongoRepository.java
285     */
286     @Override
287     public <S extends T> Flux<S> insert(Publisher<S> entities) {
288
289         Assert.notNull(entities, message: "The given Publisher of entities must not be null!");
290
291         return Flux.from(entities).flatMap(entity -> mongoOperations.insert(entity, entityInformation.getCollectionName()));
292     }
```

但是 `findAllById` 方法的两个重载方法以相同的方式运行，并且只生成一个数据库查询。

`saveAll` 方法：该方法消费 Publisher 的重载会为每个实体发出查询。

```
ReactiveMongoRepository.java | SimpleReactiveMongoRepository.java
329     */
330     @Override
331     public <S extends T> Flux<S> saveAll(Publisher<S> entityStream) {
332
333         Assert.notNull(entityStream, message: "The given Publisher of entities must not be null!");
334
335         return Flux.from(entityStream).flatMap(entity -> entityInformation.isNew(entity) ? //
336             mongoOperations.insert(entity, entityInformation.getCollectionName()).then(Mono.just(entity)) : //
337             mongoOperations.save(entity, entityInformation.getCollectionName()).then(Mono.just(entity)));
338     }
```

而其消费 Iterable 的重载，在所有实体都是新的的情况下只发出一个查询，但在其他情况下会为每个实体都发出一个查询。

```
ReactiveMongoRepository.java | SimpleReactiveMongoRepository.java
313     */
314     @Override
315     public <S extends T> Flux<S> saveAll(Iterable<S> entities) {
316
317         Assert.notNull(entities, message: "The given Iterable of entities must not be null!");
318
319         Streamable<S> source = Streamable.of(entities);
320
321         return source.stream().allMatch(entityInformation::isNew) ? //
322             mongoOperations.insert(source.stream().collect(Collectors.toList()), entityInformation.getCollectionName()) : //
323             Flux.fromIterable(entities).flatMap(this::save);
324     }
```

`deleteAll(Iterable<?extends T> entities)` 方法总是为每个实体发出一个查询，即使所有实体在 `Iterable` 容器中都可用并且不需要等待元素异步显示也是如此。

```

1 ReactiveMongoRepository.java x 2 SimpleReactiveMongoRepository.java x
398     */
399     @Override
400     public Mono<Void> deleteAll(Iterable<? extends T> entities) {
401
402         Assert.notNull(entities, message: "The given Iterable of entities must not be null!");
403
404         return Flux.fromIterable(entities).flatMap(this::delete).then();
405     }
406
407
408     /**
409      * Deletes all entities from the database.
410      * @param entityStream A Publisher of entities to be deleted.
411      * @return A Mono that completes when all entities have been deleted.
412     */
413     public Mono<Void> deleteAll(Publisher<? extends T> entityStream) {
414
415         Assert.notNull(entityStream, message: "The given Publisher of entities must not be null!");
416
417         return Flux.from(entityStream)//
418             .map(entityInformation::getRequiredId)//
419             .flatMap(this::deleteById)//
420             .then();
421     }

```

如果将 `ReactiveCrudRepository` 方法与实时生成的实现一起使用，查看实际查询会更加困难。在这种情况下，查询生成的行为方式与普通的同步 `CrudRepository` 类似。

`RepositoryFactorySupport` 为 `ReactiveCrudRepository` 生成适当的代理。

当使用 `@Query` 注解修饰方法时，`ReactiveStringBasedMongoQuery` 类用于生成查询。

`ReactivePartTreeMongoQuery` 类用于基于方法名称约定的查询生成。

将 `ReactiveMongoTemplate` 的日志级别设置为 DEBUG 时，可以跟踪发送到 MongoDB 的所有查询。

## 使用 `ReactiveMongoTemplate`

即使 `ReactiveMongoTemplate` 被用作响应式存储库的构建块，该类本身也非常通用。

有时它能比高级别的存储库更高效地使用数据库。

实现一个简单的服务，该服务使用 `ReactiveMongoTemplate` 并基于正则表达式来按标题查找图书：

```

1 import lombok.RequiredArgsConstructor;
2 import org.springframework.data.mongodb.core.ReactiveMongoOperations;
3 import org.springframework.data.mongodb.core.query.Criteria;
4 import org.springframework.data.mongodb.core.query.Query;
5 import org.springframework.stereotype.Service;
6 import reactor.core.publisher.Flux;
7
8 @Service
9 @RequiredArgsConstructor
10 public class RxMongoTemplateQueryService {
11     private static final String BOOK_COLLECTION = "book";

```

```

12     // ReactiveMongoTemplate实现该接口，并在配置了 MongoDB 数据源后出现在 Spring 上
13     // 下文中。
14     private final ReactiveMongoOperations mongoOperations;
15
16     // 使用正则表达式作为搜索条件，并返回包含结果的 Flux
17     public Flux<Book> findBooksByTitle(String title) {
18         // Query 类和 Criteria 类使用正则表达式构建实际查询
19         Query query = Query.query(new Criteria("title")
20             .regex("." + title + "."))
21             .limit(100); // 将结果数限制为 100
22         // mongoOperations 执行前面构建的查询
23         // 查询结果转换为Book.class类型的实体；查询名为 book 的集合
24         return mongoOperations.find(query, Book.class, BOOK_COLLECTION);
25     }

```

注意，可以通过提供以下方法签名来实现与普通响应式存储库相同的行为（查询限制除外），该签名遵循如下命名约定：`Flux<Book> findManyByTitleRegex(String regex)`。

在底层，`ReactiveMongoTemplate` 使用 `ReactiveMongoDatabaseFactory` 接口来获取响应式 MongoDB 连接的实例。

它使用 `MongoConverter` 接口的实例将实体转换为文档，反之亦然。

`MongoConverter` 也适用于同步 `MongoTemplate`。

`ReactiveMongoTemplate` 实现其契约的方式：

例如，`find(Query query, ...)` 方法将

`org.springframework.data.mongodb.core.query.Query` 实例映射到 `org.bson.Document` 类的实例，MongoDB 客户端使用后者工作。

`ReactiveMongoTemplate` 使用转换后的查询调用数据库客户端。

`com.mongodb.reactivestreams.client.MongoClient` 类提供了响应式 MongoDB 驱动程序的入口点，符合响应式流并通过响应式 `Publisher` 返回数据。

## 使用响应式驱动程序 (MongoDB)

Spring Data 中的响应式 MongoDB 连接基于 MongoDB 响应式流 Java 驱动程序构建。

该驱动程序提供具有非阻塞背压的异步流处理。

此外，响应式驱动程序构建在 MongoDB 异步 Java 驱动程序之上。

异步驱动程序是**低级别的**，并且具有基于回调的 API，因此它不像较高级别的响应式流驱动程序那样易于使用。

除了 MongoDB 响应式流 Java 驱动程序，还有 MongoDB RxJava 驱动程序，而后者基于**同一个异步 MongoDB 驱动程序**。

因此，针对MongoDB连接，Java生态系统准备了一个同步驱动程序、一个异步驱动程序和两个响应式驱动程序。

如果对查询过程控制的需求超过ReactiveMongoTemplate所能提供的，可以直接使用响应式驱动程序。

通过这种方法，在上述示例中使用纯响应式驱动程序的结果如下：

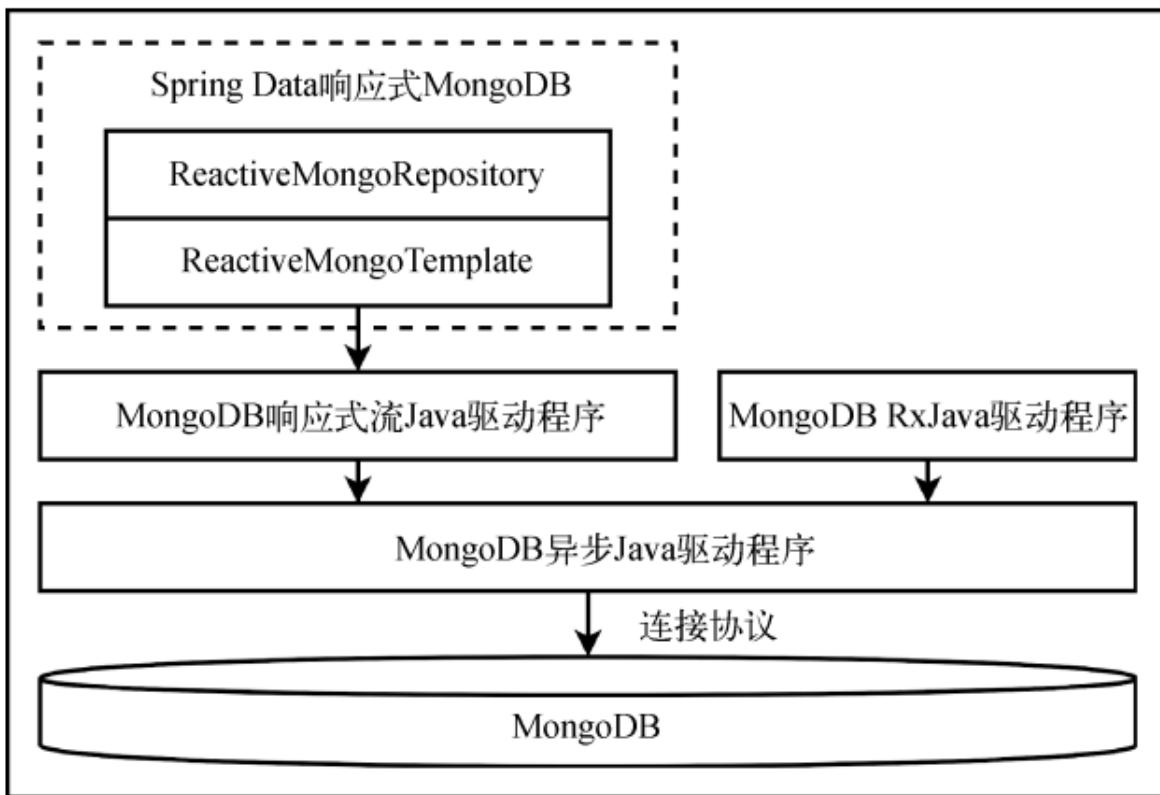
```
1 public class RxMongoDriverQueryService {
2     private final MongoClient mongoClient;
3     public Flux<Book> findBooksByTitleRegex(String regex) {
4         return Flux.defer(() -> {
5             Bson query = Filters.regex(titleRegex);
6             return mongoClient
7                 .getDatabase("test-database")
8                 .getCollection("book")
9                 .find(query);
10            })
11            .map(doc -> new Book(
12                doc.getObjectId("id"),
13                doc.getString("title"),
14                doc.getInteger("pubYear"),
15                // .....其他映射程序
16            ));
17        }
18    }
```

返回一个新的Flux实例，它将执行过程推迟到实际订阅发生的时间。在lambda中，使用com.mongodb.client.model.Filters辅助类并基于org.bson.conversions.Bson类型定义一个新查询。

尽管前面的示例中在数据库驱动程序级别进行工作，但在不需要手动处理背压，因为MongoDB响应式流Java驱动程序已经支持背压处理。

响应式MongoDB连接使用基于批大小的背压需求。虽然这种方法是一种合理的默认设置，但在使用小的需求增量时会生成许多往返。

下图强调了响应式MongoDB存储库所需的所有抽象层。



## 14.2 响应式事务

对于同步处理的情况，事务对象通常保存在 `ThreadLocal` 容器中。

但是 `ThreadLocal` 不适合用于响应式处理方式，因为用户无法控制线程切换。

事务需要将底层资源绑定到物化数据流。

在 Project Reactor 中，可以通过 [Reactor 上下文](#) 来实现这一目标。

### 基于 MongoDB 4 的响应式事务

MongoDB 从 4.0 版开始支持多文档事务 (multi-document transactions)。

Spring Data 没有任何在服务或存储库级别应用 [响应式事务](#) 的功能。但是，可以使用 `ReactiveMongoOperations` 级别（由 `ReactiveMongoTemplate` 实现）的事务进行操作。

首先，多文档事务是 MongoDB 的一项新功能。它仅适用于使用 WiredTiger 存储引擎的非分片副本集。在 MongoDB 4.0 中，没有其他配置支持多文档事务。

其次，某些 MongoDB 功能在事务中不可用，如，发出元命令和创建集合或索引都是不可能的。同时，隐式创建集合在事务中不起作用。因此，需要设置所需的数据库结构以防止错误。

此外，某些命令的行为可能有所不同。

在事务提交之前，事务外部不会显示任何数据更新。

假设必须实现一种用于在用户账户之间转账的钱包服务。每个用户都有自己的账户，且账户余额非负。用户可以将任意金额转给其他用户，但只有在账户中有足够资金时转账才会成功。

转账可以并行发生，但在转账时，系统中的资金既不能增加，也不能减少。因此，汇款人钱包的取款操作和收款人钱包的存款操作必须同时且原子地进行。此时可以使用多文档事务。

要将一笔款项从账户 A 转账到账户 B，应该执行以下操作：

1. 启动新事务。
2. 加载账户 A 的钱包。
3. 加载账户 B 的钱包。
4. 检查账户 A 的钱包中是否有足够的资金。
5. 提取转账金额并计算账户 A 的新余额。
6. 存入转账金额并计算账户 B 的新余额。
7. 保存账户 A 的钱包。
8. 保存账户 B 的钱包。
9. 提交事务。

准备MongoDB数据库：

主从搭建

主节点配置：mongo\_37017.conf

```
1 dbpath=/data/mongo/data/server1
2 bind_ip=0.0.0.0
3 port=37017
4 fork=true
5 logpath=/data/mongo/logs/server1.log
6 replset=lagouCluster
```

从节点1配置：mongo\_37018.conf

```
1 dbpath=/data/mongo/data/server2
2 bind_ip=0.0.0.0
3 port=37018
4 fork=true
5 logpath=/data/mongo/logs/server2.log
6 replset=lagouCluster
```

从节点2配置：mongo\_37019.conf

```
1 dbpath=/data/mongo/data/server3
2 bind_ip=0.0.0.0
3 port=37019
4 fork=true
5 logpath=/data/mongo/logs/server3.log
6 replset=lagouCluster
```

创建目录:

```
1 mkdir -p /data/mongo/data/server1 /data/mongo/data/server2
   /data/mongo/data/server3 /data/mongo/logs
```

启动:

```
1 mongod -f 37017.conf
2 mongod -f 37018.conf
3 mongod -f 37019.conf
```

查看进程和端口号:

```
1 ps aux | grep mongo
2 ss -nelp | grep mongo
```

初始化节点配置:

```
1 var cfg = {"_id": "lagouCluster", "protocolversion": 1, "members": [
2     {"_id": 1, "host": "node2:37017", "priority": 10},
3     {"_id": 2, "host": "node2:37018"},
4     {"_id": 3, "host": "node2:37019"}]
5 }
6
7 rs.initiate(cfg)
8 rs.status()
```

```
1 # 启用slave节点的读
2 rs.slaveOk()
3 # 查看状态
4 rs.status()
5 # 增加节点
6 rs.add("node2:37019")
7 # 删除节点
8 rs.remove("node2:37019")
```

节点说明：

PRIMARY节点：可以查询和新增数据

SECONDARY节点：只能查询，不能新增，基于priority权重可以被选为主节点

ARBITER节点：不能查询数据和新增数据，不能变为主节点

创建Spring Boot项目，添加以下依赖：

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-webflux</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>com.google.guava</groupId>
11  <artifactId>guava</artifactId>
12  <version>30.1-jre</version>
13 </dependency>
14 <dependency>
15  <groupId>org.testcontainers</groupId>
16  <artifactId>testcontainers</artifactId>
17  <version>1.15.1</version>
18  <scope>test</scope>
19 </dependency>
20 <dependency>
21  <groupId>org.projectlombok</groupId>
22  <artifactId>lombok</artifactId>
23  <optional>true</optional>
24 </dependency>
25 <dependency>
26  <groupId>org.springframework.boot</groupId>
27  <artifactId>spring-boot-starter-test</artifactId>
28  <scope>test</scope>
29 </dependency>
30 <dependency>
31  <groupId>io.projectreactor</groupId>
32  <artifactId>reactor-test</artifactId>
33  <scope>test</scope>
34 </dependency>
```

application.yml

```
1 spring:
2   data:
3     mongodb:
4       host: "node2"
5       port: 27017
6       database: "wallet"
7 #       username: "root"
```

```
8 #     password: "123456"
9 #     authentication-database: "admin"
10 #     uri:
11 #         "mongodb://192.168.2.106:27017/wallet,192.168.2.106:27018/wallet,192.168.2.1
12 #             06:27019/wallet"
13
14 logging:
15   level:
16     reactor.core.publisher.FluxUsingwhen: ERROR
17 #     org.springframework.data.mongodb.core.ReactiveMongoTemplate: DEBUG
```

映射到MongoDB文档的实体类:

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6 import lombok.With;
7 import org.bson.types.ObjectId;
8 import org.springframework.data.annotation.Id;
9 import org.springframework.data.mongodb.core.mapping.Document;
10
11 @With
12 @Data
13 @NoArgsConstructor
14 @AllArgsConstructor
15 @Document("wallet")
16 public class Wallet {
17     @Id
18     private ObjectId id;
19     private String owner;
20     private int balance;
21
22     // Some statistics
23     private int depositOperations;
24     private int withdrawOperations;
25
26     public boolean hasEnoughFunds(int amount) {
27         return balance >= amount;
28     }
29
30     public void withdraw(int amount) {
31         if (!hasEnoughFunds(amount)) {
32             throw new RuntimeException("Not enough funds!");
33         }
34         this.balance = this.balance - amount;
35         this.withdrawOperations += 1;
36     }
37
38     public void deposit(int amount) {
39         this.balance = this.balance + amount;
40         this.depositOperations += 1;
41     }
42 }
```

```
43     public static Wallet wallet(String owner, int balance) {
44         return new Wallet(new ObjectId(), owner, balance, 0, 0);
45     }
46 }
```

repository:

```
1 package com.lagou.webflux.demo.repository;
2
3 import com.lagou.webflux.demo.entity.Wallet;
4 import org.bson.types.ObjectId;
5 import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
6 import org.springframework.stereotype.Repository;
7 import reactor.core.publisher.Mono;
8
9 @Repository
10 public interface WalletRepository extends ReactiveMongoRepository<Wallet,
11 ObjectId> {
12     Mono<Wallet> findByOwner(Mono<String> owner);
13 }
```

service接口:

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Wallet;
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7 import reactor.core.publisher.Flux;
8 import reactor.core.publisher.Mono;
9
10 public interface WalletService {
11
12     Flux<String> generateClients(Integer number, Integer defaultBalance);
13
14     Mono<TxResult> transferMoney(
15         Mono<String> fromOwner,
16         Mono<String> toOwner,
17         Mono<Integer> amount);
18
19     Mono<Statistics> reportAllWallets();
20
21     /**
22      * 移除所有数据
23      * @return
24      */
25     Mono<Void> removeAllClients();
26
27     enum TxResult {
28         SUCCESS,
29         NOT_ENOUGH_FUNDS,
30         TX_CONFLICT
```

```

31     }
32
33     @NoArgsConstructor
34     @AllArgsConstructor
35     @Data
36     class Statistics {
37         private long totalAccounts;
38         private long totalBalance;
39         private long totalDeposits;
40         private long totalWithdraws;
41
42         public Statistics withWallet(Wallet w) {
43             return new Statistics(
44                 this.totalAccounts + 1,
45                 this.totalBalance + w.getBalance(),
46                 this.totalDeposits + w.getDepositOperations(),
47                 this.totalWithdraws + w.getWithdrawOperations());
48         }
49     }
50 }
```

service的基本实现：

```

1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Wallet;
4 import com.lagou.webflux.demo.repository.WalletRepository;
5 import com.lagou.webflux.demo.service.WalletService;
6 import lombok.RequiredArgsConstructor;
7 import lombok.extern.slf4j.Slf4j;
8 import reactor.core.publisher.Flux;
9 import reactor.core.publisher.Mono;
10
11 import java.util.Comparator;
12
13 import static java.lang.String.format;
14
15 @Slf4j
16 @RequiredArgsConstructor
17 public abstract class BaseWalletService implements WalletService {
18     protected final WalletRepository walletRepository;
19
20     /**
21      * 生成指定个数的账户信息
22      * @param number 账户个数
23      * @param defaultBalance 默认的余额
24      * @return 返回账户的用户名称
25     */
26     public Flux<String> generateClients(Integer number, Integer
27     defaultBalance) {
28         return walletRepository.findAll(
29             Flux.range(1, number)
30                 .map(id -> format("client-%05d", id))
31                 .map(owner -> wallet.wallet(owner, defaultBalance)))
32             .map(wallet::getOwner);
33     }
34 }
```

```

32     }
33
34     @Override
35     public Mono<Statistics> reportAllWallets() {
36         return walletRepository
37             .findAll()
38             .sort(Comparator.comparing(Wallet::getOwner))
39             .doOnNext(w ->
40                 log.info(format("%10s: %7d$ (d: %5s | w: %5s)", 
41                     w.getOwner(), w.getBalance(),
42                     w.getDepositOperations(), w.getWithdrawOperations())))
43             .reduce(new Statistics(), Statistics::withWallet);
44     }
45
46     @Override
47     public Mono<Void> removeAllClients() {
48         return walletRepository.deleteAll();
49     }

```

service的事务实现:

```

1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Wallet;
4 import com.lagou.webflux.demo.repository.WalletRepository;
5 import lombok.extern.slf4j.Slf4j;
6 import org.springframework.data.mongodb.core.ReactiveMongoContext;
7 import org.springframework.data.mongodb.core.ReactiveMongoTemplate;
8 import org.springframework.data.mongodb.core.query.Criteria;
9 import org.springframework.data.mongodb.core.query.Query;
10 import org.springframework.stereotype.Component;
11 import reactor.core.publisher.Mono;
12
13 import java.time.Duration;
14 import java.time.Instant;
15
16 import static java.time.Instant.now;
17
18 @Slf4j
19 @Component
20 public class TransactionalWalletService extends BaseWalletService {
21     private final ReactiveMongoTemplate mongoTemplate;
22
23     public TransactionalWalletService(
24         ReactiveMongoTemplate mongoTemplate,
25         WalletRepository walletRepository
26     ) {
27         super(walletRepository);
28         this.mongoTemplate = mongoTemplate;
29     }
30
31     @Override
32     public Mono<TxResult> transferMoney(Mono<String> fromOwner, Mono<String>
toOwner, Mono<Integer> requestAmount) {

```

```

33     return Mono.zip(fromOwner, toOwner, requestAmount)
34         .flatMap(data -> {
35             Instant start = now();
36             return doTransferMoney(data.getT1(), data.getT2(),
37                                     data.getT3())
38                             .onErrorReturn(TxResult.TX_CONFLICT)
39                             .doOnSuccess(result -> log.info("Transaction
40 result: {}, took: {}",
41                               result, Duration.between(start,
42                               now())));
43         });
44
45     private Mono<TxResult> doTransferMoney(String from, String to, Integer
46 amount) {
47         return mongoTemplate.inTransaction().execute(session ->
48             session
49                 .findOne(queryForOwner(from), wallet.class)
50                 .flatMap(fromWallet -> session
51                     .findOne(queryForOwner(to), wallet.class)
52                     .flatMap(toWallet -> {
53                         if (fromWallet.hasEnoughFunds(amount)) {
54                             fromWallet.withdraw(amount);
55                             toWallet.deposit(amount);
56
57                             return session.save(fromWallet)
58
59                         .then(session.save(toWallet))
60
61                         .then(ReactiveMongoContext.getSession()) // An example how to resolve
62                         the current session
63                         .doOnNext(tx ->
64                             log.info("Current session: {}", tx))
65
66                         .then(Mono.just(TxResult.SUCCESS));
67                         } else {
68                             return
69                             Mono.just(TxResult.NOT_ENOUGH_FUNDS);
70                         }
71                     }))
72                     .onErrorResume(e -> Mono.error(new
73 RuntimeException("Conflict")))
74                     .last();
75     }
76
77     private Query queryForOwner(String owner) {
78         return Query.query(new Criteria("owner").is(owner));
79     }
80 }

```

测试工具类:

```

1 package com.lagou.webflux.demo;
2

```

```
3 import com.lagou.webflux.demo.service.WalletService;
4 import lombok.Builder;
5 import lombok.RequiredArgsConstructor;
6 import lombok.ToString;
7 import lombok.extern.slf4j.Slf4j;
8 import org.junit.*;
9 import org.springframework.boot.test.autoconfigure.mongo.DataMongoTest;
10 import reactor.core.publisher.Flux;
11 import reactor.core.publisher.Mono;
12 import reactor.core.scheduler.Scheduler;
13 import reactor.core.scheduler.Schedulers;
14 import reactor.util.function.Tuple2;
15 import reactor.util.function.Tuples;
16 import java.time.Duration;
17 import java.time.Instant;
18 import java.util.List;
19 import java.util.Random;
20
21 import static java.time.Duration.between;
22 import static java.time.Instant.now;
23
24 @SuppressWarnings("Duplicates")
25 @Slf4j
26 @DataMongoTest
27 class BasewalletServiceTest {
28
29     Tuple2<Long, Long> simulateOperations(WalletService walletService) {
30         int accounts = 500;
31         int defaultBalance = 1000;
32         int iterations = 10000;
33         int parallelism = 200;
34
35         // 清空所有数据
36         walletService.removeAllClients().block();
37
38         // 创建指定个数的账户，并返回账户的用户名
39         List<String> clients = walletService.generateClients(accounts,
40         defaultBalance)
41             .doOnNext(name -> log.info("Created wallet for: {}", name))
42             .collectList() // 收集到一个List集合
43             .block(); // 阻塞，直到执行完毕
44
45         // 并行操作的调度器
46         Scheduler mongoScheduler = Schedulers
47             .newParallel("MongoOperations", parallelism);
48
49         // 记录当前时间
50         Instant startTime = now();
51         OperationalSimulation simulation = OperationalSimulation.builder()
52             .walletService(walletService) // 设置service
53             .clients(clients) // 设置全部用户名
54             .defaultBalance(defaultBalance) // 设置默认余额
55             .iterations(iterations) //
56             .simulationScheduler(mongoScheduler) // 调度器
57             .build(); // 创建simulation对象
58
59         OperationStats operations = simulation
```

```
59     .runSimulation() // 执行事务
60     .block(); // 直到执行完毕
61
62
63     log.info("--- Results -----");
64     // 收集所有账户的报告
65     walletService.Statistics statistics =
66     walletService.reportAllWallets()
67         .block();
68     log.info("Expected/actual total balance: {}$ / {}$ | Took: {}", 
69         accounts * defaultBalance, statistics.getTotalBalance(),
70         between(startTime, now()));
71     log.info("{}", statistics);
72     log.info("{}", operations);
73
74     log.info("Cleaning up database");
75     // 移除所有的账户信息
76     walletService.removeAllClients()
77         .block();
78     // 返回二元组，第一个元素是所有账户的总余额，第二个参数是执行事务之后所有账户总余额
79
80     return Tuples.of((long) accounts * defaultBalance,
81     statistics.getTotalBalance());
82 }
83
84 @Builder
85 @RequiredArgsConstructor
86 public static class OperationalSimulation {
87     private final walletService walletService;
88     private final List<String> clients;
89     private final int defaultBalance;
90     private final int iterations;
91     private final Scheduler simulationScheduler;
92
93     private final Random rnd = new Random();
94
95     public Mono<OperationStats> runSimulation() {
96         return Flux.range(0, iterations)
97             .flatMap(i -> Mono
98                 .delay(Duration.ofMillis(rnd.nextInt(10)))
99                 .publishOn(simulationScheduler)
100                .flatMap(_i -> {
101                    String fromOwner = randomOwner();
102                    String toOwner = randomOwnerExcept(fromOwner);
103                    int amount = randomTransferAmount();
104
105                    return walletService.transferMoney(
106                        Mono.just(fromOwner),
107                        Mono.just(toOwner),
108                        Mono.just(amount));
109                })
110                .reduce(OperationStats.start(), OperationStats::countTxResult);
111    }
112
113    private int randomTransferAmount() {
114        return rnd.nextInt(defaultBalance);
115    }
116
117 }
```

```

113     private String randomOwner() {
114         int from = rnd.nextInt(clients.size());
115         return clients.get(from);
116     }
117
118     private String randomOwnerExcept(String fromOwner) {
119         String toOwner;
120         do {
121             int to = rnd.nextInt(clients.size());
122             toOwner = clients.get(to);
123         } while (fromOwner.equals(toOwner));
124         return toOwner;
125     }
126 }
127
128 @ToString
129 @RequiredArgsConstructor
130 public static class OperationStats {
131     private final int successful;
132     private final int notEnoughFunds;
133     private final int conflict;
134
135     public OperationStats countTxResult(walletService.TxResult result) {
136         switch (result){
137             case SUCCESS:
138                 return new OperationStats(successful + 1, notEnoughFunds,
conflict);
139             case NOT_ENOUGH_FUNDS:
140                 return new OperationStats(successful, notEnoughFunds + 1,
conflict);
141             case TX_CONFLICT:
142                 return new OperationStats(successful, notEnoughFunds,
conflict + 1);
143             default:
144                 throw new RuntimeException("Unexpected status:" + result);
145         }
146     }
147
148     public static OperationStats start() {
149         return new OperationStats(0, 0, 0);
150     }
151 }
152 }
```

测试类：

```

1 package com.lagou.webflux.demo;
2
3 import com.lagou.webflux.demo.repository.WalletRepository;
4 import com.lagou.webflux.demo.service.WalletService;
5 import com.lagou.webflux.demo.service.impl.TransactionalWalletService;
6 import lombok.extern.slf4j.Slf4j;
7 import org.junit.Assert;
8 import org.junit.jupiter.api.DisplayName;
9 import org.junit.jupiter.api.Test;
```

```

10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.boot.test.autoconfigure.mongo.DataMongoTest;
12 import org.springframework.data.mongodb.core.ReactiveMongoTemplate;
13 import reactor.util.function.Tuple2;
14
15 @Slf4j
16 @DataMongoTest
17 class TransactionalWalletServiceTest extends BaseWalletServiceTest {
18
19     @DisplayName("Reactive transactions for data transfer")
20     @Test
21     public void testReactiveTransactionalApproach(@Autowired
22             WalletRepository walletRepository,
23             @Autowired ReactiveMongoTemplate mongoTemplate) {
24
25         // 创建TransactionalWalletService
26         WalletService walletService = new
27         TransactionalWalletService(mongoTemplate, walletRepository);
28         Tuple2<Long, Long> expectedActual =
29         simulateOperations(walletService);
30
31         // 事务执行成功之后，两个账户余额应该相等
32         Assert.assertEquals(expectedActual.getT1(), expectedActual.getT2());
33     }
34 }
```

基于事务引用正确的会话可以通过使用 Reactor 上下文实现。

ReactiveMongoTemplate.inTransaction 方法启动一个新事务并将其放入上下文中。

因此，在响应流中的任何位置都可以获得用 com.mongodb.reactivestreams.client.ClientSession 接口表示的事务的会话。

ReactiveMongoContext.getSession()辅助方法可以帮助获取会话实例。

因此，在前面的模拟中，我们进行了 10 000 次转账操作，其中 6 238 次成功，3 762 次由于资金不足而失败。此外，我们的重试策略解决了所有事务冲突，因为没有任何事务以TX\_CONFLICT 状态完成。从日志中可以明显看出，系统保持了总余额不变，即模拟前后系统中的总金额是相同的。因此，我们通过应用 MongoDB 的响应式事务来实现并发资金转账中的系统完整性。

现在，对副本集的多文档事务的支持能使用 MongoDB 作为主数据存储来实现全新的应用程序集。当然，未来版本的 MongoDB 可能支持跨分片部署的事务，并提供各种隔离级别来处理事务。但是，我们应该注意到，与简单的文档写入相比，多文档事务会产生更高的性能成本和更长的响应延迟。

## 基于 SAGA 模式的分布式事务

分布式事务可以以不同方式实现。当然，对于使用响应式范式实现的持久层，这种说法也成立。但是，鉴于 Spring Data 仅支持 MongoDB 4 的响应式事务，并且前面提到的事务支持与 Java 事务 API (Java Transaction API, JTA) 不兼容，在响应式微服务中实现分布式事务的唯一可行选择是 SAGA 模式。此外，与其他需要分布式事务的可选模式相比，SAGA 模式具有良好的可伸缩性，更适合响应式

流。

## 14.3 Spring Data响应式连接器

Spring Data为4个NoSQL数据库准备了数据库连接器，即MongoDB、Cassandra、Couchbase和Redis。

Spring Data也可能支持其他数据存储，特别是那些利用Spring WebFlux WebClient基于HTTP进行通信的数据存储。

### 1. 响应式 MongoDB 连接器

可以使用spring-boot-starter-data-mongodb-reactive Spring Boot启动器模块启用Spring Data Reactive MongoDB模块。

响应式MongoDB支持提供了一个响应式存储库和`ReactiveMongoRepository`接口定义基本存储库契约。

存储库继承了`ReactiveCrudRepository`的所有功能，并添加了对QBE的支持。

MongoDB存储库支持使用`@Query`注解的自定义查询以及带有`@Meta`注解的其他查询配置。

如果MongoDB存储库遵循命名约定，则它支持从方法名称生成查询。

MongoDB存储库的另一个显著特性是支持尾游标(tailable cursor)。

默认情况下，数据库会在消费了所有结果时自动关闭查询游标。

但是，MongoDB有固定集合(capped collections)，这些集合大小固定，支持高吞吐量操作。文档检索基于插入顺序。固定集合的工作方式与循环缓冲区类似。

固定集合也支持一个尾游标。客户端消费初始查询的所有结果后，此光标保持打开状态，当有人将新文档插入到固定集合中时，尾游标将返回新文档。

在`ReactiveMongoRepository`中，由`@Tailable`注解标记的方法会返回由`Flux<Entity>`类型表示的尾游标。

`ReactiveMongoOperations`接口和它的实现类`ReactiveMongoTemplate`级别更低，可以更精细地访问MongoDB通信。

除此之外，`ReactiveMongoTemplate`还支持MongoDB的多文档事务。此功能仅适用于WiredTiger存储引擎的非分片副本集。

响应式Spring Data MongoDB模块构建于响应式流MongoDB驱动程序之上，后者实现了响应式规范并在内部使用Project Reactor。

MongoDB响应式流Java驱动程序基于MongoDB异步Java驱动程序构建。

## 2. 响应式 Redis 连接器

Spring Data Reactive Redis 模块可以通过导入 spring-boot-starter-data-redis-reactive 启动器来启用。

Redis 连接器不提供响应式存储库。

ReactiveRedisTemplate 类成为响应式 Redis 数据访问的核心抽象。

ReactiveRedisTemplate 实现 ReactiveRedisOperations 接口，并提供所有必需的序列化/反序列化过程。

ReactiveRedisConnection 能在与 Redis 通信时使用原始字节缓冲区。

ReactiveRedisTemplate 还能订阅 Pub-Sub 通道。例如，convertAndSend(String destination, V message) 方法将给定消息发布到给定通道，并返回接收消息的客户端数。

listenToChannel(String...channels) 方法返回一个 Flux，其中包含来自感兴趣通道的消息。

因此，响应式 Redis 连接器不仅可以实现响应式数据存储，还可以提供消息传递机制。

Spring Data Redis 目前集成了 **Lettuce 驱动程序**。它是 **Redis 唯一的响应式 Java 连接器**。

Lettuce 4.x 版本使用 RxJava 进行底层实现。但是，该库的 5.x 分支切换到了 Project Reactor。

除 Couchbase 外的所有响应式连接器都具有响应式健康指标。因此，数据库运行状况检查也不应浪费任何服务器资源。

## 14.4 响应式关系型数据库连接

响应式关系型数据库连接（Reactive Relational Database Connectivity，**R2DBC**）是一项探索完全响应式数据库 API 的倡议。

Spring Data 团队领导 R2DBC 倡议，并使用它在响应式应用程序内的响应式数据访问环境中探测和验证想法。

R2DBC 在 Spring OnePlatform 2018 会议上被公开，其目标是定义具有**背压支持的响应式数据库访问 API**。Spring Data 团队在响应式 NoSQL 持久化方面获得了一些先进经验，因此决定提出对真正响应式语言级数据访问 API 的愿景。

R2DBC 项目包括以下部分。

- R2DBC 服务提供程序接口（Service Provider Interface，SPI）定义了实现驱动程序的简约 API，便于彻底减少驱动程序实现者必须遵守的 API。SPI 不适合在应用程序代码中直接使用，需要专用的客户端库。
- R2DBC 客户端提供了人性化的 API 和帮助类，可将用户请求转换为 SPI 级别。R2DBC 客户端对 R2DBC SPI 的作用与 Jdbi 库对 JDBC 的作用相同。
- R2DBC PostgreSQL 实现为 PostgreSQL 提供了 R2DBC 驱动程序。使用 Netty 框架通过 PostgreSQL 连接协议进行异步通信。背压既可以通过 TCP 流控制，也可以通过被称为门户（portal）的 PostgreSQL 特性来实现，后者实际上是一个查询内的光标。门户能完美地转换为响应式流。

并非所有关系型数据库都具有正确背压传播所需的连接协议功能。但 **TCP 流控制在所有情况下都可用**。

## 基于 Spring Data R2DBC 使用 R2DBC

Spring Data JDBC 模块中提供了基于R2DBC的 `ReactiveCrudRepository` 接口。

`SimpleR2dbcRepository` 类使用R2DBC 实现 `ReactiveCrudRepository` 接口。

`SimpleR2dbcRepository` 类不使用默认的 R2DBC 客户端，而是定义自己的客户端以使用 R2DBC SPI。

背压问题在 R2DBC SPI 级别得到了解决。

### 使用R2DBC操作PostgreSQL案例：

搭建PostgreSQL数据库：

```
1 # 安装postgre的共享库
2 sudo rpm -ivh postgresql13-libs-13.1-1PGDG.rhel7.x86_64.rpm
3 # 安装依赖
4 sudo yum install libicu -y
5 # 安装postgre客户端
6 sudo rpm -ivh postgresql13-13.1-1PGDG.rhel7.x86_64.rpm
7 # 安装postgre的服务端
8 sudo rpm -ivh postgresql13-server-13.1-1PGDG.rhel7.x86_64.rpm
```

```
1 # 修改postgres用户的密码
2 # 指定postgres的数据目录
3 # 在postgres家目录创建data目录
4 -bash-4.2$ mkdir data
5 # /var/lib/pgsql/data
6
7 # pg用户操作：初始化数据库
8 bash-4.2$ /usr/pgsql-13/bin/initdb -D /var/lib/pgsql/data
9
10 -bash-4.2$ vim /var/lib/pgsql/data/postgresql.conf
11 # 添加内容: listen_addresses='*'
12 -bash-4.2$ vim /var/lib/pgsql/data/pg_hba.conf
13 # 添加内容: host all postgres 192.168.100.1/32 trust
14
15 # 启动postgresql
16 bash-4.2$ /usr/pgsql-13/bin/pg_ctl -D /var/lib/pgsql/data -l logfile start
17
18 # 登录数据库
19 bash-4.2$ /usr/pgsql-13/bin/psql
20 psql (13.1)
21 Type "help" for help.
22 # 退出数据库登录
23 postgres=# \q
24
```

```
25 # 停止postgresql  
26 bash-4.2$ /usr/pgsql-13/bin/pg_ctl stop -m fast -D /var/lib/pgsql/data
```

使用spring data r2dbc执行插入的时候，如果设置了主键，则认为该数据存在，需要执行的是更新。

如果没有设置主键，则需要使用自增主键，此时插入数据的时候不设置主键值，系统认为是数据的插入。

设置主键自增的方法：

```
1 -- 使用SERIAL  
2 CREATE TABLE users  
3 (  
4     id SERIAL primary key ,  
5     name character varying,  
6     password character varying  
7 )  
8  
9 -- 自动创建名为users_id_seq的序列，且MAXVALUE=9223372036854775807  
10  
11 -- 先创建序列，然后设置字段的自增  
12 CREATE SEQUENCE users_id_seq  
13 START WITH 1  
14 INCREMENT BY 1  
15 NO MINVALUE  
16 NO MAXVALUE  
17 CACHE 1;  
18  
19 alter table users alter column id set default nextval('users_id_seq');
```

**PostgreSQL字段名称一定不要使用大写字母！**

新建Spring Boot项目，并添加依赖：

pom.xml

```
1 <dependency>  
2     <groupId>org.springframework.boot</groupId>  
3     <artifactId>spring-boot-starter-data-r2dbc</artifactId>  
4 </dependency>  
5 <dependency>  
6     <groupId>org.springframework.boot</groupId>  
7     <artifactId>spring-boot-starter-webflux</artifactId>  
8 </dependency>  
9 <dependency>  
10    <groupId>io.r2dbc</groupId>  
11    <artifactId>r2dbc-postgresql</artifactId>  
12    <scope>runtime</scope>-->  
13 </dependency>  
14 <dependency>
```

```
15 <groupId>org.projectlombok</groupId>
16 <artifactId>lombok</artifactId>
17 <optional>true</optional>
18 </dependency>
```

application.yml

```
1 spring:
2   r2dbc:
3     username: postgres
4     password: 123456
5     url: r2dbc:postgres://192.168.100.102:5432/mydb
```

实体类:

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.Data;
4 import org.springframework.data.annotation.Id;
5 import org.springframework.data.relational.core.mapping.Column;
6 import org.springframework.data.relational.core.mapping.Table;
7
8 @Table("book")
9 @Data
10 public class Book {
11   @Id
12   Integer id;
13   @Column("title")
14   String title;
15   @Column("publishingyear")
16   Integer publishingYear;
17
18   public Book(Integer id, String title, Integer publishingYear) {
19     this.id = id;
20     this.title = title;
21     this.publishingYear = publishingYear;
22   }
23
24   public Book(String title, Integer publishingYear) {
25     this(null, title, publishingYear);
26   }
27
28   public Book() {
29     this(null, null, null);
30   }
31 }
```

BookRepository.java

```
1 package com.lagou.webflux.demo.repository;
2
```

```
3 import com.lagou.webflux.demo.entity.Book;
4 import org.springframework.data.r2dbc.repository.Query;
5 import org.springframework.data.repository.reactive.ReactiveCrudRepository;
6 import reactor.core.publisher.Flux;
7
8 public interface BookRepository
9     extends ReactiveCrudRepository<Book, Integer> {
10
11     @Query("SELECT * FROM book WHERE publishingyear = " +
12             "(SELECT MAX(publishingyear) FROM book)")
13     Flux<Book> findTheLatestBooks();
14 }
```

BookRepository1.java

```
1 package com.lagou.webflux.demo.repository;
2
3 import org.springframework.r2dbc.core.DatabaseClient;
4 import org.springframework.stereotype.Repository;
5 import org.springframework.transaction.annotation.Transactional;
6 import reactor.core.publisher.Mono;
7
8 @Repository
9 public class BookRepository1 {
10
11     private final DatabaseClient client;
12
13     public BookRepository1(DatabaseClient client) {
14         this.client = client;
15     }
16
17     // 事务
18     @Transactional
19     public Mono<Void> insertRows() {
20
21         int i = 0;
22
23         return client.sql("insert into book(title, publishingyear) values
('hellotx3', 2022)")
24             .fetch()
25             .rowsUpdated()
26             .doOnNext(item -> {int a = 10 / i;})
27             .then(client.sql("insert into book(title, publishingyear)
values ('hellotx33', 2022)").then())
28             .then();
29     }
30
31 }
```

BookService.java

```
1 package com.lagou.webflux.demo.service;
2
3 import com.lagou.webflux.demo.entity.Book;
4 import reactor.core.publisher.Flux;
5 import reactor.core.publisher.Mono;
6
7 public interface BookService {
8
9     Mono<Book> saveBook(Book book);
10    Flux<Book> findBookAll();
11
12 }
```

### BookServiceImpl.java

```
1 package com.lagou.webflux.demo.service.impl;
2
3 import com.lagou.webflux.demo.entity.Book;
4 import com.lagou.webflux.demo.repository.BookRepository;
5 import com.lagou.webflux.demo.service.BookService;
6 import org.springframework.stereotype.Service;
7 import org.springframework.transaction.annotation.Transactional;
8 import reactor.core.publisher.Flux;
9 import reactor.core.publisher.Mono;
10
11 @Service
12 public class BookServiceImpl implements BookService {
13
14     private final BookRepository repository;
15
16     public BookServiceImpl(BookRepository repository) {
17         this.repository = repository;
18     }
19
20     @Transactional
21     @Override
22     public Mono<Book> saveBook(Book book) {
23         final Mono<Book> save = repository.save(book);
24
25         int i = 1 / 0;
26
27         return save;
28     }
29
30     @Override
31     public Flux<Book> findBookAll() {
32         return repository.findAll();
33     }
34 }
```

测试类：

```
1 package com.lagou.webflux.demo;
2
3 import com.lagou.webflux.demo.entity.Book;
4 import com.lagou.webflux.demo.repository.BookRepository;
5 import com.lagou.webflux.demo.repository.BookRepository1;
6 import com.lagou.webflux.demo.service.BookService;
7 import org.junit.jupiter.api.Test;
8 import org.springframework.beans.factory.annotation.Autowired;
9 import org.springframework.boot.test.context.SpringBootTest;
10 import reactor.core.publisher.Flux;
11 import reactor.core.publisher.Mono;
12
13 import java.time.Duration;
14
15 @SpringBootTest
16 class webflux144ApplicationTests {
17
18     @Test
19     public void test(@Autowired BookRepository repository) throws
InterruptedException {
20         System.out.println(repository);
21
22         final Mono<Book> springwebflux = repository.save(new
Book("springwebflux", 2020));
23         springwebflux.subscribe(
24             book -> System.out.println(book),
25             ex -> System.err.println(ex),
26             () -> System.out.println("处理结束"))
27         );
28         Thread.sleep(5000);
29     }
30
31     @Test
32     public void testfindTheLatestBooks(@Autowired BookRepository repository)
throws InterruptedException {
33         System.out.println(repository);
34         final Flux<Book> theLatestBooks = repository.findTheLatestBooks();
35         theLatestBooks.subscribe(System.out::println);
36         Thread.sleep(5000);
37     }
38
39     @Test
40     public void testInsertBooks(@Autowired BookRepository repository) throws
InterruptedException {
41         final Mono<Book> bookMono = repository.save(new Book("webflux",
2020));
42         bookMono.subscribe(System.out::println);
43         System.out.println("ok");
44         Thread.sleep(5000);
45     }
46
47     @Test
48     public void testFindAll(@Autowired BookRepository repository) throws
InterruptedException {
49         final Flux<Book> all = repository.findAll();
50         all.subscribe(System.out::println);
51         Thread.sleep(5000);
52     }
```

```

53
54     @Test
55     public void testService(@Autowired BookService service) {
56         service.saveBook(new Book("我的抗疫人生1", 2020))
57             .block(Duration.ofSeconds(5));
58     }
59
60     @Test
61     public void testServicefindAll(@Autowired BookService service) throws
62     InterruptedException {
62         service.findAll()
63             .subscribe(System.out::println);
64         Thread.sleep(10000);
65     }
66
67     @Test
68     public void testRepo1(@Autowired BookRepository1 bookRepository1) throws
69     InterruptedException {
70         bookRepository1.insertRows()
71             .subscribe(System.out::println);
72         Thread.sleep(5000);
73     }
74 }
```

## 14.5 Spring Data R2DBC集成MySQL

新建springboot项目

添加依赖:

```

1 <properties>
2     <java.version>11</java.version>
3 </properties>
4 <dependencies>
5     <dependency>
6         <groupId>org.springframework.boot</groupId>
7         <artifactId>spring-boot-starter-data-r2dbc</artifactId>
8     </dependency>
9     <dependency>
10        <groupId>org.springframework.boot</groupId>
11        <artifactId>spring-boot-starter-webflux</artifactId>
12    </dependency>
13    <dependency>
14        <groupId>dev.miku</groupId>
15        <artifactId>r2dbc-mysql</artifactId>
16        <scope>runtime</scope>
17    </dependency>
18    <dependency>
19        <groupId>mysql</groupId>
20        <artifactId>mysql-connector-java</artifactId>
21        <scope>runtime</scope>
```

```

22     </dependency>
23     <dependency>
24         <groupId>org.projectlombok</groupId>
25         <artifactId>lombok</artifactId>
26         <optional>true</optional>
27     </dependency>
28     <dependency>
29         <groupId>org.springframework.boot</groupId>
30         <artifactId>spring-boot-starter-test</artifactId>
31         <scope>test</scope>
32     </dependency>
33     <dependency>
34         <groupId>io.projectreactor</groupId>
35         <artifactId>reactor-test</artifactId>
36         <scope>test</scope>
37     </dependency>
38 </dependencies>
39 <build>
40     <plugins>
41         <plugin>
42             <groupId>org.springframework.boot</groupId>
43             <artifactId>spring-boot-maven-plugin</artifactId>
44         </plugin>
45     </plugins>
46 </build>

```

创建数据库spring\_r2dbc，执行如下SQL：

```
1 | CREATE DATABASE `spring_r2dbc`;
```

**创建表：**

基础字段

字段代码	字段名称	说明
id	编号	主键，自增
remark	备注	可选
active	有效标志	缺省为 1
createdAt	创建时间	默认为 CURRENT_TIMESTAMP
createdBy	创建人	
updatedAt	更新时间	默认为 CURRENT_TIMESTAMP，记录更新时自动更新
updatedBy	更新人	

除了基础字段，还包含以下主要字段：

字段代码	字段名称	说明
code	学号	
name	姓名	
gender	性别	M: 男 F: 女
birthday	生日	
address	家庭地址	

## 建表语句

```

1 CREATE TABLE `student` (
2     `id` int(11) AUTO_INCREMENT,
3     `code` varchar(50) NOT NULL,
4     `name` varchar(50) NOT NULL,
5     `gender` char(1) NOT NULL,
6     `birthday` date NOT NULL,
7     `address` varchar(300) NULL,
8     `remark` varchar(1000) NULL,
9     `active` tinyint NOT NULL DEFAULT 1,
10    `createdAt` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0),
11    `createdBy` varchar(50) NOT NULL,
12    `updatedAt` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE
CURRENT_TIMESTAMP(0),
13    `updatedBy` varchar(50) NOT NULL,
14    PRIMARY KEY (`id`),
15    UNIQUE INDEX `idx_main`(`code`)
16 );

```

## 准备测试数据

```

1 insert into student(code, name, gender, birthday, address, createdBy,
updatedBy)
2 values
3     ('S0001', 'Tom', 'M', '2001-03-05', null, 'TEST', 'TEST')
4     , ('S0002', 'Ted', 'M', '2001-06-12', null, 'TEST', 'TEST')
5     , ('S0003', 'Mary', 'F', '2001-09-12', 'Chicago', 'TEST', 'TEST')
6 ;

```

## 配置数据源

修改配置文件 application.yml，加入以下配置：

```
1 spring:
2   r2dbc:
3     url: r2dbc:mysql://192.168.100.101:3306/spring_r2dbc
4     username: root
5     password: 123456
```

## 创建实体类

代码如下：

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.Data;
4 import org.springframework.data.annotation.Id;
5 import org.springframework.data.annotation.ReadOnlyProperty;
6
7 import java.time.LocalDate;
8 import java.time.LocalDateTime;
9
10 @Data
11 public class Student {
12   @Id
13   private Long id;
14
15   private String code;
16   private String name;
17   private String gender;
18   private LocalDate birthday;
19   private String address;
20
21   private String remark;
22   private boolean active;
23
24   @ReadOnlyProperty
25   private LocalDateTime createdAt;
26   private String createdBy;
27
28   @ReadOnlyProperty
29   private LocalDateTime updatedAt;
30   private String updatedBy;
31 }
```

## 创建仓库类

Spring Data R2DBC 基本沿用了Spring Data JPA的概念，但是功能上没有 JPA 那么强大。

代码如下：

```
1 package com.example.webfluxmysqldemo.repository;
2
3 import com.example.webfluxmysqldemo.entity.Student;
4 import org.springframework.data.repository.reactive.ReactiveCrudRepository;
5
6 public interface StudentRepository extends ReactiveCrudRepository<Student,
7 Long> {
8 }
```

## 创建控制器

代码如下：

```
1 package com.example.webfluxmysqldemo.controller;
2
3 import com.example.webfluxmysqldemo.entity.Student;
4 import com.example.webfluxmysqldemo.repository.StudentRepository;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RestController;
8 import reactor.core.publisher.Flux;
9
10 @RestController
11 @RequestMapping("/api/students")
12 public class StudentController {
13     private final StudentRepository studentRepository;
14
15     public StudentController(StudentRepository studentRepository) {
16         this.studentRepository = studentRepository;
17     }
18
19     @GetMapping
20     public Flux<Student> index() {
21         return studentRepository.findAll();
22     }
23 }
```

## 启动并访问

使用<http://localhost:8080/students>访问接口。

## 14.6 SpringWebFlux集成MongoDB

从Spring Framework 5.2 M2开始，Spring通过`ReactiveTransactionManager` SPI 支持响应式事务管理。

`ReactiveTransactionManager` 是使用事务资源的响应式和非阻塞集成的事务管理抽象。它是一个会返回`Publisher` 的响应式`@Transactional`方法元注解，使用`TransactionalOperator` 实现可编程的事务管理。

两个响应式事务管理器实现是：

- R2DBC通过Spring Data R2DBC
- MongoDB通过Spring Data MongoDB

### Spring Data MongoDB

启动mongodb

```
1 mongod -f 37017.conf
2 mongod -f 37018.conf
3 mongod -f 37019.conf
```

创建Spring Boot项目

添加依赖：

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-webflux</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.projectlombok</groupId>
11  <artifactId>lombok</artifactId>
12  <optional>true</optional>
13 </dependency>
```

`application.yml` 添加n内容：

```
1 spring:
2   data:
3     mongodb:
4       #      host: node2
5       #      port: 27017
6       #      database: test-db
7       uri: mongodb://node2:37017,node2:37018,node2:37019/test-db
```

实体:

```
1 import lombok.Data;
2 import lombok.NoArgsConstructor;
3 import lombok.With;
4 import org.bson.types.ObjectId;
5 import org.springframework.data.annotation.Id;
6 import org.springframework.data.mongodb.core.index.Indexed;
7 import org.springframework.data.mongodb.core.mapping.Document;
8 import org.springframework.data.mongodb.core.mapping.Field;
9
10 import java.util.Arrays;
11 import java.util.List;
12
13 @Document(collection = "book")
14 @Data
15 @NoArgsConstructor
16 public class Book {
17   @Id
18   private ObjectId id;
19
20   @Indexed
21   private String title;
22
23   @Field("pubYear")
24   private int publishingYear;
25
26   @Indexed
27   private List<String> authors;
28
29   public Book(String title, int publishingYear, String... authors) {
30     this.title = title;
31     this.publishingYear = publishingYear;
32     this.authors = Arrays.asList(authors);
33   }
34 }
```

Repository:

```
1 import com.lagou.webflux.demo.entity.Book;
2 import org.bson.types.ObjectId;
3 import org.springframework.data.mongodb.repository.Query;
4 import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
5 import org.springframework.stereotype.Repository;
```

```

6 import reactor.core.publisher.Flux;
7
8 @Repository
9 public interface BookSpringDataMongoRepository extends
10   ReactiveMongoRepository<Book, ObjectId> {
11
12   Flux<Book> findByAuthorsOrderByPublishingYearDesc(String... authors);
13
14   @Query("{ 'authors.1': { $exists: true } }")
15   Flux<Book> booksWithFewAuthors();
16 }
```

测试类：

```

1 @SpringBootTest
2 class Webflux1461ApplicationTests {
3
4   private static final Logger log =
5     LoggerFactory.getLogger(Webflux1461ApplicationTests.class);
6
7   @Test
8   void contextLoads(@Autowired BookSpringDataMongoRepository bookRepo) {
9
10   bookRepo.saveAll(Flux.just(
11       new Book("The Martian", 2011, "Andy Weir"),
12       new Book("Blue Mars", 1996, "Kim Stanley Robinson"),
13       new Book("Artemis", 2017, "Andy Weir"),
14       new Book("The Expanse: Leviathan Wakes", 2011, "Daniel
15 Abraham", "Ty Franck"),
16       new Book("The War of the Worlds", 1898, "H. G. Wells"),
17       new Book("The Expanse: Caliban's War", 2012, "Daniel
18 Abraham", "Ty Franck"))
19     .blockLast());
20
21   log.info("Books saved in DB");
22
23   bookRepo.findAll()
24     .doOnNext(book -> System.out.println(book))
25     .blockLast();
26
27   System.out.println("=====");
28
29   bookRepo.findByAuthorsOrderByPublishingYearDesc("Daniel Abraham")
30     .doOnNext(book -> System.out.println(book))
31     .blockLast();
32   System.out.println("=====");
33
34   bookRepo.booksWithFewAuthors()
35     .doOnNext(System.out::println)
36     .blockLast();
37   System.out.println("=====");
38
39   log.info("Application finished successfully!");
40 }
```

```

39     private String toString(Iterable<Book> books) {
40         StringBuilder sb = new StringBuilder();
41         books.iterator().forEachRemaining(b ->
42             sb.append(" - ").append(b.toString()).append("\n"));
43         return sb.toString();
44     }
45
46     @Test
47     public void testInsert(@Autowired ReactiveMongoTemplate template) throws
48     InterruptedException {
49
50         CountDownLatch latch = new CountDownLatch(1);
51
52         Book book = new Book("The Martian", 2011, "Andy Weir");
53         template.insert(book)
54             .doOnNext(book1 -> System.out.println(book))
55             .flatMap(book1 -> template.findById(book.getId(),
56                     Book.class))
57                 .doOnNext(book1 -> System.out.println("找到: " + book1))
58                 .doOnTerminate(() -> latch.countDown())
59                 .subscribe();
60
61         latch.await();
62     }
63
64     @Test
65     public void testDelete(@Autowired BookSpringDataMongoRepository
66     repository) {
67         ObjectId id = new ObjectId("5fe4a3bae2c93a52c16b64a1");
68         repository.findById(id)
69             .doOnNext(book -> System.out.println("删除之前: " + book))
70             .then(repository.deleteById(id))
71             .map(book -> repository.findById(id))
72             .doOnNext(book -> System.out.println("删除之后: " + book))
73             .doOnTerminate(() -> System.out.println("运行结束"))
74             .block();
75     }
76 }

```

## 14.7 SpringWebFlux集成Redis

构造请求:

```

1 # 测试1
2 curl http://192.168.100.1:8080/city/210
3 curl -H "Content-Type: application/json" \
4     -d '{"id":210,"provinceId":211,"cityName":"beijing","description":"big
5 city"}' \
6     --request POST http://192.168.100.1:8080/city
7 curl -XDELETE http://192.168.100.1:8080/city/210

```

```

8 # 测试2
9 curl http://192.168.100.1:8080/city2/210
10 curl -H "Content-Type: application/json" \
11     -d '{"id":210,"provinceId":211,"cityName":"beijing","description":"big
12     city"}' \
13     --request POST http://192.168.100.1:8080/city2
14 curl -XDELETE http://192.168.100.1:8080/city2/210
15
16 # 测试3
17 curl http://192.168.100.1:8080/city3/5
18 curl http://192.168.100.1:8080/city3
19 curl --header "Content-Type:application/json" \
20     -d '{"id":5,"provinceId":555, "cityName":"beijing", "description":"big
21     city"}' \
22     --request POST http://192.168.100.1:8080/city3
23 curl --header "Content-Type:application/json" \
24     -d '{"id":5,"provinceId":111, "cityName":"nanjing", "description":"big
25     city"}' \
26     --request PUT http://192.168.100.1:8080/city3
27 curl -XDELETE http://192.168.100.1:8080/city3/5

```

## 1. 创建一个springboot2的项目

pom.xml代码如下：

```

1 <properties>
2     <java.version>11</java.version>
3     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
4     <project.reporting.outputEncoding>UTF-
5 </project.reporting.outputEncoding>
6 <dependencies>
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
10    </dependency>
11    <dependency>
12        <groupId>org.springframework.boot</groupId>
13        <artifactId>spring-boot-starter-webflux</artifactId>
14    </dependency>
15    <!-- Spring Boot 响应式 MongoDB 依赖 -->
16    <dependency>
17        <groupId>org.springframework.boot</groupId>
18        <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
19    </dependency>
20    <dependency>
21        <groupId>org.projectlombok</groupId>
22        <artifactId>lombok</artifactId>
23        <optional>true</optional>
24    </dependency>
25    <dependency>
26        <groupId>org.springframework.boot</groupId>

```

```
27      <artifactId>spring-boot-starter-test</artifactId>
28      <scope>test</scope>
29    </dependency>
30    <dependency>
31      <groupId>io.projectreactor</groupId>
32      <artifactId>reactor-test</artifactId>
33      <scope>test</scope>
34    </dependency>
35  </dependencies>
36 <build>
37   <plugins>
38     <plugin>
39       <groupId>org.springframework.boot</groupId>
40       <artifactId>spring-boot-maven-plugin</artifactId>
41     </plugin>
42   </plugins>
43 </build>
```

application.properties的配置如下：

```
1 spring.redis.host=192.168.100.101
2 spring.redis.port=6379
3 ## Redis服务器连接密码（默认为空）
4 #spring.redis.password=xxxxxxxx
5 # 连接超时时间（毫秒）
6 spring.redis.timeout=5000
7
8 # mongodb配置
9 spring.data.mongodb.host=192.168.100.101
10 spring.data.mongodb.port=27017
11 spring.data.mongodb.database=admin
12 spring.data.mongodb.username=admin
13 spring.data.mongodb.password=admin
```

2. 实体类：

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.Data;
4 import org.springframework.data.annotation.Id;
5
6 import java.io.Serializable;
7
8 /**
9  * city序列化，因为需要存到 Redis。
10 */
11 @Data
12 public class City implements Serializable {
13
14     private static final long serialVersionUID = -1L;
15
16     // 城市编号
17     @Id
```

```
18     private Long id;
19
20     // 省份编号
21     private Long provinceId;
22
23     // 城市名称
24     private String cityName;
25
26     // 描述
27     private String description;
28 }
```

redis的配置如下：

```
1 package com.lagou.webflux.demo.config;
2
3 import org.springframework.cache.annotation.CachingConfigurerSupport;
4 import org.springframework.cache.annotation.EnableCaching;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7 import
8 org.springframework.data.redis.connection.ReactiveRedisConnectionFactory;
9 import org.springframework.data.redis.connection.RedisConnectionFactory;
10 import org.springframework.data.redis.core.ReactiveRedisTemplate;
11 import org.springframework.data.redis.core.RedisTemplate;
12 import
13 org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer
14 ;
15 import org.springframework.data.redis.serializer.RedisSerializationContext;
16 import org.springframework.data.redis.serializer.RedisSerializer;
17 import org.springframework.data.redis.serializer.StringRedisSerializer;
18
19 @Configuration
20 @EnableCaching
21 public class MyRedisConfig extends CachingConfigurerSupport {
22
23     private static final StringRedisSerializer STRING_SERIALIZER = new
24 StringRedisSerializer();
25     private static final GenericJackson2JsonRedisSerializer
26 JACKSON_SERIALIZER = new GenericJackson2JsonRedisSerializer();
27
28     @Bean
29     public RedisTemplate<String, Object>
30     redisTemplate(RedisConnectionFactory factory) {
31         RedisTemplate<String, Object> redisTemplate = new
32 RedisTemplate<String, Object>();
33         redisTemplate.setConnectionFactory(factory);
34
35         redisTemplate.setKeySerializer(STRING_SERIALIZER);
36         redisTemplate.setValueSerializer(JACKSON_SERIALIZER);
37
38         return redisTemplate;
39     }
40 }
```

```

34     @Bean
35     public ReactiveRedisTemplate<String, Object>
36     reactiveRedisTemplate(ReactiveRedisConnectionFactory factory) {
37
38         RedisSerializer serializer = new StringRedisSerializer();
39         GenericJackson2JsonRedisSerializer valueSerializer = new
40         GenericJackson2JsonRedisSerializer();
41
42         RedisSerializationContext<String, Object> context =
43             RedisSerializationContext.newSerializationContext()
44                 .key(serializer)
45                 .value(valueSerializer)
46                 .hashKey(serializer)
47                 .hashValue(valueSerializer).build();
48
49     }
50 }
```

### 3. Redis的测试代码:

Redis的测试controller如下，使用Redis的非reactive template操作：

```

1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.City;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.data.redis.core.RedisTemplate;
6 import org.springframework.data.redis.core.ValueOperations;
7 import org.springframework.web.bind.annotation.*;
8 import reactor.core.publisher.Mono;
9
10 import java.util.concurrent.TimeUnit;
11
12 @RestController
13 @RequestMapping("/city")
14 public class CityController {
15
16     /**
17      * RedisTemplate 实现操作 Redis; 同步操作, 非异步
18      */
19     @Autowired
20     private RedisTemplate redisTemplate;
21
22     /**
23      * curl http://192.168.100.1:8080/city/210
24      * @param id
25      * @return
26      */
27     @GetMapping(value = "/{id}")
28     public Mono<City> findCityById(@PathVariable("id") Long id) {
```

```

29         String key = "city_" + id;
30         valueOperations<String, City> operations =
redisTemplate.opsForValue();
31         boolean hasKey = redisTemplate.hasKey(key);
32         City city = operations.get(key);
33
34         if (!hasKey) {
35             return Mono.create(monosink -> monosink.success(null));
36         }
37         return Mono.create(monosink -> monosink.success(city));
38     }
39
40 /**
41 * curl -H "Content-Type: application/json" \
42 * -d
'{"id":210,"provinceId":211,"cityName":"beijing","description":"big city"}'
\
43 * --request POST http://192.168.100.1:8080/city
44 *
45 * @param city
46 * @return
47 */
48 @PostMapping
49 public Mono<City> saveCity(@RequestBody City city) {
    redisTemplate.opsForValue().set("city_" + city.getId(), city, 60,
TimeUnit.SECONDS);
    return Mono.create(monosink -> monosink.success(city));
50 }
51
52 /**
53 * curl -XDELETE http://192.168.100.1:8080/city/210
54 * @param id
55 * @return
56 */
57 @DeleteMapping(value = "/{id}")
58 public Mono<Long> deleteCity(@PathVariable("id") Long id) {
    String key = "city_" + id;
    boolean hasKey = redisTemplate.hasKey(key);
    if (hasKey) {
        redisTemplate.delete(key);
    }
    return Mono.create(monosink -> monosink.success(id));
59 }
60
61 }
62
63 }
64
65 }
66
67 }
68
69 }

```

#### 4. 添加Redis的reactive template

上面的redis操作不具有reactive的特性，下面是具有reactive特性的redis操作，代码如下：

```

1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.City;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.data.redis.core.ReactiveRedisTemplate;
6 import org.springframework.data.redis.core.ReactiveValueOperations;

```

```
7 import org.springframework.web.bind.annotation.*;
8 import reactor.core.publisher.Mono;
9
10 @RestController
11 @RequestMapping("/city2")
12 public class ReactiveCityController {
13
14     /**
15      *
16      * 持 Reactive 的操作类为 ReactiveRedisTemplate
17      * ReactiveValueOperations 是 String (或 value) 的操作视图,
18      * 操作视图还有
19      *   ReactiveHashOperations
20      *   ReactiveListOperations
21      *   ReactiveSetOperations
22      *   ReactiveZSetOperations 等。
23     */
24     @Autowired
25     private ReactiveRedisTemplate reactiveRedisTemplate;
26
27     /**
28      * curl http://192.168.100.1:8080/city2/210
29      * @param id
30      * @return
31     */
32     @GetMapping(value = "/{id}")
33     public Mono<City> findCityById(@PathVariable("id") Long id) {
34         String key = "city_" + id;
35         ReactiveValueOperations<String, City> operations =
36         reactiveRedisTemplate.opsForValue();
37         Mono<City> city = operations.get(key);
38         return city;
39     }
40
41     /**
42      * curl -H "Content-Type: application/json" \
43      * -d
44      ' {"id":210,"provinceId":211,"cityName":"beijing","description":"big city"}'
45     */
46     /**
47      * --request POST http://192.168.100.1:8080/city2
48      *
49      * @param city
50      * @return
51     */
52     @PostMapping
53     public Mono<City> saveCity(@RequestBody City city) {
54         return reactiveRedisTemplate.opsForValue()
55             .getAndSet("city_" + city.getId(), city);
56     }
57
58     /**
59      * curl -XDELETE http://192.168.100.1:8080/city2/210
60      * @param id
61      * @return
62     */
63     @DeleteMapping(value = "/{id}")
64     public Mono<Long> deleteCity(@PathVariable("id") Long id) {
65         String key = "city_" + id;
```

```
62         return reactiveRedisTemplate.delete(key);  
63     }  
64  
65 }
```

## 5. 整合Redis和MongoDB

DAO层的代码如下：

```
1 package com.lagou.webflux.demo.repository;  
2  
3 import com.lagou.webflux.demo.entity.City;  
4 import org.springframework.data.mongodb.repository.ReactiveMongoRepository;  
5 import org.springframework.stereotype.Repository;  
6  
7 @Repository  
8 public interface CityRepository extends ReactiveMongoRepository<City,  
9 ObjectId> {  
10 }
```

新增一个handler，代码如下：

```
1 package com.lagou.webflux.demo.handler;  
2  
3 import com.lagou.webflux.demo.entity.City;  
4 import com.lagou.webflux.demo.repository.CityRepository;  
5 import org.slf4j.Logger;  
6 import org.slf4j.LoggerFactory;  
7 import org.springframework.beans.factory.annotation.Autowired;  
8 import org.springframework.data.redis.core.RedisTemplate;  
9 import org.springframework.data.redis.core.ValueOperations;  
10 import org.springframework.stereotype.Component;  
11 import reactor.core.publisher.Flux;  
12 import reactor.core.publisher.Mono;  
13  
14 @Component  
15 public class CityHandler {  
16  
17     private static final Logger LOGGER =  
18     LoggerFactory.getLogger(CityHandler.class);  
19  
20     @Autowired  
21     private RedisTemplate redisTemplate;  
22  
23     private final CityRepository cityRepository;  
24  
25     @Autowired  
26     public CityHandler(CityRepository cityRepository) {  
27         this.cityRepository = cityRepository;  
28     }  
29  
30     public Mono<City> save(City city) {  
31         return cityRepository.save(city);  
32     }
```

```
31     }
32
33     /**
34      * 如果缓存存在，从缓存中获取城市信息；
35      * 如果缓存不存在，从 DB 中获取城市信息，然后插入缓存
36      *
37      * @param id
38      * @return
39      */
40     public Mono<City> findCityById(Long id) {
41         // 从缓存中获取城市信息
42         String key = "city_" + id;
43         valueOperations<String, City> operations =
44             redisTemplate.opsForValue();
45         // 缓存存在
46         boolean hasKey = redisTemplate.hasKey(key);
47         if (hasKey) {
48             City city = operations.get(key);
49
50             LOGGER.info("CityHandler.findCityById() : 从缓存中获取了城市 >> "
51             + city.toString());
52             return Mono.create(cityMonoSink -> cityMonoSink.success(city));
53         }
54
55         // 从 MongoDB 中获取城市信息
56         Mono<City> cityMono = cityRepository.findById(id);
57         if (cityMono == null)
58             return cityMono;
59
60         // 插入缓存
61         cityMono.subscribe(cityObj -> {
62             operations.set(key, cityObj);
63             LOGGER.info("CityHandler.findCityById() : 城市插入缓存 >> " +
64             cityObj.toString());
65         });
66
67         return cityMono;
68     }
69
70
71     public Flux<City> findAllCity() {
72         return cityRepository.findAll().cache();
73     }
74
75     public Mono<City> modifyCity(City city) {
76         Mono<City> cityMono = cityRepository.save(city);
77         // 缓存存在，删除缓存
78         String key = "city_" + city.getId();
79         boolean hasKey = redisTemplate.hasKey(key);
80         if (hasKey) {
81             redisTemplate.delete(key);
82
83             LOGGER.info("CityHandler.modifyCity() : 从缓存中删除城市 ID >> "
84             + city.getId());
85         }
86
87         return cityMono;
88     }
89
90 }
```

```

85     /**
86      * 如果缓存存在，删除；
87      * 如果缓存不存在，不操作
88      */
89     public Mono<Long> deleteCity(Long id) {
90         cityRepository.deleteById(id);
91         String key = "city_" + id;
92         boolean hasKey = redisTemplate.hasKey(key);
93         if (hasKey) {
94             redisTemplate.delete(key);
95             LOGGER.info("CityHandler.deleteCity() : 从缓存中删除城市 ID >> "
+ id);
96         }
97         return Mono.create(cityMonesink -> cityMonesink.success(id));
98     }
99
100 }

```

新增一个controller，代码如下：

```

1 package com.lagou.webflux.demo.controller;
2
3 import com.lagou.webflux.demo.entity.City;
4 import com.lagou.webflux.demo.handler.CityHandler;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.*;
7 import reactor.core.publisher.Flux;
8 import reactor.core.publisher.Mono;
9
10 @RestController
11 @RequestMapping("/city3")
12 public class ReactiveMongoCityController {
13
14     @Autowired
15     private CityHandler cityHandler;
16
17     /**
18      * curl http://192.168.100.1:8080/city3/5
19      * @param id
20      * @return
21      */
22     @GetMapping(value = "/{id}")
23     public Mono<City> findCityById(@PathVariable("id") Long id) {
24         return cityHandler.findCityById(id);
25     }
26
27     /**
28      * curl http://192.168.100.1:8080/city3
29      * @return
30      */
31     @GetMapping
32     public Flux<City> findAllCity() {
33         return cityHandler.findAllCity();
34     }
35

```

```

36  /**
37   * curl --header "Content-Type:application/json" \
38   * -d '{"id":5,"provinceId":555, "cityName":"beijing",
39   "description":"big city"}' \
40   * --request POST http://192.168.100.1:8080/city3
41   * @param city
42   * @return
43   */
44   @PostMapping
45   public Mono<City> saveCity(@RequestBody City city) {
46     return cityHandler.save(city);
47   }
48
49 /**
50  * curl --header "Content-Type:application/json" \
51  * -d '{"id":5,"provinceId":111, "cityName":"nanjing",
52  "description":"big city"}' \
53  * --request PUT http://192.168.100.1:8080/city3
54  * @param city
55  * @return
56  */
57   @PutMapping
58   public Mono<City> modifyCity(@RequestBody City city) {
59     return cityHandler.modifyCity(city);
60   }
61
62 /**
63  * curl -XDELETE http://192.168.100.1:8080/city3/5
64  * @param id
65  * @return
66  */
67   @DeleteMapping(value = "/{id}")
68   public Mono<Long> deleteCity(@PathVariable("id") Long id) {
69     return cityHandler.deleteCity(id);
70   }

```

## 14.8 使用rxjava2-jdbc库将同步持久化库转换为响应式持久化库

David Moten创建的rxjava2-jdbc库，可以非阻塞、响应式的方式封装 JDBC 驱动。

该库基于 RxJava 2 构建，并使用专用线程池和非阻塞连接池，因此，请求不会在等待连接可用时阻塞线程。

一旦连接可用，查询就开始在连接上执行并阻塞线程。

该库具有流式 DSL，可以发送 SQL 语句并以响应式流的方式接收结果。

创建项目：

建表语句:

```
1 create table wallet
2 (
3     id int auto_increment primary key,
4     owner varchar(255) not null,
5     balance int not null,
6     deposits int not null,
7     withdraws int not null
8 );
9
10 create table book (
11     id varchar(64) primary key,
12     title varchar(255) not null,
13     publishing_year integer not null
14 );
15
16 insert into book values('98cf43e-1967-47a1-ace7-8399a29866a0', 'The
Martian', 2011);
17 insert into book values('99cf43e-1111-3333-ace7-8399a29866a0', 'Blue Mars',
1996);
18 insert into book values('11111111-1967-2343-7777-000000000000', 'The Case for
Mars', 1996);
19 insert into book values('99999999-1967-47a1-aaaa-8399a29866a0', 'The War of
Worlds', 1897);
20 insert into book values('99cf43e-1967-3344-e34a-22222233333', 'Edison''s
Conquest of Mars', 1947);
```

添加依赖:

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-webflux</artifactId>
4 </dependency>
5 <dependency>
6     <groupId>io.reactivex.rxjava2</groupId>
7     <artifactId>rxjava</artifactId>
8     <version>2.2.20</version>
9 </dependency>
10 <dependency>
11     <groupId>com.github.davidmoten</groupId>
12     <artifactId>rxjava2-jdbc</artifactId>
13     <version>0.2.7</version>
14 </dependency>
15 <dependency>
16     <groupId>org.projectlombok</groupId>
17     <artifactId>lombok</artifactId>
18 </dependency>
19 <dependency>
20     <groupId>com.google.guava</groupId>
21     <artifactId>guava</artifactId>
22     <version>30.1-jre</version>
23 </dependency>
```

```
24 <dependency>
25   <groupId>mysql</groupId>
26   <artifactId>mysql-connector-java</artifactId>
27   <version>5.1.49</version>
28 </dependency>
```

application.yml:

```
1 spring:
2   datasource:
3     url: jdbc:mysql://node2/book?useSSL=false
4     driver-class-name: com.mysql.jdbc.Driver
5     username: root
6     password: 123456
7
8   rxjava2jdbc:
9     pool:
10      size: 10
```

实体bean:

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.RequiredArgsConstructor;
4 import org.davidmoten.rx.jdbc.annotations.Column;
5 import org.davidmoten.rx.jdbc.annotations.Query;
6
7 import java.util.UUID;
8
9 @Query("select id, title, publishing_year from book order by
10 publishing_year")
11 public interface Book {
12   @Column
13   String id();
14
15   @Column
16   String title();
17
18   @Column
19   Integer publishing_year();
20
21   static Book of(String title, Integer publishingYear) {
22     return new Impl(UUID.randomUUID().toString(), title,
23 publishingYear);
24   }
25
26   @RequiredArgsConstructor
27   class Impl implements Book {
28     private final String id;
29     private final String title;
30     private final Integer publishingYear;
31
32     @Override
```

```

31     public String id() {
32         return id;
33     }
34
35     @Override
36     public String title() {
37         return title;
38     }
39
40     @Override
41     public Integer publishing_year() {
42         return publishingYear;
43     }
44 }
45 }
```

repository:

```

1 package com.lagou.webflux.demo.repository;
2
3 import com.lagou.webflux.demo.entity.Book;
4 import io.reactivex.Flowable;
5 import io.reactivex.Maybe;
6 import io.reactivex.Single;
7 import lombok.RequiredArgsConstructor;
8 import org.davidmoten.rx.jdbc.Database;
9 import org.davidmoten.rx.jdbc.tuple.Tuple2;
10 import org.reactivestreams.Publisher;
11 import org.springframework.stereotype.Component;
12
13 @Component
14 @RequiredArgsConstructor
15 public class RxBookRepository {
16     private static final String SAVE_QUERY =
17         "insert into book (id, title, publishing_year) " +
18         "values(:id, :title, :publishing_year) " +
19         "on duplicate key " +
20         "update title=:title, publishing_year=:publishing_year";
21
22     private static final String SELECT_BY_ID =
23         "select * from book where id=:id";
24
25     private static final String SELECT_BY_TITLE =
26         "select * from book where title=:title";
27
28     private static final String SELECT_BY_YEAR_BETWEEN =
29         "select * from book where " +
30         "publishing_year >= :from and publishing_year <= :to";
31
32     private final Database database;
33
34     public Flowable<Book> save(Flowable<Book> books) {
35         return books
36             .flatMap(book -> save(book).toFlowable());
37     }
}
```

```

38
39     public Single<Book> save(Book book) {
40         return database
41             .update(SAVE_QUERY)
42             .parameter("id", book.id())
43             .parameter("title", book.title())
44             .parameter("publishing_year", book.publishing_year())
45             .counts()
46             .ignoreElements()
47             .andThen(single.just(book));
48     }
49
50     public Flowable<Book> findAll() {
51         return database
52             .select(Book.class)
53             .get();
54     }
55
56     public Maybe<Book> findById(String id) {
57         return database
58             .select(SELECT_BY_ID)
59             .parameter("id", id)
60             .autoMap(Book.class)
61             .firstElement();
62     }
63
64     public Maybe<Book> findByTitle(Publisher<String> titlePublisher) {
65         return Flowable.fromPublisher(titlePublisher)
66             .firstElement()
67             .flatMap(title -> database
68                 .select(SELECT_BY_TITLE)
69                 .parameter("title", title)
70                 .autoMap(Book.class)
71                 .firstElement());
72     }
73
74     public Flowable<Book> findByYearBetween(
75         Single<Integer> from,
76         Single<Integer> to
77     ) {
78         return single
79             .zip(from, to, Tuple2::new)
80             .flatMapPublisher(tuple -> database
81                 .select(SELECT_BY_YEAR_BETWEEN)
82                 .parameter("from", tuple._1())
83                 .parameter("to", tuple._2())
84                 .autoMap(Book.class));
85     }
86 }
```

Configuration:

```

1 package com.lagou.webflux.demo.config;
2
3 import org.davidmoten.rx.jdbc.Database;
```

```

4 import org.springframework.beans.factory.annotation.Value;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 @Configuration
9 public class DatabaseConfiguration {
10
11     @Bean
12     public Database database(
13         @Value("${spring.datasource.url}") String url,
14         @Value("${spring.datasource.username}") String username,
15         @Value("${spring.datasource.password}") String password,
16         @Value("${rxjava2jdbc.pool.size}") int poolSize
17     ) {
18
19         Database database = Database.nonBlocking().url(url)
20             .user(username)
21             .password(password)
22             .maxPoolSize(poolSize)
23             .build();
24         return database;
25     }
26
27 }

```

测试案例:

```

1 @Autowired
2 RxBookRepository repository;
3 @Test
4 void contextLoads() {
5 }
6 @Test
7 public void testfindAll() throws InterruptedException {
8     final Flowable<Book> allBooks = repository.findAll();
9     allBooks.subscribe(
10         book -> System.out.println(book),
11         ex -> {
12             System.err.println(ex);
13         },
14         () -> {
15             System.out.println("执行完成");
16         }
17     );
18     Thread.sleep(5000);
19 }
20 @Test
21 public void testfindById() throws InterruptedException {
22     repository.findById("11111111-1967-2343-7777-000000000000")
23         .subscribe(
24             book -> System.out.println(book),
25             ex -> {
26                 System.err.println(ex);
27             },
28             () -> {

```

```
29                     System.out.println("执行完成");
30                 }
31             );
32         Thread.sleep(5000);
33     }
34     @Test
35     public void testFindByTitle() throws InterruptedException {
36         Set<String> names = new HashSet<>();
37         names.add("The Case for Mars");
38         repository.findByTitle(Flowable.fromIterable(names))
39             .subscribe(
40                 book -> System.out.println(book),
41                 ex -> {
42                     System.err.println(ex);
43                 },
44                 () -> {
45                     System.out.println("执行完成");
46                 }
47             );
48         Thread.sleep(5000);
49     }
50     @Test
51     public void testFindByYearBetween() throws InterruptedException {
52         repository.findByYearBetween(Single.just(1990), Single.just(2020))
53             .subscribe(
54                 book -> System.out.println(book),
55                 ex -> {
56                     System.err.println(ex);
57                 },
58                 () -> {
59                     System.out.println("执行完成");
60                 }
61             );
62         Thread.sleep(5000);
63     }
64     @Test
65     public void testSave1() throws InterruptedException {
66         repository.save(new Book() {
67             @Override
68             public String id() {
69                 return "mybook-1";
70             }
71             @Override
72             public String title() {
73                 return "mytitle-1";
74             }
75             @Override
76             public Integer publishing_year() {
77                 return 2020;
78             }
79         }).subscribe();
80         Thread.sleep(2000);
81     }
82     @Test
83     public void testSave2() {
84         repository.save(Flowable.just(new Book() {
85             @Override
86             public String id() {
```

```

87         return "mybook-2";
88     }
89     @Override
90     public String title() {
91         return "mytitle-2";
92     }
93     @Override
94     public Integer publishing_year() {
95         return 2021;
96     }
97 }).blockingFirst();
98 }

```

rxjava2-jdbc 库支持大多数 JDBC 驱动程序。此外，该库还有一些事务支持。事务内的所有操作都必须在同一连接上执行。事务的提交/回滚会自动发生。

rxjava2-jdbc 库很简洁，减少了一些潜在的线程块，并且可以响应式地使用关系型数据库。

但是，到目前为止，它仍然是新的，可能无法处理复杂的响应式工作流，尤其是那些涉及事务的工作流。rxjava2-jdbc 库还需要所有 SQL 查询的定义。

## 14.9 包装同步CrudRepository将同步持久化库转换为响应式持久化库

当已经有一个包含所有必需的数据访问机制的 CrudRepository 实例（不需要手动查询或实体映射），却不能在响应式应用程序中直接使用它。

此时可以编写自己的响应式适配器，它的行为类似于 rxjava2-jdbc，但处于存储库级别。

在应用此方法时要小心 JPA。在使用延迟加载时，我们会很快遇到代理问题。那么，让我们假设有以下由JPA 定义的 Book 实体：

创建项目：

创建数据库：

```

1 CREATE TABLE `book` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `title` varchar(255) DEFAULT NULL,
4   `publishing_year` int(255) DEFAULT NULL,
5   PRIMARY KEY (`id`)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

创建SpringBoot项目

添加依赖：

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-webflux</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.boot</groupId>
7   <artifactId>spring-boot-starter-data-jpa</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.projectlombok</groupId>
11  <artifactId>lombok</artifactId>
12 </dependency>
13 <dependency>
14  <groupId>mysql</groupId>
15  <artifactId>mysql-connector-java</artifactId>
16  <version>5.1.49</version>
17 </dependency>
```

### application.yml

```
1 spring:
2   datasource:
3     driver-class-name: com.mysql.jdbc.Driver
4     username: root
5     password: 123456
6     url: jdbc:mysql://node2/book?useSSL=false
7 logging:
8   level:
9     org.springframework.jdbc.datasource: debug
```

### 实体

```
1 package com.lagou.webflux.demo.entity;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 import javax.persistence.*;
8
9 @Data
10 @NoArgsConstructor
11 @AllArgsConstructor
12 @Entity
13 @Table(name = "book")
14 public class Book {
15   @Id
16   @GeneratedValue(strategy = GenerationType.IDENTITY)
17   private Integer id;
18
19   private String title;
20 }
```

```
21     private Integer publishingYear;
22
23     public Book(String title, int publishingYear) {
24         this(null, title, publishingYear);
25     }
26 }
```

JPA的repository

```
1 package com.lagou.webflux.demo.repository;
2
3 import com.lagou.webflux.demo.entity.Book;
4 import org.springframework.data.jpa.repository.Query;
5 import org.springframework.data.repository.CrudRepository;
6 import org.springframework.stereotype.Repository;
7
8 @Repository
9 public interface BookJpaRepository extends CrudRepository<Book, Integer> {
10
11     Iterable<Book> findByIdBetween(int lower, int upper);
12
13     @Query("SELECT b FROM Book b WHERE " +
14             "LENGTH(b.title)=(SELECT MIN(LENGTH(b2.title)) FROM Book b2)")
15     Iterable<Book> findShortestTitle();
16
17 }
```

适配器，负责将阻塞式访问适配为响应式访问：

```
1 package com.lagou.webflux.demo.adapter;
2
3 import lombok.RequiredArgsConstructor;
4 import org.reactivestreams.Publisher;
5 import org.springframework.data.repository.CrudRepository;
6 import org.springframework.data.repository.reactive.ReactiveCrudRepository;
7 import reactor.core.publisher.Flux;
8 import reactor.core.publisher.Mono;
9 import reactor.core.scheduler.Scheduler;
10
11 @RequiredArgsConstructor
12 public abstract class
13 ReactiveCrudRepositoryAdapter<T, ID, I extends CrudRepository<T, ID>>
14     implements ReactiveCrudRepository<T, ID> {
15
16     protected final I delegate;
17     protected final Scheduler scheduler;
18
19     @Override
20     public <S extends T> Mono<S> save(S entity) {
21         return Mono
22             .fromCallable(() -> delegate.save(entity))
23             .subscribeOn(scheduler);
24     }
25 }
```

```
25
26     @Override
27     public <S extends T> Flux<S> saveAll(Iterable<S> entities) {
28         return Mono.fromCallable(() -> delegate.saveAll(entities))
29             .flatMapMany(Flux::fromIterable)
30             .subscribeOn(scheduler);
31     }
32
33     @Override
34     public <S extends T> Flux<S> saveAll(Publisher<S> entityStream) {
35         return Flux.from(entityStream)
36             .flatMap(entity -> Mono.fromCallable(() ->
37                 delegate.save(entity)))
38             .subscribeOn(scheduler);
39     }
40
41     @Override
42     public Mono<T> findById(ID id) {
43         return Mono.fromCallable(() -> delegate.findById(id))
44             .flatMap(result -> result
45                 .map(Mono::just)
46                 .orElseGet(Mono::empty))
47             .subscribeOn(scheduler);
48     }
49
50     @Override
51     public Mono<T> findById(Publisher<ID> id) {
52         return Mono.from(id)
53             .flatMap(actualId ->
54                 delegate.findById(actualId)
55                     .map(Mono::just)
56                     .orElseGet(Mono::empty))
57             .subscribeOn(scheduler);
58     }
59
60     @Override
61     public Mono<Boolean> existsById(ID id) {
62         return Mono
63             .fromCallable(() -> delegate.existsById(id))
64             .subscribeOn(scheduler);
65     }
66
67     @Override
68     public Mono<Boolean> existsById(Publisher<ID> id) {
69         return Mono.from(id)
70             .flatMap(actualId ->
71                 Mono.fromCallable(() ->
72                     delegate.existsById(actualId)))
73             .subscribeOn(scheduler);
74     }
75
76     @Override
77     public Flux<T> findAll() {
78         return Mono
79             .fromCallable(delegate::findAll)
80             .flatMapMany(Flux::fromIterable)
81             .subscribeOn(scheduler);
82     }
```

```
81
82     @Override
83     public Flux<T> findAllById(Iterable<ID> ids) {
84         return Mono
85             .fromCallable(() -> delegate.findAllById(ids))
86             .flatMapMany(Flux::fromIterable)
87             .subscribeOn(scheduler);
88     }
89
90     @Override
91     public Flux<T> findAllById(Publisher<ID> idStream) {
92         return Flux
93             .from(idStream)
94             .buffer()
95             .flatMap(ids ->
96                 Flux.fromIterable(delegate.findAllById(ids)))
97             .subscribeOn(scheduler);
98     }
99
100    @Override
101    public Mono<Long> count() {
102        return Mono
103            .fromCallable(delegate::count)
104            .subscribeOn(scheduler);
105    }
106
107    @Override
108    public Mono<Void> deleteById(ID id) {
109        return Mono
110            .<Void>fromRunnable(() -> delegate.deleteById(id))
111            .subscribeOn(scheduler);
112    }
113
114    @Override
115    public Mono<Void> deleteById(Publisher<ID> id) {
116        return Mono.from(id)
117            .flatMap(actualId ->
118                Mono
119                    .<Void>fromRunnable(() ->
120                        delegate.deleteById(actualId))
121                    .subscribeOn(scheduler)
122            );
123
124    @Override
125    public Mono<Void> delete(T entity) {
126        return Mono
127            .<Void>fromRunnable(() -> delegate.delete(entity))
128            .subscribeOn(scheduler);
129    }
130
131    @Override
132    public Mono<Void> deleteAll(Iterable<? extends T> entities) {
133        return Mono
134            .<Void>fromRunnable(() -> delegate.deleteAll(entities))
135            .subscribeOn(scheduler);
136    }
```

```

137     @Override
138     public Mono<Void> deleteAll(Publisher<? extends T> entityStream) {
139         return Flux.from(entityStream)
140             .flatMap(entity -> Mono
141                 .fromRunnable(() -> delegate.delete(entity)))
142                 .subscribeOn(scheduler))
143             .then();
144     }
145
146     @Override
147     public Mono<Void> deleteAll() {
148         return Mono
149             .<Void>fromRunnable(delegate::deleteAll)
150             .subscribeOn(scheduler);
151     }
152 }
```

响应式Repository:

```

1 package com.lagou.webflux.demo.repository;
2
3 import com.lagou.webflux.demo.adapter.ReactiveCrudRepositoryAdapter;
4 import com.lagou.webflux.demo.entity.Book;
5 import org.reactivestreams.Publisher;
6 import org.springframework.stereotype.Component;
7 import reactor.core.publisher.Flux;
8 import reactor.core.publisher.Mono;
9 import reactor.core.scheduler.Scheduler;
10
11 @Component
12 public class RxBookRepository extends
13     ReactiveCrudRepositoryAdapter<Book, Integer, BookJpaRepository> {
14
15     public RxBookRepository(
16         BookJpaRepository delegate,
17         Scheduler scheduler
18     ) {
19         super(delegate, scheduler);
20     }
21
22     public Flux<Book> findByIdBetween(
23         Publisher<Integer> lowerPublisher,
24         Publisher<Integer> upperPublisher
25     ) {
26         return Mono.zip(
27             Mono.from(lowerPublisher),
28             Mono.from(upperPublisher)
29         ).flatMapMany(
30             item ->
31
32             Flux.fromIterable(delegate.findByIdBetween(item.getT1(), item.getT2()))
33                 .subscribeOn(scheduler)
34         ).subscribeOn(scheduler);
35     }
36 }
```

```
36     public Flux<Book> findShortestTitle() {
37         return Mono.fromCallable(delegate::findShortestTitle)
38             .subscribeOn(scheduler)
39             .flatMapMany(Flux::fromIterable);
40     }
41 }
```

配置信息：

```
1 package com.lagou.webflux.demo.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5 import reactor.core.scheduler.Scheduler;
6 import reactor.core.scheduler.Schedulers;
7
8 @Configuration
9 public class RxPersistenceConfiguration {
10     @Bean
11     public Scheduler jpaScheduler() {
12         return Schedulers.newParallel("JPA", 10);
13     }
14 }
```

主程序：

```
1 package com.lagou.webflux.demo;
2
3 import com.lagou.webflux.demo.config.RxPersistenceConfiguration;
4 import com.lagou.webflux.demo.entity.Book;
5 import com.lagou.webflux.demo.repository.RxBookRepository;
6 import lombok.RequiredArgsConstructor;
7 import lombok.extern.slf4j.Slf4j;
8 import org.springframework.boot.CommandLineRunner;
9 import org.springframework.boot.SpringApplication;
10 import org.springframework.boot.autoconfigure.SpringBootApplication;
11 import org.springframework.context.annotation.Import;
12 import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
13 import reactor.core.publisher.Flux;
14 import reactor.core.publisher.Mono;
15
16 import java.time.Duration;
17
18
19 @Slf4j
20 @EnableJpaRepositories
21 @RequiredArgsConstructor
22 @SpringBootApplication
23 @Import({
24     RxPersistenceConfiguration.class
25 })
26 public class Webflux149Application implements CommandLineRunner {
27 }
```

```

28     private final RxBookRepository bookRepository;
29
30     public static void main(String[] args) {
31         SpringApplication.run(webflux149Application.class, args);
32     }
33
34     @Override
35     public void run(String... args) {
36         Flux<Book> books = Flux.just(
37             new Book("The Martian", 2011),
38             new Book("Blue Mars", 1996),
39             new Book("The War of the Worlds", 1898),
40             new Book("Artemis", 2016),
41             new Book("The Expanse: Leviathan Wakes", 2011),
42             new Book("The Expanse: Caliban's War", 2012)
43         );
44
45         bookRepository
46             .saveAll(books)
47             .count()
48             .doOnNext(amount -> log.info("{} books saved in DB",
amount))
49             .block();
50
51         Flux<Book> allBooks = bookRepository.findAll();
52         reportResults("All books in DB:", allBooks);
53
54         Flux<Book> andyWeirBooks = bookRepository
55             .findByIdBetween(Mono.just(17), Mono.just(22));
56         reportResults("Books with ids (17..22):", andyWeirBooks);
57
58         Flux<Book> booksWithFewAuthors = bookRepository.findShortestTitle();
59         reportResults("Books with the shortest title:",
booksWithFewAuthors);
60
61         Mono.delay(Duration.ofSeconds(5))
62             .subscribe(e -> log.info("Application finished
successfully!"));
63     }
64
65     private void reportResults(String message, Flux<Book> books) {
66         books.map(Book::toString)
67             .reduce(
68                 new StringBuffer(),
69                 (sb, b) -> sb.append(" - ")
70                     .append(b)
71                     .append("\n"))
72             .doOnNext(sb -> log.info(message + "\n{}", sb))
73             .subscribe();
74     }
75
76 }
```

并非所有阻塞功能都可以轻松映射。例如，JPA 延迟加载很可能被描述的方法所破坏。此外，与 rxjava2-jdbc 库中类似，需要额外的工作以支持事务。或者，需要以一定的粒度包装同步操作，而在该粒度中没有跨越多个阻塞调用的事务扩展。

这里描述的方法并没有将阻塞请求神奇地转换为响应式非阻塞执行。构成 JPA 调度程序的某些线程仍将被阻塞。但是，对调度程序的详细监控以及合理的池管理应该有助于在应用程序的性能和资源使用之间创建可接受的平衡。

## 15 测试响应式应用程序

### 15.1 使用StepVerifier测试响应式流

出于测试目的，Reactor 提供了额外的 reactor-test 模块，该模块提供了 StepVerifier。StepVerifier 提供了一个流式 API，用于为任何 Publisher 构建验证流程。

#### 15.1.1 StepVerifier要点

验证 Publisher 主要有两种方法。第一种是 `StepVerifier.<T>create(Publisher<T> source)`。使用此技术构建的测试如下所示：

```
1 StepVerifier
2     .create(Flux.just("foo", "bar"))
3     .expectSubscription()
4     .expectNext("foo")
5     .expectNext("bar")
6     .expectComplete()
7     .verify();
```

在此示例中，Publisher 应生成两个特定元素，后续操作将验证特定元素是否已传递给最终订阅者。

该类提供的构建器技术可以**定义验证过程中事件发生的顺序**。

根据前面的代码，第一个发出的事件必须是与订阅相关的事件，紧跟其后的事件必须是 `foo` 和 `bar` 字符串。

最后，`StepVerifier#expectCompletion` 定义终止信号的存在。

在此例中，必须是 `Subscriber#onComplete` 的调用，或者成功完成给定的 Flux。

要执行验证，或者说对创建流进行订阅，就必须调用 `.verify()` 方法。

`verify()` 是一个阻塞调用，它阻塞执行，直到流发出所有预期的事件。

通过使用这种简单地技术，可以使用可计数的元素和事件来验证 Publisher。但是，用大量元素来验证流程是很困难的。

如果检查的是该发布者已发出元素是否达到特定数量，可以使用 `.expectNextCount()`。

如下代码：

```
1 StepVerifier
2     // .create(Flux.range(0, 101))
3     .create(Flux.range(0, 100))
4     // .create(Flux.range(0, 99))
5     .expectSubscription()
6     .expectNext(0) // 期望下一个元素与指定的相等
7     .expectNextCount(98) // 从上个期望或从开始订阅开始，期望收到指定个数的元素
8     .expectNext(99) // 期望下一个元素与指定的相等
9     .expectComplete() // 期望收到onComplete信号
10    .verify(); // 阻塞验证
```

尽管 `.expectNextCount()` 方法解决了一部分问题，但在某些情况下，仅仅检查发出元素的数量是不够的。

例如，在验证负责按特定规则过滤或选择元素的代码时，检查所有发出的项是否与过滤规则匹配非常重要。

为此，`StepVerifier` 可以使用 Java Hamcrest 等工具立即记录发出的数据及其验证。

如下代码：

```
1 Publisher<Wallet> usersWallets = findAllUsersWallets();
2
3 StepVerifier.create(usersWallets)
4     .expectSubscription()
5     .recordWith(ArrayList::new)
6     .expectNextCount(1)
7     .consumeRecordedWith(
8         wallets -> assertThat(
9             wallets,
10            everyItem(hasProperty("owner", equalTo("admin"))))
11        )
12    )
13    .expectComplete()
14    .verify();
```

与前面的示例相反，每个期望仅涵盖一个元素或指定数量元素的验证，`.consumeRecordedWith()` 可以验证给定 Publisher 发布的所有元素。应该注意的是 `.consumeRecordedWith()` 只有在指定了 `.recordWith()` 时才有效。反过来，我们应该仔细定义存储记录的集合类。对于多线程发布者而言，用于记录事件的集合类型应该支持并发访问，因此在这些情况下，最好使用 `.recordWith(ConcurrentLinkedQueue :: new)` 而不是 `.recordWith(ArrayList :: new)`，因为与 `ArrayList` 相比，`ConcurrentLinkedQueue` 是线程安全的。

除此之外，还有其他功能相似的方法。例如，对下一个元素的期望的定义，如以下代码所示：

```
1 StepVerifier
2     .create(Flux.just("alpha-foo", "betta-bar"))
3     .expectSubscription()
4     .expectNextMatches(e -> e.startsWith("alpha"))
5     .expectNextMatches(e -> e.startsWith("betta"))
6     .expectComplete()
7     .verify();
```

.expectNextMatches() 和 .expectNext() 之间的唯一区别是，前者可以定义自定义的匹配器 Predicate，这使其比后者更灵活。这是因为 .expectNext() 基于元素之间的比较，而这种比较使用元素的 .equals() 方法。

类似地，.assertNext() 和 .consumeNextwith() 使编写自定义断言成为可能。要注意，.assertNext() 是 .consumeNextwith() 的别名。.expectNextMatches() 和 .assertNext() 之间的区别在于前者接受 Predicate，必须返回 true 或 false，而后者接收可能抛出异常的Consumer，并且捕获消费者抛出的任何 AssertionError，然后通过 .verify() 方法抛出。

如下面的代码所示：

```
1 StepVerifier
2     .create(findUsersUSDWallet())
3     .expectSubscription()
4     .assertNext(wallet -> assertThat(wallet, hasProperty("currency",
5     equalTo("USD"))))
5     .expectComplete().verify();
```

最后，只剩下未覆盖的错误情况，这也是正常系统生命周期的一部分。可以检查错误信号的API方法不是很多，最简单的是 .expectError() 方法，该方法没有参数。

如以下代码所示：

```
1 StepVerifier.create(Flux.error(new RuntimeException("Error")))
2     .expectError()
3     .verify();
```

在某些情况下，测试特定错误类型至关重要。例如，如果用户在登录期间输入了错误的凭据，则安全服务应发出 BadCredentialsException.class。为了验证发出的错误，我们可以使用 .expectError(Class<? extends Throwable>)。

如以下代码所示：

```
1 StepVerifier.create(securityService.login("admin", "wrong"))
2     .expectSubscription()
3     .expectError(BadCredentialsException.class)
4     .verify();
```

还可以使用名为 .expectErrorMatches() 和 .consumeErrorwith() 的扩展，它们能与发出的 Throwable 进行直接交互。

### 15.1.2 使用StepVerifier进行高级测试

发布者测试的第一步是验证无界 Publisher。根据响应式流规范，无限流意味着流永远不会调用 `subscriber#onComplete()` 方法。由于 StepVerifier 将无限期地等待完成信号，因此，测试将被阻塞，直到它被杀死。

为了解决这个问题，StepVerifier 提供了一个取消 API，在满足某些期望时，它可以取消对源的订阅。

如下面的代码所示：

```
1 Flux<String> websocketPublisher = ...;
2 StepVerifier.create(websocketPublisher)
3     .expectSubscription()
4     .expectNext("Connected")
5     .expectNext("Price: $12.00")
6     .thenCancel() // 取消订阅
7     .verify();
```

上述代码表示，在收到 Connected 以及 Price:\$ 12.00 消息后，我们将断开或取消订阅 WebSocket。

系统验证过程的另一个关键阶段是检查 Publisher 的背压行为。例如，通过 WebSocket与外部系统交互会产生一个只推式的 Publisher。防止此类行为的一种简单方法是使用 `.onBackpressureBuffer()` 操作符保护下游。要使用所选的背压策略检查系统是否按预期运行，必须手动控制用户需求。为此，StepVerifier 提供了 `.thenRequest()` 方法，它允许我们控制用户需求。

这由以下代码描述：

```
1 Flux<String> websocketPublisher = ...;
2 Class<Exception> expectedErrorClass =
3     reactor.core.Exceptions.failWithOverflow().getClass();
4 StepVerifier.create(websocketPublisher.onBackpressureBuffer(5), 0) // 使用背压控制
5     .expectSubscription()
6     .thenRequest(1)
7     .expectNext("Connected")
8     .thenRequest(1)
9     .expectNext("Price: $12.00")
10    .expectError(expectedErrorClass)
11    .verify();
```

在前面的示例中，使用的是 `StepVerifier.create()` 方法的重载，它接收初始订阅者的请求作为第二个参数。在单参数方法的重载中，默认需求是 `Long.MAX_VALUE`，即无限需求。

如果元素生成过程需要一些额外的外部交互，我们可以使用 `.then()` 方法。

TestPublisher 实现了响应式流 Publisher，可以直接触发 `onNext()`、`onComplete()` 和 `onError()` 事件以进行测试。

下一个示例演示了如何在测试执行期间触发新事件：

```
1 TestPublisher<String> idsPublisher = TestPublisher.create();
2 StepVerifier.create(walletsRepository.findAllById(idsPublisher))
3     .expectSubscription()
4     .then(() -> idsPublisher.next("1"))
5     .assertNext(w -> assertThat(w, hasProperty("id", equalTo("1"))))
6     .then(() -> idsPublisher.next("2"))
7     .assertNext(w -> assertThat(w, hasProperty("id", equalTo("2"))))
8     .then(idsPublisher::complete)
9     .expectComplete()
10    .verify();
```

通过这种方式，可以验证在该操作之后是否找到已发出的 ID，以及 walletsRepository 的行为是否与预期一致。

### 15.1.3 处理虚拟时间

尽管测试的本质在于覆盖业务逻辑，但还有另一个非常重要的部分应该被考虑在内。要理解这个特性，我们首先应该考虑以下示例代码：

```
1 public Flux<String> sendWithInterval() {
2     return Flux.interval(Duration.ofMinutes(1))
3         .zipWith(Flux.just("a", "b", "c"))
4         .map(Tuple2::getT2);
5 }
```

要使用 StepVerifier 验证此类代码，我们最终可能实现以下测试用例：

```
1 StepVerifier.create(sendWithInterval())
2     .expectSubscription()
3     .expectNext("a", "b", "c")
4     .expectComplete()
5     .verify();
```

正在进行的测试将被传递给之前的 sendWithInterval() 实现，而该实现是真正想要获取的。但是，此测试存在问题。如果运行几次，测试的平均持续时间超过 3 分钟。

发生这种情况是因为 sendWithInterval() 方法在每个元素之前产生 3 个延迟一分钟的事件。

在时间间隔或调度时间为几小时或几天的情况下，系统的验证可能花费大量时间，这在当今的持续集成中是不可接受的。为了解决这个问题，Reactor Test 模块提供了用虚拟时间替换实际时间的能力，如下面的代码所示：

```
1 // 使用虚拟时间
2 StepVerifier.withVirtualTime(() -> sendWithInterval())
3 // 场景验证.....
```

当使用.withVirtualTime()构建器方法时，使用 reactor.test.scheduler.VirtualTimeScheduler 显式替换 Reactor 中的每个 Scheduler。表示 Flux.interval 也将在该 Scheduler 上运行。可以使用 VirtualTimeScheduler#advanceTimeBy 完成对时间的所有控制。

如以下代码所示：

```
1 StepVerifier.withVirtualTime(() -> sendWithInterval())
2     .expectSubscription()
3     .then(
4         () -> virtualTimeScheduler.get().advanceTimeBy(Duration.ofMinutes(3))
5     )
6     .expectNext("a", "b", "c")
7     .expectComplete()
8     .verify();
```

将 .then() 与 VirtualTimeScheduler API 结合使用以使时间适当提前。如果运行该测试，它只需要几毫秒而不是几分钟！这个结果要好得多，因为测试表现现在与数据产生的实际时间间隔无关。

还可以用.thenAwait()替换.then()和 VirtualTimeScheduler 的组合，它们的行为完全相同。

为了限制在验证方案上花费的时间，可以使用.verify(Duration t)重载方法。

如果测试在允许的持续时间内无法完成验证，它将抛出 AssertionError。此外，.verify()方法会返回验证过程实际花费的时间。以下代码描述了这样一个用例：

```
1 Duration took = StepVerifier
2     .withVirtualTime(
3         () -> sendWithInterval()
4     )
5     .expectSubscription()
6     .thenAwait(Duration.ofMinutes(3))
7     .expectNext("a", "b", "c")
8     .expectComplete()
9     .verify();
10
11 System.out.println("Verification took: " + took);
```

如果重点是检查在指定的等待时间内是否没有生成事件，则可以使用一个名为.expectNoEvents()的附加 API 方法。使用此方法，可以检查事件是否按照指定的时间间隔生成，如下所示：

```
1 StepVerifier.withVirtualTime(  
2     () -> sendWithInterval()  
3 )  
4     .expectSubscription()  
5     .expectNoEvent(Duration.ofMinutes(1)) // 用于表示在指定的时间内没有收到事件  
6     .expectNext("a")  
7     .expectNoEvent(Duration.ofMinutes(1))  
8     .expectNext("b")  
9     .expectNoEvent(Duration.ofMinutes(1))  
10    .expectNext("c")  
11    .expectComplete()  
12    .verify();
```

没有参数的.thenAwait()方法的主要思路是：触发尚未执行的任务以及计划在当前虚拟时间或该时间之前执行的任务。

例如，要在下一次设置中接收第一个预定事件，Flux.interval(Duration.ofMillis(0), Duration.ofMillis(1000))将需要额外调用.thenAwait()，如下面的代码所示：

```
1 StepVerifier.withVirtualTime(  
2     () -> Flux.interval(Duration.ofMillis(0), Duration.ofMillis(1000))  
3             .zipWith(Flux.just("a", "b", "c"))  
4             .map(Tuple2::getT2)  
5 )  
6     .expectSubscription()  
7     .thenAwait()  
8     .expectNext("a")  
9     .expectNoEvent(Duration.ofMillis(1000))  
10    .expectNext("b")  
11    .expectNoEvent(Duration.ofMillis(1000))  
12    .expectNext("c")  
13    .expectComplete()  
14    .verify();
```

如果没有.thenAwait()，测试将永远挂起。

#### 15.1.4 验证响应式上下文

最后，最不常见的验证是 Reactor 的 Context。假设要验证身份验证服务的响应式 API。为了验证用户身份，LoginService 希望订阅者提供一个包含身份验证信息的 Context：

```
1 stepVerifier
2     .create(securityService.login("admin", "admin"))
3     .expectSubscription()
4     .expectAccessibleContext()
5     .hasKey("security")
6     .then()
7     .expectComplete()
8     .verify();
```

.expectAccessibleContext() 验证只可能在一种情况下失败，即返回的 Publisher 不是 Reactor 类型 (Flux 或 Mono) 时。

因此，只有可访问的上下文存在时，后续的 Context 验证才会被执行。除了.hasKey()，还有许多其他方法可以对当前上下文进行详细验证。要退出上下文验证，构建器提供了.then()方法。

## 15.2 测试WebFlux

### 15.2.1 使用 WebTestClient 测试控制器

如测试支付服务。在这种场景下，假设支付服务支持 /payments 端点的 GET 和POST 方法。第一个 HTTP 调用负责检索当前用户的已执行支付列表。而第二个可以提交新的支付。

该 REST 控制器的实现如下所示：

```
1 @RestController
2 @RequestMapping("/payments")
3 public class PaymentController {
4     private final PaymentService paymentService;
5
6     public PaymentController(PaymentService paymentService) {
7         this.paymentService = paymentService;
8     }
9
10    @GetMapping("/")
11    public Flux<Payment> list() {
12        return paymentService.list();
13    }
14
15    @PostMapping("/")
16    public Mono<String> send(Mono<Payment> payment) {
17        return paymentService.send(payment);
18    }
19 }
```

WebTestClient 类似于 org.springframework.test.web.servlet.MockMvc。这些测试 Web 客户端之间的唯一区别是WebTestClient 旨在测试 WebFlux 端点。

例如，使用 WebTestClient 和 Mockito 库，可以通过以下方式编写用于检索用户支付列表的验证：

```
1  @Test
2  public void verifyRespondWithExpectedPayments() {
3
4      PaymentService paymentService = Mockito.mock(PaymentService.class);
5      PaymentController controller = new PaymentController(paymentService);
6      prepareMockResponse(paymentService);
7
8      webTestClient
9          .bindToController(controller)
10         .build()
11         .get()
12         .uri("/payments/")
13         .exchange()
14         .expectHeader()
15         .contentTypeCompatibleWith(APPLICATION_JSON)
16         .expectStatus()
17         .is2xxSuccessful()
18         .returnResult(Payment.class)
19         .getResponseBody()
20         .as(StepVerifier::create)
21         .expectNextCount(5)
22         .expectComplete()
23         .verify();
24
25 }
```

在此示例中，使用 WebTestClient 构建了对 PaymentController 的验证。反过来，可以使用 WebTestClient 流式 API 检查响应的状态码和消息头的正确性。此外，可以使用 getResponseBody() 来获取 Flux 响应，最后使用 StepVerifier 对其进行验证。此示例显示，两个工具可以轻松地相互集成。

从前面的示例中可以看到，模拟了 PaymentService，并且在测试 PaymentController 时不与外部服务通信。但是，要检查系统完整性，就必须运行完整的组件，而不仅仅是其中几层。

要运行完整的集成测试，就需要启动整个应用程序。为此，可以将 @SpringBootTest 注解与 @AutoConfigureWebTestClient 注解结合起来使用。WebTestClient 提供与 HTTP 服务器建立 HTTP 连接的功能。此外，WebTestClient 可以借助模拟的请求和响应对象直接绑定到基于 WebFlux 的应用程序，而无须 HTTP 服务器。它在测试 WebFlux 应用程序中扮演的角色与 TestRestTemplate 在 Web MVC 应用程序中扮演的角色一样，如以下代码所示：

```
1  @RunWith(SpringRunner.class)
2  @SpringBootTest
3  @AutoConfigureWebTestClient
4  public class PaymentControllerTests {
5
6      @Autowired
7      webTestClient client;
8
9      // ...
10 }
```

此时，不需要配置 WebTestClient。由 Spring Boot自动配置。

## 第四部分 Spring WebFlux源码分析

### 16 Spring WebFlux源码解析——socket包

`org.springframework.web.reactive.socket`

响应式WebSocket交互的抽象和支持类。

#### WebSocketHandler

一个WebSocket会话处理程序。

```
1 public interface WebSocketHandler {  
2  
3     /**  
4      * 返回此处理程序支持的子协议列表。  
5      * 默认情况下返回一个空列表。  
6      */  
7     default List<String> getSubProtocols() {  
8         return Collections.emptyList();  
9     }  
10  
11    /**  
12     * 处理WebSocket会话。  
13     * @param session the session to handle  
14     * @return completion {@code Mono<Void>} to indicate the outcome of the  
15     * websocket session handling.  
16     */  
17     Mono<Void> handle(WebSocketSession session);  
18  
19 }
```

#### WebSocketSession

表示具有响应式流输入和输出的WebSocket会话。

在服务器端，可以通过将请求映射到WebSocketHandler来处理WebSocket会话，并确保在Spring配置中注册了WebSocketHandlerAdapter策略。

在客户端，可以将WebSocketHandler提供给WebSocketClient。

```
1 public interface WebSocketSession {  
2  
3     /**  
4      * Return the id for the session.  
5      * 返回会话的ID。  
6      */  
7     String getId();  
8  
9     /**  
10      *从握手请求中返回信息。  
11      * Return information from the handshake request.  
12      */  
13     HandshakeInfo getHandshakeInfo();  
14  
15     /**  
16      * 返回一个DataBuffer Factory来创建消息有效载荷。  
17      * Return a {@code DataBuffer} Factory to create message payloads.  
18      * @return the buffer factory for the session  
19      */  
20     DataBufferFactory bufferFactory();  
21  
22     /**  
23      * 获取传入消息的流。  
24      * Get the flux of incoming messages.  
25      */  
26     Flux<WebSocketMessage> receive();  
27  
28     /**  
29      * 将给定的消息写入WebSocket连接。  
30      * write the given messages to the websocket connection.  
31      * @param messages the messages to write  
32      */  
33     Mono<Void> send(Publisher<WebSocketMessage> messages);  
34  
35     /**  
36      * 用CloseStatus.NORMAL关闭WebSocket会话。  
37      * Close the websocket session with {@link CloseStatus#NORMAL}.  
38      */  
39     default Mono<Void> close() {  
40         return close(CloseStatus.NORMAL);  
41     }  
42  
43     /**  
44      * 关闭具有给定状态的WebSocket会话。  
45      * close the websocket session with the given status.  
46      * @param status the close status  
47      */  
48     Mono<Void> close(CloseStatus status);  
49  
50  
51     // WebSocketMessage factory methods  
52  
53     /**  
54      * Factory方法使用会话的bufferFactory() 创建文本WebSocketMessage。  
55      */  
56     WebSocketMessage textMessage(String payload);  
57  
58     /**
```

```

59     * Factory方法使用会话的bufferFactory () 创建二进制WebSocketMessage sage。
60     */
61     WebSocketMessage binaryMessage(Function<DataBufferFactory, DataBuffer>
payloadFactory);
62
63     /**
64      * 工厂方法使用会话的bufferFactory () 创建一个ping *WebSocketMessage。
65      */
66     WebSocketMessage pingMessage(Function<DataBufferFactory, DataBuffer>
payloadFactory);
67
68     /**
69      * Factory方法使用会话的bufferFactory () 创建pong *WebSocketMessage。
70      */
71     WebSocketMessage pongMessage(Function<DataBufferFactory, DataBuffer>
payloadFactory);
72
73 }

```

## WebSocketMessage

WebSocket消息的表示。

```

1 public class WebSocketMessage {
2
3     private final Type type;
4
5     private final DataBuffer payload;
6
7
8     /**
9      * WebSocketMessage的构造函数。
10     * 请参阅WebSocketSession中的静态工厂方法,
11     * 或使用 WebSocketSession.bufferFactory() 创建有效内容, 然后调用此构造函数。
12     */
13     public WebSocketMessage(Type type, DataBuffer payload) {
14         Assert.notNull(type, "'type' must not be null");
15         Assert.notNull(payload, "'payload' must not be null");
16         this.type = type;
17         this.payload = payload;
18     }
19
20
21     /**
22      * Return the message type (text, binary, etc).
23      */
24     public Type getType() {
25         return this.type;
26     }
27
28     /**
29      * Return the message payload.
30      */
31     public DataBuffer getPayload() {

```

```
32         return this.payload;
33     }
34
35     /**
36      * Return the message payload as UTF-8 text. This is a useful for text
37      * websocket messages.
38     */
39     public String getPayloadAsText() {
40         byte[] bytes = new byte[this.payload.readableByteCount()];
41         this.payload.read(bytes);
42         return new String(bytes, StandardCharsets.UTF_8);
43     }
44
45     /**
46      * 保留消息有效载荷的数据缓冲区，这在运行时（例如Netty）和池缓冲区中很有用。一个快捷
47      * 方式：
48      */
49     public WebSocketMessage retain() {
50         DataBufferUtils.retain(this.payload);
51         return this;
52     }
53
54     /**
55      * 释放放在运行时（如Netty）使用池缓冲区（如Netty）有用的有效载荷DataBuffer。一个快
56      * 捷方式：
57      */
58     public void release() {
59         DataBufferUtils.release(this.payload);
60     }
61
62     @Override
63     public boolean equals(Object other) {
64         if (this == other) {
65             return true;
66         }
67         if (!(other instanceof WebSocketMessage)) {
68             return false;
69         }
70         WebSocketMessage otherMessage = (WebSocketMessage) other;
71         return (this.type.equals(otherMessage.type) &&
72                 Objectutils.nullSafeEquals(this.payload,
73                     otherMessage.payload));
74     }
75
76     @Override
77     public int hashCode() {
78         return this.type.hashCode() * 29 + this.payload.hashCode();
79     }
80
81     /**
82      * websocket 消息类型。
83      */
84     public enum Type { TEXT, BINARY, PING, PONG }
85 }
```

## CloseStatus

表示WebSocket“关闭”的状态码和原因。 1xxx范围内的状态码由协议预先定义。

```
1 public final class CloseStatus {  
2  
3     /**  
4      * "1000 indicates a normal closure, meaning that the purpose for which  
5      * the connection  
6      * was established has been fulfilled."  
7      */  
8      public static final CloseStatus NORMAL = new CloseStatus(1000);  
9  
10     /**  
11      * "1001 indicates that an endpoint is "going away", such as a server  
12      * going down or a  
13      * browser having navigated away from a page."  
14      */  
15      public static final CloseStatus GOING_AWAY = new CloseStatus(1001);  
16  
17     /**  
18      * "1002 indicates that an endpoint is terminating the connection due  
19      * to a protocol  
20      * error."  
21      */  
22      public static final CloseStatus PROTOCOL_ERROR = new  
23      CloseStatus(1002);  
24  
25     /**  
26      * "1003 indicates that an endpoint is terminating the connection  
27      * because it has  
28      * received a type of data it cannot accept (e.g., an endpoint that  
29      * understands only  
30      * text data MAY send this if it receives a binary message)."  
31      */  
32      public static final CloseStatus NOT_ACCEPTABLE = new CloseStatus(1003);  
33  
34     // 10004: Reserved.  
35     // The specific meaning might be defined in the future.  
36  
37     /**  
38      * "1005 is a reserved value and MUST NOT be set as a status code in a  
39      * Close control  
40      * frame by an endpoint. It is designated for use in applications  
41      * expecting a status  
42      * code to indicate that no status code was actually present."  
43      */  
44      public static final CloseStatus NO_STATUS_CODE = new CloseStatus(1005);  
45  
46     /**  
47      * "1006 is a reserved value and MUST NOT be set as a status code in a  
48      * Close control
```

```
40     * frame by an endpoint. It is designated for use in applications
41     * expecting a status
42     * code to indicate that the connection was closed abnormally, e.g.,
43     * without sending
44     * or receiving a Close control frame."
45     */
46     public static final CloseStatus NO_CLOSE_FRAME = new CloseStatus(1006);
47
48 /**
49     * "1007 indicates that an endpoint is terminating the connection
50     * because it has
51     * received data within a message that was not consistent with the type
52     * of the message
53     * (e.g., non-UTF-8 [RFC3629] data within a text message)."
54     */
55     public static final CloseStatus BAD_DATA = new CloseStatus(1007);
56
57 /**
58     * "1008 indicates that an endpoint is terminating the connection
59     * because it has
60     * received a message that violates its policy. This is a generic
61     * status code that can
62     * be returned when there is no other more suitable status code (e.g.,
63     * 1003 or 1009)
64     * or if there is a need to hide specific details about the policy."
65     */
66     public static final CloseStatus POLICY_VIOLATION = new
67     CloseStatus(1008);
68
69 /**
70     * "1009 indicates that an endpoint is terminating the connection
71     * because it has
72     * received a message that is too big for it to process."
73     */
74     public static final CloseStatus TOO_BIG_TO_PROCESS = new
75     CloseStatus(1009);
76
77 /**
78     * "1010 indicates that an endpoint (client) is terminating the
79     * connection because it
80     * has expected the server to negotiate one or more extension, but the
81     * server didn't
82     * return them in the response message of the WebSocket handshake. The
83     * list of
84     * extensions that are needed SHOULD appear in the /reason/ part of the
85     * Close frame.
86     * Note that this status code is not used by the server, because it can
87     * fail the
88     * WebSocket handshake instead."
89     */
90     public static final CloseStatus REQUIRED_EXTENSION = new
91     CloseStatus(1010);
92
93 /**
94     * "1011 indicates that a server is terminating the connection because
95     * it encountered
96     * an unexpected condition that prevented it from fulfilling the
97     * request."
98 
```

```
80     */
81     public static final CloseStatus SERVER_ERROR = new CloseStatus(1011);
82
83     /**
84      * "1012 indicates that the service is restarted. A client may
85      * reconnect, and if it
86      * chooses to do, should reconnect using a randomized delay of 5 -
87      * 30s."
88      */
89     public static final CloseStatus SERVICE_RESTARTED = new
90     CloseStatus(1012);
91
92     /**
93      * "1013 indicates that the service is experiencing overload. A client
94      * should only
95      * connect to a different IP (when there are multiple for the target)
96      * or reconnect to
97      * the same IP upon user action."
98      */
99     public static final CloseStatus SERVICE_OVERLOAD = new
100    CloseStatus(1013);
101
102    /**
103     * "1015 is a reserved value and MUST NOT be set as a status code in a
104     * Close control
105     * frame by an endpoint. It is designated for use in applications
106     * expecting a status
107     * code to indicate that the connection was closed due to a failure to
108     * perform a TLS
109     * handshake (e.g., the server certificate can't be verified)."
110     */
111    public static final CloseStatus TLS_HANDSHAKE_FAILURE = new
112    CloseStatus(1015);
113
114
115    private final int code;
116
117    @Nullable
118    private final String reason;
119
120    /**
121     * Create a new {@link CloseStatus} instance.
122     * @param code the status code
123     */
124    public CloseStatus(int code) {
125        this(code, null);
126    }
127
128    /**
129     * Create a new {@link CloseStatus} instance.
130     * @param code the status code
131     * @param reason the reason
132     */
133    public CloseStatus(int code, @Nullable String reason) {
134        Assert.isTrue((code >= 1000 && code < 5000), "Invalid status
135        code");
136        this.code = code;
```

```
127     this.reason = reason;
128 }
129
130
131 /**
132 * Return the status code.
133 */
134 public int getCode() {
135     return this.code;
136 }
137
138 /**
139 * Return the reason, or {@code null} if none.
140 */
141 @Nullable
142 public String getReason() {
143     return this.reason;
144 }
145
146 /**
147 * Create a new {@link CloseStatus} from this one with the specified
148 * reason.
149 * @param reason the reason
150 * @return a new {@link CloseStatus} instance
151 */
152 public CloseStatus withReason(String reason) {
153     Assert.hasText(reason, "Reason must not be empty");
154     return new CloseStatus(this.code, reason);
155 }
156
157 public boolean equalsCode(CloseStatus other) {
158     return (this.code == other.code);
159 }
160
161 @Override
162 public boolean equals(Object other) {
163     if (this == other) {
164         return true;
165     }
166     if (!(other instanceof CloseStatus)) {
167         return false;
168     }
169     CloseStatus otherStatus = (CloseStatus) other;
170     return (this.code == otherStatus.code &&
171             ObjectUtils.nullSafeEquals(this.reason,
172             otherStatus.reason));
173 }
174
175 @Override
176 public int hashCode() {
177     return this.code * 29 + ObjectUtils.nullSafeHashCode(this.reason);
178 }
179
180 @Override
181 public String toString() {
182     return "CloseStatus[code=" + this.code + ", reason=" + this.reason
183     + "]";
184 }
```

```
182 }  
183  
184 }
```

状态码	常量	表示
1000	NORMAL	1000表示一个正常的闭包，这意味着建立连接的目的已经完成
1001	GOING_AWAY	1001表示端点正在“消失”，比如服务器关闭或浏览器从页面上离开。
1002	PROTOCOL_ERROR	1002表示由于协议错误，端点正在终止连接
1003	NOT_ACCEPTABLE	1003表示端点正在终止连接，因为它接收了一种无法接受的数据类型(例如，一个只理解文本数据的端点，如果接收到二进制消息，则可以发送此数据)
1004	Reserved	保留，未来可能在定义
1005	NO_STATUS_CODE	1005是一个保留值，不能在端点的闭合控制帧中设置为状态码。
1006	NO_CLOSE_FRAME	1006是一个保留值，不能在端点的闭合控制帧中设置为状态码
1007	BAD_DATA	1007表示端点正在终止连接，因为它在一条消息中接收到与消息类型不一致的数据(例如，文本消息中的非utf - 8[RFC3629]数据)。
1008	POLICY_VIOLATION	1008表示端点正在终止连接，因为它收到了违反其策略的消息。
1009	TOO_BIG_TO_PROCESS	1009表示端点正在终止连接，因为它收到了一个太大的消息，无法处理。
10010	REQUIRED_EXTENSION	1010表示端点(客户端)终止了连接，因为它期望服务器可以协商一个或多个扩展，但是服务器并没有在WebSocket握手的响应消息中返回它们
10011	SERVER_ERROR	1011表明服务器正在终止连接，因为它遇到了一个意想不到的情况，阻止它完成请求
10012	SERVICE_RESTARTED	1012表示该服务重新启动
10013	SERVICE_OVERLOAD	1013显示服务正在经历过载。
10015	TLS_HANDSHAKE_FAILURE	1015是一个保留的值，不能在端点的闭环控制帧中设置为状态码

## HandshakeInfo

与启动WebSocketSession会话的握手请求相关的简单信息容器

```
1 public class HandshakeInfo {
2
3     private final URI uri;
4
5     private final Mono<Principal> principalMono;
6
7     private final HttpHeaders headers;
8
9     @Nullable
10    private final String protocol;
11
12
13    /**
14     * Constructor with information about the handshake.
15     * @param uri the endpoint URL
16     * @param headers request headers for server or response headers or
17     * client
18     * @param principal the principal for the session
19     * @param protocol the negotiated sub-protocol (may be {@code null})
20     */
21    public HandshakeInfo(URI uri, HttpHeaders headers, Mono<Principal>
principal, @Nullable String protocol) {
22        Assert.notNull(uri, "URI is required");
23        Assert.notNull(headers, "HttpHeaders are required");
24        Assert.notNull(principal, "Principal is required");
25        this.uri = uri;
26        this.headers = headers;
27        this.principalMono = principal;
28        this.protocol = protocol;
29    }
30
31
32    /**
33     * 返回WebSocket端点的URL
34     * Return the URL for the websocket endpoint.
35     */
36    public URI getUri() {
37        return this.uri;
38    }
39
40    /**
41     * Return the handshake HTTP headers. Those are the request headers for
42     * a
43     * server session and the response headers for a client session.
44     */
45    public HttpHeaders getHeaders() {
46        return this.headers;
47    }
48
49    /**
50     * 返回与握手HTTP请求相关的主体。
51     * Return the principal associated with the handshake HTTP request.
52     */
53    public Mono<Principal> getPrincipal() {
54        return this.principalMono;
55    }
56}
```

```
55  /**
56  * 在握手时协商的子协议，如果没有，则为null。
57  * The sub-protocol negotiated at handshake time, or {@code null} if
58  * none.
59  * @see <a href="https://tools.ietf.org/html/rfc6455#section-1.9">
60  * https://tools.ietf.org/html/rfc6455#section-1.9</a>
61  */
62  @Nullable
63  public String getSubProtocol() {
64      return this.protocol;
65  }
66
67  @Override
68  public String toString() {
69      return "HandshakeInfo[uri=" + this.uri + ", headers=" + this.headers
70      + "]";
71  }
72 }
```

## Package org.springframework.web.reactive.socket.adapter

将Spring的Reactive WebSocket API与WebSocket运行时相适配的类。

### AbstractWebSocketSession<T>

WebSocketSession实现的便捷基类，包含公共字段并暴露给外界来访问。还实现了WebSocketMessage工厂方法。

### AbstractListenerWebSocketSession

在事件侦听器WebSocket API之间架设的WebSocketSession实现的基类

```
1  public abstract class AbstractWebSocketSession<T> implements
2  WebSocketSession {
3
4      private final T delegate;
5
6      private final String id;
7
8      private final HandshakeInfo handshakeInfo;
9
10     private final DataBufferFactory bufferFactory;
11
12     /**
13      * 在握手时协商的子协议，如果没有，则为null。
14      * The sub-protocol negotiated at handshake time, or {@code null} if
15      * none.
16      * @see <a href="https://tools.ietf.org/html/rfc6455#section-1.9">
17      * https://tools.ietf.org/html/rfc6455#section-1.9</a>
18      */
19
20     @Nullable
21     public String getSubProtocol() {
22         return this.protocol;
23     }
24
25     @Override
26     public String toString() {
27         return "HandshakeInfo[uri=" + this.uri + ", headers=" + this.headers
28         + "]";
29     }
30 }
```

```
13     * Create a new instance and associate the given attributes with it.
14     */
15     protected AbstractWebSocketSession(T delegate, String id, HandshakeInfo
handshakeInfo,
16                                         DataBufferFactory bufferFactory) {
17
18         Assert.notNull(delegate, "Native session is required.");
19         Assert.notNull(id, "Session id is required.");
20         Assert.notNull(handshakeInfo, "HandshakeInfo is required.");
21         Assert.notNull(bufferFactory, "DataBuffer factory is required.");
22
23         this.delegate = delegate;
24         this.id = id;
25         this.handshakeInfo = handshakeInfo;
26         this.bufferFactory = bufferFactory;
27     }
28
29
30     protected T getDelegate() {
31         return this.delegate;
32     }
33
34     @Override
35     public String getId() {
36         return this.id;
37     }
38
39     @Override
40     public HandshakeInfo getHandshakeInfo() {
41         return this.handshakeInfo;
42     }
43
44     // 返回一个DataBuffer Factory来创建消息有效载荷。
45     @Override
46     public DataBufferFactory bufferFactory() {
47         return this.bufferFactory;
48     }
49
50     // 获取传入消息的流。
51     @Override
52     public abstract Flux<WebSocketMessage> receive();
53
54     // 将给定的消息写入WebSocket连接。
55     @Override
56     public abstract Mono<Void> send(Publisher<WebSocketMessage> messages);
57
58
59     // WebSocketMessage factory methods
60
61     @Override
62     public WebSocketMessage textMessage(String payload) {
63         byte[] bytes = payload.getBytes(StandardCharsets.UTF_8);
64         DataBuffer buffer = bufferFactory().wrap(bytes);
65         return new WebSocketMessage(WebSocketMessage.Type.TEXT, buffer);
66     }
67
68 }
```

```

69     // Factory方法使用websocketSession.bufferFactory() 为会话创建二进制
70     // websocketMessage。
71     @Override
72     public WebSocketMessage binaryMessage(Function<DataBufferFactory,
73                                         DataBuffer> payloadFactory) {
74         DataBuffer payload = payloadFactory.apply(bufferFactory());
75         return new WebSocketMessage(WebSocketMessage.Type.BINARY, payload);
76     }
77
78     //工厂方法使用websocketSession.bufferFactory() 为会话创建一个ping
79     // websocketMessage。
80
81     @Override
82     public WebSocketMessage pingMessage(Function<DataBufferFactory,
83                                         DataBuffer> payloadFactory) {
84         DataBuffer payload = payloadFactory.apply(bufferFactory());
85         return new WebSocketMessage(WebSocketMessage.Type.PING, payload);
86     }
87
88     //工厂方法创建一个使用websocketSession.bufferFactory pong websocketMessage会
89     //话()。
90
91     @Override
92     public WebSocketMessage pongMessage(Function<DataBufferFactory,
93                                         DataBuffer> payloadFactory) {
94         DataBuffer payload = payloadFactory.apply(bufferFactory());
95         return new WebSocketMessage(WebSocketMessage.Type.PONG, payload);
96     }

```

## AbstractListenerWebSocketSession

在事件侦听器WebSocket API（例如Java WebSocket API JSR-356, Jetty, Undertow）和Reactive Streams之间进行桥接的WebSocketSession实现的基类。

也是订阅者的实现，因此，它可以用作会话处理的完成订阅。

```

1  public abstract class AbstractListenerWebSocketSession<T> extends
2      AbstractWebSocketSession<T>
3      implements Subscriber<Void> {
4
5      /**
6      * The "back-pressure" buffer size to use if the underlying websocket
7      * API
8      * does not have flow control for receiving messages.
9      */

```

```

8     private static final int RECEIVE_BUFFER_SIZE = 8192;
9
10
11    @Nullable
12    private final MonoProcessor<Void> completionMono;
13
14    private final WebSocketReceivePublisher receivePublisher = new
15    WebSocketReceivePublisher();
16
17    @Nullable
18    private volatile WebSocketSendProcessor sendProcessor;
19
20    private final AtomicBoolean sendCalled = new AtomicBoolean();
21
22 /**
23 * Base constructor.
24 * @param delegate the native WebSocket session, channel, or connection
25 * @param id the session id
26 * @param handshakeInfo the handshake info
27 * @param bufferFactory the DataBuffer factor for the current
28 connection
29 */
30 public AbstractListenerWebSocketSession(T delegate, String id,
31 HandshakeInfo handshakeInfo,
32                                     DataBufferFactory
33 bufferFactory) {
34
35 /**
36 * Alternative constructor with completion {@code Mono<Void>} to
37 propagate
38 * the session completion (success or error) (for client-side use).
39 */
40 public AbstractListenerWebSocketSession(T delegate, String id,
41 HandshakeInfo handshakeInfo,
42                                     DataBufferFactory
43 bufferFactory, @Nullable MonoProcessor<Void> completionMono) {
44
45
46
47    protected WebSocketSendProcessor getSendProcessor() {
48        WebSocketSendProcessor sendProcessor = this.sendProcessor;
49        Assert.state(sendProcessor != null, "No WebSocketSendProcessor
50 available");
51        return sendProcessor;
52    }
53
54    @Override
55    public Flux<WebSocketMessage> receive() {
56        return canSuspendReceiving() ?

```

```
57     Flux.from(this.receivePublisher).onBackpressureBuffer(RECEIVE_BUFFER_SIZE)
58     ;
59
60     @Override
61     public Mono<Void> send(Publisher<WebSocketMessage> messages) {
62         if (this.sendCalled.compareAndSet(false, true)) {
63             WebSocketSendProcessor sendProcessor = new
64             WebSocketSendProcessor();
65             this.sendProcessor = sendProcessor;
66             return Mono.from(subscriber -> {
67                 messages.subscribe(sendProcessor);
68                 sendProcessor.subscribe(subscriber);
69             });
70         }
71         else {
72             return Mono.error(new IllegalStateException("send() has already
73             been called"));
74         }
75     }
76
77     /**
78      * 底层的WebSocket API是否具有流量控制功能，可以暂停和恢复接收消息。
79      */
80     protected abstract boolean canSuspendReceiving();
81
82     /**
83      * Suspend receiving until received message(s) are processed and more
84      * demand
85      * is generated by the downstream Subscriber.
86      * <p><strong>Note:</strong> if the underlying websocket API does not
87      * provide
88      * flow control for receiving messages, and this method should be a no-
89      * op
90      * and {@link #canSuspendReceiving()} should return {@code false}.
91      */
92     protected abstract void suspendReceiving();
93
94     /**
95      * Resume receiving new message(s) after demand is generated by the
96      * downstream Subscriber.
97      * <p><strong>Note:</strong> if the underlying websocket API does not
98      * provide
99      * flow control for receiving messages, and this method should be a no-
100     * op
101     * and {@link #canSuspendReceiving()} should return {@code false}.
102     */
103     protected abstract void resumeReceiving();
104
105     /**
106      * Send the given WebSocket message.
107      */
108     protected abstract boolean sendMessage(WebSocketMessage message) throws
109     IOException;
110
111     // WebSocketHandler adapter delegate methods
```

```
105  
106     /** Handle a message callback from the webSocketHandler adapter */  
107     void handleMessage(Type type, WebSocketMessage message) {  
108         this.receivePublisher.handleMessage(message);  
109     }  
110  
111     /** Handle an error callback from the webSocketHandler adapter */  
112     void handleError(Throwable ex) {  
113         this.receivePublisher.onError(ex);  
114         WebSocketSendProcessor sendProcessor = this.sendProcessor;  
115         if (sendProcessor != null) {  
116             sendProcessor.cancel();  
117             sendProcessor.onError(ex);  
118         }  
119     }  
120  
121     /** Handle a close callback from the webSocketHandler adapter */  
122     void handleClose(CloseStatus reason) {  
123         this.receivePublisher.onAllDataRead();  
124         WebSocketSendProcessor sendProcessor = this.sendProcessor;  
125         if (sendProcessor != null) {  
126             sendProcessor.cancel();  
127             sendProcessor onComplete();  
128         }  
129     }  
130  
131  
132     // Subscriber<Void> implementation  
133  
134     @Override  
135     public void onSubscribe(Subscription subscription) {  
136         subscription.request(Long.MAX_VALUE);  
137     }  
138  
139     @Override  
140     public void onNext(Void aVoid) {  
141         // no op  
142     }  
143  
144     @Override  
145     public void onError(Throwable ex) {  
146         if (this.completionMono != null) {  
147             this.completionMono.onError(ex);  
148         }  
149         int code = CloseStatus.SERVER_ERROR.getCode();  
150         close(new Closestatus(code, ex.getMessage()));  
151     }  
152  
153     @Override  
154     public void onComplete() {  
155         if (this.completionMono != null) {  
156             this.completionMono.onComplete();  
157         }  
158         close();  
159     }  
160  
161
```

```
162     private final class WebSocketReceivePublisher extends
163         AbstractListenerReadPublisher<WebSocketMessage> {
164
165         @Nullable
166         private volatile WebSocketMessage webSocketMessage;
167
168         @Override
169         protected void checkOnDataAvailable() {
170             if (this.webSocketMessage != null) {
171                 onDataAvailable();
172             }
173         }
174
175         @Override
176         @Nullable
177         protected WebSocketMessage read() throws IOException {
178             if (this.webSocketMessage != null) {
179                 WebSocketMessage result = this.webSocketMessage;
180                 this.webSocketMessage = null;
181                 resumeReceiving();
182                 return result;
183             }
184             return null;
185         }
186
187         void handleMessage(WebSocketMessage webSocketMessage) {
188             this.webSocketMessage = webSocketMessage;
189             suspendReceiving();
190             onDataAvailable();
191         }
192     }
193
194
195     protected final class WebSocketSendProcessor extends
196         AbstractListenerWriteProcessor<WebSocketMessage> {
197
198         private volatile boolean isReady = true;
199
200         @Override
201         protected boolean write(WebSocketMessage message) throws
202             IOException {
203             return sendMessage(message);
204         }
205
206         @Override
207         protected void releaseData() {
208             this.currentData = null;
209         }
210
211         @Override
212         protected boolean isEmpty(WebSocketMessage message) {
213             return (message.getPayload().readableByteCount() == 0);
214         }
215
216         @Override
217         protected boolean isWritePossible() {
218             return (this.isReady && this.currentData != null);
```

```
217     }
218
219     /**
220      * Sub-classes can invoke this before sending a message (false) and
221      * after receiving the async send callback (true) effective
222      * translating
223      *      *      *      *      *      *      *      *      *
224      *      *      *      *      *      *      *      *      *
225      *      *      *      *      *      *      *      *      *
226      *      *      *      *      *      *      *      *      *
227      *      *      *      *      *      *      *      *      *
228      *      *      *      *      *      *      *      *      *
229 }
```

这两个类是由子类来实现的。不同的服务器有不同的实现。

```
1 @Override
2 public abstract Flux<WebSocketMessage> receive();
3
4 @Override
5 public abstract Mono<Void> send(Publisher<WebSocketMessage> messages);
```

又分为两个分支，一个是NettyWebSocketSessionSupport下的ReactorNettyWebSocketSession

## NettyWebSocketSessionSupport

基于Netty的WebSocketSession适配器的基类，它提供了将Netty WebSocketFrames转换为WebSocketMessages和从WebSocketMessages转换的便利方法。

```
1 public abstract class NettyWebSocketSessionSupport<T> extends
AbstractWebSocketSession<T> {
2
3     /**
4      * 默认的最大大小用于聚集入站WebSocket帧。
5      * The default max size for aggregating inbound websocket frames.
6      */
7     protected static final int DEFAULT_FRAME_MAX_SIZE = 64 * 1024;
8
9
10    private static final Map<Class<?>, WebSocketMessage.Type> MESSAGE_TYPES;
11
12    static {
13        MESSAGE_TYPES = new HashMap<>(4);
14        MESSAGE_TYPES.put(TextWebSocketFrame.class,
WebSocketMessage.Type.TEXT);
```

```

15     MESSAGE_TYPES.put(BinaryWebSocketFrame.class,
16         WebSocketMessage.Type.BINARY);
17     MESSAGE_TYPES.put(PingWebSocketFrame.class,
18         WebSocketMessage.Type.PING);
19     MESSAGE_TYPES.put(PongWebSocketFrame.class,
20         WebSocketMessage.Type.PONG);
21 }
22
23
24
25
26 //返回一个DataBuffer Factory来创建消息有效载荷。
27 @Override
28 public NettyDataBufferFactory bufferFactory() {
29     return (NettyDataBufferFactory) super.bufferFactory();
30 }
31
32
33 protected WebSocketMessage toMessage(WebSocketFrame frame) {
34     DataBuffer payload = bufferFactory().wrap(frame.content());
35     return new WebSocketMessage(MESSAGE_TYPES.get(frame.getClass()),
36         payload);
37 }
38
39 protected WebSocketFrame toFrame(WebSocketMessage message) {
40     ByteBuf byteBuf =
41         NettyDataBufferFactory.toByteBuf(message.getPayload());
42     if (WebSocketMessage.Type.TEXT.equals(message.getType())) {
43         return new TextWebSocketFrame(byteBuf);
44     }
45     else if (WebSocketMessage.Type.BINARY.equals(message.getType())) {
46         return new BinaryWebSocketFrame(byteBuf);
47     }
48     else if (WebSocketMessage.Type.PING.equals(message.getType())) {
49         return new PingWebSocketFrame(byteBuf);
50     }
51     else if (WebSocketMessage.Type.PONG.equals(message.getType())) {
52         return new PongWebSocketFrame(byteBuf);
53     }
54     else {
55         throw new IllegalArgumentException("Unexpected message type: " +
56             message.getType());
57     }
58 }

```

## ReactorNettyWebSocketSession

Spring WebSocketSession实现，适配响应式Netty的WebSocket NettyInbound和NettyOutbound。

```
1 public class ReactorNettyWebSocketSession
2     extends
3         NettyWebSocketSessionSupport<ReactorNettyWebSocketSession.WebSocketConnectio
4 n> {
5
6
7     public ReactorNettyWebSocketSession(WebSocketInbound inbound,
8         WebSocketOutbound outbound,
9             HandshakeInfo info,
10            NettyDataBufferFactory bufferFactory) {
11
12         super(new WebSocketConnection(inbound, outbound), info,
13             bufferFactory);
14     }
15
16
17     //获取传入消息的流。
18     @Override
19     public Flux<WebSocketMessage> receive() {
20         return getDelegate().getInbound()
21             .aggregateFrames(DEFAULT_FRAME_MAX_SIZE)
22             .receiveFrames()
23             .map(super::toMessage);
24     }
25
26
27     //将给定的消息写入WebSocket连接。
28     @Override
29     public Mono<Void> send(Publisher<WebSocketMessage> messages) {
30         Flux<WebSocketFrame> frames =
31             Flux.from(messages).map(this::toFrame);
32         return getDelegate().getOutbound()
33             .options(NettyPipeline.SendOptions::flushOnEach)
34             .sendObject(frames)
35             .then();
36     }
37
38
39     //关闭具有给定状态的WebSocket会话。
40     @Override
41     public Mono<Void> close(CloseStatus status) {
42         return Mono.error(new UnsupportedOperationException(
43             "Currently in Reactor Netty applications are expected to use the
44             " +
45             "Cancellation returned from subscribing to the \"receive\"-side
46             Flux " +
47             "in order to close the websocket session."));
48     }
49
50
51     /**
52      * Simple container for {@link NettyInbound} and {@link NettyOutbound}.
53      */
54
55     public static class WebSocketConnection {
56
57         private final WebSocketInbound inbound;
```

```

48     private final WebsocketOutbound outbound;
49
50
51     public WebSocketConnection(WebSocketInbound inbound,
52         WebsocketOutbound outbound) {
53         this.inbound = inbound;
54         this.outbound = outbound;
55     }
56
57     public WebSocketInbound getInbound() {
58         return this.inbound;
59     }
60
61     public WebsocketOutbound getOutbound() {
62         return this.outbound;
63     }
64 }
65 }
```

基于AbstractListenerWebSocketSession的几种不同的实现：

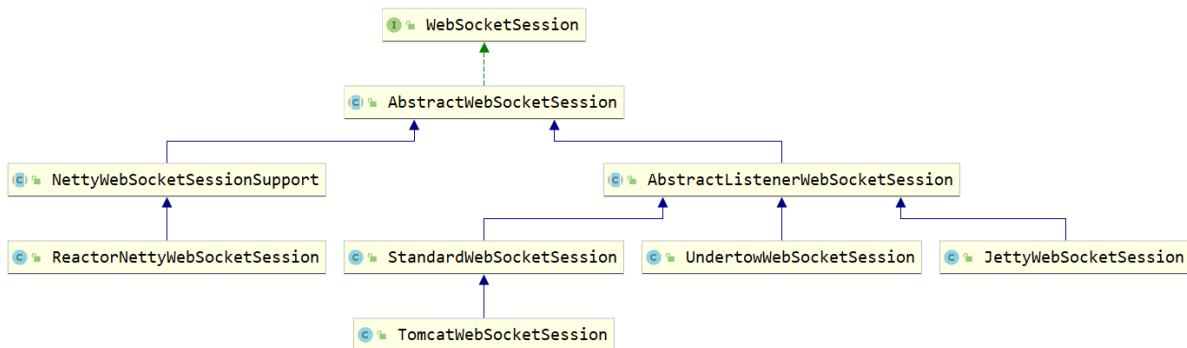
### StandardWebSocketSession

为标准Java (JSR 356) 会话跳转WebSocketSession适配器。

### StandardWebSocketHandlerAdapter

Java WebSocket API (JSR-356) 的适配器，它将事件委托给一个被动的WebSocketHandler及其会话。

另外几个也是差不多，都有不同的实现。



`package org.springframework.web.reactive.socket.client;`

### WebSocketClient

响应式风格处理WebSocket会话的协议。

```
1 public interface websocketClient {
2
3     // 具有自定义标头的执行
4     Mono<Void> execute(URI url, WebSocketHandler handler);
5
6     //对给定的URL执行握手请求，并使用给定的handler处理生成的websocket会话。
7     Mono<Void> execute(URI url, HttpHeaders headers, WebSocketHandler
8     handler);
9 }
```

## WebSocketClientSupport

WebSocketClient实现的基类。

```
1 public class websocketClientSupport {
2
3     private static final String SEC_WEBSOCKET_PROTOCOL = "Sec-WebSocket-
4     Protocol";
5
6     protected final Log logger = LogFactory.getLog(getClass());
7
8     protected List<String> beforeHandshake(URI url, HttpHeaders
9     requestHeaders, WebSocketHandler handler) {
10         if (logger.isDebugEnabled()) {
11             logger.debug("Executing handshake to " + url);
12         }
13         return handler.getSubProtocols();
14     }
15
16     protected HandshakeInfo afterHandshake(URI url, HttpHeaders
17     responseHeaders) {
18         if (logger.isDebugEnabled()) {
19             logger.debug("Handshake response: " + url + ", " +
20             responseHeaders);
21         }
22         String protocol = responseHeaders.getFirst(SEC_WEBSOCKET_PROTOCOL);
23         return new HandshakeInfo(url, responseHeaders, Mono.empty(),
24         protocol);
25     }
26 }
```

## ReactorNettyWebSocketClient

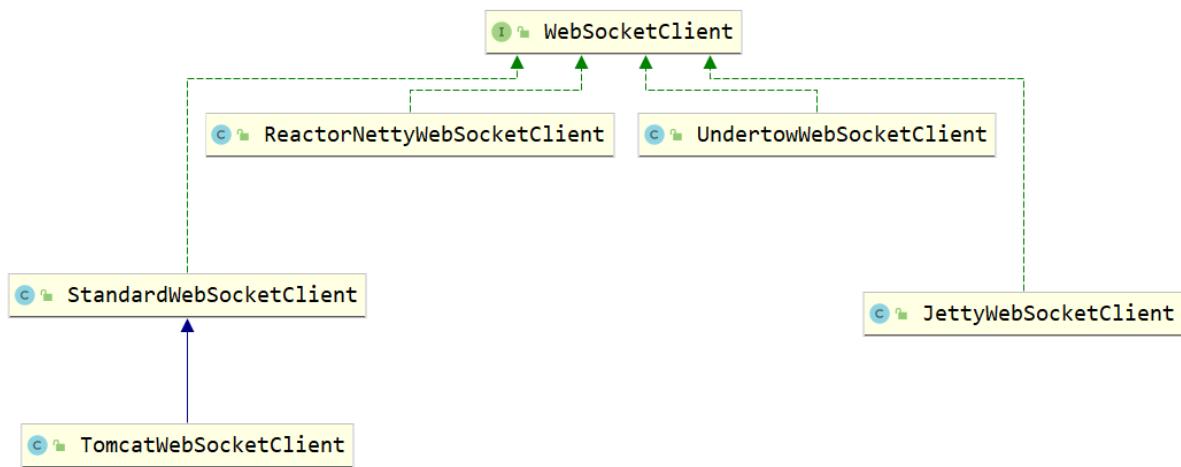
用于Reactor Netty的WebSocketClient实现。

```
1 public class ReactorNettyWebSocketClient extends WebSocketClientSupport
2     implements WebSocketClient {
3
4
5     /**
6      * 默认构造器
7      */
8     public ReactorNettyWebSocketClient() {
9         this(options -> {});
10    }
11
12    /**
13     * Constructor that accepts an {@link HttpClientOptions.Builder}
14     * consumer
15     * to supply to {@link HttpClient#create(Consumer)}.
16     */
17    public ReactorNettyWebSocketClient(Consumer<? super
18     HttpClientOptions.Builder> clientOptions) {
19        this.httpClient = HttpClient.create(clientOptions);
20    }
21
22    /**
23     * 返回配置好的HttpClient
24     */
25    public HttpClient getHttpClient() {
26        return this.httpClient;
27    }
28
29    //对给定的URL执行握手请求，并使用给定的处理器处理生成的WebSocket会话。
30    @Override
31    public Mono<Void> execute(URI url, WebSocketHandler handler) {
32        return execute(url, new HttpHeaders(), handler);
33    }
34
35    @Override
36    public Mono<Void> execute(URI url, HttpHeaders headers, WebSocketHandler
37    handler) {
38        List<String> protocols = beforeHandshake(url, headers, handler);
39
40        return getHttpClient()
41            .ws(url.toString(),
42                nettyHeaders -> setNettyHeaders(headers, nettyHeaders),
43                StringUtils.collectionToCommaDelimitedString(protocols))
44            .flatMap(response -> {
45                HandshakeInfo info = afterHandshake(url,
46                    toHttpHeaders(response));
47                ByteBufAllocator allocator = response.channel().alloc();
48                NettyDataBufferFactory factory = new
49                NettyDataBufferFactory(allocator);
50                return response.receivewebsocket((in, out) -> {
51                    WebSocketSession session = new
52                    ReactorNettyWebSocketSession(in, out, info, factory);
53                    return handler.handle(session);
54                });
55            });
56    }
```

```

52     }
53
54     private void setNettyHeaders(HttpHeaders headers,
55         io.netty.handler.codec.http.HttpHeaders nettyHeaders) {
56         headers.forEach(nettyHeaders::set);
57     }
58
59     private HttpHeaders toHttpHeaders(HttpClientResponse response) {
60         HttpHeaders headers = new HttpHeaders();
61         response.responseHeaders().forEach(entry -> {
62             String name = entry.getKey();
63             headers.put(name, response.responseHeaders().getAll(name));
64         });
65         return headers;
66     }
67 }

```



`org.springframework.web.reactive.socket.server.support;`

WebSocket请求的服务器端支持类。

### RequestUpgradeStrategy

根据底层网络运行时将HTTP请求升级到WebSocket会话的策略。

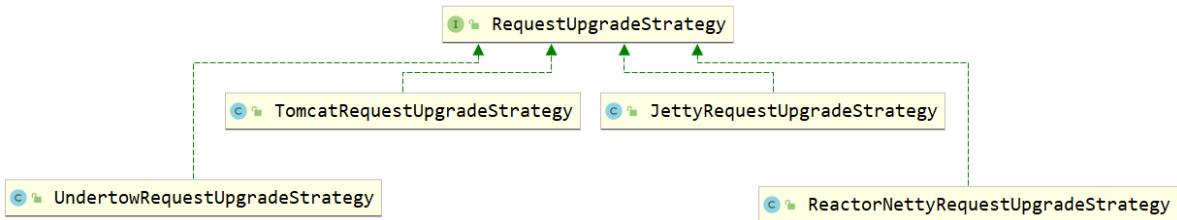
```
1 public interface RequestUpgradeStrategy {  
2     /**  
3      * 升级到websocket会话并使用给定的处理器处理它。  
4      * @param exchange the current exchange  
5      * @param webSocketHandler handler for the WebSocket session  
6      * @param subProtocol the selected sub-protocol got the handler  
7      * @return completion {@code Mono<Void>} to indicate the outcome of the  
8      * WebSocket session handling.  
9      */  
10     Mono<Void> upgrade(ServerWebExchange exchange, WebSocketHandler  
11         webSocketHandler, @Nullable String subProtocol);  
12 }
```

### ReactorNettyRequestUpgradeStrategy

持有RequestUpgradeStrategy的实现。

```
1 public class ReactorNettyRequestUpgradeStrategy implements  
2     RequestUpgradeStrategy {  
3     @Override  
4     public Mono<Void> upgrade(ServerWebExchange exchange, WebSocketHandler  
5         handler, @Nullable String subProtocol) {  
6         ReactorServerHttpResponse response = (ReactorServerHttpResponse)  
7             exchange.getResponse();  
8         HandshakeInfo info = getHandshakeInfo(exchange, subProtocol);  
9         NettyDataBufferFactory bufferFactory = (NettyDataBufferFactory)  
10            response.bufferFactory();  
11  
12         return response.getReactorResponse().sendwebsocket(subProtocol,  
13             (in, out) ->  
14             handler.handle(new ReactorNettyWebSocketSession(in, out, info,  
15                 bufferFactory)));  
16     }  
17  
18     private HandshakeInfo getHandshakeInfo(ServerWebExchange exchange,  
19         @Nullable String protocol) {  
20         ServerHttpRequest request = exchange.getRequest();  
21         Mono<Principal> principal = exchange.getPrincipal();  
22         return new HandshakeInfo(request.getURI(), request.getHeaders(),  
23             principal, protocol);  
24     }  
25 }
```

upgrade方法把接收到的http请求协议转换为websocket协议。



## WebSocketService

一种委托websocket相关的HTTP请求的服务。

对于WebSocket端点，这意味着要处理初始的WebSocket HTTP握手请求。对于SockJS端点，这可能意味着处理所有在SockJS协议中定义的HTTP请求。

```

1 public interface WebSocketService {
2     /**
3      * 处理HTTP请求并使用给定的websocketHandler。
4      * @param exchange the current exchange
5      * @param websocketHandler handler for websocket session
6      * @return a completion Mono for the websocket session handling
7      */
8     Mono<Void> handleRequest(ServerWebExchange exchange, WebSocketHandler
9     websocketHandler);
  
```

## HandshakeWebSocketService

WebSocketService的实现。

```

1 public class HandshakeWebSocketService implements WebSocketService,
2 Lifecycle {
3     private static final String SEC_WEBSOCKET_KEY = "Sec-WebSocket-Key";
4
5     private static final String SEC_WEBSOCKET_PROTOCOL = "Sec-WebSocket-
6     Protocol";
7
8     private static final boolean tomcatPresent = Classutils.isPresent(
9         "org.apache.tomcat.websocket.server.WsHttpUpgradeHandler",
10        HandshakeWebSocketService.class.getClassLoader());
11
12     private static final boolean jettyPresent = Classutils.isPresent(
13         "org.eclipse.jetty.websocket.server.WebSocketServerFactory",
14        HandshakeWebSocketService.class.getClassLoader());
15
16     private static final boolean undertowPresent = Classutils.isPresent(
17         "io.undertow.websocket.WebSocketProtocolHandshakeHandler",
18        HandshakeWebSocketService.class.getClassLoader());
19
20     private static final boolean reactorNettyPresent =
21         Classutils.isPresent(
  
```

```
20         "reactor.ipc.netty.http.server.HttpServerResponse",
21         HandshakewebSocketService.class.getClassLoader());
22
23
24     protected static final Log logger =
25         LogFactory.getLog(HandshakewebSocketService.class);
26
27     private final RequestUpgradeStrategy upgradeStrategy;
28
29     private volatile boolean running = false;
30
31
32     /**
33      * 默认构造函数自动，基于类路径检测的RequestUpgradeStrategy的发现使用。
34      * Default constructor automatic, classpath detection based discovery of
35      * the
36      * {@link RequestUpgradeStrategy} to use.
37     */
38     public HandshakewebSocketService() {
39         this(initUpgradeStrategy());
40     }
41
42     /**
43      * 使用RequestUpgradeStrategy的替代构造函数。
44      * @param upgradeStrategy the strategy to use
45      */
46     public HandshakewebSocketService(RequestUpgradeStrategy
47         upgradeStrategy) {
48         Assert.notNull(upgradeStrategy, "RequestUpgradeStrategy is
49         required");
50         this.upgradeStrategy = upgradeStrategy;
51     }
52
53     private static RequestUpgradeStrategy initUpgradeStrategy() {
54         String className;
55         if (tomcatPresent) {
56             className = "TomcatRequestUpgradeStrategy";
57         }
58         else if (jettyPresent) {
59             className = "JettyRequestUpgradeStrategy";
60         }
61         else if (undertowPresent) {
62             className = "UndertowRequestUpgradeStrategy";
63         }
64         else if (reactorNettyPresent) {
65             // As late as possible (Reactor Netty commonly used for
66             // web client)
67             className = "ReactorNettyRequestUpgradeStrategy";
68         }
69         else {
70             throw new IllegalStateException("No suitable default
71             RequestUpgradeStrategy found");
72         }
73
74         try {
75             className =
76                 "org.springframework.web.reactive.socket.server.upgrade." + className;
```

```
71         class<?> clazz = ClassUtils.forName(className,
72 HandshakeWebSocketService.class.getClassLoader());
73         return (RequestUpgradeStrategy)
74             ReflectionUtils.accessibleConstructor(clazz).newInstance();
75     }
76     catch (Throwable ex) {
77         throw new IllegalStateException(
78             "Failed to instantiate RequestUpgradeStrategy: " +
79 className, ex);
80     }
81 }
82 /**
83 * Return the {@link RequestUpgradeStrategy} for websocket requests.
84 */
85 public RequestUpgradeStrategy getUpgradeStrategy() {
86     return this.upgradeStrategy;
87 }
88 @Override
89 public boolean isRunning() {
90     return this.running;
91 }
92 @Override
93 public void start() {
94     if (!isRunning()) {
95         this.running = true;
96         doStart();
97     }
98 }
99 @
100 protected void doStart() {
101     if (getUpgradeStrategy() instanceof Lifecycle) {
102         ((Lifecycle) getUpgradeStrategy()).start();
103     }
104 }
105 @Override
106 public void stop() {
107     if (isRunning()) {
108         this.running = false;
109         doStop();
110     }
111 }
112 }
113 @
114 protected void doStop() {
115     if (getUpgradeStrategy() instanceof Lifecycle) {
116         ((Lifecycle) getUpgradeStrategy()).stop();
117     }
118 }
119 }
120
121 //处理HTTP请求并使用给定的websocketHandler。
122 @Override
123 public Mono<Void> handleRequest(ServerWebExchange exchange,
124 WebSocketHandler handler) {
```

```

125     ServerHttpRequest request = exchange.getRequest();
126     HttpMethod method = request.getMethod();
127     HttpHeaders headers = request.getHeaders();
128
129     if (logger.isDebugEnabled()) {
130         logger.debug("Handling " + request.getURI() + " with headers: "
131         + headers);
132     }
133
134     if (HttpMethod.GET != method) {
135         return Mono.error(new MethodNotAllowedException(
136             request.getMethodValue(),
137             Collections.singleton(HttpMethod.GET)));
138     }
139
140     if (!"websocket".equalsIgnoreCase(headers.getUpgrade())) {
141         return handleBadRequest("Invalid 'Upgrade' header: " +
142             headers);
143     }
144
145     List<String> connectionValue = headers.getConnection();
146     if (!connectionValue.contains("Upgrade") &&
147         !connectionValue.contains("upgrade")) {
148         return handleBadRequest("Invalid 'Connection' header: " +
149             headers);
150     }
151
152     String key = headers.getFirst(SEC_WEBSOCKET_KEY);
153     if (key == null) {
154         return handleBadRequest("Missing \"Sec-WebSocket-Key\" "
155             header);
156     }
157
158     String protocol = selectProtocol(headers, handler);
159     return this.upgradeStrategy.upgrade(exchange, handler, protocol);
160 }
161
162
163     @Nullable
164     private String selectProtocol(HttpHeaders headers, WebSocketHandler
165         handler) {
166         String protocolHeader = headers.getFirst(SEC_WEBSOCKET_PROTOCOL);
167         if (protocolHeader != null) {
168             List<String> supportedProtocols = handler.getSubProtocols();
169             for (String protocol :
170                 StringUtils.commaDelimitedListToStringArray(protocolHeader)) {
171                 if (supportedProtocols.contains(protocol)) {
172                     return protocol;
173                 }
174             }
175         }
176         return null;

```

```
175     }
176 }
```

WebSocketService实现通过委托给RequestUpgradeStrategy处理WebSocket HTTP握手请求，该请求可以从类路径自动检测（无参数构造函数），但也可以显式配置。

## 17 Spring WebFlux源码解析——support包

```
org.springframework.web.reactive.socket.server.support
```

Spring WebFlux的支持类。

### AbstractAnnotationConfigDispatcherHandlerInitializer

getConfigClasses()需要具体实现类。

更多的模板和定制方法由AbstractDispatcherHandlerInitializer提供。

```
1 public abstract class AbstractAnnotationConfigDispatcherHandlerInitializer
2     extends AbstractDispatcherHandlerInitializer {
3
4     /**
5      * 创建要提供给DispatcherHandler的应用程序上下文。
6      * 返回的上下文被委托给Spring的
7      DispatcherHandler.DispatcherHandler (ApplicationContext)。因此，它通常包含控制
8
9      * 器，视图解析器和其他web相关的bean。
10     * 该实现创建一个AnnotationConfigApplicationContext，为其提供由
11     getConfigClasses () 返回的带注释的类。
12     */
13     @Override
14     protected ApplicationContext createApplicationContext() {
15         AnnotationConfigApplicationContext servletAppContext = new
16         AnnotationConfigApplicationContext();
17         Class<?>[] configClasses = getConfigClasses();
18         if (!ObjectUtils.isEmpty(configClasses)) {
19             servletAppContext.register(configClasses);
20         }
21         return servletAppContext;
22     }
23
24     /**
25      * 为应用程序上下文指定@Configuration或者@Component类。
26      * 返回dispatcher servlet应用程序上下文的配置类
27      */
28     protected abstract Class<?>[] getConfigClasses();
29 }
```

## AbstractDispatcherHandlerInitializer

WebApplicationInitializer实现的基类，在Servlet上下文中注册一个DispatcherHandler，并将其包装在一个ServletHttpHandlerAdapter中。

大多数应用程序应该考虑扩展Spring Java配置，  
AbstractAnnotationConfigDispatcherHandlerInitializer子类。

```
1 public abstract class AbstractDispatcherHandlerInitializer implements
2 WebApplicationInitializer {
3
4     /**
5      * 默认的servlet名字。可以通过覆盖{@link #getServletName}来自定义。
6      */
7     public static final String DEFAULT_SERVLET_NAME = "dispatcher-handler";
8
9     /**
10      * 默认的servlet映射。. 可以通过覆盖{@link #getServletMapping()}来自定义
11      */
12     public static final String DEFAULT_SERVLET_MAPPING = "/";
13
14     //配置给定的ServletContext (servlets, filters, listeners context-params
15     //and attributes ) 来初始化此web应用程序
16     @Override
17     public void onStartup(ServletContext servletContext) throws
18     ServletException {
19         registerDispatcherHandler(servletContext);
20     }
21
22     /**
23      * 针对给定的servlet上下文注册一个DispatcherHandler。
24      * 此方法将创建一个DispatcherHandler，并使用从createApplicationContext () 返回
25      * 的应用程序上下文对其进行初始化。创
26          * 建的处理器将被包装在一个ServletHttpHandlerAdapter servlet中，其名称
27          * 由getServletName () 返回，将其映射到从
28          * getServletMapping () 返回的模式。
29          * 进一步的自定义可以通过覆盖
30          * customizeRegistration (ServletRegistration.Dynamic) 或
31          * createDispatcherHandler (ApplicationContext) 来实现。
32      */
33     protected void registerDispatcherHandler(ServletContext servletContext)
34     {
35         String servletName = getServletName();
36         Assert.hasLength(servletName, "getServletName() must not return
empty or null");
37
38         ApplicationContext applicationContext = createApplicationContext();
39         Assert.notNull(applicationContext,
40                         "createApplicationContext() did not return an
application " +
41                         "context for servlet [" + servletName + "]");
42
43         refreshApplicationContext(applicationContext);
```

```
37     registerCloseListener(servletContext, applicationContext);
38
39     WebHandler dispatcherHandler =
40     createDispatcherHandler(applicationContext);
41     Assert.assertNotNull(dispatcherHandler,
42         "createDispatcherHandler() did not return a
43     webHandler for servlet [" + servletName + "]");
44
45     ServletHttpHandlerAdapter handlerAdapter =
46     createHandlerAdapter(dispatcherHandler);
47     Assert.assertNotNull(handlerAdapter,
48         "createHttpHandler() did not return a
49     ServletHttpHandlerAdapter for servlet [" + servletName + "]");
50
51     ServletRegistration.Dynamic registration =
52     servletContext.addServlet(servletName, handlerAdapter);
53     Assert.assertNotNull(registration,
54         "Failed to register servlet with name '" +
55     servletName + "'." +
56         "Check if there is another servlet registered under
57     the same name.");
58
59     registration.setLoadOnStartup(1);
60     registration.addMapping(getServletMapping());
61     registration.setAsyncSupported(true);
62
63     customizeRegistration(registration);
64 }
65
66 /**
67 * Return the name under which the {@link ServletHttpHandlerAdapter}
68 * will be registered.
69 * Defaults to {@link #DEFAULT_SERVLET_NAME}.
70 * @see #registerDispatcherHandler(ServletContext)
71 */
72 protected String getServletName() {
73     return DEFAULT_SERVLET_NAME;
74 }
75
76 /**
77 * 创建要提供给DispatcherHandler的应用程序上下文。
78 * 返回的上下文被委托给Spring的
79 DispatcherHandler.DispatcherHandler (ApplicationContext)。因此，它通常包含控制
80     * 器，视图解析器和其他web相关的bean。
81 */
82 protected abstract ApplicationContext createApplicationContext();
83
84 /**
85 * 如有必要，刷新给定的应用程序上下文。
86 */
87 protected void refreshApplicationContext(ApplicationContext context) {
88     if (context instanceof ConfigurableApplicationContext) {
89         ConfigurableApplicationContext cac =
90 (ConfigurableApplicationContext) context;
91         if (!cac.isActive()) {
92             cac.refresh();
93         }
94     }
95 }
```

```
85     }
86
87     /**
88      * 使用指定的ApplicationContext创建DispatcherHandler（或其他类型的
89      * WebHandler派生调度程序）。
90      */
91     protected WebHandler createDispatcherHandler(ApplicationContext
92     applicationContext) {
93         return new DispatcherHandler(applicationContext);
94     }
95
96     /**
97      * 创建一个ServletHttpHandlerAdapter。 默认实现返回一个
98      * ServletHttpHandlerAdapter和提供的webHandler。
99      */
100    protected ServletHttpHandlerAdapter createHandlerAdapter(WebHandler
101    webHandler) {
102        HttpHandler httpHandler = new HttpWebHandlerAdapter(webHandler);
103        return new ServletHttpHandlerAdapter(httpHandler);
104    }
105
106    /**
107     * Specify the servlet mapping for the {@code
108     * ServletHttpHandlerAdapter}.
109     * <p>Default implementation returns {@code /}.
110     * @see #registerDispatcherHandler(ServletContext)
111     */
112     protected String getServletMapping() {
113         return DEFAULT_SERVLET_MAPPING;
114     }
115
116     /**
117      * 一旦registerDispatcherHandler (ServletContext) 完成，可以选择执行进一步的注
118      * 册自定义配置定制。
119      */
120     protected void customizeRegistration(ServletRegistration.Dynamic
121     registration) {
122
123     /**
124      * Register a {@link ServletContextListener} that closes the given
125      * application context
126      */
127     protected void registerCloseListener(ServletContext servletContext,
128     ApplicationContext applicationContext) {
129         if (applicationContext instanceof ConfigurableApplicationContext) {
130             ConfigurableApplicationContext context =
131             (ConfigurableApplicationContext) applicationContext;
132             ServletContextDestroyedListener listener = new
133             ServletContextDestroyedListener(context);
134             servletContext.addListener(listener);
135         }
136     }
137
138     private static class ServletContextDestroyedListener implements
139     ServletContextListener {
```

```

131     private final ConfigurableApplicationContext applicationContext;
132
133     public
134         ServletContextDestroyedListener(ConfigurableApplicationContext
135             applicationContext) {
136             this.applicationContext = applicationContext;
137         }
138
139         @Override
140         public void contextInitialized(ServletContextEvent sce) {
141         }
142
143         @Override
144         public void contextDestroyed(ServletContextEvent sce) {
145             this.applicationContext.close();
146         }
147     }

```

### AbstractServletHttpHandlerAdapterInitializer

在Servlet上下文中注册ServletHttpHandlerAdapter的WebApplicationInitializer实现的基类。

```

1  public abstract class AbstractServletHttpHandlerAdapterInitializer
2      implements WebApplicationInitializer {
3
4      /**
5       * The default servlet name. Can be customized by overriding {@link
6       * #getServletName}.
7       */
8
9      @Override
10     public void onStartup(ServletContext servletContext) throws
11         ServletException {
12         registerHandlerAdapter(servletContext);
13     }
14
15     /**
16      * Register a {@link ServletHttpHandlerAdapter} against the given
17      * servlet context.
18      * <p>This method will create a {@code HttpHandler} using {@link
19      * #createHttpHandler()},
20      * and use it to create a {@code ServletHttpHandlerAdapter} with the
21      * name returned by
22      * {@link #getServletName()}, and mapping it to the patterns
23      * returned from {@link #getServletMappings()}.

```

```

24     */
25     protected void registerHandlerAdapter(ServletContext servletContext) {
26         String servletName = getServletName();
27         Assert.hasLength(servletName, "getServletName() must not return
empty or null");
28
29         HttpHandler httpHandler = createHttpHandler();
30         Assert.notNull(httpHandler,
31                         "createHttpHandler() did not return a HttpHandler for
servlet [" + servletName + "]");
32
33         ServletHttpHandlerAdapter servlet = createServlet(httpHandler);
34         Assert.notNull(servlet,
35                         "createHttpHandler() did not return a
ServletHttpHandlerAdapter for servlet [" + servletName + "]");
36
37         ServletRegistration.Dynamic registration =
38         servletContext.addServlet(servletName, servlet);
39         Assert.notNull(registration,
40                         "Failed to register servlet with name '" +
servletName + "'." +
41                         "Check if there is another servlet registered under
the same name.");
42
43         registration.setLoadOnStartup(1);
44         registration.addMapping(getServletMappings());
45         registration.setAsyncSupported(true);
46
47         customizeRegistration(registration);
48     }
49
50     /**
51      * Return the name under which the {@link ServletHttpHandlerAdapter}
will be registered.
52      * Defaults to {@link #DEFAULT_SERVLET_NAME}.
53      * @see #registerHandlerAdapter(ServletContext)
54      */
55     protected String getServletName() {
56         return DEFAULT_SERVLET_NAME;
57     }
58
59     /**
60      * Create the {@link HttpHandler}.
61      */
62     protected abstract HttpHandler createHttpHandler();
63
64     /**
65      * Create a {@link ServletHttpHandlerAdapter} with the specified .
66      * <p>Default implementation returns a {@code ServletHttpHandlerAdapter}
with the provided
67      * {@code httpHandler}.
68      */
69     protected ServletHttpHandlerAdapter createServlet(HttpHandler
httpHandler) {
70         return new ServletHttpHandlerAdapter(httpHandler);
71     }
72     /**

```

```

73     * Specify the servlet mapping(s) for the {@code
74     ServletHttpHandlerAdapter} &mdash;
75     * for example {@code "/"}, {@code "/app"}, etc.
76     * @see #registerHandlerAdapter(ServletContext)
77     */
78     protected abstract String[] getServletMappings();
79
80     /**
81     * Optionally perform further registration customization once
82     * {@link #registerHandlerAdapter(ServletContext)} has completed.
83     * @param registration the {@code DispatcherServlet} registration to be
84     * customized
85     * @see #registerHandlerAdapter(ServletContext)
86     */
87     protected void customizeRegistration(ServletRegistration.Dynamic
registration) {
88
89 }

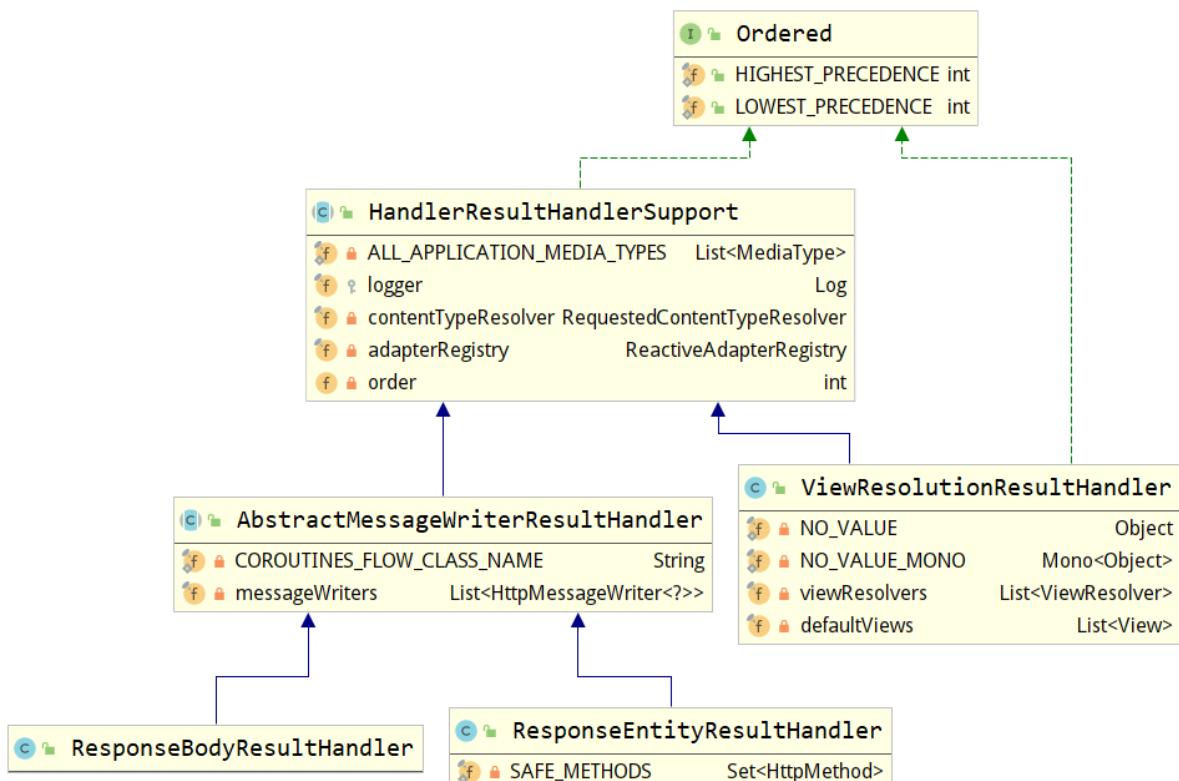
```

## 18 Spring WebFlux源码解析——result包

`org.springframework.web.reactive.result`

支持各种编程模型样式，包括调用不同类型的handler。

类图：



### HandlerResultHandlerSupport

HandlerResultHandler的基类，支持内容协商和访问ReactiveAdapter注册表。

```

1 | public abstract class HandlerResultHandlerSupport implements Ordered {

```

```
2      private static final MediaType MEDIA_TYPE_APPLICATION_ALL = new
3      MediaType("application");
4
5
6      private final RequestedContentTypeResolver contentTypeResolver;
7
8      private final ReactiveAdapterRegistry adapterRegistry;
9
10     private int order = LOWEST_PRECEDENCE;
11
12
13     protected HandlerResultHandlersSupport(RequestedContentTypeResolver
14     contentTypeResolver,
15                                         ReactiveAdapterRegistry
16     adapterRegistry) {
17
18         Assert.notNull(contentTypeResolver, "RequestedContentTypeResolver
19         is required");
20         Assert.notNull(adapterRegistry, "ReactiveAdapterRegistry is
21         required");
22         this.contentTypeResolver = contentTypeResolver;
23         this.adapterRegistry = adapterRegistry;
24     }
25
26
27     /**
28     * Return the configured {@link ReactiveAdapterRegistry}.
29     */
30     public ReactiveAdapterRegistry getAdapterRegistry() {
31         return this.adapterRegistry;
32     }
33
34     /**
35     * Return the configured {@link RequestedContentTypeResolver}.
36     */
37     public RequestedContentTypeResolver getContentResolver() {
38         return this.contentTypeResolver;
39     }
40
41     /**
42     * Set the order for this result handler relative to others.
43     * <p>By default set to {@link Ordered#LOWEST_PRECEDENCE}, however see
44     * Javadoc of sub-classes which may change this default.
45     * @param order the order
46     */
47     public void setorder(int order) {
48         this.order = order;
49     }
50
51
52     @Override
53     public int getOrder() {
54         return this.order;
55     }
56
57     /**
58     * Get a {@code ReactiveAdapter} for the top-level return value type.
59     */
```

```

55     * @return the matching adapter or {@code null}
56     */
57     @Nullable
58     protected ReactiveAdapter getAdapter(HandlerResult result) {
59         Class<?> returnType = result.getReturnType().getRawClass();
60         return getAdapterRegistry().getAdapter(returnType,
61             result.getReturnValue());
62     }
63     /**
64      * Select the best media type for the current request through a content
65      * negotiation algorithm.
66      * @param exchange the current request
67      * @param producibleTypesSupplier the media types that can be produced for
68      * the current request
69      * @return the selected media type or {@code null}
70     */
71     @Nullable
72     protected MediaType selectMediaType(ServerWebExchange exchange,
73                                         Supplier<List<MediaType>>
74                                         producibleTypesSupplier) {
75
76         List<MediaType> acceptableTypes = getAcceptableTypes(exchange);
77         List<MediaType> producibleTypes = getProducibleTypes(exchange,
78                                                 producibleTypesSupplier);
79
80         Set<MediaType> compatibleMediaTypes = new LinkedHashSet<>();
81         for (MediaType acceptable : acceptableTypes) {
82             for (MediaType producible : producibleTypes) {
83                 if (acceptable.isCompatibleWith(producible)) {
84
85                     compatibleMediaTypes.add(selectMoreSpecificMediaType(acceptable,
86                                         producible));
87                 }
88             }
89         }
90
91         List<MediaType> result = new ArrayList<>(compatibleMediaTypes);
92         MediaType.sortBySpecificityAndQuality(result);
93
94         for (MediaType mediaType : result) {
95             if (mediaType.isConcrete()) {
96                 return mediaType;
97             }
98             else if (mediaType.equals(MediaType.ALL) ||
99             mediaType.equals(MEDIA_TYPE_APPLICATION_ALL)) {
100                 return MediaType.APPLICATION_OCTET_STREAM;
101             }
102         }
103
104         return null;
105     }
106
107     private List<MediaType> getAcceptableTypes(ServerWebExchange exchange)
108     {
109         List<MediaType> mediaTypes =
110             getContentTypeResolver().resolveMediaTypes(exchange);

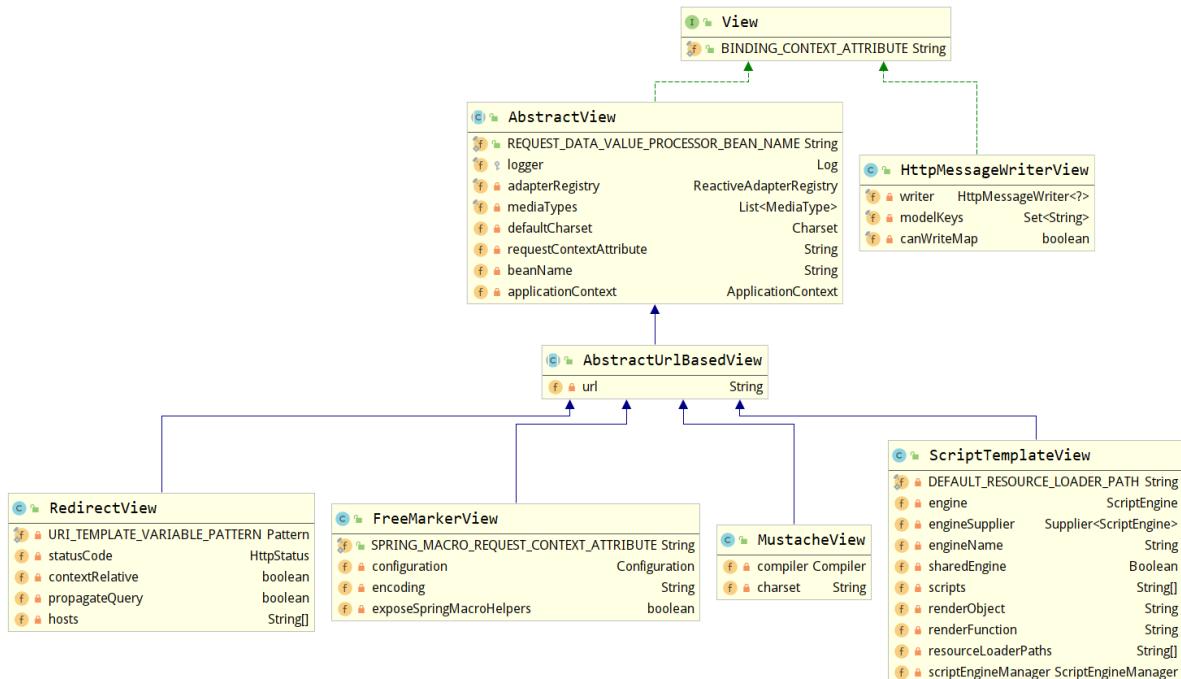
```

```

103         return (mediaTypes.isEmpty() ?
104             Collections.singletonList(MediaType.ALL) : mediaTypes);
105
106     @SuppressWarnings("unchecked")
107     private List<MediaType> getProducibleTypes(ServerWebExchange exchange,
108                                                 Supplier<List<MediaType>>
109                                                 producibleTypesSupplier) {
110
111         Set<MediaType> mediaTypes =
112             exchange.getAttribute(HandlerMapping.PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);
113         return (mediaTypes != null ? new ArrayList<>(mediaTypes) :
114             producibleTypesSupplier.get());
115     }
116
117     private MediaType selectMoreSpecificMediaType(MediaType acceptable,
118                                                   MediaType producible) {
119         producible = producible.copyQualityValue(acceptable);
120         Comparator<MediaType> comparator =
121             MediaType.SPECIFICITY_COMPARATOR;
122         return (comparator.compare(acceptable, producible) <= 0 ?
123             acceptable : producible);
124     }
125 }

```

## 接口 View 的类图



## View

通过视图解析支持结果处理。

将HandlerResult呈现给HTTP响应的约定。

与Encoder相比，Encoder是一个单实例对象，并对给定类型的任何对象进行编码。

因此，视图通常是通过名称来选择的，并使用ViewResolver来解析。

例如将其与HTML模板匹配。此外，视图可以基于模型中包含的多个属性呈现。

视图还可以选择从模型中选择一个属性，使用任何现有的编码器来呈现替代媒体类型。

```
1 // 返回此视图支持的媒体类型列表，或空列表。
2 List<MediaType> getSupportedMediaTypes();
3
4 // 这个视图是否通过执行重定向来呈现。
5 default boolean isRedirectView() {
6     return false;
7 }
8
9 // 根据给定的HandlerResult呈现视图。实现可以访问和使用模型，或者仅在其中使用一个特定的属性。
10 Mono<Void> render(@Nullable Map<String, ?> model, @Nullable MediaType
contentType, ServerWebExchange exchange);
```

## AbstractView

View实现的基类。

```
1 public abstract class AbstractView implements View, ApplicationContextAware
{
2
3     /** 在 bean factory 有名的requestDataValueProcessor */
4     public static final String REQUEST_DATA_VALUE_PROCESSOR_BEAN_NAME =
"requestDataValueProcessor";
5
6
7     /** 可用于子类的日志记录器 */
8     protected final Log logger = LoggerFactory.getLog(getClass());
9
10    private static final Object NO_VALUE = new Object();
11
12
13    private final List<MediaType> mediaTypes = new ArrayList<>(4);
14
15    private final ReactiveAdapterRegistry adapterRegistry;
16
17    private Charset defaultCharset = StandardCharsets.UTF_8;
18
19    @Nullable
20    private String requestContextAttribute;
21
22    @Nullable
23    private ApplicationContext applicationContext;
24
25
26    public AbstractView() {
27        this(new ReactiveAdapterRegistry());
```

```
28     }
29
30     public AbstractView(ReactiveAdapterRegistry registry) {
31         this.mediaTypes.add(ViewResolverSupport.DEFAULT_CONTENT_TYPE);
32         this.adapterRegistry = registry;
33     }
34
35
36     /**
37      * Set the supported media types for this view.
38      * Default is "text/html; charset=UTF-8".
39      */
40     public void setSupportedMediaTypes(@Nullable List<MediaType>
supportedMediaTypes) {
41         Assert.notEmpty(supportedMediaTypes, "MediaType List must not be
empty");
42         this.mediaTypes.clear();
43         if (supportedMediaTypes != null) {
44             this.mediaTypes.addAll(supportedMediaTypes);
45         }
46     }
47
48     /**
49      * Return the configured media types supported by this view.
50      */
51     @Override
52     public List<MediaType> getSupportedMediaTypes() {
53         return this.mediaTypes;
54     }
55
56     /**
57      * Set the default charset for this view, used when the
58      * {@linkplain #setSupportedMediaTypes(List)} content type} does not
59      * contain one.
60      * Default is {@linkplain StandardCharsets#UTF_8 UTF 8}.
61      */
62     public void setDefaultCharset(Charset defaultCharset) {
63         Assert.notNull(defaultCharset, "'defaultCharset' must not be
null");
64         this.defaultCharset = defaultCharset;
65     }
66
67     /**
68      * Return the default charset, used when the
69      * {@linkplain #setSupportedMediaTypes(List)} content type} does not
70      * contain one.
71      */
72     public Charset getDefaultCharset() {
73         return this.defaultCharset;
74     }
75
76     /**
77      * Set the name of the RequestContext attribute for this view.
78      * Default is none.
79      */
80     public void setRequestContextAttribute(@Nullable String
requestContextAttribute) {
81         this.requestContextAttribute = requestContextAttribute;
82     }
83 }
```

```

80    }
81
82    /**
83     * Return the name of the RequestContext attribute, if any.
84     */
85    @Nullable
86    public String getRequestContextAttribute() {
87        return this.requestContextAttribute;
88    }
89
90    @Override
91    public void setApplicationContext(@Nullable ApplicationContext
92                                   applicationContext) {
93        this.applicationContext = applicationContext;
94    }
95
96    @Nullable
97    public ApplicationContext getApplicationContext() {
98        return this.applicationContext;
99    }
100
101   /**
102    * Obtain the ApplicationContext for actual use.
103    * @return the ApplicationContext (never {@code null})
104    * @throws IllegalStateException in case of no ApplicationContext set
105    */
106   protected final ApplicationContext obtainApplicationContext() {
107       ApplicationContext applicationContext = getApplicationContext();
108       Assert.state(applicationContext != null, "No ApplicationContext");
109       return applicationContext;
110   }
111
112   /**
113    * Prepare the model to render.
114    * @param model Map with name Strings as keys and corresponding model
115    * objects as values (Map can also be {@code null} in case of empty
116    * model)
117    * @param contentType the content type selected to render with which
118    * should
119    * match one of the {@link #getSupportedContentTypes()} supported media
120    * types}.
121    * @param exchange the current exchange
122    * @return {@code Mono} to represent when and if rendering succeeds
123    */
124   @Override
125   public Mono<Void> render(@Nullable Map<String, ?> model, @Nullable
126                           MediaType contentType,
127                           ServerWebExchange exchange) {
128
129       if (logger.isTraceEnabled()) {
130           logger.trace("Rendering view with model " + model);
131       }
132
133       if (contentType != null) {
134
135           exchange.getResponse().getHeaders().setContentType(contentType);
136       }

```

```

132
133     return getModelAttributes(model, exchange).flatMap(mergedModel -> {
134         // Expose RequestContext?
135         if (this.requestContextAttribute != null) {
136             mergedModel.put(this.requestContextAttribute,
137                 createRequestContext(exchange, mergedModel));
138         }
139     });
140 }
141
142 /**
143 * Prepare the model to use for rendering.
144 * <p>The default implementation creates a combined output Map that
145 includes
146 * model as well as static attributes with the former taking
147 precedence.
148 */
149 protected Mono<Map<String, Object>> getModelAttributes(@Nullable
150 Map<String, ?> model,
151 ServerWebExchange exchange) {
152
153     int size = (model != null ? model.size() : 0);
154
155     Map<String, Object> attributes = new LinkedHashMap<>(size);
156     if (model != null) {
157         attributes.putAll(model);
158     }
159
160     return
161         resolveAsyncAttributes(attributes).then(Mono.just(attributes));
162 }
163
164 /**
165 * By default, resolve async attributes supported by the
166 * {@link ReactiveAdapterRegistry} to their blocking counterparts.
167 * <p>View implementations capable of taking advantage of reactive
168 types
169 * can override this method if needed.
170 * @return {@code Mono} for the completion of async attributes
171 resolution
172 */
173 protected Mono<Void> resolveAsyncAttributes(Map<String, Object> model)
174 {
175
176     List<String> names = new ArrayList<>();
177     List<Mono<?>> valueMones = new ArrayList<>();
178
179     for (Map.Entry<String, ?> entry : model.entrySet()) {
180         Object value = entry.getValue();
181         if (value == null) {
182             continue;
183         }
184         ReactiveAdapter adapter = this.adapterRegistry.getAdapter(null,
185             value);
186         if (adapter != null) {
187             names.add(entry.getKey());
188         }
189     }
190 }
```

```

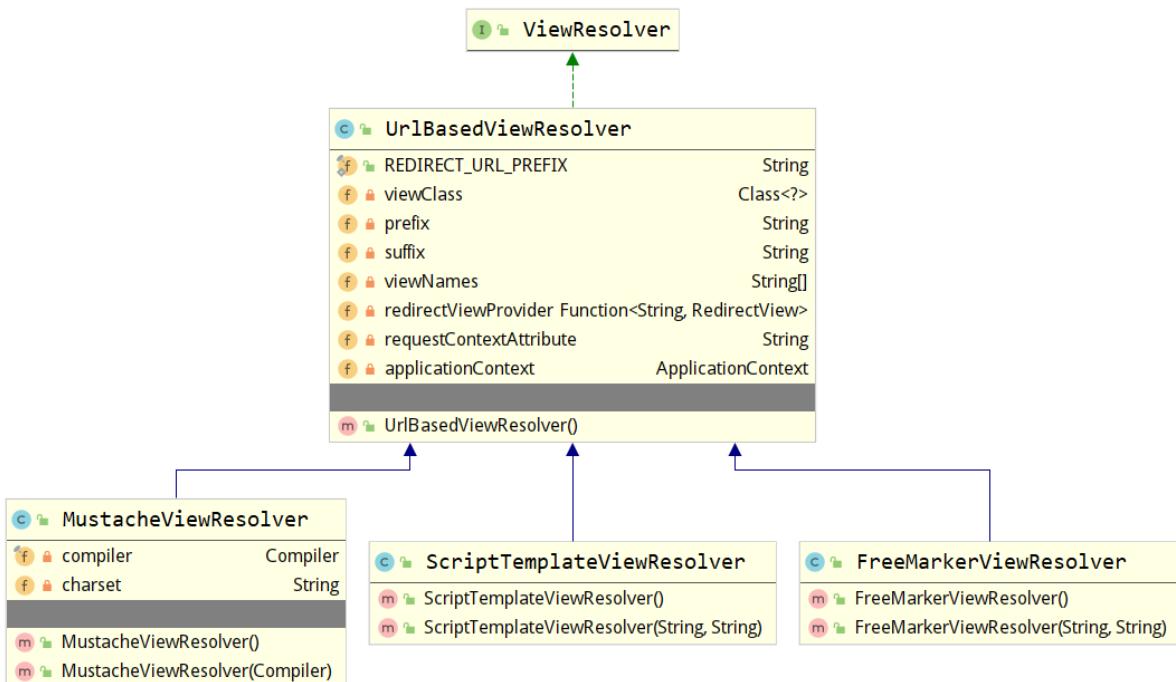
180             if (adapter.isMultivalue()) {
181                 Flux<Object> fluxvalue =
182                     Flux.from(adapter.toPublisher(value));
183             }
184             else {
185                 Mono<Object> monovalue =
186                     Mono.from(adapter.toPublisher(value));
187             }
188         }
189     }
190
191     if (names.isEmpty()) {
192         return Mono.empty();
193     }
194
195     return Mono.zip(valueMonos,
196                     values -> {
197             for (int i=0; i < values.length; i++) {
198                 if (values[i] != NO_VALUE) {
199                     model.put(names.get(i), values[i]);
200                 }
201                 else {
202                     model.remove(names.get(i));
203                 }
204             }
205             return NO_VALUE;
206         })
207         .then();
208     }
209
210 /**
211 * Create a RequestContext to expose under the specified attribute
212 name.
213 * <p>The default implementation creates a standard RequestContext
214 instance
215 * for the given request and model. Can be overridden in subclasses for
216 * custom instances.
217 * @param exchange current exchange
218 * @param model combined output Map (never {@code null}),
219 * with dynamic values taking precedence over static attributes
220 * @return the RequestContext instance
221 * @see #setRequestContextAttribute
222 */
223 protected RequestContext createRequestContext(ServerWebExchange
224 exchange, Map<String, Object> model) {
225     return new RequestContext(exchange, model,
226     obtainApplicationContext(), getRequestDataValueProcessor());
227 }
```

```

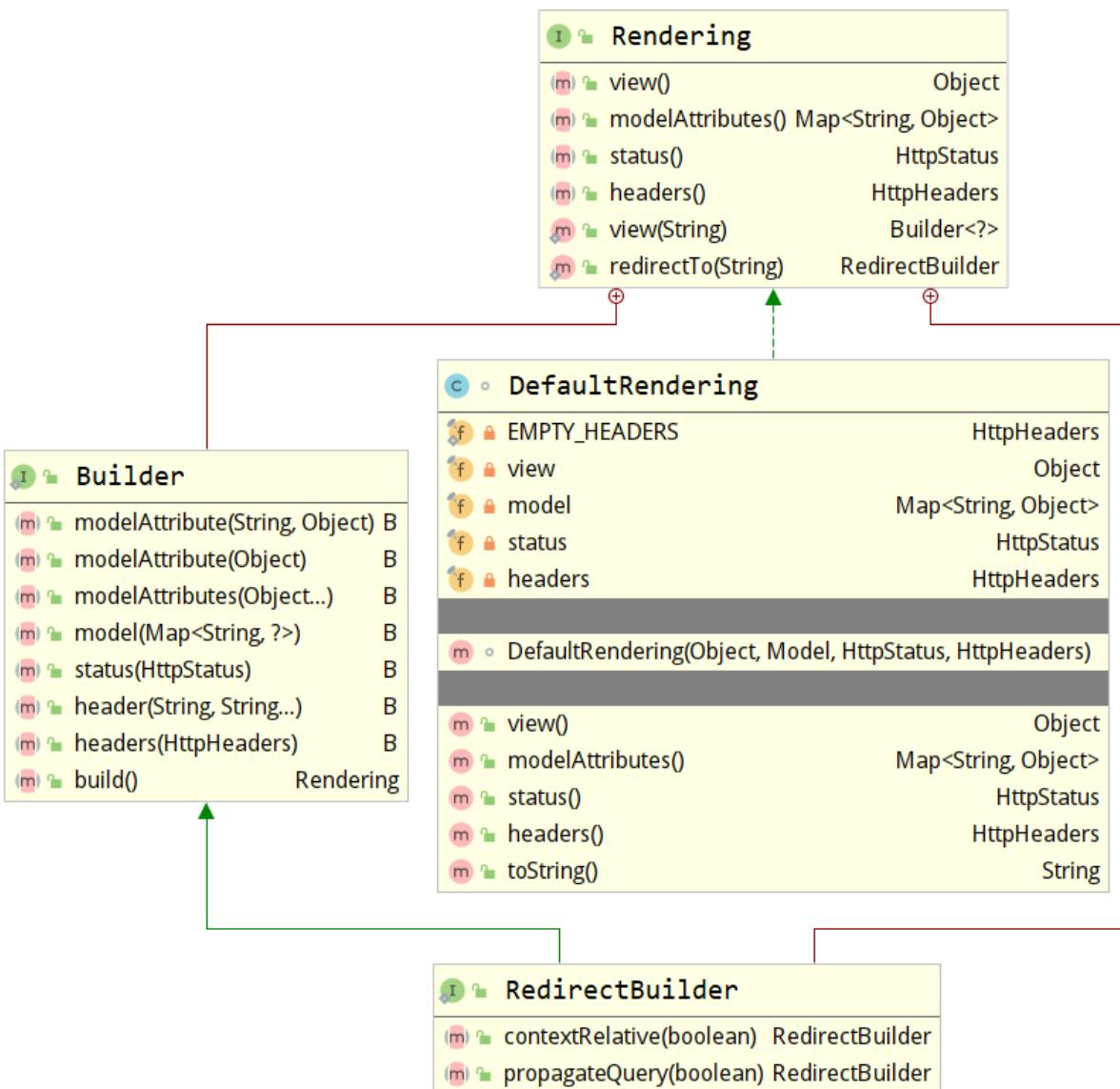
228     * Spring configuration} for a {@code RequestDataValueProcessor} bean
229     with
230     * the name {@link #REQUEST_DATA_VALUE_PROCESSOR_BEAN_NAME}.
231     * @return the RequestDataValueProcessor, or null if there is none at
232     * the
233     * application context.
234     */
235     @Nullable
236     protected RequestDataValueProcessor getRequestDataValueProcessor() {
237         ApplicationContext context = getApplicationContext();
238         if (context != null &&
239             context.containsBean(REQUEST_DATA_VALUE_PROCESSOR_BEAN_NAME)) {
240             return context.getBean(REQUEST_DATA_VALUE_PROCESSOR_BEAN_NAME,
241             RequestDataValueProcessor.class);
242         }
243         return null;
244     }
245
246     /**
247      * Subclasses must implement this method to actually render the view.
248      * @param renderAttributes combined output Map (never {@code null}),
249      * with dynamic values taking precedence over static attributes
250      * @param contentType the content type selected to render with which
251      * should
252      * match one of the {@link #getSupportedMediaTypes()} supported media
253      * types}.
254      * @param exchange current exchange @return {@code Mono} to represent
255      * when
256      * and if rendering succeeds
257      */
258     protected abstract Mono<Void> renderInternal(Map<String, Object>
259     renderAttributes,
260                                         @Nullable MediaType
261                                         contentType, ServerWebExchange exchange);
262
263
264     @Override
265     public String toString() {
266         return getClass().getName();
267     }
268 }

```

## ViewResolver



## Rendering



## SimpleHandlerAdapter

HandlerAdapter，它允许使用普通的WebHandler与一般的DispatcherHandler一起使用。

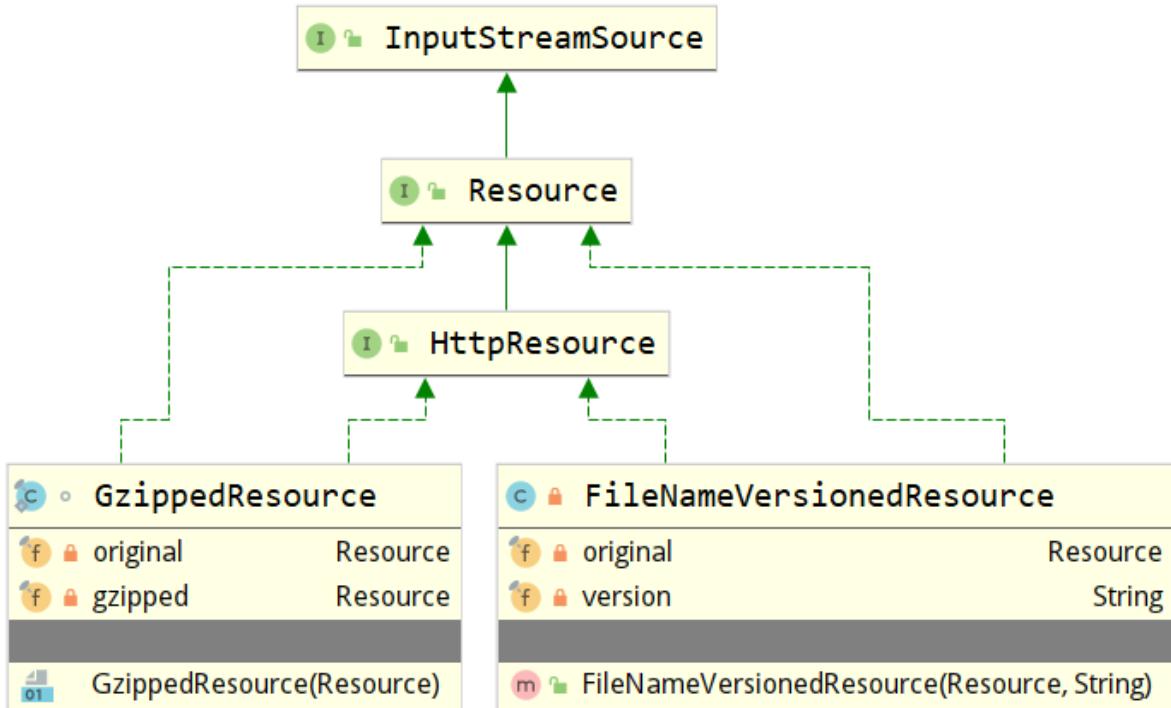
```
1 public class SimpleHandlerAdapter implements HandlerAdapter {
2
3     private static final MethodParameter RETURN_TYPE;
4
5     static {
6         try {
7             Method method = WebHandler.class.getMethod("handle",
8                 ServerWebExchange.class);
8             RETURN_TYPE = new MethodParameter(method, -1);
9         }
10        catch (NoSuchMethodException ex) {
11            throw new IllegalStateException(
12                "Failed to initialize the return type for webHandler: " +
13                ex.getMessage());
14        }
15    }
16
17    //这个HandlerAdapter是否支持给定的handler。
18    @Override
19    public boolean supports(Object handler) {
20        return WebHandler.class.isAssignableFrom(handler.getClass());
21    }
22
23    //鼓励实现处理由调用handler所产生的异常，并在必要时返回代表错误响应的替代结果。
24    @Override
25    public Mono<HandlerResult> handle(ServerWebExchange exchange, Object
26 handler) {
27        WebHandler webHandler = (WebHandler) handler;
28        Mono<Void> mono = webHandler.handle(exchange);
29        return mono.then(Mono.empty());
30    }
31}
```

## 19 Spring WebFlux源码解析——resource包

org.springframework.web.reactive.resource

支持为静态资源提供服务的类。

Resource类图：



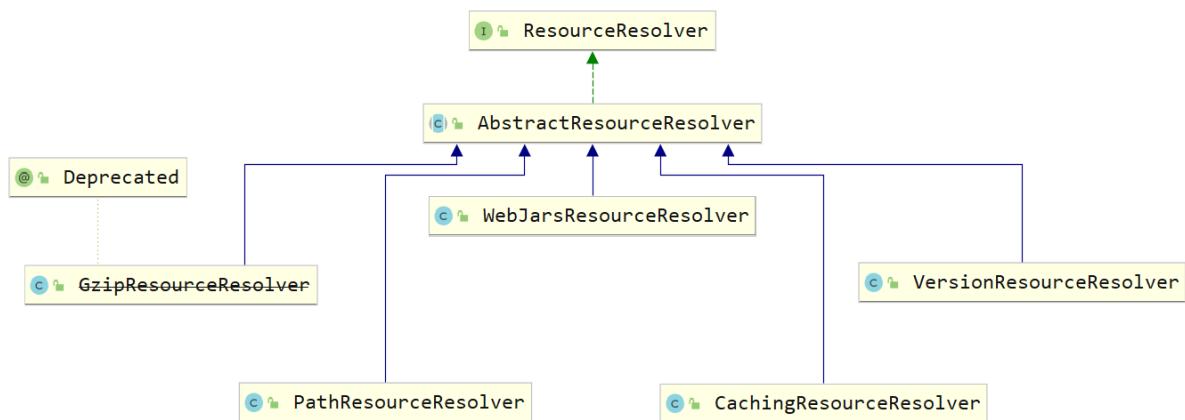
将资源写入HTTP响应的扩展接口。

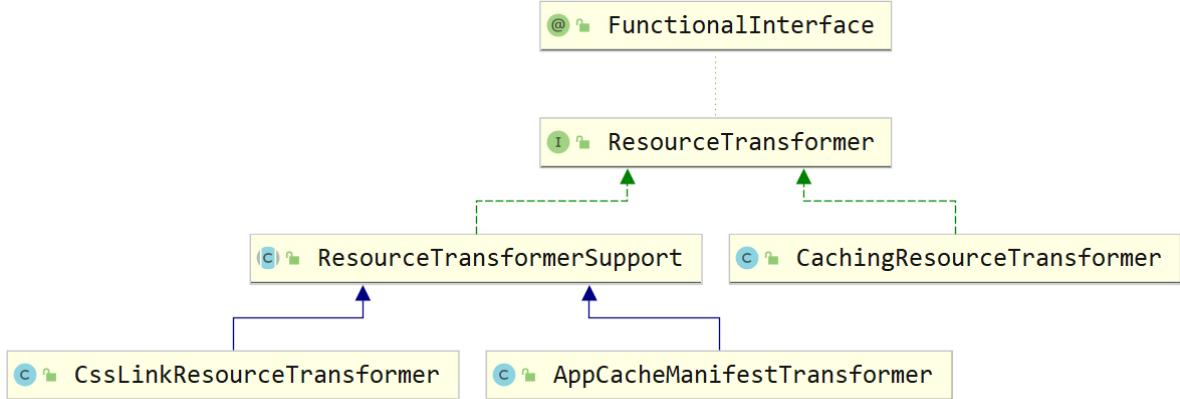
HTTP头将被提供给服务当前资源的HTTP响应。返回HttpHeaders

```

1 public interface HttpResource extends Resource {
2
3     /**
4      * The HTTP headers to be contributed to the HTTP response
5      * that serves the current resource.
6      * @return the HTTP response headers
7      */
8     HttpHeaders getResponseHeaders();
9 }
  
```

## ResourceResolver





解决对服务器端资源的请求的策略。

提供解决传入请求到实际资源的机制，以及获取客户端在请求资源时应该使用的公共URL路径。

```

1 | Mono<Resource> resolveResource(@Nullable ServerWebExchange exchange, String
2 |   requestPath,
|     List<? extends Resource> locations, ResourceResolverChain chain);
  
```

将提供的请求和请求路径解析为存在于其中一个给定资源位置下的资源。

```

1 | Mono<String> resolveUrlPath(String resourcePath, List<? extends Resource>
2 |   locations,
|     ResourceResolverChain chain);
3 | }
  
```

解析面向外部的公用URL路径，供客户端用来访问位于给定内部资源路径的资源。在向客户端呈现URL链接时这很有用。

### ResourceResolverChain

调用ResourceResolvers链的协定，其中每个解析器都被赋予一个引用链，允许它在必要时进行委托。

```

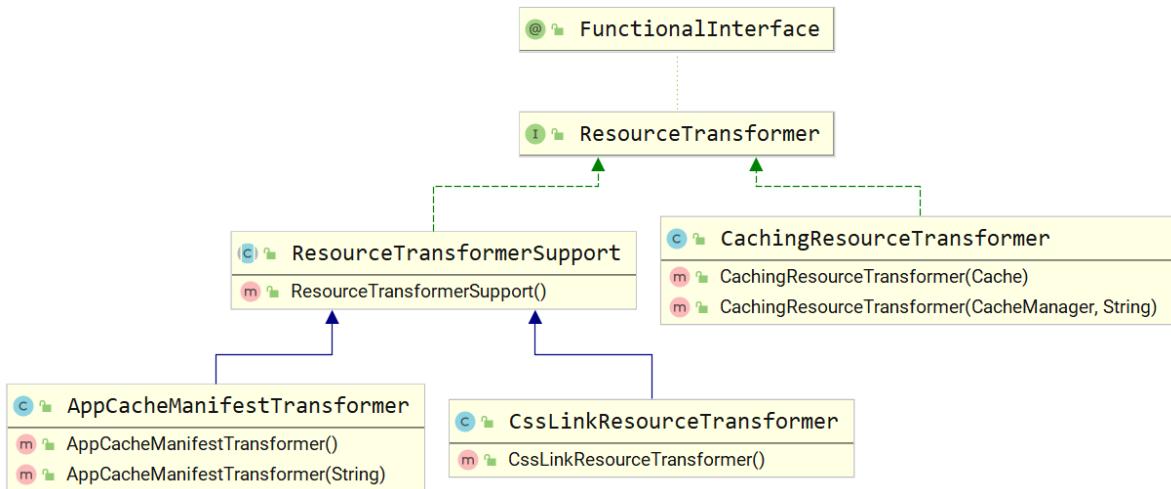
1 | Mono<Resource> resolveResource(@Nullable ServerWebExchange exchange, String
2 |   requestPath,
|     List<? extends Resource> locations);
  
```

将提供的请求和请求路径解析为存在于其中一个给定资源位置下的资源。

```
1 | Mono<String> resolveUrlPath(String resourcePath, List<? extends Resource>
  locations);}
```

解析面向外部的公用URL路径，供客户端用来自访问位于给定内部资源路径的资源。

## ResourceTransformer



转换资源内容的抽象。

转换给定的资源。

```
1 @FunctionalInterface
2 public interface ResourceTransformer {
3
4     /**
5      * Transform the given resource.
6      * @param exchange the current exchange
7      * @param resource the resource to transform
8      * @param transformerChain the chain of remaining transformers to
9      * delegate to
10     * @return the transformed resource (never empty)
11     */
12     Mono<Resource> transform(ServerWebExchange exchange, Resource resource,
13                             ResourceTransformerChain transformerChain);
14 }
```

## ResourceTransformerChain

一个调用ResourceTransformers链的协议，每个解析器都被赋予一个链，让它在必要时进行委托。

- `getResolverChain()` 返回用于解析正在转换的资源的ResourceResolverChain。这可能需要解析相关的资源，例如链接到其他资源。
- `transform(ServerWebExchange exchange, Resource resource)` 转换给定的资源。

```
1 public interface ResourceTransformerChain {  
2  
3     /**  
4      * Return the {@code ResourceResolverChain} that was used to resolve the  
5      * {@code Resource} being transformed. This may be needed for resolving  
6      * related resources, e.g. links to other resources.  
7      */  
8     ResourceResolverChain getResolverChain();  
9  
10    /**  
11     * Transform the given resource.  
12     * @param exchange the current exchange  
13     * @param resource the candidate resource to transform  
14     * @return the transformed or the same resource, never empty  
15     */  
16    Mono<Resource> transform(ServerWebExchange exchange, Resource resource);  
17  
18 }
```

```
1 @FunctionalInterface  
2 protected static interface CssLinkResourceTransformer.LinkParser
```

提取表示链接的内容块。

```
1 @FunctionalInterface  
2 protected interface LinkParser {  
3  
4     void parse(String cssContent, SortedSet<ContentChunkInfo> result);  
5  
6 }
```

## AbstractResourceResolver

基于 ResourceResolver 提供一致的日志记录。

```
1 // 将提供的请求和请求路径解析为存在于其中一个给定资源位置下的资源。  
2 @Override  
3 public Mono<Resource> resolveResource(@Nullable ServerWebExchange exchange,  
4                                         String requestPath,  
5                                         List<? extends Resource> locations, ResourceResolverChain chain) {  
6  
7     if (logger.isTraceEnabled()) {  
8         logger.trace("Resolving resource for request path \"{}\" + requestPath  
+ \"\");  
9     }  
10    return resolveResourceInternal(exchange, requestPath, locations, chain);  
11 }
```

```

1 // 解析面向外部的公用URL路径，供客户端用来访问位于给定内部资源路径的资源。
2 @Override
3 public Mono<String> resolveUrlPath(String resourceUrlPath, List<? extends
4 Resource> locations,
5                                     ResourceResolverChain chain) {
6
7     if (logger.isTraceEnabled()) {
8         logger.trace("Resolving public URL for resource path \'" +
9             resourceUrlPath + "\'");
10    }
11
12
13
14 protected abstract Mono<Resource> resolveResourceInternal(@Nullable
15 ServerWebExchange exchange,
16
17                                         String
18 requestPath, List<? extends Resource> locations, ResourceResolverChain
19 chain);
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
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
187
188
189
189
190
191
192
193
194
195
196
197
197
198
199
199
200
201
202
203
204
205
206
207
207
208
209
209
210
211
212
213
214
215
215
216
216
217
217
218
218
219
219
220
220
221
221
222
222
223
223
224
224
225
225
226
226
227
227
228
228
229
229
230
230
231
231
232
232
233
233
234
234
235
235
236
236
237
237
238
238
239
239
240
240
241
241
242
242
243
243
244
244
245
245
246
246
247
247
248
248
249
249
250
250
251
251
252
252
253
253
254
254
255
255
256
256
257
257
258
258
259
259
260
260
261
261
262
262
263
263
264
264
265
265
266
266
267
267
268
268
269
269
270
270
271
271
272
272
273
273
274
274
275
275
276
276
277
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
139
```

确定静态资源的版本并使用and或or从URL路径中提取的策略。

```
1 // 从请求路径中提取资源版本。  
2 @Nullable  
3 String extractVersion(String requestPath);
```

```
1 // 从请求路径中删除版本。假定给定的版本是通过extractVersion (String) 提取的。  
2 String removeversion(String requestPath, String version);
```

```
1 // 给给定的请求路径添加一个版本。  
2 String addversion(String requestPath, String version);
```

```
1 // 确定给定资源的版本  
2 Mono<String> getResourceversion(Resource resource);
```

### VersionStrategy 的实现类：

#### AbstractFileNameVersionStrategy

基于文件名后缀的抽象基类，基于VersionStrategy实现，例如“static/ myresource-version.js”

```
1 // 从请求路径中提取资源版本。  
2 @Override  
3 public String extractVersion(String requestPath) {  
4     Matcher matcher = pattern.matcher(requestPath);  
5     if (matcher.find()) {  
6         String match = matcher.group(1);  
7         return (match.contains("-") ? match.substring(match.lastIndexOf('-')  
+ 1) : match);  
8     }  
9     else {  
10         return null;  
11     }  
12 }
```

```
1 // 从请求路径中删除版本
2 @Override
3 public String removeversion(String requestPath, String version) {
4     return StringUtils.delete(requestPath, "-" + version);
5 }
```

```
1 // 给给定的请求路径添加一个版本。
2 @Override
3 public String addversion(String requestPath, String version) {
4     String basefilename = StringUtils.stripFilenameExtension(requestPath);
5     String extension = StringUtils.getFilenameExtension(requestPath);
6     return (basefilename + '-' + version + '.' + extension);
7 }
```

## AbstractPrefixVersionStrategy

用于在URL路径中插入前缀的版本策略实现的抽象基类。例如：“version/static/myresource.js”。

```
1 protected final Log logger = LoggerFactory.getLog(getClass());
2
3
4 private final String prefix;
5
6
7 protected AbstractPrefixVersionStrategy(String version) {
8     Assert.hasText(version, "'version' must not be empty");
9     this.prefix = version;
10 }
11
12
13 @Override
14 public String extractVersion(String requestPath) {
15     return requestPath.startsWith(this.prefix) ? this.prefix : null;
16 }
17
18 @Override
19 public String removeVersion(String requestPath, String version) {
20     return requestPath.substring(this.prefix.length());
21 }
22
23 @Override
24 public String addVersion(String path, String version) {
25     if (path.startsWith(".")) {
26         return path;
27     }
28     else if (this.prefix.endsWith("/") || path.startsWith("/")) {
29         return this.prefix + path;
30     }
31     else {
32         return this.prefix + '/' + path;
33     }
34 }
```

## 具体实现类 ContentVersionStrategy

从资源的内容中计算Hex MD5散列的版本策略，并将其附加到文件名。“styles/main-e36d2e05253c6c7085a91522ce43a0b4.css”。

```

1 public class ContentVersionStrategy extends AbstractFileNameVersionStrategy {
2
3     private static final DataBufferFactory dataBufferFactory = new
4     DefaultDataBufferFactory();
5
6     // 确定给定资源的版本。
7     @Override
8     public Mono<String> getResourceversion(Resource resource) {
9         return DataBufferUtils.read(resource, dataBufferFactory,
10             StreamUtils.BUFFER_SIZE)
11             .reduce(DataBuffer::write)
12             .map(buffer -> {
13                 byte[] result = new byte[buffer.readableByteCount()];
14                 buffer.read(result);
15                 DataBufferUtils.release(buffer);
16                 return Digestutils.md5DigestAsHex(result);
17             });
18     }

```

## 具体实现类 FixedVersionStrategy

依赖于固定版本作为请求路径前缀的VersionStrategy，例如减少SHA，版本名称，发布日期等

例如当ContentVersionStrategy无法使用时，例如使用负责加载JavaScript资源并需要知道其相对路径的JavaScript模块加载器时，这非常有用。

```

1 public class FixedVersionStrategy extends AbstractPrefixVersionStrategy {
2
3     private final Mono<String> versionMono;
4
5     /**
6      * Create a new FixedVersionStrategy with the given version string.
7      * @param version the fixed version string to use
8      */
9     public FixedVersionStrategy(String version) {
10         super(version);
11         this.versionMono = Mono.just(version);
12     }
13 }

```

```

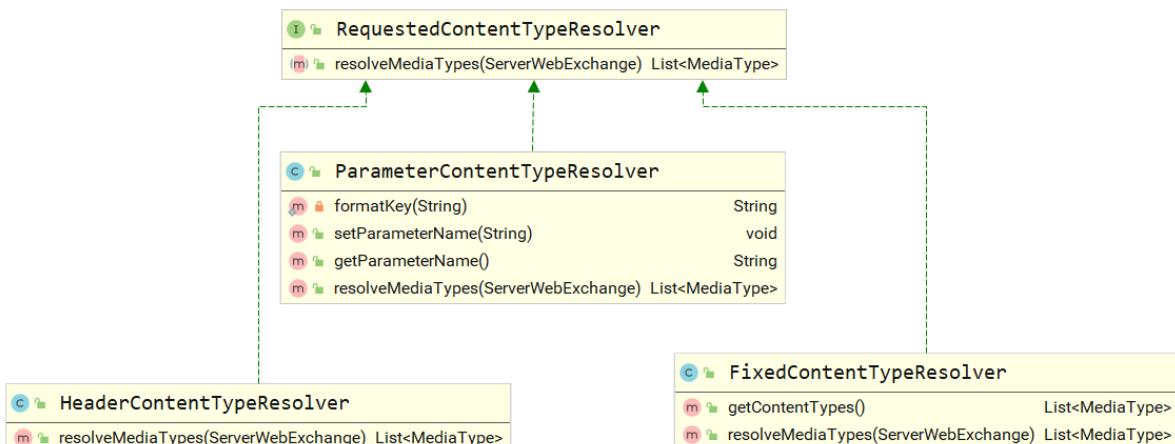
15
16     @Override
17     public Mono<String> getResourceVersion(Resource resource) {
18         return this.versionMono;
19     }
20
21 }

```

## 20 Spring WebFlux源码解析——accept包

`org.springframework.web.reactive.accept`

`RequestedContentTypeResolver` 策略和实现来解析给定请求的请求内容类型。



### 接口`RequestedContentTypeResolver`

为`ServerWebExchange`解决请求的媒体类型的策略。

```

1 public interface RequestedContentTypeResolver {
2
3     /**
4      * Resolve the given request to a list of requested media types. The
5      * returned
6      *      * list is ordered by specificity first and by quality parameter second.
7      *      * @param exchange the current exchange
8      *      * @return the requested media types or an empty list
9      *      * @throws NotAcceptableStatusException if the requested media type is
10     * invalid
11     */
12     List<MediaType> resolveMediaTypes(ServerWebExchange exchange);
13 }

```

将给定的请求解析为请求的媒体类型列表。返回的列表首先按特异性排序，然后按质量参数排序。

### 类org.springframework.web.reactive.accept.FixedContentTypeResolver

可以用作“最后一行”策略，当客户端没有请求任何媒体类型时提供回调。

```
1 public class FixedContentTypeResolver implements
2 RequestedContentTypeResolver {
3
4     private static final Log logger =
5     LogFactory.getLog(FixedContentTypeResolver.class);
6
7
8     /**
9      * 具有单个默认MediaType的构造函数。.
10     */
11    public FixedContentTypeResolver(MediaType mediaType) {
12        this(Collections.singletonList(mediaType));
13    }
14
15    /**
16     * 使用默认的MediaType排序列表的构造函数返回用于支持各种内容类型的应用程序。
17     * 如果目标不存在，并且不支持任何其他默认媒体类型，请考虑在最后附加MediaType.ALL。
18     */
19    public FixedContentTypeResolver(List<MediaType> mediaTypes) {
20        this.mediaTypes = Collections.unmodifiableList(mediaTypes);
21    }
22
23
24    /**
25     * 返回配置的媒体类型列表。
26     */
27    public List<MediaType> getSupportedContentTypes() {
28        return this.mediaTypes;
29    }
30
31
32
33    @Override
34    public List<MediaType> resolveMediaTypes(ServerWebExchange exchange) {
35        if (logger.isDebugEnabled()) {
36            logger.debug("Requested media types: " + this.mediaTypes);
37        }
38        return this.mediaTypes;
39    }
40
41 }
```

### org.springframework.web.reactive.accept.HeaderContentTypeResolver

解析器查看请求的“Accept”请求报文头。

```

1 public class HeaderContentTypeResolver implements
2 RequestedContentTypeResolver {
3
4     @Override
5     public List<MediaType> resolveMediaTypes(ServerWebExchange exchange)
6     throws NotAcceptableStatusException {
7         try {
8             List<MediaType> mediaTypes =
9                 exchange.getRequest().getHeaders().getAccept();
10            MediaType.sortBySpecificityAndQuality(mediaTypes);
11            return mediaTypes;
12        }
13        catch (InvalidMediaTypeException ex) {
14            String value =
15                exchange.getRequest().getHeaders().getFirst("Accept");
16            throw new NotAcceptableStatusException(
17                "Could not parse 'Accept' header [" + value + "]: " +
18                ex.getMessage());
19        }
20    }
21 }

```

## org.springframework.web.reactive.accept.ParameterContentTypeResolver

解析查询参数并使用它查找匹配的MediaType的解析器。

查找键可以注册，也可以作为一个后备的MediaTypeFactory执行查找。

```

1 public class ParameterContentTypeResolver implements
2 RequestedContentTypeResolver {
3
4     /** Primary lookup for media types by key (e.g. "json" ->
5      * "application/json") */
6     private final Map<String, MediaType> mediaTypes = new
7     ConcurrentHashMap<>(64);
8
9     private String parameterName = "format";
10
11
12     public ParameterContentTypeResolver(Map<String, MediaType> mediaTypes) {
13         mediaTypes.forEach((key, value) ->
14             this.mediaTypes.put(formatKey(key), value));
15     }
16
17     private static String formatKey(String key) {
18         return key.toLowerCase(Locale.ENGLISH);
19     }
20
21     /**
22      * Set the name of the parameter to use to determine requested media
23      * types.
24     */
25 }

```

```

20     * <p>By default this is set to {@literal "format"}.
21     */
22     public void setParameterName(String parameterName) {
23         Assert.notNull(parameterName, "'parameterName' is required");
24         this.parameterName = parameterName;
25     }
26
27     public String getParameterName() {
28         return this.parameterName;
29     }
30
31
32     @Override
33     public List<MediaType> resolveMediaTypes(ServerWebExchange exchange)
34     throws NotAcceptableStatusException {
35         String key =
36             exchange.getRequest().getQueryParams().getFirst(getParameterName());
37         if (!StringUtils.hasText(key)) {
38             return Collections.emptyList();
39         }
40         key = formatKey(key);
41         MediaType match = this.mediaTypes.get(key);
42         if (match == null) {
43             match = MediaTypeFactory.getMediaType("filename." + key)
44                 .orElseThrow(() -> {
45                     List<MediaType> supported = new ArrayList<>
46                         (this.mediaTypes.values());
47                     return new NotAcceptableStatusException(supported);
48                 });
49         }
50     }

```

## org.springframework.web.reactive.accept.RequestedContentTypeResolverBuilder

Builder的复合RequestedContentTypeResolver;

代表其他解析器实现一个不同的策略来确定请求的内容类型,例如Accept报文头,查询参数,或其他。

使用生成器方法在所需的顺序中添加解析器。

对于给定的请求,首先解析返回一个非空的列表。

默认情况下,如果没有显式解析器配置,构建器将添加HeaderContentTypeResolver。

```

1  public class RequestedContentTypeResolverBuilder {
2
3      private final List<Supplier<RequestedContentTypeResolver>> candidates =
4          new ArrayList<>();
5
6      /**

```

```

7   * Add a resolver to get the requested content type from a query
8   * parameter.
9   */
10  public ParameterResolverConfigurer parameterResolver() {
11      ParameterResolverConfigurer parameterBuilder = new
12      ParameterResolverConfigurer();
13      this.candidates.add(parameterBuilder::createResolver);
14      return parameterBuilder;
15  }
16
17  /**
18   * Add resolver to get the requested content type from the
19   * {@literal "Accept"} header.
20   */
21  public void headerResolver() {
22      this.candidates.add(HeaderContentTypeResolver::new);
23  }
24
25  /**
26   * Add resolver that returns a fixed set of media types.
27   * @param mediaTypes the media types to use
28   */
29  public void fixedResolver(MediaType... mediaTypes) {
30      this.candidates.add(() -> new
31      FixedContentTypeResolver(Arrays.asList(mediaTypes)));
32  }
33
34  /**
35   * Add a custom resolver.
36   * @param resolver the resolver to add
37   */
38  public void resolver(RequestedContentTypeResolver resolver) {
39      this.candidates.add(() -> resolver);
40  }
41
42  /**
43   * Build a {@link RequestedContentTypeResolver} that delegates to the
44   * list
45   * of resolvers configured through this builder.
46   */
47  public RequestedContentTypeResolver build() {
48
49      List<RequestedContentTypeResolver> resolvers =
50      this.candidates.isEmpty() ?
51          Collections.singletonList(new HeaderContentTypeResolver()) :
52
53      this.candidates.stream().map(Supplier::get).collect(Collectors.toList());
54
55      return exchange -> {
56          for (RequestedContentTypeResolver resolver : resolvers) {
57              List<MediaType> type =
58                  resolver.resolveMediaTypes(exchange);
59              if (type.isEmpty() || (type.size() == 1 &&
60                  type.contains(MediaType.ALL))) {
61                  continue;
62              }
63              return type;
64          }
65      };
66  }

```

```

58         }
59         return Collections.emptyList();
60     };
61 }
62
63
64 /**
65 * Helper to create and configure {@link ParameterContentTypeResolver}.
66 */
67 public static class ParameterResolverConfigurer {
68
69     private final Map<String, MediaType> mediaTypes = new HashMap<>();
70
71     @Nullable
72     private String parameterName;
73
74     /**
75      * Configure a mapping between a lookup key (extracted from a query
76      * parameter value) and a corresponding {@code MediaType}.
77      * @param key the lookup key
78      * @param mediaType the MediaType for that key
79      */
80     public ParameterResolverConfigurer mediaType(String key, MediaType
mediaType) {
81         this.mediaTypes.put(key, mediaType);
82         return this;
83     }
84
85     /**
86      * Map-based variant of {@link #mediaType(String, MediaType)}.
87      * @param mediaTypes the mappings to copy
88      */
89     public ParameterResolverConfigurer mediaType(Map<String, MediaType>
mediaTypes) {
90         this.mediaTypes.putAll(mediaTypes);
91         return this;
92     }
93
94     /**
95      * Set the name of the parameter to use to determine requested
media types.
96      * <p>By default this is set to {@literal "format"}.
97      */
98     public ParameterResolverConfigurer parameterName(String
parameterName) {
99         this.parameterName = parameterName;
100        return this;
101    }
102
103    /**
104     * Private factory method to create the resolver.
105     */
106    private RequestedContentTypeResolver createResolver() {
107        ParameterContentTypeResolver resolver = new
ParameterContentTypeResolver(this.mediaTypes);
108        if (this.parameterName != null) {
109            resolver.setParameterName(this.parameterName);
110        }

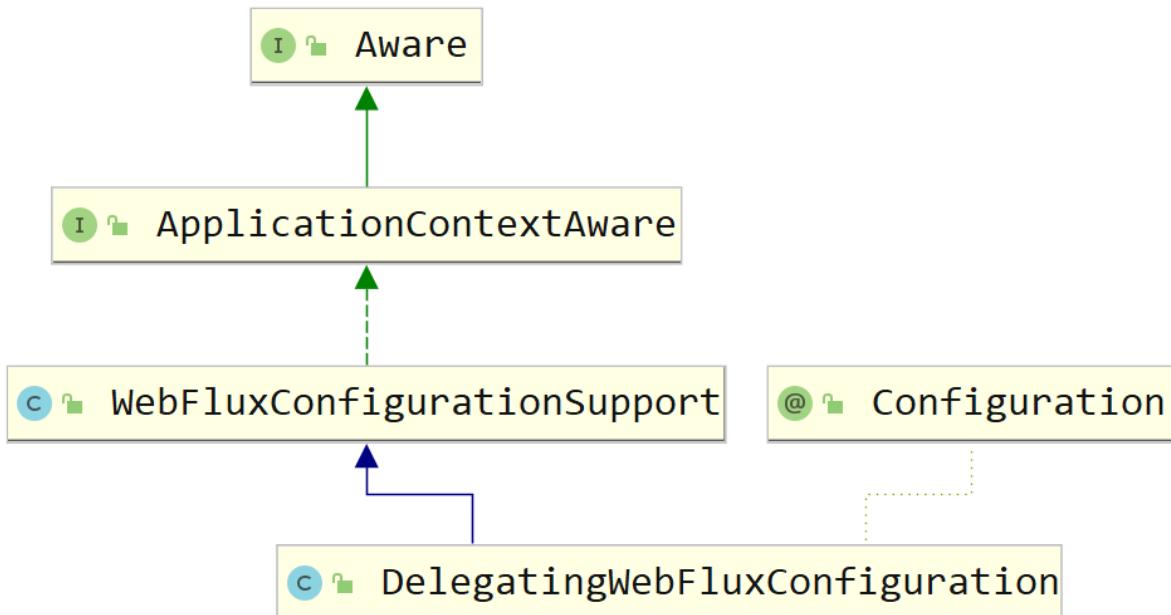
```

```
111     return resolver;
112 }
113 }
114 }
115 }
```

## 21 Spring WebFlux源码解析——config包

org.springframework.web.reactive.config

类图：



Spring WebFlux 配置基础架构。

将注解 @EnablewebFlux 添加到 @Configuration 类中，从 webFluxConfigurationSupport 导入 Spring Web Reactive 配置。

例如：

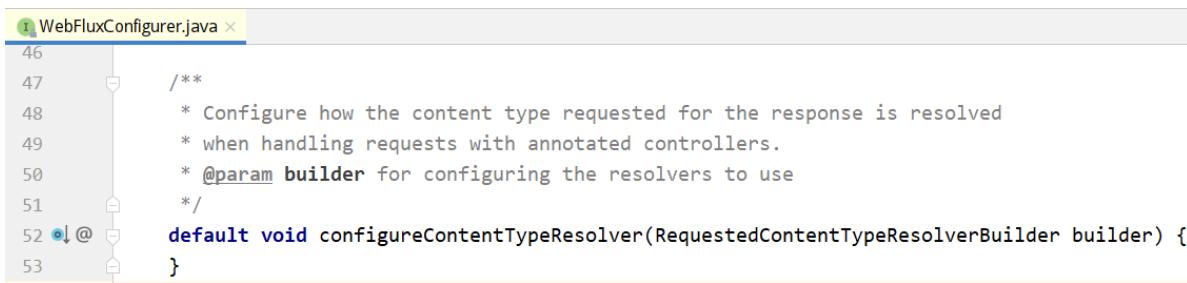
```
1 @Configuration
2 @EnablewebFlux
3 @ComponentScan(basePackageClasses = MyConfiguration.class)
4 public class MyConfiguration {
5
6 }
```

接口 `WebFluxConfigurer`

定义了回调方法来自定义通过 `@EnablewebFlux` 来启用WebFlux应用程序的配置。

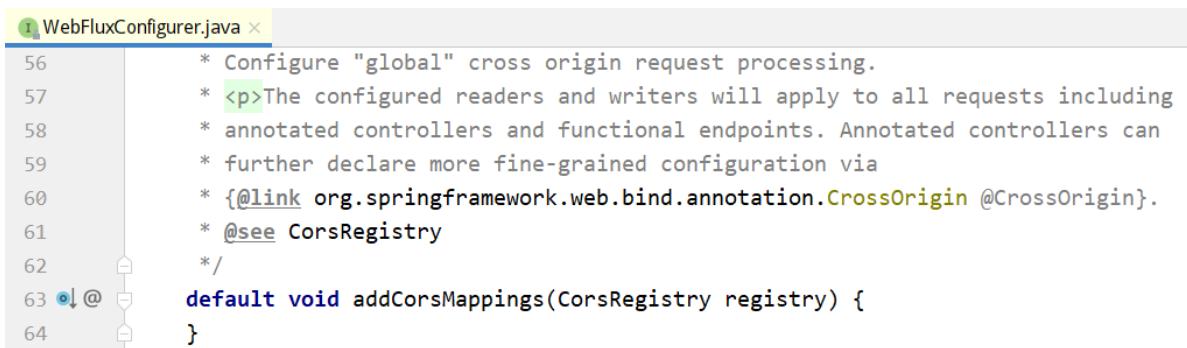
@EnableWebFlux 注释的配置类可以实现这个接口的调用，并提供一个定制默认配置的机会。

可以实现这个接口，并根据需要覆盖相关的方法。



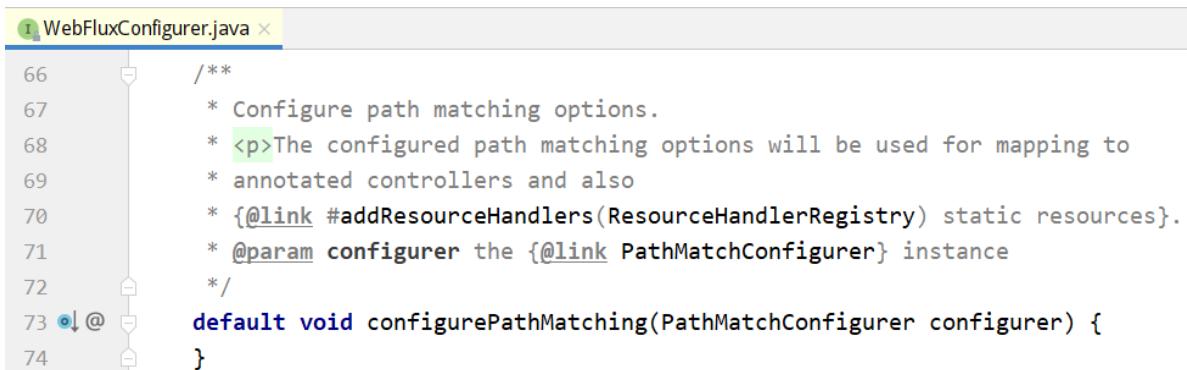
```
46
47     /**
48      * Configure how the content type requested for the response is resolved
49      * when handling requests with annotated controllers.
50      * @param builder for configuring the resolvers to use
51      */
52     default void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
53 }
```

配置如何解决响应请求的内容类型。



```
56     /**
57      * Configure "global" cross origin request processing.
58      * <p>The configured readers and writers will apply to all requests including
59      * annotated controllers and functional endpoints. Annotated controllers can
60      * further declare more fine-grained configuration via
61      * {@link org.springframework.web.bind.annotation.CrossOrigin @CrossOrigin}.
62      * @see CorsRegistry
63      */
64     default void addCorsMappings(CorsRegistry registry) {
65 }
```

配置全局的跨域请求处理。配置的reader和writer对所有的请求包括注解控制器和函数式端点都有效。注解控制器可以通过@CrossOrigin注解进一步进行配置。



```
66     /**
67      * Configure path matching options.
68      * <p>The configured path matching options will be used for mapping to
69      * annotated controllers and also
70      * {@link #addResourceHandlers(ResourceHandlerRegistry) static resources}.
71      * @param configurer the {@link PathMatchConfigurer} instance
72      */
73     default void configurePathMatching(PathMatchConfigurer configurer) {
74 }
```

配置路径匹配选项。HandlerMappings与路径匹配选项。

为静态资源和注解控制器配置路径匹配选项。

default void addResourceHandlers(ResourceHandlerRegistry registry)

为服务静态资源添加资源处理程序。

default void configureArgumentResolvers(ArgumentResolverConfigurer configurer)

配置自定义控制器方法参数的解析器。

```
default void configureHttpMessageCodecs(ServerCodecConfigurer configurer)
```

配置自定义HTTP消息读取器和编写器或重写内置的。

```
default void addFormatters(FormatterRegistry registry)
```

添加自定义转换器和格式化程序，用于执行控制器方法参数的类型转换和格式化。

```
@Nullable
```

```
default MessageCodesResolver getMessageCodesResolver()
```

供一个自定义MessageCodesResolver用于数据绑定，而不是DataBinder中默认创建的那个。

```
default void configureViewResolvers(ViewResolverRegistry registry)
```

配置视图解析以处理控制器方法的返回值，这些方法依赖于解析视图来呈现响应。

默认情况下，所有的控制器方法都依赖于视图解析，除非使用@ResponseBody或显式返回 ResponseEntity。

视图可以显式地指定为字符串返回值或隐式，例如void返回值。

## 类org.springframework.web.reactive.config.CorsRegistration

协助创建映射到路径模式的CorsConfiguration实例。

默认情况下，当最大时间设置为30分钟时，允许GET，HEAD和POST请求的所有来源，标题和凭证。

### 构造方法：

```
CorsRegistration(java.lang.String pathPattern)
```

创建一个新的CorsRegistration，允许指定路径的最大时间设置为1800秒（30分钟）的GET，HEAD和POST请求的所有来源，标题和凭证。

CORS配置应该适用的路径;支持精确的路径映射URI（如“/ admin”）以及Ant样式的路径模式（如“/ admin / \*\*”）。

### 方法：

- allowCredentials(boolean allowCredentials) 是否支持用户凭据。
- allowedHeaders(java.lang.String... headers) 设置请求可以在实际请求中使用的标题列表。如果它是一个：cache - control、content - language、Expires、last - modified或Pragma，根据CORS规范，则不需要列出header名称。  
默认情况下，所有标题都是允许的。
- allowedMethods(java.lang.String... methods) 设置允许的HTTP方法
- allowedOrigins(java.lang.String... origins) 设置允许的来源 特殊值“\*”允许所有域，默认情况下所有的来源都是允许的。
- exposedHeaders(java.lang.String... headers) 设置“简单”标题以外的响应标题列表
- getCorsConfiguration()

- getPathPattern()
- maxAge(long maxAge) 配置客户端可以缓存请求响应的时间，以秒为单位。

## org.springframework.web.reactive.config.CorsRegistry

CorsRegistry注册CorsConfiguration映射到路径模式。

为指定的路径模式启用跨域请求处理

### 方法:

- addMapping(java.lang.String pathPattern) 为指定的路径模式启用跨域请求处理。  
支持精确的路径映射URI（如“/ admin”）以及Ant样式的路径模式（如“/ admin / \*\*”）。  
默认情况下，允许所有来源，所有标题，凭据和GET，HEAD和POST方法，并且最大时间设置为30分钟。

```

1 private final List<CorsRegistration> registrations = new ArrayList<>();
2
3 public CorsRegistration addMapping(String pathPattern) {
4     CorsRegistration registration = new CorsRegistration(pathPattern);
5     this.registrations.add(registration);
6     return registration;
7 }
8
9 protected Map<String, CorsConfiguration> getCorsConfigurations() {
10     Map<String, CorsConfiguration> configs = new LinkedHashMap<>(
11         this.registrations.size());
12     for (CorsRegistration registration : this.registrations) {
13         configs.put(registration.getPathPattern(),
14             registration.getCorsConfiguration());
15     }
16     return configs;
17 }
```

## org.springframework.web.reactive.config.PathMatchConfigurer

配置HandlerMapping的路径匹配选项。

### 方法

- **isUseTrailingSlashMatch()** 是否与URL匹配，而不管是否存在尾部斜线。如果启用，映射到“/ users”的方法也匹配“/ users /”。默认值是true。
- setUseCaseSensitiveMatch(java.lang.Boolean caseSensitiveMatch) **是否与网址匹配，而不考虑其情况。**
- setUseTrailingSlashMatch(java.lang.Boolean trailingSlashMatch) 是否与URL匹配，而不管是否以斜杠结尾。

```

1  @Nullable
2  private Boolean trailingSlashMatch;
3
4  @Nullable
5  private Boolean caseSensitiveMatch;
6
7  public PathMatchConfigurer setUseCaseSensitiveMatch(Boolean
8  caseSensitiveMatch) {
9      this.caseSensitiveMatch = caseSensitiveMatch;
10     return this;
11 }
12
13 public PathMatchConfigurer setUseTrailingSlashMatch(Boolean
14 trailingSlashMatch) {
15     this.trailingSlashMatch = trailingSlashMatch;
16     return this;
17 }
18
19 @Nullable
20 protected Boolean isUseTrailingSlashMatch() {
21     return this.trailingSlashMatch;
22 }
23
24 @Nullable
25 protected Boolean isUseCaseSensitiveMatch() {
26     return this.caseSensitiveMatch;
27 }

```

## org.springframework.web.reactive.config.ResourceChainRegistration

完成资源和转换器的注册

### 方法

- addResolver(ResourceResolver resolver) 将一个资源解析器添加到链中。
- addTransformer(ResourceTransformer transformer) 向链中添加资源转换器器。

```

1  private static final String DEFAULT_CACHE_NAME = "spring-resource-chain-
2  cache";
3
4  private static final boolean isWebJarsAssetLocatorPresent =
5  ClassUtils.isPresent(
6      "org.webjars.WebJarAssetLocator",
7      ResourceChainRegistration.class.getClassLoader());
8
9  private final List<ResourceResolver> resolvers = new ArrayList<>(4);
10
11 private final List<ResourceTransformer> transformers = new ArrayList<>(4);
12
13 private boolean hasVersionResolver;
14
15 private boolean hasPathResolver;

```

```

13
14     private boolean hasCssLinkTransformer;
15
16     private boolean hasWebjarsResolver;
17
18     public ResourceChainRegistration(boolean cacheResources) {
19         this(cacheResources, cacheResources ? new
20             ConcurrentHashMapCache(DEFAULT_CACHE_NAME) : null);
21     }
22
23     public ResourceChainRegistration(boolean cacheResources, @Nullable Cache
24         cache) {
25         Assert.isTrue(!cacheResources || cache != null, "'cache' is required
26         when cacheResources=true");
27         if (cacheResources) {
28             this.resolvers.add(new CachingResourceResolver(cache));
29             this.transformers.add(new CachingResourceTransformer(cache));
30         }
31     }
32
33     public ResourceChainRegistration addResolver(ResourceResolver resolver) {
34         Assert.notNull(resolver, "The provided ResourceResolver should not be
35         null");
36         this.resolvers.add(resolver);
37         if (resolver instanceof VersionResourceResolver) {
38             this.hasVersionResolver = true;
39         }
40         else if (resolver instanceof PathResourceResolver) {
41             this.hasPathResolver = true;
42         }
43         else if (resolver instanceof WebJarsResourceResolver) {
44             this.hasWebjarsResolver = true;
45         }
46         return this;
47     }
48
49     public ResourceChainRegistration addTransformer(ResourceTransformer
50         transformer) {
51         Assert.notNull(transformer, "The provided ResourceTransformer should not
52         be null");
53         this.transformers.add(transformer);
54         if (transformer instanceof CssLinkResourceTransformer) {
55             this.hasCssLinkTransformer = true;
56         }
57         return this;
58     }
59
60     protected List<ResourceResolver> getResourceResolvers() {
61         if (!this.hasPathResolver) {
62             List<ResourceResolver> result = new ArrayList<>(this.resolvers);
63             if (isWebJarsAssetLocatorPresent && !this.hasWebjarsResolver) {
64                 result.add(new WebJarsResourceResolver());
65             }
66             result.add(new PathResourceResolver());
67             return result;
68         }
69         return this.resolvers;
70     }

```

```

65
66 protected List<ResourceTransformer> getResourceTransformers() {
67     if (this.hasVersionResolver && !this.hasCssLinkTransformer) {
68         List<ResourceTransformer> result = new ArrayList<>(
69             this.transformers);
70         boolean hasTransformers = !this.transformers.isEmpty();
71         boolean hasCaching = hasTransformers && this.transformers.get(0)
72             instanceof CachingResourceTransformer;
73         result.add(hasCaching ? 1 : 0, new CssLinkResourceTransformer());
74     }
75     return this.transformers;
76 }

```

## org.springframework.web.reactive.config.ResourceHandlerRegistry

通过Spring WebFlux存储资源处理程序的注册，以提供静态资源（如图像，css文件和其他），包括设置优化的高速缓存头，以便在Web浏览器中进行高效加载。资源可以从Web应用程序根目录下的位置，类路径和其他位置提供。

使用addResourceHandler(String ...) 提供应为其调用处理程序以提供静态资源（例如“/resources/\*\*”）的URL路径模式。

在返回的ResourceHandlerRegistration上使用其他方法来添加一个或多个从（例如“/”，“classpath： / META-INF / public-web-resources”）静态内容的位置，或者指定一个缓存期间服务的资源。

### 构造器

- ResourceHandlerRegistry(ResourceLoader resourceLoader)  
为给定资源加载器（通常是应用程序上下文）创建新的资源处理程序注册表。

### 方法

- addResourceHandler(java.lang.String... patterns) 添加一个资源处理程序，用于根据指定的URL路径模式提供静态资源。处理程序将针对每个与指定路径模式匹配的传入请求进行调用。像“/static/\*\*”或“/css/{filename}:w+.css”的模式是允许的。有关语法的更多详细信息，请参阅PathPattern。
- getHandlerMapping() 返回映射资源处理程序的处理程序映射；或者在没有注册的情况下为空。
- hasMappingForPattern(java.lang.String pathPattern) 资源处理程序是否已经注册了给定的路径模式。
- setOrder(int order) 指定相对于Spring配置中配置的其他HandlerMappings进行资源处理的顺序。使用的默认值是Integer.MAX\_VALUE-1。

```

1 private final ResourceLoader resourceLoader;
2
3 private final List<ResourceHandlerRegistration> registrations = new
ArrayList<>();

```

```

4
5     private int order = Integer.MAX_VALUE -1;
6
7     public ResourceHandlerRegistry(ResourceLoader resourceLoader) {
8         this.resourceLoader = resourceLoader;
9     }
10
11    public ResourceHandlerRegistration addResourceHandler(String... patterns) {
12        ResourceHandlerRegistration registration = new
13        ResourceHandlerRegistration(this.resourceLoader, patterns);
14        this.registrations.add(registration);
15        return registration;
16    }
17
18    public boolean hasMappingForPattern(String pathPattern) {
19        for (ResourceHandlerRegistration registration : this.registrations) {
20            if
21            (Arrays.asList(registration.getPathPatterns()).contains(pathPattern)) {
22                return true;
23            }
24        }
25
26    public ResourceHandlerRegistry setOrder(int order) {
27        this.order = order;
28        return this;
29    }
30
31 @Nullable
32 protected AbstractUrlHandlerMapping getHandlerMapping() {
33     if (this.registrations.isEmpty()) {
34         return null;
35     }
36     Map<String, WebHandler> urlMap = new LinkedHashMap<>();
37     for (ResourceHandlerRegistration registration : this.registrations) {
38         for (String pathPattern : registration.getPathPatterns()) {
39             ResourceWebHandler handler = registration.getRequestHandler();
40             try {
41                 handler.afterPropertiesSet();
42             }
43             catch (Throwable ex) {
44                 throw new BeanInitializationException("Failed to init
ResourceHttpRequestHandler", ex);
45             }
46             urlMap.put(pathPattern, handler);
47         }
48     }
49     SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();
50     handlerMapping.setOrder(this.order);
51     handlerMapping.setUrlMap(urlMap);
52     return handlerMapping;
53 }
```

创建和配置静态资源处理程序。

## 构造器

- ResourceHandlerRegistration(ResourceLoader resourceLoader, java.lang.String... pathPatterns)  
创建一个ResourceHandlerRegistration实例。

- resourceLoader : 用于将字符串位置转换为资源的资源加载器
- pathPatterns - 一个或多个资源URL路径模式

## 方法

- addResourceLocations(java.lang.String... resourceLocations)
  - 添加一个或多个资源位置来服务静态内容。每个位置必须指向一个有效的目录。多个位置可以指定为逗号分隔的列表，并且位置将按照指定的顺序对给定的资源进行检查。
  - 例如，`{/，“classpath： / META-INF / public-web-resources /”}`允许资源从Web应用程序的根目录和任何包含/ META-INF / public-web-resources /的目录，Web应用程序根目录中的资源优先。
- setCacheControl(CacheControl cacheControl) 指定应该由资源处理器使用的 CacheControl。

```
1 private final ResourceLoader resourceLoader;
2
3 private final String[] pathPatterns;
4
5 private final List<Resource> locations = new ArrayList<>();
6
7 @Nullable
8 private CacheControl cacheControl;
9
10 @Nullable
11 private ResourceChainRegistration resourceChainRegistration;
12
13 public ResourceHandlerRegistration(ResourceLoader resourceLoader, String...
14     pathPatterns) {
15     Assert.notNull(resourceLoader, "ResourceLoader is required");
16     Assert.notEmpty(pathPatterns, "At least one path pattern is required for
17     resource handling");
18     this.resourceLoader = resourceLoader;
19     this.pathPatterns = pathPatterns;
20 }
21
22 public ResourceHandlerRegistration addResourceLocations(String...
23     resourceLocations) {
24     for (String location : resourceLocations) {
25         this.locations.add(this.resourceLoader.getResource(location));
26     }
27     return this;
28 }
29
30 public ResourceHandlerRegistration setCacheControl(CacheControl
31     cacheControl) {
```

```

28     this.cacheControl = cacheControl;
29     return this;
30 }
31
32 public ResourceChainRegistration resourceChain(boolean cacheResources) {
33     this.resourceChainRegistration = new
ResourceChainRegistration(cacheResources);
34     return this.resourceChainRegistration;
35 }
36
37 public ResourceChainRegistration resourceChain(boolean cacheResources, Cache
cache) {
38     this.resourceChainRegistration = new
ResourceChainRegistration(cacheResources, cache);
39     return this.resourceChainRegistration;
40 }
41
42
43 protected String[] getPathPatterns() {
44     return this.pathPatterns;
45 }
46
47
48 protected ResourceWebHandler getRequestHandler() {
49     ResourcewebHandler handler = new ResourcewebHandler();
50     if (this.resourceChainRegistration != null) {
51
handler.setResourceResolvers(this.resourceChainRegistration.getResourceReso
lvers());
52
handler.setResourceTransformers(this.resourceChainRegistration.getResourceT
ransformers());
53     }
54     handler.setLocations(this.locations);
55     if (this.cacheControl != null) {
56         handler.setCacheControl(this.cacheControl);
57     }
58     return handler;
59 }

```

## org.springframework.web.reactive.config.UrlBasedViewResolverRegistration

配置UrlBasedViewResolver的属性。

### 方法

- prefix(java.lang.String prefix) 设置在构建URL时附加到**视图名称的前缀**。
- suffix(java.lang.String suffix) 设置在构建URL时附加到**视图名称的后缀**。
- viewClass(java.lang.Class<?> viewClass) 设置应该用于**创建视图的视图类**。
- viewNames(java.lang.String... viewNames) 设置可由该视图解析器处理的视图名称（或名称模式）。视图名称可以包含简单的通配符，这样'my\*', '\* Report'和'\*Repo\*' 将全部匹配视图名称'myReport'。

```

1 private final UrlBasedViewResolver viewResolver;
2
3 public UrlBasedViewResolverRegistration(UrlBasedViewResolver viewResolver) {
4     Assert.notNull(viewResolver, "viewResolver must not be null");
5     this.viewResolver = viewResolver;
6 }
7
8 public UrlBasedViewResolverRegistration prefix(String prefix) {
9     this.viewResolver.setPrefix(prefix);
10    return this;
11 }
12
13 public UrlBasedViewResolverRegistration suffix(String suffix) {
14     this.viewResolver.setSuffix(suffix);
15    return this;
16 }
17
18
19 public UrlBasedViewResolverRegistration viewClass(Class<?> viewClass) {
20     this.viewResolver.setViewClass(viewClass);
21    return this;
22 }
23
24
25 public UrlBasedViewResolverRegistration viewNames(String... viewNames) {
26     this.viewResolver.setViewNames(viewNames);
27    return this;
28 }
29
30 protected UrlBasedViewResolver getViewResolver() {
31    return this.viewResolver;
32 }

```

## org.springframework.web.reactive.config.ViewResolverRegistry

配置一个ViewResolver的链，支持不同的模板机制。

还可以根据所请求的内容类型配置defaultView以进行渲染，例如， JSON， XML等

### 方法

- defaultViews(View... defaultViews)

设置与任何视图名称关联的默认视图，并根据请求的内容类型的最佳匹配进行选择。使用HttpMessageWriterView来调整和使用任何现有的HttpMessageWriter（例如JSON， XML）作为一个视图。

- freeMarker()

注册一个带有.ftl后缀的FreeMarkerViewResolver。

- hasRegistrations()

是否有任何视图解析器已被注册。

- order(int order)

设置ViewResolutionResultHandler的顺序。默认情况下，此属性未设置，这意味着结果处理程序将按顺序排列。

- viewResolver(ViewResolver viewResolver)

注册一个ViewResolver bean实例。这可能有助于配置第三方解析器实现，或者在这个类中不公开一些需要设置的高级属性时，可以替代其他注册方法。

```
1  @Nullable
2  private final ApplicationContext applicationContext;
3
4  private final List<ViewResolver> viewResolvers = new ArrayList<>(4);
5
6  private final List<View> defaultViews = new ArrayList<>(4);
7
8
9  @Nullable
10 private Integer order;
11
12 public ViewResolverRegistry(@Nullable ApplicationContext applicationContext)
13 {
13     this.applicationContext = applicationContext;
14 }
15
16 public UrlBasedViewResolverRegistration freemarker() {
17     if (!checkBeanOfType(FreeMarkerConfigurer.class)) {
18         throw new BeanInitializationException("In addition to a FreeMarker
view resolver " +
19                                         "there must also be a single
FreeMarkerConfig bean in this web application context " +
20                                         "(or its parent):
FreeMarkerConfigurer is the usual implementation. " +
21                                         "This bean may be given any
name.");
22     }
23     FreeMarkerRegistration registration = new FreeMarkerRegistration();
24     UrlBasedViewResolver resolver = registration.getViewResolver();
25     if (this.applicationContext != null) {
26         resolver.setApplicationContext(this.applicationContext);
27     }
28     this.viewResolvers.add(resolver);
29     return registration;
30 }
31
32 public void viewResolver(ViewResolver viewResolver) {
33     this.viewResolvers.add(viewResolver);
34 }
35
36 public void defaultViews(View... defaultViews) {
37     this.defaultViews.addAll(Arrays.asList(defaultViews));
38 }
39
40
41 public boolean hasRegistrations() {
42     return (!this.viewResolvers.isEmpty());
43 }
```

```

44
45     public void order(int order) {
46         this.order = order;
47     }
48
49     private boolean checkBeanOfType(Class<?> beanType) {
50         return (this.applicationContext == null ||
51
52             !objectUtils.isEmpty(BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
53                 this.applicationContext, beanType, false, false)));
54     }
55
56     protected int getOrder() {
57         return (this.order != null ? this.order : ordered.LOWEST_PRECEDENCE);
58     }
59
60     protected List<ViewResolver> getViewResolvers() {
61         return this.viewResolvers;
62     }
63
64     protected List<View> getDefaultViews() {
65         return this.defaultviews;
66     }
67
68     private static class FreeMarkerRegistration extends
69         UrlBasedViewResolverRegistration {
70
71         public FreeMarkerRegistration() {
72             super(new FreeMarkerViewResolver());
73             getViewResolver().setSuffix(".ftl");
74         }
75     }

```

## org.springframework.web.reactive.config.WebFluxConfigurationSupport

### 所有实现的接口 ApplicationContextAware

Spring WebFlux配置的主要类。直接导入或扩展和重写受保护的方法来自定义配置。

```

1  @Nullable
2  private Map<String, CorsConfiguration> corsConfigurations;
3
4  @Nullable
5  private PathMatchConfigurer pathMatchConfigurer;
6
7  @Nullable
8  private ViewResolverRegistry viewResolverRegistry;
9
10 @Nullable
11 private ApplicationContext applicationContext;
12
13 @Override

```

```
14 public void setApplicationContext(@Nullable ApplicationContext
15     applicationContext) {
16     this.applicationContext = applicationContext;
17 }
18
19 @Nullable
20 public final ApplicationContext getApplicationContext() {
21     return this.applicationContext;
22 }
23
24 @Bean
25 public DispatcherHandler webHandler() {
26     return new DispatcherHandler();
27 }
28
29 @Bean
30 @Order(0)
31 public WebExceptionHandler responseStatusExceptionHandler() {
32     return new ResponseStatusExceptionHandler();
33 }
34
35 @Bean
36 public RequestMappingHandlerMapping requestMappingHandlerMapping() {
37     RequestMappingHandlerMapping mapping =
38         createRequestMappingHandlerMapping();
39     mapping.setOrder(0);
40     mapping.setContentTypeResolver(webFluxContentTypeResolver());
41     mapping.setCorsConfigurations(getCorsConfigurations());
42
43     PathMatchConfigurer configurer = getPathMatchConfigurer();
44     Boolean useTrailingSlashMatch = configurer.isUseTrailingSlashMatch();
45     Boolean useCaseSensitiveMatch = configurer.isUseCaseSensitiveMatch();
46     if (useTrailingSlashMatch != null) {
47         mapping.setUseTrailingSlashMatch(useTrailingSlashMatch);
48     }
49     if (useCaseSensitiveMatch != null) {
50         mapping.setUseCaseSensitiveMatch(useCaseSensitiveMatch);
51     }
52     return mapping;
53 }
54
55 protected RequestMappingHandlerMapping createRequestMappingHandlerMapping()
56 {
57     return new RequestMappingHandlerMapping();
58 }
59
60 @Bean
61 public RequestedContentTypeResolver webFluxContentTypeResolver() {
62     RequestedContentTypeResolverBuilder builder = new
63     RequestedContentTypeResolverBuilder();
64     configureContentTypeResolver(builder);
65     return builder.build();
66 }
67
68 protected void
69 configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
70 }
```

```
67
68
69 protected final Map<String, CorsConfiguration> getCorsConfigurations() {
70     if (this.corsConfigurations == null) {
71         CorsRegistry registry = new CorsRegistry();
72         addCorsMappings(registry);
73         this.corsConfigurations = registry.getCorsConfigurations();
74     }
75     return this.corsConfigurations;
76 }
77
78 protected void addCorsMappings(CorsRegistry registry) {
79 }
80
81 protected final PathMatchConfigurer getPathMatchConfigurer() {
82     if (this.pathMatchConfigurer == null) {
83         this.pathMatchConfigurer = new PathMatchConfigurer();
84         configurePathMatching(this.pathMatchConfigurer);
85     }
86     return this.pathMatchConfigurer;
87 }
88
89 public void configurePathMatching(PathMatchConfigurer configurer) {
90 }
91
92 @Bean
93 public RouterFunctionMapping routerFunctionMapping() {
94     RouterFunctionMapping mapping = createRouterFunctionMapping();
95     mapping.setOrder(-1); // go before RequestMappingHandlerMapping
96     mapping.setMessageReaders(serverCodecConfigurer().getReaders());
97     mapping.setCorsConfigurations(getCorsConfigurations());
98
99     return mapping;
100 }
101
102 @Bean
103 public HandlerMapping resourceHandlerMapping() {
104     ResourceLoader resourceLoader = this.applicationContext;
105     if (resourceLoader == null) {
106         resourceLoader = new DefaultResourceLoader();
107     }
108     ResourceHandlerRegistry registry = new
109     ResourceHandlerRegistry(resourceLoader);
110     addResourceHandlers(registry);
111
112     AbstractHandlerMapping handlerMapping = registry.getHandlerMapping();
113     if (handlerMapping != null) {
114         PathMatchConfigurer configurer = getPathMatchConfigurer();
115         Boolean useTrailingSlashMatch =
116             configurer.isUseTrailingSlashMatch();
117         Boolean useCaseSensitiveMatch =
118             configurer.isUseCaseSensitiveMatch();
119         if (useTrailingSlashMatch != null) {
120             handlerMapping.setUseTrailingSlashMatch(useTrailingSlashMatch);
121         }
122         if (useCaseSensitiveMatch != null) {
123             handlerMapping.setUseCaseSensitiveMatch(useCaseSensitiveMatch);
124         }
125     }
126 }
```

```
122     }
123     else {
124         handlerMapping = new EmptyHandlerMapping();
125     }
126     return handlerMapping;
127 }
128 ...
129 ...
130 ...
```

## org.springframework.web.reactive.config.WebFluxConfigurerComposite

### 所有实现的接口WebFluxConfigurer

代理一个或多个WebFluxConfigurer。

```
1 public class WebFluxConfigurerComposite implements WebFluxConfigurer {
2
3     private final List<WebFluxConfigurer> delegates = new ArrayList<>();
4
5     public void addWebFluxConfigurers(List<WebFluxConfigurer> configurers) {
6         if (!CollectionUtils.isEmpty(configurers)) {
7             this.delegates.addAll(configurers);
8         }
9     }
10
11     @Override
12     public void
13         configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
14         this.delegates.forEach(delegate ->
15             delegate.configureContentTypeResolver(builder));
16     }
17
18     @Override
19     public void addCorsMappings(CorsRegistry registry) {
20         this.delegates.forEach(delegate ->
21             delegate.addCorsMappings(registry));
22     }
23
24     @Override
25     public void configurePathMatching(PathMatchConfigurer configurer) {
26         this.delegates.forEach(delegate ->
27             delegate.configurePathMatching(configurer));
28     }
29
30     @Override
31     public void addResourceHandlers(ResourceHandlerRegistry registry) {
32         this.delegates.forEach(delegate ->
33             delegate.addResourceHandlers(registry));
34     }
35
36     @Override
```

```
32     public void configureArgumentResolvers(ArgumentResolverConfigurer
33         configurer) {
34         this.delegates.forEach(delegate ->
35             delegate.configureArgumentResolvers(configurer));
36     }
37
38     @Override
39     public void configureHttpMessageCodecs(ServerCodecConfigurer configurer)
40     {
41         this.delegates.forEach(delegate ->
42             delegate.configureHttpMessageCodecs(configurer));
43     }
44
45     @Override
46     public void addFormatters(FormatterRegistry registry) {
47         this.delegates.forEach(delegate ->
48             delegate.addFormatters(registry));
49     }
50
51     @Override
52     public Validator getValidator() {
53         return createSingleBean(WebFluxConfigurer::getValidator,
54             Validator.class);
55     }
56
57     @Override
58     public MessageCodesResolver getMessageCodesResolver() {
59         return createSingleBean(WebFluxConfigurer::getMessageCodesResolver,
60             MessageCodesResolver.class);
61     }
62
63     @Nullable
64     private <T> T createSingleBean(Function<WebFluxConfigurer, T> factory,
65         Class<T> beanType) {
66         List<T> result = this.delegates.stream().map(factory).filter(t -> t
67             != null).collect(Collectors.toList());
68         if (result.isEmpty()) {
69             return null;
70         }
71         else if (result.size() == 1) {
72             return result.get(0);
73         }
74         else {
75             throw new IllegalStateException("More than one WebFluxConfigurer
76 implements " +
77                 beanType.getSimpleName() + "
78             factory method.");
79         }
80     }
81 }
```

## org.springframework.web.reactive.config.DelegatingWebFluxConfiguration

- java.lang.Object
  - org.springframework.web.reactive.config.WebFluxConfigurationSupport
    - org.springframework.web.reactive.config.DelegatingWebFluxConfiguration

WebFluxConfigurationSupport的一个子类，用于检测并代理所有类型为WebFluxConfigurer的bean，允许它们自定义由WebFluxConfigurationSupport提供的配置。

这是由@EnableWebFlux实际导入的类。

```
1 @Configuration
2 public class DelegatingWebFluxConfiguration extends
3     WebFluxConfigurationSupport {
4
5     private final WebFluxConfigurerComposite configurers = new
6     WebFluxConfigurerComposite();
7
8     @Autowired(required = false)
9     public void setConfigurers(List<WebFluxConfigurer> configurers) {
10        if (!CollectionUtils.isEmpty(configurers)) {
11            this.configurers.addWebFluxConfigurers(configurers);
12        }
13    }
14
15    @Override
16    protected void
17    configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {
18        this.configurers.configureContentTypeResolver(builder);
19    }
20
21    @Override
22    protected void addCorsMappings(CorsRegistry registry) {
23        this.configurers.addCorsMappings(registry);
24    }
25
26    @Override
27    public void configurePathMatching(PathMatchConfigurer configurer) {
28        this.configurers.configurePathMatching(configurer);
29    }
30
31    @Override
32    protected void addResourceHandlers(ResourceHandlerRegistry registry) {
33        this.configurers.addResourceHandlers(registry);
34    }
35
36    @Override
37    protected void configureArgumentResolvers(ArgumentResolverConfigurer
38 configurer) {
39        this.configurers.configureArgumentResolvers(configurer);
40    }
```

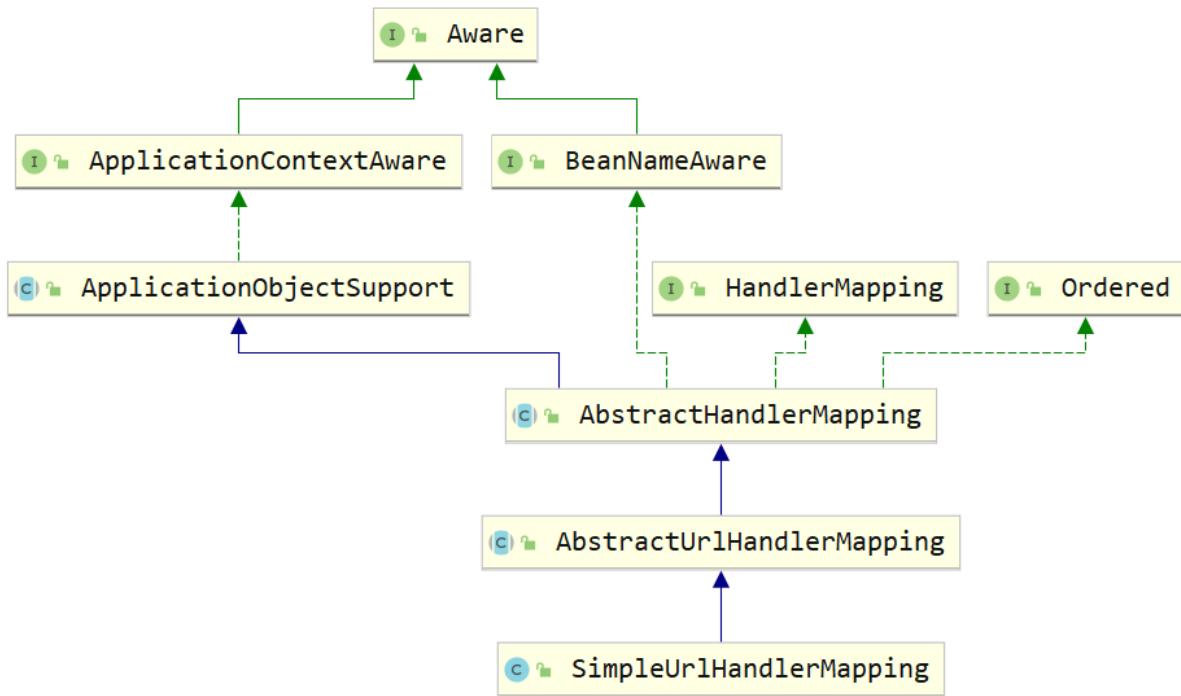
```
37
38     @Override
39     protected void configureHttpMessageCodecs(ServerCodecConfigurer
configurer) {
40         this.configurers.configureHttpMessageCodecs(configurer);
41     }
42
43     @Override
44     protected void addFormatters(FormatterRegistry registry) {
45         this.configurers.addFormatters(registry);
46     }
47
48     @Override
49     protected Validator getvalidator() {
50         Validator validator = this.configurers.getValidator();
51         return (validator != null ? validator : super.getValidator());
52     }
53
54     @Override
55     protected MessageCodesResolver getMessageCodesResolver() {
56         MessageCodesResolver messageCodesResolver =
this.configurers.getMessageCodesResolver();
57         return (messageCodesResolver != null ? messageCodesResolver :
super.getMessageCodesResolver());
58     }
59
60     @Override
61     protected void configureViewResolvers(ViewResolverRegistry registry) {
62         this.configurers.configureViewResolvers(registry);
63     }
64 }
```

## 22 Spring WebFlux源码解析——handler包

org.springframework.web.reactive.handler

### Spring Webflux --handler

提供包括抽象基类在内的HandlerMapping实现。



## AbstractHandlerMapping

HandlerMapping实现的抽象基类。

### 接口

- java.lang.Object
  - org.springframework.context.support.ApplicationObjectSupport
    - org.springframework.web.reactive.handler.AbstractHandlerMapping

实现了HandlerMapping, Ordered接口

```

1 | public abstract class AbstractHandlerMapping extends
2 |     ApplicationObjectSupport implements HandlerMapping, Ordered
  
```

```

1 | protected abstract Mono<?> getHandlerInternal(ServerWebExchange exchange);
  
```

查找给定请求的handler，如果找不到特定的请求，则返回一个空的Mono。这个方法被getHandler(org.springframework.web.server.ServerWebExchange)调用。

```

1  @Nullable
2  protected CorsConfiguration getCorsConfiguration(Object handler,
3      ServerWebExchange exchange) {
4      if (handler instanceof CorsConfigurationSource) {
5          return ((CorsConfigurationSource)
6              handler).getCorsConfiguration(exchange);
7      }
8      return null;
9  }

```

检索给定handle的CORS配置。

```

1  @Override
2  public Mono<Object> getHandler(ServerWebExchange exchange) {
3      return getHandlerInternal(exchange).map(handler -> {
4          if (CorsUtils.isCorsRequest(exchange.getRequest())) {
5              CorsConfiguration configA =
6                  this.globalCorsConfigSource.getCorsConfiguration(exchange);
7              CorsConfiguration configB = getCorsConfiguration(handler,
8                  exchange);
9              CorsConfiguration config = (configA != null ?
10                  configA.combine(configB) : configB);
11             if (!getCorsProcessor().process(config, exchange) ||
12                 Corsutils.isPreFlightRequest(exchange.getRequest())) {
13                 return REQUEST_HANDLED_HANDLER;
14             }
15         }
16     }
17     return handler;
18 });
19 }

```

抽象类实现的主要的具体方法,来获得具体的Handle, 实现了HandlerMapping中的getHandler,  
`Mono<Object> getHandler(ServerWebExchange exchange);`

## AbstractUrlHandlerMapping

**基于URL映射的HandlerMapping实现的抽象基类。**

### 接口

- `java.lang.Object`
  - `org.springframework.context.support.ApplicationObjectSupport`
    - `org.springframework.web.reactive.handler.AbstractHandlerMapping`
      - `org.springframework.web.reactive.handler.AbstractUrlHandlerMapping`

支持直接匹配, 例如注册的“/test”匹配“/test”, 以及各种ant样式匹配, 例如, “/test\*”匹配“/test”和“/team”, “/test/\*”匹配“/test”下的所有路径。

将搜索所有路径模式以查找当前请求路径的最具体匹配。

最具体的模式定义为**使用最少捕获变量和通配符的最长路径模式**。

```
1 | private final Map<PathPattern, Object> handlerMap = new LinkedHashMap<>();
```

返回注册路径模式和handler的只读视图。

注册路径模式和handler，可能是一个实际的handler实例或延迟初始化handler的bean名称。

```
1 | public final Map<PathPattern, Object> getHandlerMap() {  
2 |     return Collections.unmodifiableMap(this.handlerMap);  
3 | }
```

下面两个方法实现了handler的注册，会把所有的路径映射，和handler实例放在handlerMap中

```
1 | protected void registerHandler(String[] urlPaths, String beanName)  
2 |         throws BeansException, IllegalStateException {  
3 |  
4 |     Assert.notNull(urlPaths, "URL path array must not be null");  
5 |     for (String urlPath : urlPaths) {  
6 |         registerHandler(urlPath, beanName);  
7 |     }  
8 | }  
9 |  
10 |  
11 | protected void registerHandler(String urlPath, Object handler)  
12 |         throws BeansException, IllegalStateException {  
13 |  
14 |     Assert.notNull(urlPath, "URL path must not be null");  
15 |     Assert.notNull(handler, "Handler object must not be null");  
16 |     Object resolvedHandler = handler;  
17 |  
18 |     // Parse path pattern  
19 |     urlPath = prependLeadingslash(urlPath);  
20 |     PathPattern pattern = getPathPatternParser().parse(urlPath);  
21 |     if (this.handlerMap.containsKey(pattern)) {  
22 |         Object existingHandler = this.handlerMap.get(pattern);  
23 |         if (existingHandler != null) {  
24 |             if (existingHandler != resolvedHandler) {  
25 |                 throw new IllegalStateException(  
26 |                     "Cannot map " + getHandlerDescription(handler) + "  
27 | to [" + urlPath + "]: " +  
28 |                     "there is already " +  
29 |                     getHandlerDescription(existingHandler) + " mapped.");  
30 |             }  
31 |         }  
32 |     }  
33 |  
34 |     // Eagerly resolve handler if referencing singleton via name.  
35 |     if (!this.lazyInitHandlers && handler instanceof String) {
```

```

34     String handlerName = (String) handler;
35     if (obtainApplicationContext().isSingleton(handlerName)) {
36         resolvedHandler =
37             obtainApplicationContext().getBean(handlerName);
38     }
39
40     // Register resolved handler
41     this.handlerMap.put(pattern, resolvedHandler);
42     if (logger.isInfoEnabled()) {
43         logger.info("Mapped URL path [" + urlPath + "] onto " +
44             getHandlerDescription(handler));
45     }

```

预处理映射的路径，如果不以 / 开头就加上 /

```

1 private static String prependLeadingSlash(String pattern) {
2     if (StringUtils.hasLength(pattern) && !pattern.startsWith("/")) {
3         return "/" + pattern;
4     }
5     else {
6         return pattern;
7     }
8 }

```

在这一步的时候开始获取内部的handler，查找给定请求的handler，如果找不到就返回一个空的 Mono。

```

1 @Override
2 public Mono<Object> getHandlerInternal(ServerWebExchange exchange) {
3     PathContainer lookupPath =
4         exchange.getRequest().getPath().pathWithinApplication();
5     Object handler;
6     try {
7         handler = lookupHandler(lookupPath, exchange);
8     }
9     catch (Exception ex) {
10         return Mono.error(ex);
11     }
12
13     if (handler != null && logger.isDebugEnabled()) {
14         logger.debug("Mapping [" + lookupPath + "] to " + handler);
15     }
16     else if (handler == null && logger.isTraceEnabled()) {
17         logger.trace("No handler mapping found for [" + lookupPath + "]");
18     }
19
20     return Mono.justOrEmpty(handler);
21 }

```

获取到handler的类，先获取请求的url地址，调用 `lookupHandler(lookupPath, exchange)` 去找这个handler。

```
1  @Nullable
2  protected Object lookupHandler(PathContainer lookupPath, ServerWebExchange
3   exchange)
4   throws Exception {
5
6      return this.handlerMap.entrySet().stream()
7       .filter(entry -> entry.getKey().matches(lookupPath))
8       .sorted((entry1, entry2) ->
9          PathPattern.SPECIFICITY_COMPARATOR.compare(entry1.getKey(), entry2.getKey()))
10      .findFirst()
11      .map(entry -> {
12          PathPattern pattern = entry.getKey();
13          if (logger.isDebugEnabled()) {
14              logger.debug("Matching pattern for request [" + lookupPath + "] is " + pattern);
15          }
16          PathContainer pathwithinMapping =
17          pattern.extractPathwithinPattern(lookupPath);
18          return handleMatch(entry.getValue(), pattern, pathwithinMapping,
19          exchange);
20      })
21      .orElse(null);
22 }
```

调用 `handleMatch(entry.getValue(), pattern, pathwithinMapping, exchange)` 来匹配 handler，验证过后，设置到ServerWebExchange中最后返回。

```
1  private Object handleMatch(Object handler, PathPattern bestMatch,
2   PathContainer pathwithinMapping,
3   ServerWebExchange exchange) {
4
5     // Bean name or resolved handler?
6     if (handler instanceof String) {
7         String handlerName = (String) handler;
8         handler = obtainApplicationContext().getBean(handlerName);
9     }
10
11    validateHandler(handler, exchange);
12
13    exchange.getAttributes().put(BEST_MATCHING_HANDLER_ATTRIBUTE, handler);
14    exchange.getAttributes().put(BEST_MATCHING_PATTERN_ATTRIBUTE,
15    bestMatch);
16    exchange.getAttributes().put(PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE,
17    pathwithinMapping);
18
19    return handler;
20 }
```

## SimpleUrlHandlerMapping

HandlerMapping的实现，来把url请求映射到对应的 request handler 的bean

支持映射到bean实例和映射到bean名称；非单例的handler需要映射到bean名称。

“urlMap”属性适合用bean实例填充处理程序映射。可以通过java.util.Properties类接受的形式，通过“映射”属性设置映射到bean名称，如下所示：

```
/welcome.html=ticketController  
/show.html=ticketController
```

语法是PATH = HANDLER\_BEAN\_NAME。如果路径不是以斜杠开始的，则给它自动补充一个斜杠。

支持直接匹配，例如注册的“/ test”匹配“/ test”，以及各种ant样式匹配，例如，“/ test \*”匹配“/ test”和“/ team”，“/ test / \*”匹配“/ test”下的所有路径。

## 接口

- java.lang.Object
  - org.springframework.context.support.ApplicationObjectSupport
  - org.springframework.web.reactive.handler.AbstractHandlerMapping
    - org.springframework.web.reactive.handler.AbstractUrlHandlerMapping
    - org.springframework.web.reactive.handler.SimpleUrlHandlerMapping

```
1 private final Map<String, Object> urlMap = new LinkedHashMap<>();  
2  
3  
4 // 程序启动遍历的时候把加载到的所有映射路径，和handle设置到urlMap  
5 public void setUrlMap(Map<String, ?> urlMap) {  
6     this.urlMap.putAll(urlMap);  
7 }  
8  
9 // 获得所有的urlMap，允许urlMap访问URL路径映射，可以添加或覆盖特定条目。  
10 public Map<String, ?> getUrlMap() {  
11     return this.urlMap;  
12 }  
13  
14  
15 // 初始化程序上下文，除了父类的初始化，还调用了registerHandler  
16 @Override  
17 public void initApplicationContext() throws BeansException {  
18     super.initApplicationContext();  
19     registerHandlers(this.urlMap);  
20 }
```

开始注册handler，注册urlMap中为相应路径指定的所有的handler。

如果handler不能注册，抛出 BeansException

如果有注册的handler有冲突，比如两个相同的，抛出 `java.lang.IllegalStateException`

```
1 protected void registerHandlers(Map<String, Object> urlMap) throws
2 BeansException {
3     if (urlMap.isEmpty()) {
4         logger.warn("Neither 'urlMap' nor 'mappings' set on
5 simpleUrlHandlerMapping");
6     }
7     else {
8         for (Map.Entry<String, Object> entry : urlMap.entrySet()) {
9             String url = entry.getKey();
10            Object handler = entry.getValue();
11            // Prepend with slash if not already present.
12            if (!url.startsWith("/")) {
13                url = "/" + url;
14            }
15            // Remove whitespace from handler bean name.
16            if (handler instanceof String) {
17                handler = ((String) handler).trim();
18            }
19            registerHandler(url, handler);
20        }
21    }
22 }
```

调用的 `registerHandler(url, handler)` 就是刚刚抽象类 `AbstractUrlHandlerMapping` 中的 `registerHandler`方法。

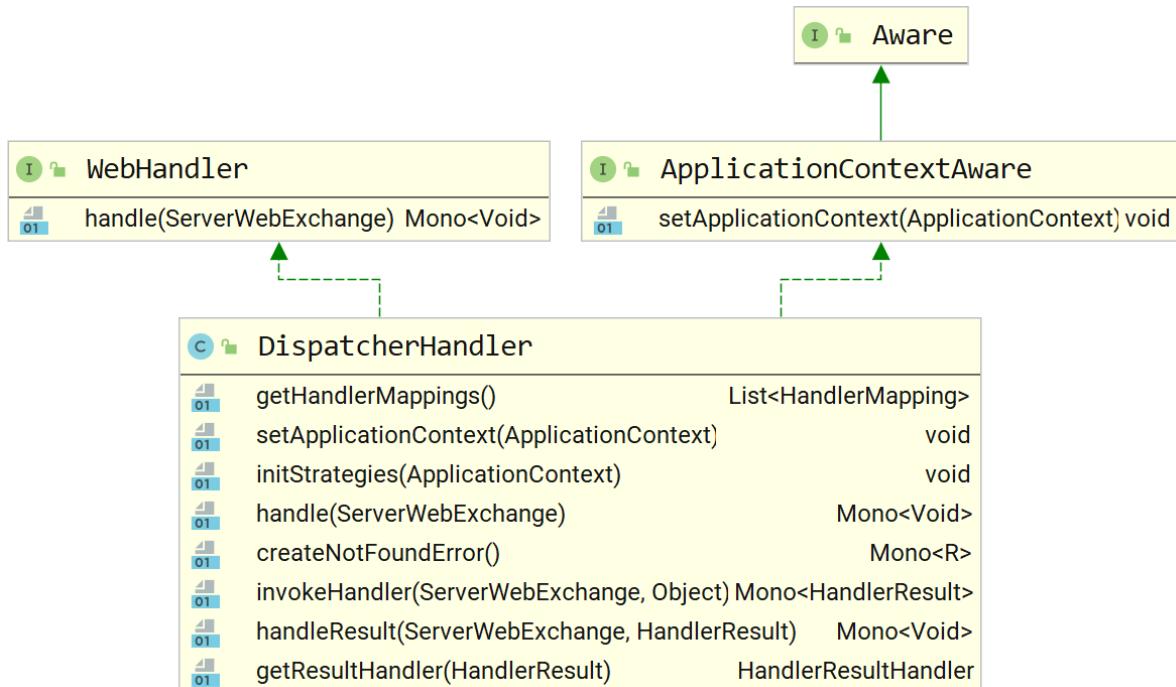
## 23 Spring WebFlux源码解析——reactive包

`org.springframework.web.reactive`

`Spring-webflux` 模块的顶级软件包包含 `DispatcherHandler`，它是WebFlux服务器端点处理的主要入口点，包括用于将请求映射到处理程序的关键协议，调用它们并处理结果。

该模块为响应式服务器端点提供两种编程模型。一个基于注释 `@Controller`，另一个基于功能路由和处理。

该模块还包含对响应式 `webclient` 以及客户端和服务器，响应式 `websocket` 支持。



### (Interface) HandlerAdapter

将 `DispatcherHandler` 与调用处理程序的细节解耦，以支持任何处理程序类型。

40



`boolean supports(Object handler);`

判断该 `HandlerAdapter` 是否支持给定的 `handler`。

58



`Mono<HandlerResult> handle(ServerWebExchange exchange, Object handler);`

调用给定的 `handler` 处理请求。

返回值：`Mono<HandlerResult>`，一个单独的 `HandlerResult` 或没有，如果请求已完全处理，不需要进一步处理。

### 所有已知的实现类：

`HandlerFunctionAdapter`

`RequestMappingHandlerAdapter`

`SimpleHandlerAdapter`

`WebSocketHandlerAdapter`

### HandlerMapping

#### 所有已知的实现类：

`AbstractHandlerMapping`

`AbstractHandlerMethodMapping`

```
AbstractUrlHandlerMapping  
RequestMappingHandlerMapping  
RequestMappingInfoHandlerMapping  
RouterFunctionMapping  
SimpleUrlHandlerMapping
```

由定义请求和处理程序对象之间的映射的对象实现的接口。

```
37 @ String BEST_MATCHING_HANDLER_ATTRIBUTE = HandlerMapping.class.getName() + ".bestMatchingHandler";
```

包含最佳匹配模式的映射处理程序的属性名称。

```
43 @ String BEST_MATCHING_PATTERN_ATTRIBUTE = HandlerMapping.class.getName() + ".bestMatchingPattern";
```

在处理程序映射中包含最佳匹配模式的属性的名称。

```
75 @ String MATRIX_VARIABLES_ATTRIBUTE = HandlerMapping.class.getName() + ".matrixVariables";
```

包含具有URI变量名称的映射的属性的名称以及每个URI变量的相应MultiValueMap。

```
54 @ String PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE = HandlerMapping.class.getName() + ".pathWithinHandlerMapping";
```

包含处理程序映射中的路径的属性的名称，在匹配的情况下，例如“/ static / \*\*”或完全相关的URI。

```
84 @ String PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE = HandlerMapping.class.getName() + ".producibleMediaTypes";
```

包含适用于映射处理程序的可生产的MediaType集合的属性的名称。

```
64 @ String URI_TEMPLATE_VARIABLES_ATTRIBUTE = HandlerMapping.class.getName() + ".uriTemplateVariables";
```

包含URI模板的属性的名称将变量名称映射到值。

```
93 ⏵ Mono<Object> getHandler(ServerWebExchange exchange);
```

返回此请求的处理程序(handler)。

返回 Mono<Object>：一个发出一个值的Mono，如果请求无法解析到一个处理程序，则没有。

## HandlerResultHandler

所有已知实现类：

```
ResponseBodyResultHandler  
ResponseEntityResultHandler
```

```
ServerResponseResultHandler
```

```
ViewResolutionResultHandler
```

处理 HandlerResult，通常由 HandlerAdapter 返回。

37

```
boolean supports(HandlerResult result);
```

该对象是否可以使用给定的 HandlerResult

46

```
Mono<Void> handleResult(ServerWebExchange exchange, HandlerResult result);
```

处理给定的结果修改响应标头，需要时将数据写入响应。

result - 处理的结果

Mono <Void> 来指示请求处理完成。

## Class

### BindingContext

将请求数据绑定到对象上，并提供对具有控制器特定属性的共享模型的访问的上下文。

提供为特定目标创建 webExchangeDataBinder 的方法，命令对象将数据绑定和验证应用，或者没有目标对象用于从请求值进行简单类型转换。用于请求的默认模型的容器。

60

61

62

```
public BindingContext() {  
    this(initializer: null);  
}
```

创建一个新的BindingContext。

68

69

70

```
public BindingContext(@Nullable WebBindingInitializer initializer) {  
    this.initializer = initializer;  
}
```

使用给定的初始化程序创建一个新的BindingContext。

initializer - 要应用的绑定初始化程序（可能为null）

76

77

78

```
public Model getModel() {  
    return this.model;  
}
```

返回默认的数据模型。

```
90     public WebExchangeDataBinder createDataBinder(ServerWebExchange exchange, @Nullable Object target, String name) {
91         WebExchangeDataBinder dataBinder = new ExtendedWebExchangeDataBinder(target, name);
92         if (this.initializer != null) {
93             this.initializer.initBinder(dataBinder);
94         }
95         return initDataBinder(dataBinder, exchange);
96     }
```

创建一个 `WebExchangeDataBinder`，以在目标命令对象上应用数据绑定和验证。

```
102     protected WebExchangeDataBinder initDataBinder(WebExchangeDataBinder binder, ServerWebExchange exchange) {
103         return binder;
104     }
```

初始化给定交换的数据绑定实例。

## HandlerResult

表示调用处理程序或处理程序方法的结果。

```
56     public HandlerResult(Object handler, @Nullable Object returnValue, MethodParameter returnType) {
57         this(handler, returnValue, returnType, context: null);
58     }
```

创建一个新的HandlerResult

- `handler` - 处理请求的handler对象
- `returnValue` - handler的返回值，可能是null值
- `returnType` - 返回值类型

```
82     public Object getHandler() {
83         return this.handler;
84     }
```

返回处理请求的处理程序。

```
89     @Nullable
90     public Object getReturnValue() {
91         return this.returnValue;
92     }
```

返回从处理程序返回的值（如果有）。

```
100     public ResolvableType getReturnType() {  
101         return this.returnType;  
102     }
```

返回处理器返回值的类型。

```
115     public BindingContext getBindingContext() {  
116         return this.bindingContext;  
117     }
```

返回用于请求处理的BindingContext。

### DispatcherHandler (程序入口，核心)

HTTP请求处理器/控制器的中央调度程序

将请求分发给处理器。

DispatcherHandler 从 Spring 配置中发现需要的代理组件。它在应用程序上下文中检测到以下内容：

- HandlerMapping -- 将请求映射到处理对象
- HandlerAdapter -- 适配相应的处理器
- HandlerResultHandler -- 处理处理器返回值

DispatcherHandler 也设计为一个Spring bean本身，并实现了 ApplicationContextAware 以访问其运行的上下文。

DispatcherHandler 被声明为bean的名称为“webHandler”，那么它将被 webHttpHandlerBuilder.applicationContext (org.springframework.context.ApplicationContext) 发现，与 WebFilter，WebExceptionHandler 等一起创建一个处理链。

@EnableWebFlux 配置中包含 DispatcherHandler bean 的声明。

### 构造器

DispatcherHandler()

创建一个新的 DispatcherHandler，通过 setApplicationContext(org.springframework.context.ApplicationContext) 配置一个 ApplicationContext。

```
DispatcherHandler(ApplicationContext applicationContext)
```

为给定的ApplicationContext创建一个新的DispatcherHandler。

```
@Nullable  
public final java.util.List getHandlerMappings()
```

返回在注入的上下文中类型检测到的所有 HandlerMapping bean，并排序。

注意：如果在 setApplicationContext (ApplicationContext) 之前调用，此方法可能返回 null。返回的是具有配置的映射的不可变列表或空值

```
public void setApplicationContext(ApplicationContext applicationContext)
```

设置此对象运行的ApplicationContext。通常，此调用将用于初始化对象。

在普通bean属性的采集之后但是在初始化回调之前调用，例如 InitializingBean.afterPropertiesSet() 或自定义init方法。在 ResourceLoaderAware.setResourceLoader (org.springframework.core.io.ResourceLoader) , ApplicationEventPublisherAware.setApplicationEventPublisher (org.springframework.context.ApplicationEventPublisher) 和MessageSourceAware (如果适用) 后调用。

覆盖了接口 ApplicationContextAware 中的 setApplicationContext

applicationContext - 该对象要使用的ApplicationContext对象

```
protected void initStrategies(ApplicationContext context)
```

## 初始化环境

```
public reactor.core.publisher.Mono<java.lang.Void> handle(ServerWebExchange  
exchange)
```

处理Web服务器交换。

在接口 WebHandler 中定义了 handle 方法，在这里来实现 Mono <Void> 来指示请求处理完实现：

```
1  @Nullable
2  private List<HandlerMapping> handlerMappings;
3
4  @Nullable
5  private List<HandlerAdapter> handlerAdapters;
6
7  @Nullable
8  private List<HandlerResultHandler> resultHandlers;
9
```

首先定义了可以为空的三个List列表，分别是 `handlerMappings` , `handlerAdapters` , `resultHandlers`

```
1  public DispatcherHandler(ApplicationContext applicationContext) {
2      initStrategies(applicationContext);
3  }
4
5  @Override
6  public void setApplicationContext(ApplicationContext applicationContext) {
7      initStrategies(applicationContext);
8  }
9
10
11 protected void initStrategies(ApplicationContext context) {
12     Map<String, HandlerMapping> mappingBeans =
13     BeanFactoryUtils.beansOfTypeIncludingAncestors(
14         context, HandlerMapping.class, true, false);
15
16     ArrayList<HandlerMapping> mappings = new ArrayList<>(
17         mappingBeans.values());
18     AnnotationAwareOrderComparator.sort(mappings);
19     this.handlerMappings = Collections.unmodifiableList(mappings);
20
21     Map<String, HandlerAdapter> adapterBeans =
22     BeanFactoryUtils.beansOfTypeIncludingAncestors(
23         context, HandlerAdapter.class, true, false);
24
25     this.handlerAdapters = new ArrayList<>(adapterBeans.values());
26     AnnotationAwareOrderComparator.sort(this.handlerAdapters);
27
28     Map<String, HandlerResultHandler> beans =
29     BeanFactoryUtils.beansOfTypeIncludingAncestors(
30         context, HandlerResultHandler.class, true, false);
31
32     this.resultHandlers = new ArrayList<>(beans.values());
33     AnnotationAwareOrderComparator.sort(this.resultHandlers);
34 }
```

在实例化DispatcherHandler之后，开始调用 `initStrategies (applicationContext)`，这个方法，`beansOfTypeIncludingAncestors` 这个方法来通过解析和反射等方法来收集 `context` 中所有的bean（可以通过注解或者xml配置Spring Bean，Springboot都是注解一般），来创建ioc容器，上面代码中的 `mappingBeans` `adapterBeans` `beans` 都是如此，创建好之后来让之前最初创建的三个List变量集合引用，并且对他们进行排序。

入口类的核心方法：

```
1  @Override
2  public Mono<Void> handle(ServerWebExchange exchange) {
3      if (logger.isDebugEnabled()) {
4          ServerHttpRequest request = exchange.getRequest();
5          logger.debug("Processing " + request.getMethodValue() + " request
for [" + request.getURI() + "]");
6      }
7      if (this.handlerMappings == null) {
8          return Mono.error(HANDLER_NOT_FOUND_EXCEPTION);
9      }
10     return Flux.fromIterable(this.handlerMappings)
11         .concatMap(mapping -> mapping.getHandler(exchange))
12         .next()
13         .switchIfEmpty(Mono.error(HANDLER_NOT_FOUND_EXCEPTION))
14         .flatMap(handler -> invokeHandler(exchange, handler))
15         .flatMap(result -> handleResult(exchange, result));
16 }
17
18
19 package org.springframework.web.server;
20
21 public interface WebHandler {
22
23     /**
24      * Handle the web server exchange.
25      * @param exchange the current server exchange
26      * @return {@code Mono<Void>} to indicate when request handling is
complete
27      */
28     Mono<Void> handle(ServerWebExchange exchange);
29 }
30 }
```

handle方法实现 package `org.springframework.web.server` 中的 `WebHandler` 中的接口，先获取请求中的 `request`，

接下来 `Flux` 开始遍历 `handlerMappings`，从 `ServerWebExchange` 中获得相应的 `Handle`，调用 `invokeHandler(exchange, handler)` 来匹配真正要使用的那个 `HandlerAdapter`，来执行 `handle` 方法处理请求逻辑，

`return handlerAdapter.handle(exchange, handler);`，这个过程中，别的类已经处理完了，并且返回了结果(还没看到哪些类)，返回一个 `Mono<HandlerResult>`，代码如下：

```
1 private Mono<HandlerResult> invokeHandler(ServerWebExchange exchange, Object
2   handler) {
3     if (this.handlerAdapters != null) {
4       for (HandlerAdapter handlerAdapter : this.handlerAdapters) {
5         if (handlerAdapter.supports(handler)) {
6           return handlerAdapter.handle(exchange, handler);
7         }
8       }
9     }
10    return Mono.error(new IllegalStateException("No HandlerAdapter: " +
11      handler));
12 }
```

最后就是调用 `handleResult(exchange, result)` 这个方法，代码如下：

```
1 private HandlerResultHandler getResultHandler(HandlerResult handlerResult) {
2   if (this.resultHandlers != null) {
3     for (HandlerResultHandler resultHandler : this.resultHandlers) {
4       if (resultHandler.supports(handlerResult)) {
5         return resultHandler;
6       }
7     }
8   }
9   throw new IllegalStateException("No HandlerResultHandler for " +
10    handlerResult.getReturnValue());
11 }
```

同理，也是要遍历来匹配真正的 `resultHandler`，然后获得这个，调用 `handlerResult.getReturnValue()`；来返回结果。