
Reactive Stream 编程

WebFlux

课程讲义

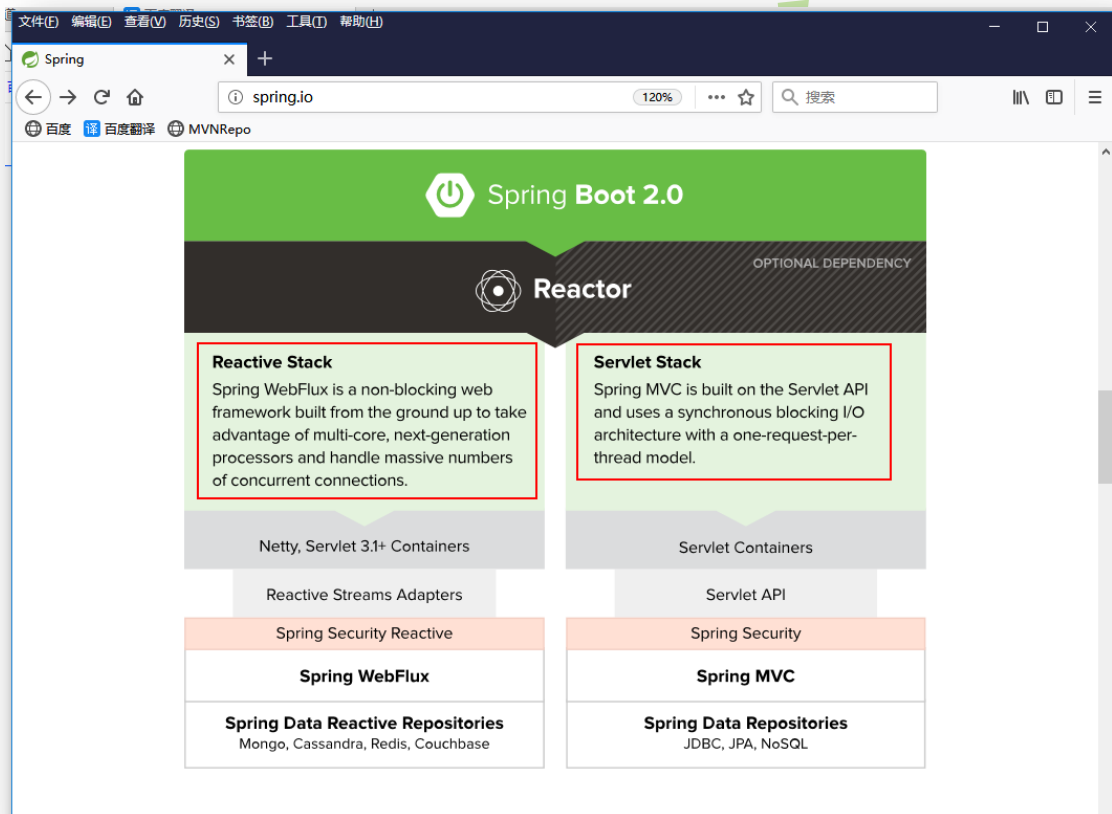
主讲: Reythor 雷

2019

Reactive Stream 编程 WebFlux

第1章 WebFlux 基础

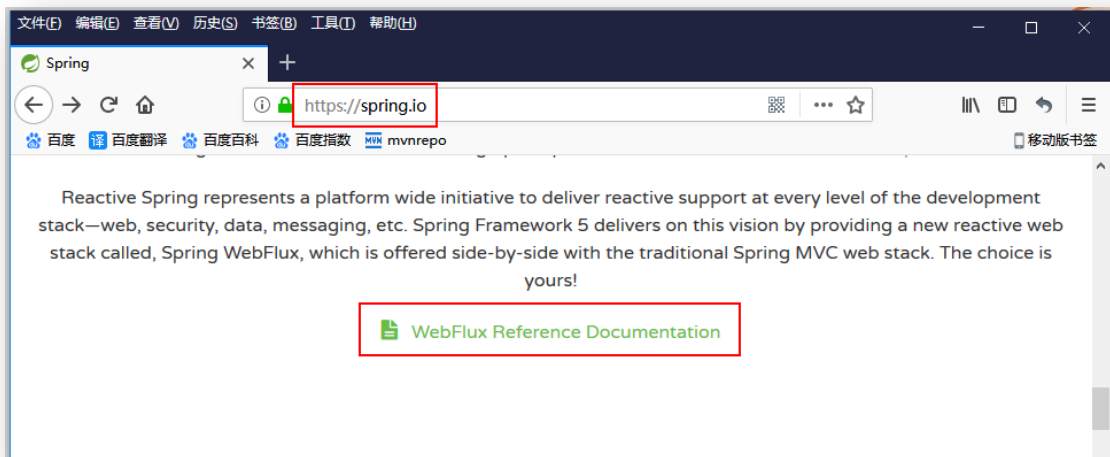
1.1 WebFlux 简介



【原文】Spring WebFlux is a non-blocking web framework built from **the ground up**(自底向上) to **take advantage of**(利用) **multi-core**(多核), **next-generation processors**(下一代处理器) and handle massive(大量) numbers of **concurrent connections**(并发连接).

【翻译】Spring WebFlux 是一个自底向上构建的非阻塞 Web 框架，用于利用多核、下一代处理器处理高并发连接。

打开 Spring 官网，可以看到 WebFlux 在线参考文档入口地址。



1. Spring WebFlux

The original web framework included in the Spring Framework, Spring Web MVC, was purpose-built for the Servlet API and Servlet containers. The reactive-stack web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports [Reactive Streams](#) back pressure, and runs on such servers as Netty, Undertow, and Servlet 3.1+ containers.

Both web frameworks mirror the names of their source modules ([spring-webmvc](#) and [spring-webflux](#)) and co-exist side by side in the Spring Framework. Each module is optional. Applications can use one or the other module or, in some cases, both — for example, Spring MVC controllers with the reactive [WebClient](#).

【原文】 The original(原始的) web framework included in the Spring Framework, Spring Web MVC, **was purpose built for**(专为***构建) the Servlet API and Servlet containers. The reactive-stack web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams(反应式流) back pressure(背压), and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers.

【翻译】 原始的 web 框架包含在 Spring 框架中，即 Spring Web MVC，是专为 Servlet API 与 Servlet 容器构建的。Reactive-stack web 框架,即 Spring WebFlux，最近被添加到了 Spring 5.0 版本中。它是完全地非阻塞的，支持 Reactive Streams 背压，运行在诸如 Netty、Undertow 与 Server3.1+容器中。

【原文】 Both web frameworks mirror(反映) the names of their source modules(源模块) [spring-webmvc](#) and [spring-webflux](#) and co-exist(共存) side by side(一起) in the Spring Framework. Each module is optional. Applications may use one or the other module, or in some cases both — e.g.(例如, 发音与意义与 For example 相同) Spring MVC controllers with the reactive [WebClient](#).

【翻译】 两个 Web 框架都反映了它们源模块的名称：[spring-webmvc](#) 与 [spring-webflux](#)，并且它们共存于 Spring 框架之中。每一个模块都是可选的。应用程序可以选择一个或另一个模块，

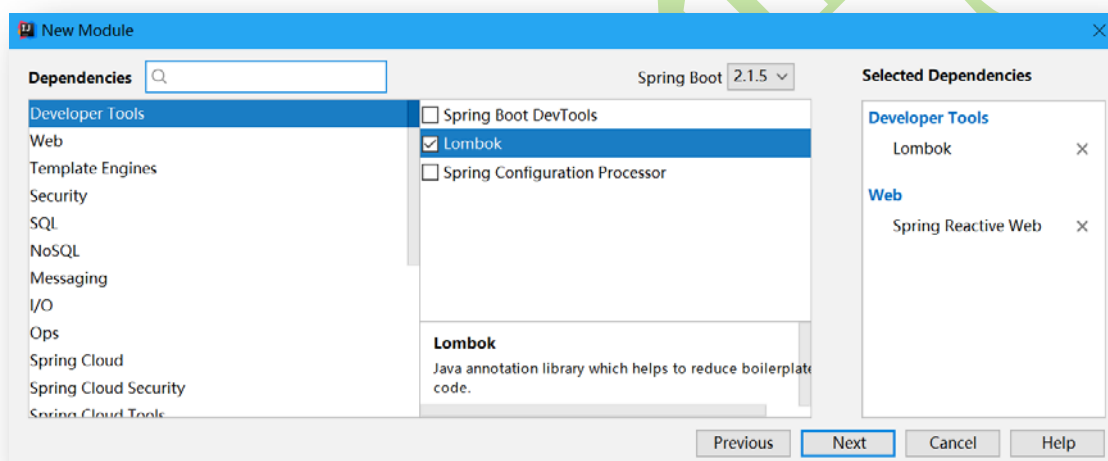
或者在某些情况下两个同时使用。例如，SpringMVC 使用反应式 Web 客户端进行控制。

1.2 WebFlux 牛刀小试

1.2.1 第一个 WebFlux 工程 02-firstwebflux

(1) 创建工程

创建一个 Spring Initializr 工程，Spring Boot 的版本要求最低为 2.0.0。不要添加原来的 web 依赖，而是要添加 Reactive Web，即 flux 依赖，并添加上 Lombok 依赖。命名为 02-firstwebflux。



(2) 定义处理器

为了对比效果，这里首先定义两个处理器方法：一个普通处理器方法，一个 Reactive 处理器方法。

```
@RestController
public class SomeController {

    @GetMapping("/common")
    public String commonHandle() {
        return "common handler";
    }

    @GetMapping("/mono")
    public Mono<String> monoHandle() {
        // Mono表示包含0或1个元素的异步序列
        // 静态方法just()可用于指定该异步序列中所包含的元素
        return Mono.just("mono handler");
    }
}
```

1.2.2 添加耗时操作 03-primary

(1) 创建工程

复制 02-firstwebflux 工程，并重命名为 03-primary。

(2) 定义耗时操作启用 **lombok** 日志

```
@Slf4j    // Lombok的日志
@RestController
public class SomeController {

    // 定义耗时操作
    private String doSome(String common) {
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return common;
    }
}
```

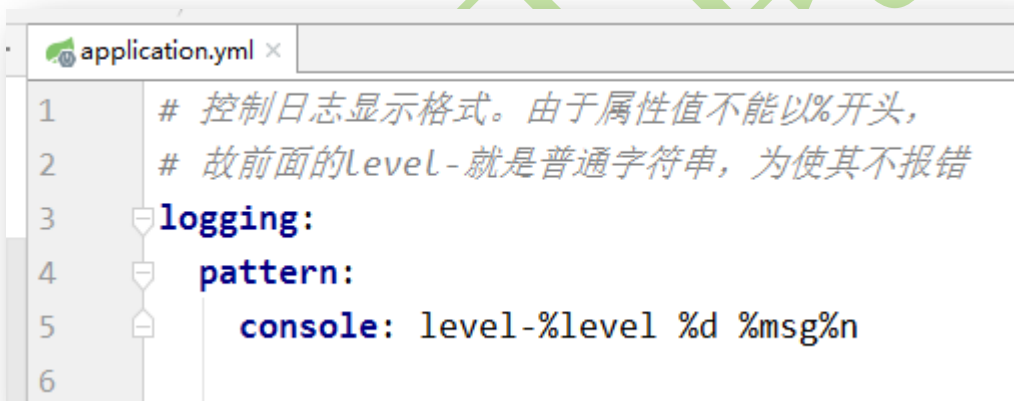
(3) 修改两个处理器方法

```
@GetMapping("/common")
public String commonHandle() {
    log.info("common--start");
    // 执行耗时操作
    String result = doSome("common-handler");
    log.info("common--end");
    return result;
}
```

```
@GetMapping("/mono")
public Mono<String> monoHandle() {
    Log.info("mono--start");
    // 执行耗时操作
    Mono<String> mono = Mono.fromSupplier(() -> doSome("mono-haneler"));
    Log.info("mono--end");
    return mono;
}
```

(4) 修改配置文件

为了便于观察日志输出，这里指定了日志的输出格式。



```
application.yml x
1      # 控制日志显示格式。由于属性值不能以%开头，
2      # 故前面的level-就是普通字符串，为使其不报错
3      logging:
4      pattern:
5          console: level-%level %d %msg%n
6
```

1.2.3 返回 Flux

Mono 表示包含 0 或 1 个元素的异步序列，Flux 则表示包含 0 或 N 个元素的异步序列。

(1) 直接指定 Flux 序列元素

在处理器中添加如下处理器方法。

```
@GetMapping("/flux")
public Flux<String> fluxHandle() {
    // 通过静态方法指定序列所包含的元素
    return Flux.just("reading", "swimming", "Fitness");
}
```

(2) 数组转 Flux

```
@GetMapping("/array")
public Flux<String> fluxHandle(@RequestParam String[] interests) {
    // 将数组转为Flux
    return Flux.fromArray(interests);
}
```

(3) 集合转 Flux

```
@GetMapping("/list")
public Flux<String> fluxHandle(@RequestParam List<String> interests) {
    // 将list转为Stream, 再将Stream转为Flux
    return Flux.fromStream(interests.stream());
}
```

(4) Flux 执行耗时操作

Stream 流中的每个元素将调用一次耗时操作 `doSome()`，即若 `interests` 集合中若存在三个元素，则其就会调用三次 `doSome()` 方法。


```
@GetMapping("/time")
public Flux<String> timeHandle(@RequestParam List<String> interests) {
    log.info("flux--start");
    // 将Flux的每个元素映射为一个doSome()耗时操作
    Flux<String> flux = Flux.fromStream(
        interests.stream().map(i -> doSome("elem-" + i))
    );
    log.info("flux--end");
    return flux;
}
```

(5) SSE

SSE, Server-Sent Event, 服务端推送事件。

```
@GetMapping(value = "/sse", produces = "text/event-stream")
public Flux<String> sseHandle() {
    // 通过静态方法指定序列所包含的元素
    return Flux.just("reading", "swimming", "Fitness");
}
```

1.3 SSE

反应式流编程中经常与 SSE 相结合使用，所以我们这里学习一下 SSE 相关知识。

1.3.1 SSE 简介

SSE (Server-Sent Event, 服务端推送事件), HTML5 规范中的一个组成部分, 一个子规范。由于这是官方特性, 主流浏览器对其支持是较好的 (除了火狐)。

SSE 与 WebSocket 对比:

WebSocket: 是双工通道。

SSE: 是单工通道, 只能是服务端向客户端发送消息。

1.3.2 SSE 技术规范

SSE 规范比较简单，主要由两个部分组成：

- 服务端与浏览器之间的通讯协议
- 浏览器中可供 JavaScript 使用的 EventSource 对象

(1) 通讯协议

这个通讯协议是基于**纯文本**的简单协议。服务器端的响应内容类型必须是“text/event-stream”。响应文本的内容是一个事件流，事件流是一个简单的文本流，仅支持 UTF-8 格式的编码。

事件流由不同的事件组成。不同事件间通过仅包含回车符和换行符的空行（“\r\n”）来分隔。

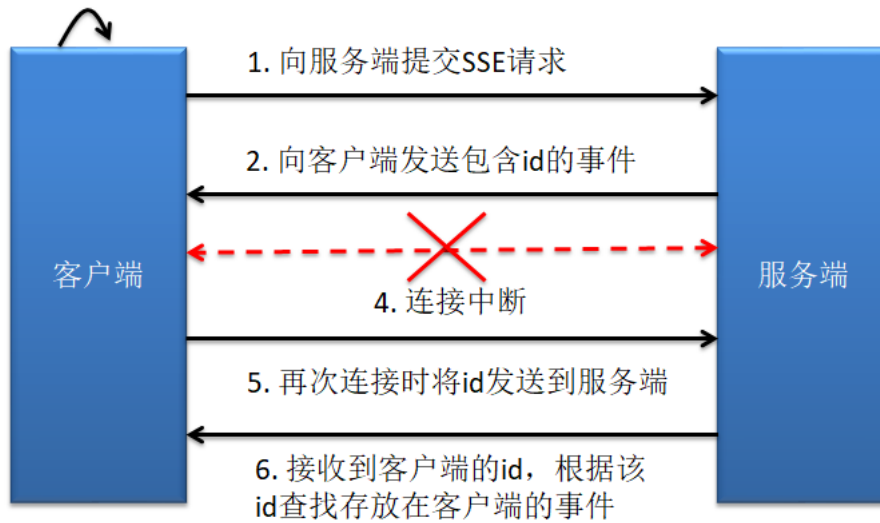
每个事件可以由多行构成，每行由类型和数据两部分组成。类型与数据通过冒号（“:”）进行分隔，冒号前的为类型，冒号后的为其对应的值。每个事件可以包含如下类型的行：

- 类型为 data，表示该行是事件所包含的数据。以 data 开头的行可以出现多次。所有这些行都是该事件的数据。
- 类型为 event，表示该行用来声明事件名称。浏览器在收到数据时，会产生对应名称的事件。
- 类型为空白，表示该行是注释，会在处理时被忽略。
- 类型为 retry，表示浏览器在连接断开之后进行重连的等待时间。
- 类型为 id，表示事件的标识符，标识符用于连接中断后的继连。

```
data: china           // 该事件仅包含数据
data: Beijing
data: haidian

: this is custom event // 注释
event: myevent        // 该事件指定了名称
data: shanghai
id: 101
retry: 3s
```

3. 将id存放在http头的
Last-Event-ID属性中



事件id的事件跟踪功能示意图

(2) EventSource 对象

对于服务端发送的带有事件的响应,浏览器需要在 JavaScript 中使用 EventSource 对象进行处理。EventSource 使用的是标准的事件监听器方式(注意,这里的事件并不是响应中所带的事件,而是浏览器上所发生的事件)。当相应的事件发生时,只需使 EventSource 对象调用相应的事件处理方法即可。EventSource 提供了三个标准事件。

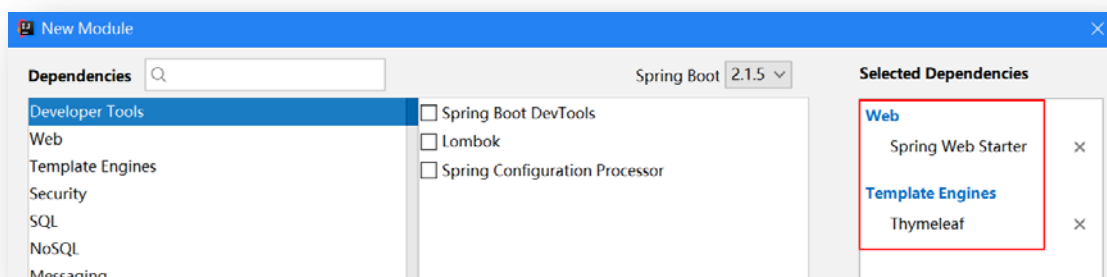
事件名称	事件解发条件	事件处理方法
open	当浏览器成功与服务端建立连接时触发	onopen()
message	当收到服务端发送的事件时触发	onmessage() addEventListener()
error	当发生异常时触发	onerror()

EventSource 提供的标准事件

1.3.3 SSE 举例 04-sse

(1) 创建工程

创建一个 Spring Boot 工程，导入 web 依赖与 Thymeleaf 依赖。



(2) 定义处理器

A、定义返回普通响应的处理器方法

首先定义一个向客户端返回普通响应的处理器，其目的主要是用于对比，对比一下使用 SSE 前后，浏览器的区别。

```
@Controller
public class SomeController {

    // 向客户端发送普通响应
    @RequestMapping("/common")
    public void commonHandle(HttpServletResponse response)
        throws IOException {

        PrintWriter out = response.getWriter();
        for(int i=0; i<10; i++) {
            out.print("data:" + i + "\n");
            out.print("\r\n");
            out.flush();

            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

B、定义返回默认 SSE 响应的处理器方法

```
// 向客户端发送默认事件的SSE响应
@RequestMapping("/sse/default")
public void defaultHandle(HttpServletResponse response)
    throws IOException {

    // 根据SSE规范进行设置
    response.setContentType("text/event-stream");
    response.setCharacterEncoding("UTF-8");

    // 以下代码未修改
    PrintWriter out = response.getWriter();
    for(int i=0; i<10; i++) {
        out.print("data:" + i + "\n");
        out.print("\r\n");
        out.flush();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

(3) 定义处理默认事件的客户端页面

A、定义 defaultsse.html

在 src/main/resources/templates 目录中定义 defaultsse.html 页面, 仅包含如下 JS 代码块。其演示了 EventSource 的 onmessage() 方法的用法。

```

1 <!DOCTYPE html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8">
5     <title>default</title>
6 </head>
7
8 <script type="text/javascript">
9     var es = new EventSource("sse/default");
10    es.onmessage = function (evt) {
11        console.log("my-msg", evt.data, evt);
12
13        if(evt.data == 9) {
14            es.close();
15        }
16    }
17 </script>
18
19 <body>
20 </body>
21 </html>

```

B、修改处理器

由于 Thymeleaf 页面不能通过浏览器访问，需要通过处理器来访问，所以在处理器中添加如下的处理器方法。

```

// 跳转到defaultsse.html页面
@RequestMapping("/default")
public String defaultSSEHandle() {
    return "/defaultsse";
}

```

(4) 定义返回自定义 SSE 响应的处理器方法

复制 defaultHandler(), 并重命名为 customHandle()。在其中仅添加了一行内容：指定事件名称。

```
// 向客户端发送自定义事件的SSE响应
@RequestMapping("/sse/custom")
public void customHandle(HttpServletResponse response)
    throws IOException {

    // 根据SSE规范进行设置
    response.setContentType("text/event-stream");
    response.setCharacterEncoding("UTF-8");

    PrintWriter out = response.getWriter();
    for(int i=0; i<10; i++) {
        // 设置自定义事件名称
        out.print("event:china\n");
        out.print("data:" + i + "\n");
        out.print("\r\n");
        out.flush();

        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```


(5) 定义处理自定义事件的客户端页面

A、定义 customsse.html

```

1  <!DOCTYPE html>
2  <html lang="en" xmlns:th="http://www.thymeleaf.org">
3  <head>
4      <meta charset="UTF-8">
5      <title>custom</title>
6  </head>
7
8  <script type="text/javascript">
9      var es = new EventSource("sse/custom");
10     es.addEventListener("china", function (evt) {
11         console.log("my-msg", evt.data);
12
13         if(evt.data == 9) {
14             es.close();
15         }
16     });
17 </script>
18
19 <body>
20 </body>
21 </html>
22

```

B、修改处理器

```

// 跳转到customsse.html页面
@RequestMapping("/custom")
public String customSSEHandle() {
    return "/customsse";
}

```

1.4 Reactive Stream

- Reactive Stream: 是反应式编程的规范。
- RxJava: 出现在 Reactive Stream 规范之前。
- RxJava2: 出现在 Reactive Stream 规范之后。
- Reactor: 完全基于 Reactive Stream 规范的

1.4.1 Reactive Stream 概述

(1) 推拉模型与发布/订阅模型

在流处理机制中发布/订阅模型可以分为 push（推送）模型和 pull（拉取）模型。push 模型中，发布者将元素主动推送给订阅者。而 pull 模式中，订阅者会向发布者主动索要。

(2) 异步系统与背压

在同步系统中发布者与订阅者的工作效率相当，发布者发布一个消息后阻塞，等待订阅者消费。订阅者消费完后，订阅者阻塞，等待发布者发布。这种同步式处理方式效率很低。

由于同步式处理方式效率很低，一般使用的是异步处理机制。即发布者发布消息，与消费者消费消息的速度是不一样的。那么它们间是如何协调工作的呢？有两种情况：

- 情况一：当订阅者消费速度比发布者发布速度快时，会出现订阅者无消息可消费的情况。
- 情况二：当发布者发布比订阅者消费快时，会出现消息堆积的情况。有两大类解决方案。
 - ◆ 改变订阅者。
 - ◆ 改变发布者。由订阅者控制发布者发布的速度。这种解决方案称为背压（Back Pressure）。使用背压策略可确保较快的发布者不会压制较慢的订阅者。

(3) 反应式流模型

反应式流从 2013 年开始，作为提供非阻塞背压的异步流处理标准的倡议，旨在解决处理元素流（即消息流、数据流）的问题——如何将元素流从发布者传递到订阅者，而不需要发布者阻塞，不需要订阅者有无边界缓冲区，不需要订阅者丢弃无法处理的元素。

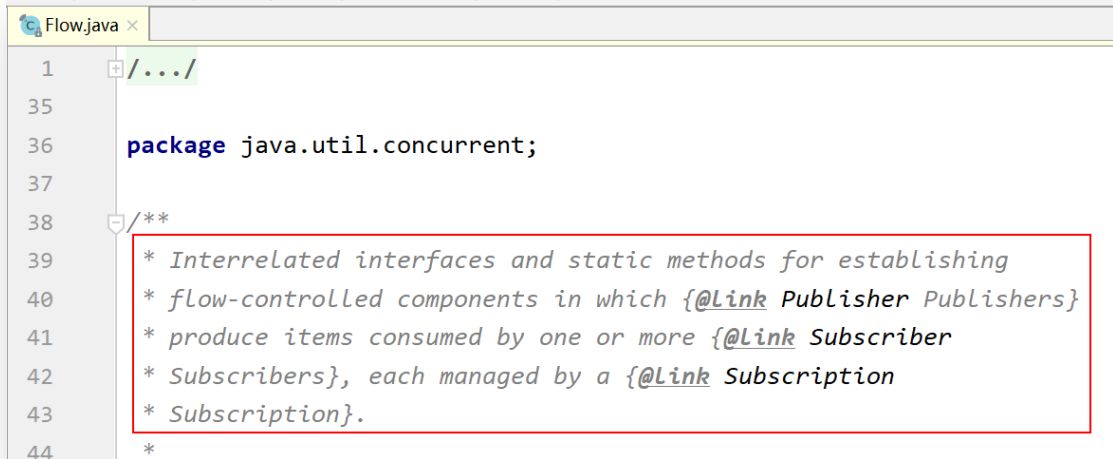
反应式流模型可以解决这个问题，该模型非常简单：订阅者向发布者发送异步请求，订阅 n 个元素；然后发布者向订阅者异步发送 n 个或少于 n 个的元素。反应式流会在 pull 模型和 push 模型流处理机制之间动态切换。当发布者快、订阅者慢时，它使用 pull 模型；当发布者慢、订阅者快时，它使用 push 模型。即谁慢谁占主动。

2015 年发布了用于处理反应式流的规范和 Java API。

1.4.2 反应式流规范

在 Java 中反应式流规范，是通过 JDK 的 `java.util.concurrent.Flow` 类中声明的四个内部接口来定义的。这套规范最初是定义在 JDK9 中的。

(1) Flow 类



【原文】Interrelated（相关联的） interfaces and static methods for establishing（创建） flow-controlled components in which Publishers produce items consumed by one or more Subscribers, each managed by a Subscription.

【翻译】（这个类用于）创建流控制组件的相关的接口与方法，其中 Publishers 生成由一个或多个 Subscriber 消费的 items，每个 items 由 Subscription 管理。

(2) Publisher<T>接口

```
public final class Flow {

    private Flow() {} // uninstantiable

    /**
     * A producer of items (and related control messages) received by
     * Subscribers. Each current {@link Subscriber} receives the same
     * items (via method {@code onNext}) in the same order, unless
     * drops or errors are encountered. If a Publisher encounters an
     * error that does not allow items to be issued to a Subscriber,
     * that Subscriber receives {@code onError}, and then receives no
     * further messages. Otherwise, when it is known that no further
     * messages will be issued to it, a subscriber receives {@code
     * onComplete}. Publishers ensure that Subscriber method
     * invocations for each subscription are strictly ordered in <i>happens-before</i>
     * order.
     *
     * 

Publishers may vary in policy about whether drops (failures
     * to issue an item because of resource limitations) are treated
     * as unrecoverable errors. Publishers may also vary about
     * whether Subscribers receive items that were produced or
     * available before they subscribed.


     *
     * @param <T> the published item type
     */
    @FunctionalInterface
    public static interface Publisher<T> {
```

该段注释不仅说明什么是 Publisher，还说明了其与 Subscriber 的关系，确切地说，是 Subscriber 接口中的方法都会在什么时候被触发。

【原文】A producer of items (and related control messages) received by Subscribers. Each current Subscriber receives the same items (via method *onNext*) in the same order, unless drops or errors are encountered. If a Publisher encounters an error that does not allow items to be issued to a Subscriber, that Subscriber receives *onError*, and then receives no further messages. Otherwise, when it is known that no further messages will be issued to it, a subscriber receives *onComplete*. Publishers ensure that Subscriber method invocations for each subscription are strictly ordered in order.

【翻译】（这是一个）被订阅者接收的 items（与相关控制信息）的生产者。每个当前的订阅者以相同的顺序接受相同的 items（通过 *onNext()* 方法），除非 items 被删除或发生了错误。

如果一个发布者发生了“不允许 items 发布给订阅者”的错误，那么订阅者将触发 `onError()` 方法，并且不再接受消息。否则（发布者没有发生错误），当发布者没有消息再发布给订阅者时，订阅者将触发 `onComplete()` 方法。发布者可以确保每一个订阅的订阅者方法调用都严格按照顺序进行。

A、subscribe()

```
public static interface Publisher<T> {
    /**
     * Adds the given Subscriber if possible. If already
     * subscribed, or the attempt to subscribe fails due to policy
     * violations or errors, the Subscriber's {@code onError}
     * method is invoked with an {@link IllegalStateException}.
     * Otherwise, the Subscriber's {@code onSubscribe} method is
     * invoked with a new {@link Subscription}. Subscribers may
     * enable receiving items by invoking the {@code request}
     * method of this Subscription, and may unsubscribe by
     * invoking its {@code cancel} method.
     *
     * @param subscriber the subscriber
     * @throws NullPointerException if subscriber is null
     */
    public void subscribe(Subscriber<? super T> subscriber);
}
```

这是 `publicsher` 接口的唯一的方法，用于添加给定的订阅者。即建立订阅者与生产者间的订阅关系。

【原文】Adds the given Subscriber if possible. If already subscribed, or the attempt to subscribe fails due to policy violations or errors, the Subscriber's `{@code onError}` method is invoked with an `{@link IllegalStateException}`. Otherwise, the Subscriber's `{@code onSubscribe}` method is invoked with a new `{@link Subscription}`. Subscribers may enable receiving items by invoking the `{@code request}` method of this Subscription, and may unsubscribe by invoking its `{@code cancel}` method.

【翻译】如果可能，添加给定的订阅者。如果已订阅，或者由于策略违反或错误而订阅失败，则使用 `IllegalStateException` 调用订阅者的 `onError()` 方法。否则（订阅成功），订阅者的 `onSubscribe()` 方法伴随着新的订阅关系而被调用。订阅者可以通过调用此订阅令牌的 `request()` 方法来接收 item，也可以通过调用其 `cancel()` 方法来取消订阅关系。

(3) Subscriber<T>接口

```
/**
 * A receiver of messages. The methods in this interface are
 * invoked in strict sequential order for each {@link
 * Subscription}.
 *
 * @param <T> the subscribed item type
 */
public static interface Subscriber<T> {
    /**
```

【原文】A receiver of messages. The methods in this interface are invoked in strict sequential order for each Subscription.

【翻译】（这是一个）消息的接收者。对于每个订阅关系，接口的方法将严格按照串行顺序被调用。

A、onSubscribe()

```
public static interface Subscriber<T> {
    /**
     * Method invoked prior to invoking any other Subscriber
     * methods for the given Subscription. If this method throws
     * an exception, resulting behavior is not guaranteed, but may
     * cause the Subscription not to be established or to be cancelled.
     *
     * <p>Typically, implementations of this method invoke {@code
     * subscription.request} to enable receiving items.
     *
     * @param subscription a new subscription
     */
    public void onSubscribe(Subscription subscription);
```

【原文】Method invoked **prior to**（先于，在先） invoking any other Subscriber methods for the given Subscription.

【翻译】（这是一个）对于给定订阅关系的订阅者的其它方法调用之前先被调用的方法。

该方法是由 Publisher 的 subscribe()方法触发的，即订阅关系一旦确立，就会触发该方法的执行。

B、onNext()

```
/**
 * Method invoked with a Subscription's next item. If this
 * method throws an exception, resulting behavior is not
 * guaranteed, but may cause the Subscription to be cancelled.
 *
 * @param item the item
 */
public void onNext(T item);
```

【原文】Method invoked with a Subscription's next item. If this method throws an exception, resulting behavior is not guaranteed, but may cause the Subscription to be cancelled.

【翻译】（这是一个）调用订阅令牌的下一个 item 的方法。如果该方法抛出异常，结果行为将不能被保证，但可能引起订阅关系的取消。

onNext()方法是上一个 onNext()执行完毕后触发的。

C、onError()

```
/**
 * Method invoked upon an unrecoverable error encountered by a
 * Publisher or Subscription, after which no other Subscriber
 * methods are invoked by the Subscription. If this method
 * itself throws an exception, resulting behavior is
 * undefined.
 *
 * @param throwable the exception
 */
public void onError(Throwable throwable);
```

【原文】Method invoked upon an unrecoverable error encountered by a Publisher or Subscription, after which no other Subscriber methods are invoked by the Subscription. If this

method itself throws an exception, resulting behavior is undefined.

【翻译】（这是一个）当发布者或订阅令牌发生不可恢复的错误时调用的方法，在该错误之后，订阅令牌不会再调用其他订阅者方法。如果该方法本身发生异常，结果行为未定义。

D、onComplete()

```
/**
 * Method invoked when it is known that no additional
 * Subscriber method invocations will occur for a Subscription
 * that is not already terminated by error, after which no
 * other Subscriber methods are invoked by the Subscription.
 * If this method throws an exception, resulting behavior is
 * undefined.
 */
public void onComplete();
```

【原文】Method invoked when it is known that no additional Subscriber method invocations will occur for a Subscription that is not already terminated by error, after which no other Subscriber methods are invoked by the Subscription. If this method throws an exception, resulting behavior is undefined.

【翻译】对于不是被错误终止的订阅关系，当订阅者知道没有其它订阅者方法调用再发生时，该方法被调用。对于该订阅关系，该方法之后将不再有其它订阅者方法被调用。如果该方法本身发生异常，结果行为未定义。

当 Publisher 的 close()方法执行完毕，会触发该方法的执行。

(4) Subscription 接口

```
/**
 * Message control linking a {@link Publisher} and {@link
 * Subscriber}. Subscribers receive items only when requested,
 * and may cancel at any time. The methods in this interface are
 * intended to be invoked only by their Subscribers; usages in
 * other contexts have undefined effects.
 */
public static interface Subscription {
    /**
```

【原文】 Message control linking a Publisher and Subscriber. Subscribers receive items only when requested, and may cancel at any time. The methods in this interface are intended to be invoked only by their Subscribers; usages in other contexts have undefined effects.

【翻译】(这是一个)连接发布者与订阅者的消息控制器。只有当请求时订阅者才可接收 items, 并且可以在任何时间取消。这个接口中的方法只能被它的订阅者调用。在其它上下文环境中的用法没有定义其效果 (即在其它类中调用该接口对象的方法最终的执行效果未定义)。

A、request()

```
/**
 * Adds the given number {@code n} of items to the current
 * unfulfilled demand for this subscription. If {@code n} is
 * less than or equal to zero, the Subscriber will receive an
 * {@code onError} signal with an {@link
 * IllegalArgumentException} argument. Otherwise, the
 * Subscriber will receive up to {@code n} additional {@code
 * onNext} invocations (or fewer if terminated).
 *
 * @param n the increment of demand; a value of {@code
 * Long.MAX_VALUE} may be considered as effectively unbounded
 */
public void request(long n);
```

【原文】 Adds the given number {@code n} of items to the current unfulfilled demand for this subscription. If {@code n} is less than or equal to zero, the Subscriber will receive an {@code onError} signal with an {@link IllegalArgumentException} argument. Otherwise, the Subscriber will receive up to {@code n} additional {@code onNext} invocations (or fewer if terminated).

【翻译】（这个方法用于）给未满足需求的订阅令牌添加指定数量 n 的 items。若 n 小于等于 0，订阅者将触发 onError()方法，并被标记为 IllegalArgumentException 异常。否则（即 n 大于 0），订阅者将触发 n 次 onNext()调用（或更少，如果终止）。

B、cancel()

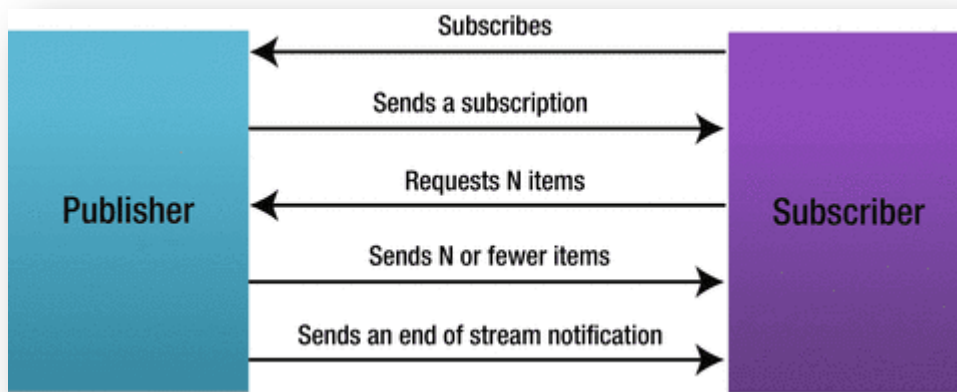
```
/**
 * Causes the Subscriber to (eventually) stop receiving
 * messages. Implementation is best-effort -- additional
 * messages may be received after invoking this method.
 * A cancelled subscription need not ever receive an
 * {@code onComplete} or {@code onError} signal.
 */
public void cancel();
```

【原文】 Causes the Subscriber to (eventually) stop receiving messages. Implementation is best-effort -- additional messages may be received after invoking this method. A cancelled subscription need not ever receive an {@code onComplete} or {@code onError} signal.

【翻译】（这个方法会）导致订阅者最终停止接收消息。实现会尽力而为（尽力不再接收其它消息）——该方法被调用后可能还会收到其它消息。一个被取消的订阅令牌不再需要接收 onComplete()或 onError()的信号。

（5）三个接口的关系

三个接口的关系如下图所示，但真正细节上的关系，即这些接口中包含方法的执行关系，要远比这个图复杂。具体见后面代码的演示。



（6）Processor<T, R>接口

Processor，即处理器，充当订阅者和发布者的处理阶段。Processor 接口继承了 Publisher 和 Subscriber 接口，对于发布者来说，Processor 是订阅者，对于订阅者来说，Processor 是发布者。

Processor 用于转换发布者/订阅者管道中的元素。Processor<T, R>会将来自于发布者的 T 类型的消息数据，接收并转换为 R 类型的数据，并将转换后的 R 类型数据发布给订阅者。一个发布者可以拥有多个处理器。



1.4.3 发布者类 SubmissionPublisher

通常情况下，我们会使用 JDK 中已经定义好的一个发布者类 SubmissionPublisher，该类就可以完成一个简单的消息生成与发布。从该类的注释第一段中可以了解到其简介。

```

51 /**
52  * A {@link Flow.Publisher} that asynchronously issues submitted
53  * (non-null) items to current subscribers until it is closed. Each
54  * current subscriber receives newly submitted items in the same order
55  * unless drops or exceptions are encountered. Using a
56  * SubmissionPublisher allows item generators to act as compliant <a
57  * href="http://www.reactive-streams.org/">reactive-streams</a>
58  * Publishers relying on drop handling and/or blocking for flow
59  * control.
60  */

```

【原文】A {@link Flow.Publisher} that asynchronously(异步地) issues(放出、发布) **submitted (non-null) items**(已被提交的非空元素) to current subscribers(订阅者) until it is closed. Each current subscriber receives **newly submitted items**(新的已被提交的元素) in the same order unless(除非) drops(丢弃) or exceptions are encountered. Using a SubmissionPublisher allows **item generators**(元素生成器) to act(工作) as(以...方式) compliant(遵从) *reactive-streams*. Publishers **relying on**(依靠) **drop handling**(丢弃处理) and/or blocking for **flow control**(流量控制).

【翻译】(当前的 SubmissionPublisher 类)是一个发布者，它能够以异步方式将已被提交的非空元素发布给订阅者，直到该发布者被关闭。每个当前订阅者都会以相同的顺序接收新提交的元素，除非遇到删除或异常。使用 SubmissionPublisher 对象允许元素生成器以符合响应流的方式工作。发布者依靠丢弃处理和/或阻塞来进行流量控制。

【总结】SubmissionPublisher 是一个发布者，能够以响应流的方式生成消息元素，以异步方式发布消息元素。

(1) submit()

该发布者类中有一个很重要的方法 `submit()`，用于发布指定消息到订阅令牌。

```
/**
 * Publishes the given item to each current subscriber by
 * asynchronously invoking its {@link Flow.Subscriber#onNext(Object)
 * onNext} method, blocking uninterruptibly while resources for any
 * subscriber are unavailable. This method returns an estimate of
 * the maximum lag (number of items submitted but not yet consumed)
 * among all current subscribers. This value is at least one
 * (accounting for this submitted item) if there are any
 * subscribers, else zero.
 *
 * <p>If the Executor for this publisher throws a
 * RejectedExecutionException (or any other RuntimeException or
 * Error) when attempting to asynchronously notify subscribers,
 * then this exception is rethrown, in which case not all
 * subscribers will have been issued this item.
 *
 * @param item the (non-null) item to publish
 * @return the estimated maximum lag among subscribers
 * @throws IllegalStateException if closed
 * @throws NullPointerException if item is null
 * @throws RejectedExecutionException if thrown by Executor
 */
public int submit(T item) {
    return doOffer(item, Long.MAX_VALUE, null);
}
```

【原文】 Publishes the given item to each current subscriber by asynchronously invoking its {onNext} method, blocking uninterruptibly while resources for any subscriber are unavailable.

【翻译】 将给定 item 发布到每个当前的订阅者，并通过异步方式调用其 onNext()方法。当任何为订阅者准备的资源不可用时会阻塞，并且阻塞将不会被打断。

(2) close()

```
/**
 * Unless already closed, issues {@link
 * Flow.Subscriber#onComplete() onComplete} signals to current
 * subscribers, and disallows subsequent attempts to publish.
 * Upon return, this method does NOT guarantee that all
 * subscribers have yet completed.
 */
public void close() {
    if (!closed) {
```

【原文】 Unless already closed, issues {@link onComplete} signals to current subscribers, and disallows subsequent attempts to publish. Upon return, this method does *NOT* guarantee that all subscribers have yet completed.

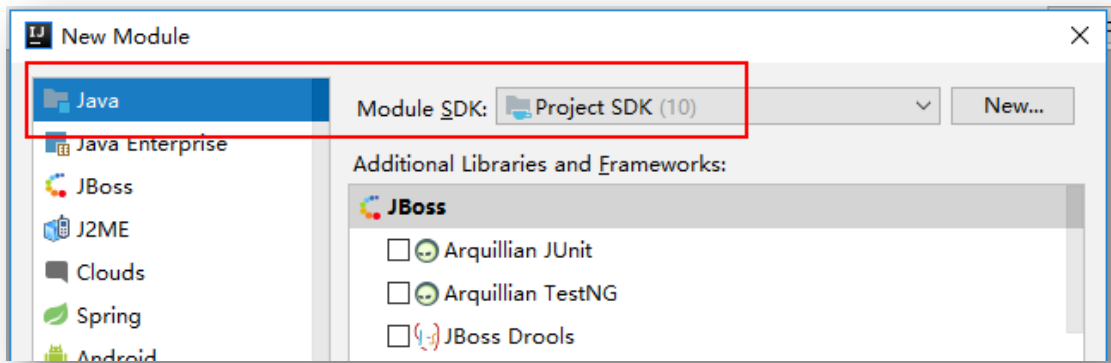
【翻译】 除非已经关闭，否则会向当前所有订阅者发出 onComplete()信号，并且不允许后续发布尝试。返回时，该方法不保证所有订阅者都已完成。

1.4.4 “发布-订阅”模式反应式流编程 05-reactiveStream

(1) 创建工程

创建一个普通的 Java 工程，无需是 Maven 工程，并命名为 reactiveStream。注意，要求 JDK 为 9 或 10。

当前工程中要自定义一个订阅者，而发布者则使用 JDK 自带的 SubmissionPublisher。



(2) 代码编写

A、定义订阅者

由于订阅的消息为 Integer 类型，所以 Flow.Subscriber 的泛型为 Integer 类型。

```
/**
 * 订阅者
 */
public class SomeSubscriber implements Flow.Subscriber<Integer> {
    // 声明订阅令牌
    private Flow.Subscription subscription;

    // 订阅者中第一个被执行的方法
    public void onSubscribe(Flow.Subscription subscription) {
        System.out.println("=== 执行订阅者的onSubscribe()方法 ===");
        this.subscription = subscription;
        // 首次订阅8条消息
        this.subscription.request(8);
    }
}
```

// 订阅者每接收一次消息，就会执行该方法一次

```
public void onNext(Integer item) {
    System.out.println("订阅者正在处理的消息数据为: " + item);
    try {
        TimeUnit.MICROSECONDS.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // 再次订阅10条消息，即每消费1条消息，就会再订阅10条消息
    this.subscription.request(10);
}
```

// 当订阅、消费过程中出现异常时，该方法被触发

```
public void onError(Throwable throwable) {
    System.out.println(" --- 执行onError()方法 ---");
    throwable.printStackTrace();
    this.subscription.cancel();
}
```

// 当所有消息被全部消费完毕后，会触发该方法的执行

```
public void onComplete() {
    System.out.println("发布者已关闭，订阅者将所有消息全部处理完成");
}
```

B、定义测试类

```
public class SomeTest {
    public static void main(String[] args) {
        SubmissionPublisher<Integer> publisher = null;
        try {
            // 使用JDK内置的发布者
            publisher = new SubmissionPublisher();
            // 创建订阅者
            SomeSubscriber subscriber = new SomeSubscriber();
            // 建立订阅关系
            publisher.subscribe(subscriber);
            // 生产300条消息
            for(int i = 0; i < 300; ++i) {
                // 生成一个[0, 100)的随机整数
                int item = new Random().nextInt(100);
                System.out.println("生产出第" + i + "条消息" + item);
                // 发布消息, 发布者缓存满时阻塞
                publisher.submit(item);
            }
        } finally {
            // 所有消息发送完毕, 关闭发布者
            publisher.close();
        }
        try {
            System.out.println("主线程开始等待");
            TimeUnit.SECONDS.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

1.4.5 “发布-处理-订阅”模式响应流编程 05-reactiveStream

前面的例子是，发布者发布的所有 `Integer` 消息均会被订阅者全部消费。但这里有一个需求：将发布者发布的大于 50 的消息过滤掉，并将小于 50 的 `Integer` 消息转换为 `String` 后发布给订阅者。注意，要求发布者必须要发布所有消息，不能让发布者自己过滤掉大于 50 的消息，大于 50 的消息也必须发布。

此时就需要借助处理器 `Processor` 来完成了。处理器将发布者的整数数据，经过处理器过滤处理后，变为了 `String` 数据。这样的话，订阅者所消费的数据就成为了 `String` 类型了。

(1) 创建工程

还在前面的工程中定义即可。

(2) 定义处理器

```
public class SomeProcessor extends SubmissionPublisher<String>
    implements Flow.Processor<Integer, String> {

    // 声明订阅令牌
    private Flow.Subscription subscription;

    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        this.subscription.request(8);
    }

    public void onNext(Integer item) {
        System.out.println("--- 处理器正在处理的消息数据为: " + item);
        if(item < 50) {
            this.submit("消息已处理: " + item);
        }
        try {
            TimeUnit.MICROSECONDS.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.subscription.request(10);
    }

    public void onError(Throwable throwable) {
        throwable.printStackTrace();
        this.subscription.cancel();
    }

    public void onComplete() {
        System.out.println("处理器已将消息处理完毕");
        this.close();
    }
}
```

(3) 修改订阅者

```
/**
 * 订阅者
 */
public class SomeSubscriber implements Flow.Subscriber<String> {
    // 声明订阅令牌
    private Flow.Subscription subscription;

    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        this.subscription.request(8);
    }

    public void onNext(String item) {
        System.out.println("订阅者正在处理的消息数据为: " + item);
        try {
            TimeUnit.MICROSECONDS.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.subscription.request(10);
    }

    public void onError(Throwable throwable) {
        throwable.printStackTrace();
        this.subscription.cancel();
    }

    public void onComplete() {
        System.out.println("发布者已关闭, 订阅者将所有消息全部处理完成");
    }
}
```

(4) 修改测试类

```
public class SomeTest {

    public static void main(String[] args) {
        SubmissionPublisher<Integer> publisher = null;
        try {
            publisher = new SubmissionPublisher();
            // 创建订阅者
            SomeSubscriber subscriber = new SomeSubscriber();
            // 创建处理器
            SomeProcessor processor = new SomeProcessor();
            // 建立订阅关系
            publisher.subscribe(processor);
            processor.subscribe(subscriber);
            for(int i = 0; i < 300; ++i) {
                int item = new Random().nextInt(100);
                System.out.println("开始生产消息" + i);
                publisher.submit(item);
            }
        } finally {
            publisher.close();
        }
        try {
            System.out.println("主线程开始等待");
            TimeUnit.SECONDS.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

第2章 WebFlux 服务端开发

本系统要实现的功能是：通过 WebFlux 实现对 MongoDB 的 CRUD 操作。

2.1 使用传统处理器开发

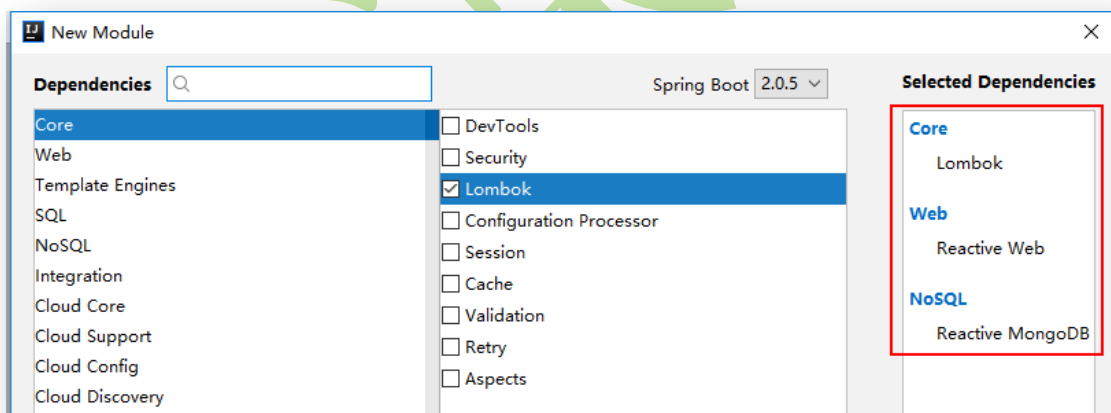
使用传统处理器开发，指的是使用 `@Controller` 注解的类作为处理器类，使用 `@RequestMapping` 进行请求与处理器方法映射，来开发 WebFlux 服务端的开发方式。

2.1.1 基本结构搭建 06-webflux-common

(1) 创建工程

创建一个 Spring Initializr 工程，Spring Boot 的版本要求最低为 2.0.0。不要添加原来的 web 依赖，而是要添加 Reactive Web，即 flux 依赖，添加 Lombok 依赖。另外，再添加上 Reactive MongoDB 依赖。

工程命名为 06-webflux-common。



(2) 修改启动类

```
@EnableReactiveMongoRepositories // 开启MongoDB的spring-data-jpa
@SpringBootApplication
public class WebfluxCommonApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebfluxCommonApplication.class, args);
    }
}
```

(3) 定义实体类

MongoDB 中表的 id 一般为 String 类型。@Document 与 @Id 均为 spring data jpa 中的注解，可以完成自动建表并指定表的主键。

```
@Data
// 指定在MongoDB中生成的表
@Document(collection = "t_student")
public class Student {

    @Id
    // MongoDB表中的id一般为String类型
    private String id;
    private String name;
    private int age;
}
```

(4) 定义 Repository 接口

第一个泛型为该 JPA 操作的实体类，第二个泛型为实体类的主键类型。

```
public interface StudentRepository
    extends ReactiveMongoRepository<Student, String> {
}
```

(5) 定义处理器

```
@RestController
@RequestMapping("/student")
public class StudentController {
    @Autowired
    private StudentRepository repository;

    // 一次性返回数据
    @GetMapping("/all")
    public Flux<Student> getAll() {
        return repository.findAll();
    }

    // 以SSE形式实时性返回数据
    @GetMapping(value = "/sse/all", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Student> getAllSse() {
        return repository.findAll();
    }
}
```

（6）数据库连接配置

```
application.yml
1  spring:
2    data:
3      mongodb:
4        uri: mongodb://localhost:27017/test
5
```

2.1.2 CURD 的实现

（1）添加对象

```
// 添加数据
@PostMapping("/save")
public Mono<Student> saveStudent(@RequestBody Student student) {
    return repository.save(student);
}
```

（2）无状态数据删除

所谓无状态删除，即指定的要删除的对象无论是否存在，其响应码均为 200，我们无法知道是否真正删除了数据。

```
// 无状态删除
@DeleteMapping("/delcomm/{id}")
public Mono<Void> deleteStudent(@PathVariable("id") String id) {
    return repository.deleteById(id);
}
```

(3) 有状态数据删除

所谓有状态删除，即指若删除的对象存在，且删除成功，则返回响应码 200，否则返回响应码 404。通过响应码就可以判断删除操作是否成功。

```
// 有状态删除
@DeleteMapping("/delstat/{id}")
// Mono<ResponseEntity<Void>>表示方法返回值为Mono序列，
// 其包含的元素为ResponseEntity对象，该对象中仅为包含响应状态码
public Mono<ResponseEntity<Void>> deleteStatStudent(@PathVariable("id") String id) {
    // 首先根据id查找是否存在该对象。
    // 若存在，则删除之，并返回一个Mono，其元素ResponseEntity中包含200状态码；
    // 若不存在，则返回一个由defaultIfEmpty()返回的Mono，其元素ResponseEntity中包含404状态码
    return repository.findById(id)
        .flatMap(stu -> repository.delete(stu).
            then(Mono.just(new ResponseEntity<Void>(HttpStatus.OK))))
        .defaultIfEmpty(new ResponseEntity<Void>(HttpStatus.NOT_FOUND));
}
```

(4) 修改数据

对于执行修改操作的处理器方法，我们可以这样定义其返回值：若修改成功，则返回修改后的对象数据；若指定的 id 对象不存在，则返回 404。


```
@PutMapping("/update/{id}")
public Mono<ResponseEntity<Student>> updateStudent(@PathVariable("id") String id,
                                                    @RequestBody Student student) {
    return repository.findById(id)
        .flatMap(stu -> {
            stu.setName(student.getName());
            stu.setAge(student.getAge());
            return repository.save(stu);
        })
        .map(stu -> new ResponseEntity<Student>(stu, HttpStatus.OK))
        .defaultIfEmpty(new ResponseEntity<Student>(HttpStatus.NOT_FOUND));
}
```

(5) 根据 id 查询

对于执行根据 id 进行查询操作的处理器方法，我们可以这样定义其返回值：若有查询结果，则返回查询到的对象数据；若没有查询结果，则返回 404。

```
// 根据id查询
@GetMapping("/find/{id}")
public Mono<ResponseEntity<Student>> updateStudent(@PathVariable("id") String id) {
    return repository.findById(id)
        .map(stu -> new ResponseEntity<Student>(stu, HttpStatus.OK))
        .defaultIfEmpty(new ResponseEntity<Student>(HttpStatus.NOT_FOUND));
}
```

(6) 根据年龄上下限查询

A、修改 StudentRepository 接口

在其中添加一个抽象方法。

```
public interface StudentRepository
    extends ReactiveMongoRepository<Student, String> {

    /**
     * 根据年龄上下限查询
     * @param below 年龄下限(不包含此下限)
     * @param top 年龄上限(不包含此上限)
     * @return
     */
    Flux<Student> findByAgeBetween(int below, int top);

}
```

B、修改处理器

```
// 根据年龄上下限查询：一次性返回
@GetMapping("/age/{below}/{top}")
public Flux<Student> findStudentByAge(@PathVariable("below") int below,
    @PathVariable("top") int top) {
    return repository.findByAgeBetween(below, top);
}
```

```
// 根据年龄上下限查询：实时性返回
@GetMapping(value = "/sse/age/{below}/{top}",
    produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Student> findStudentByAgeSSE(@PathVariable("below") int below,
    @PathVariable("top") int top) {
    return repository.findByAgeBetween(below, top);
}
```

(7) 使用 MongoDB 的原始查询语句

A、修改 StudentRepository 接口

在接口中添加如下抽象方法。

```
/**
 * 使用MongoDB的原始查询实现根据年龄上下限查询
 * @param below
 * @param top
 * @return
 */
@Query("{ 'age' : { '$gte' : ?0, '$lte' : ?1 } }")
Flux<Student> queryByAge(int below, int top);
```

B、修改处理器

```
// 使用MongoDB的原始查询实现根据年龄上下限查询：一次性返回
@GetMapping("/age/query/{below}/{top}")
public Flux<Student> queryStudentByAge(@PathVariable("below") int below,
                                       @PathVariable("top") int top) {
    return repository.queryByAge(below, top);
}
```

```
// 使用MongoDB的原始查询实现根据年龄上下限查询：实时性返回
@GetMapping(value = "/sse/age/query/{below}/{top}",
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<Student> queryStudentByAgeSSE(@PathVariable("below") int below,
                                          @PathVariable("top") int top) {
    return repository.queryByAge(below, top);
}
```

2.1.3 参数校验

为了保证数据在进入业务运算时的正确性，我们会对参数首先进行校验。一般情况下，我们可以直接使用 Hibernate Validator 中已经定义好的校验注解，若不能满足需求，也可以自定义校验逻辑。

(1) 使用 Hibernate 注解校验

Hibernate Validator 中已经定义好了很多通用的校验注解，我们可以直接使用。

A、代码中添加注解

a、修改实体类 Student

在要验证的属性上添加相应注解，即验证规则。

```
@Data
// 指定在MongoDB中生成的表
@Document(collection = "t_student")
public class Student {

    @Id
    // MongoDB表中的id一般为String类型
    private String id;

    @NotBlank(message = "姓名不能为空")
    private String name;

    @Range(min = 15, max = 50, message = "年龄必须在{min}-{max}范围")
    private int age;
}
```

b、修改处理器

```
// 添加数据
@PostMapping("/save")
public Mono<Student> saveStudent(@Valid @RequestBody Student student) {
    return repository.save(student);
}

// 添加数据
@PostMapping("/save2")
public Mono<Student> saveStudent2(@Valid Student student) {
    return repository.save(student);
}
```

```
@PutMapping("/update/{id}")
public Mono<ResponseEntity<Student>> updateStudent(@PathVariable("id") String id,
    @Valid @RequestBody Student student) {
    return repository.findById(id)
        .flatMap(stu -> {
            stu.setName(student.getName());
            stu.setAge(student.getAge());
            return repository.save(stu);
        })
        .map(stu -> new ResponseEntity<Student>(stu, HttpStatus.OK))
        .defaultIfEmpty(new ResponseEntity<Student>(HttpStatus.NOT_FOUND));
}
```

B、添加校验切面

```
// 表示该类为处理器通知切面，处理器方法将作为连接点
@ControllerAdvice
public class ParamValidateAdvice {

    @ExceptionHandler
    public ResponseEntity<String> validateHandle(WebExchangeBindException ex) {
        return new ResponseEntity<String>(exToStr(ex), HttpStatus.BAD_REQUEST);
    }

    // 将所有的异常对象转换为一个String
    private String exToStr(WebExchangeBindException ex) {
        return ex.getFieldErrors().stream()
            .map(e -> e.getField() + ":" + e.getDefaultMessage())
            .reduce("", (s1, s2) -> s1 + "\n" + s2);
    }
}
```

C、常用 Hibernate 校验注解

每一个 Hibernate 校验注解均有一个 message 属性，用于设置验证失败后的提示信息。

注解	适用的数据类型	说明
@AssertFalse	Boolean, boolean	验证注解的元素值是 false
@AssertTrue	Boolean, boolean	验证注解的元素值是 true
@DecimalMax (value=x)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number and CharSequence.	验证注解的元素值小于等于@ DecimalMax 指定的 value 值
@DecimalMin (value=x)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number and CharSequence.	验证注解的元素值小于等于@ DecimalMin 指定的 value 值

@Digits(integer=整数位数, fraction=小数位数)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number and CharSequence.	验证注解的元素值的整数位数和小数位数上限
@Future	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time date/time API is on the class path: any implementations of ReadablePartial and ReadableInstant.	验证注解的元素值(日期类型)比当前时间晚
@Max (value=x)	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of CharSequence (the numeric value represented by the character sequence is evaluated), any sub-type of Number.	验证注解的元素值小于等于@Max 指定的 value 值
@Min (value=x)	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of CharSequence (the numeric value represented by the char sequence is evaluated), any sub-type of Number.	验证注解的元素值大于等于@Min 指定的 value 值
@NotNull	Any type	验证注解的元素值不是 null
@Null	Any type	验证注解的元素值是 null
@Past	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time date/time API is on the class path: any implementations of ReadablePartial and ReadableInstant.	验证注解的元素值(日期类型)比当前时间早
@Pattern(regex=正则表达式, flag=)	String. Additionally supported by HV: any sub-type of CharSequence.	验证注解的元素值与指定的正则表达式匹配

@Size(min=最小值, max=最大值)	String, Collection, Map and arrays. Additionally supported by HV: any sub-type of CharSequence.	验证注解的元素值的在 min 和 max (包含) 指定区间之内, 如字符串长度、集合大小
@Valid	Any non-primitive type (引用类型)	验证关联的对象, 如账户对象里有一个订单对象, 指定验证订单对象
@NotEmpty	CharSequence, Collection, Map and Arrays	验证注解的元素值不为 null 且不为空 (字符串长度不为 0、集合大小不为 0)
@Range(min=最小值, max=最大值)	CharSequence, Collection, Map and Arrays, BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types	验证注解的元素值在最小值和最大值之间
@NotBlank	CharSequence	验证注解的元素值不为空 (不为 null、去除首位空格后长度为 0), 不同于 @NotEmpty, @NotBlank 只应用于字符串且在比较时会去除字符串的空格
@Length(min=下限, max=上限)	CharSequence	验证注解的元素值长度在 min 和 max 区间内
@Email	CharSequence	验证注解的元素值是 Email, 也可以通过正则表达式和 flag 指定自定义的 email 格式

(2) 自定义校验逻辑

对于有些情况, 我们使用注解无法满足校验需求, 例如, 我们需要使 Student 的 name 值不能是 admin 或 administrator, 此时, 可自己定义校验逻辑。

A、定义异常类

```
@Getter
@Setter
public class StudentException extends RuntimeException {
    private String errField;
    private String errValue;

    public StudentException() {
        super();
    }

    public StudentException(String message, String errField, String errValue) {
        super(message);
        this.errField = errField;
        this.errValue = errValue;
    }
}
```

B、定义校验工具类

```
public class ValidateUtil {
    // 无效姓名列表
    private static final String[] INVALID_NAMES = {"admin", "administrator"};

    // 验证姓名
    public static void validateName(String name) {
        Stream.of(INVALID_NAMES)
            .filter(invalideName -> name.equalsIgnoreCase(invalideName))
            .findAny()
            .ifPresent(invalideName -> {
                throw new StudentException("name", invalideName, "使用了非法姓名");
            });
    }
}
```

C、修改校验切面

在校验切面类中添加如下异常处理方法。

```
// 表示该类为处理器通知切面，处理器方法将作为连接点
@ControllerAdvice
public class ParamValidateAdvice {

    @ExceptionHandler
    public ResponseEntity<String> validateHandle(StudentException ex) {
        // 获取异常对象中的数据
        String message = ex.getMessage();
        String errField = ex.getErrField();
        String errValue = ex.getErrValue();
        // 重新构建异常信息
        String exMsg = message + "【" + errField + ":" + errValue + "】";

        return new ResponseEntity<String>(exMsg, HttpStatus.BAD_REQUEST);
    }
}
```

D、修改处理器

修改具有 Student 类型参数的处理器方法，在其中添加校验代码。

```
// 添加数据
@PostMapping("/save")
public Mono<Student> saveStudent(@Valid @RequestBody Student student) {
    // 验证姓名的合法性
    ValidateUtil.validateName(student.getName());

    return repository.save(student);
}

// 添加数据
@PostMapping("/save2")
public Mono<Student> saveStudent2(@Valid Student student) {
    // 验证姓名的合法性
    ValidateUtil.validateName(student.getName());

    return repository.save(student);
}
```

```
@PutMapping("/update/{id}")
public Mono<ResponseEntity<Student>> updateStudent(@PathVariable("id") String id,
                                                    @Valid @RequestBody Student student) {
    // 验证姓名的合法性
    ValidateUtil.validateName(student.getName());

    return repository.findById(id)
        .flatMap(stu -> {
            stu.setName(student.getName());
            stu.setAge(student.getAge());
            return repository.save(stu);
        })
        .map(stu -> new ResponseEntity<Student>(stu, HttpStatus.OK))
        .defaultIfEmpty(new ResponseEntity<Student>(HttpStatus.NOT_FOUND));
}
```

2.2 使用 Router Functions 开发

使用 Router Functions 开发，指的是使用由@Component 注解的普通类作为处理器类，使用 Router 进行请求与处理器方法映射，来开发 WebFlux 服务端的开发方式。

2.2.1 基本结构搭建 07-webflux-router

(1) 创建工程

复制前面的 06-webflux-common 工程，并重命名为 07-webflux-router。
删除其中的 StudentController 类及 ParamValidateAdvice 类，其它代码保留。

(2) 定义路由器

```
@Configuration
public class StudentRouter {

    @Bean
    RouterFunction<ServerResponse> customRouter(StudentHandler handler) {
        return RouterFunctions
            .nest(RequestPredicates.path("/student"),
                RouterFunctions.route(RequestPredicates.GET("/all"),
                    handler::findAllHandler)
            );
    }
}
```

(3) 定义处理器

这里的处理器并不是之前使用@Controller注解的处理器类，而是一个使用@Component注解的普通类。通过对前面路由器中route()方法的第二个参数分析可知，该处理器就是一个HandlerFunction类。

```
@Component
public class StudentHandler {

    @Autowired
    private StudentRepository repository;

    // 查询所有
    public Mono<ServerResponse> findAllHandler(ServerRequest request) {
        return ServerResponse
            // 指定响应码为200
            .ok()
            // 指定请求体中的内容类型为UTF8编码的JSON数据
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            // 构建响应体
            .body(repository.findAll(), Student.class);
    }
}
```

2.2.2 CURD 的实现

(1) 添加对象

A、修改路由器

```
@Bean
RouterFunction<ServerResponse> customRouter(StudentHandler handler) {
    return RouterFunctions
        .nest(RequestPredicates.path("/student"),
            RouterFunctions.route(RequestPredicates.GET("/all"),
                handler::findAllHandler)
            .andRoute(RequestPredicates.POST("/save")
                .and(RequestPredicates.accept(MediaType.APPLICATION_JSON_UTF8)),
                handler::saveHandler)
        );
}
```

B、修改处理器

在处理器中添加如下处理器方法。

```
// 添加数据
public Mono<ServerResponse> saveHandler(ServerRequest request) {
    // 从请求中获取要添加的数据
    Mono<Student> studentMono = request.bodyToMono(Student.class);

    return ServerResponse
        .ok()
        .contentType(MediaType.APPLICATION_JSON_UTF8)
        .body(repository.saveAll(studentMono), Student.class);
}
```

(2) 有状态删除对象

当前删除是一个有状态删除：若删除成功，则返回 200，否则返回 404。

A、修改路由器

在路由方法中添加如下路由规则。

```
@Bean
RouterFunction<ServerResponse> customRouter(StudentHandler handler) {
    return RouterFunctions
        .nest(RequestPredicates.path("/student"),
            RouterFunctions.route(RequestPredicates.GET("/all"),
                handler::findAllHandler)
            .andRoute(RequestPredicates.POST("/save")
                .and(RequestPredicates.accept(MediaType.APPLICATION_JSON_UTF8)),
                handler::saveHandler)
            .andRoute(RequestPredicates.DELETE("/del/{id}"), handler::deleteByIdHandler)
        );
}
```

B、修改处理器

```
// 根据id删除
public Mono<ServerResponse> deleteByIdHandler(ServerRequest request) {
    // 从请求的路径变量中获取id
    String id = request.pathVariable("id");

    return repository
        .findById(id)
        .flatMap(stu -> repository.delete(stu)
            .then(ServerResponse.ok().build()))
        .switchIfEmpty(ServerResponse.notFound().build());
}
```

(3) 修改对象

A、修改路由器

在路由方法中添加如下路由规则。

```
@Bean
RouterFunction<ServerResponse> customRouter(StudentHandler handler) {
    return RouterFunctions
        .nest(RequestPredicates.path("/student"),
            RouterFunctions.route(RequestPredicates.GET("/all"),
                handler::findAllHandler)
            .andRoute(RequestPredicates.POST("/save")
                .and(RequestPredicates.accept(MediaType.APPLICATION_JSON_UTF8)),
                handler::saveHandler)
            .andRoute(RequestPredicates.DELETE("/del/{id}"), handler::deleteByIdHandler)
            .andRoute(RequestPredicates.PUT("/update/{id}"), handler::updateHandler)
        );
}
```

B、修改处理器

这里实现的逻辑是：若指定的 id 对象不存在，则指定 id 作为新的 id 完成插入；否则完成修改。

```
// 修改
public Mono<ServerResponse> updateHandler(ServerRequest request) {
    // 从请求的路径变量中获取id
    String id = request.pathVariable("id");
    // 从请求中获取要修改的数据
    Mono<Student> studentMono = request.bodyToMono(Student.class);

    return studentMono
        .flatMap(stu -> {
            stu.setId(id);
            return ServerResponse
                .ok()
                .contentType(MediaType.APPLICATION_JSON_UTF8)
                .body(repository.save(stu), Student.class);
        });
}
```

2.2.3 参数校验

由于这里的处理器方法只有 `ServerRequest` 一个参数，所以无法使用注解方式的参数校验，即无法使用 `Hibernate Validator`。但可以使用自定义的参数校验。

(1) 修改实体类

去掉实体类中的 `Hibernate Validator` 注解。


```
@Data
// 指定在MongoDB中生成的表
@Document(collection = "t_student")
public class Student {

    @Id
    // MongoDB表中的id一般为String类型
    private String id;
    private String name;
    private int age;
}
```

(2) 修改处理器

A、对添加对象的验证

在处理器类中添加如下具有验证功能的处理器方法。

```
// 添加数据(验证姓名是否合法)
public Mono<ServerResponse> saveHandlerValidate(ServerRequest request) {
    // 从请求中获取要添加的数据
    Mono<Student> studentMono = request.bodyToMono(Student.class);

    return studentMono.flatMap(stu -> {
        // 验证姓名
        ValidateUtil.validateName(stu.getName());
        return ServerResponse
            .ok()
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            .body(repository.saveAll(studentMono), Student.class);
    });
}
```

B、对修改对象的验证

在处理器类中添加如下具有验证功能的处理器方法。

```
// 修改(验证姓名是否合法)
public Mono<ServerResponse> updateHandlerValidate(ServerRequest request) {
    // 从请求的路径变量中获取id
    String id = request.pathVariable("id");
    // 从请求中获取要修改的数据
    Mono<Student> studentMono = request.bodyToMono(Student.class);

    return studentMono
        .flatMap(stu -> {
            // 验证姓名
            ValidateUtil.validateName(stu.getName());
            stu.setId(id);
            return ServerResponse
                .ok()
                .contentType(MediaType.APPLICATION_JSON_UTF8)
                .body(repository.save(stu), Student.class);
        });
}
```

(3) 修改路由器

将路由器中相应的 Lambda 方法引用修改为具有姓名验证功能的方法。

```
@Bean
RouterFunction<ServerResponse> customRouter(StudentHandler handler) {
    return RouterFunctions
        .nest(RequestPredicates.path("/student"),
            RouterFunctions.route(RequestPredicates.GET("/all"),
                handler::findAllHandler)
            .andRoute(RequestPredicates.POST("/save")
                .and(RequestPredicates.accept(MediaType.APPLICATION_JSON_UTF8)),
                handler::saveHandlerValidate)
            .andRoute(RequestPredicates.DELETE("/del/{id}"), handler::deleteByIdHandler)
            .andRoute(RequestPredicates.PUT("/update/{id}"), handler::updateHandlerValidate)
        );
}
```

(4) 定义异常处理器

异常处理器指的是当发生异常时，其会捕获到异常，并对异常信息进行处理。

```
@Component
@Order(-99)
public class CustomExceptionHandler implements WebExceptionHandler {
    @Override
    public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {
        // 获取HTTP响应对象
        ServerHttpResponse response = exchange.getResponse();
        // 设置响应码为400
        response.setStatusCode(HttpStatus.BAD_REQUEST);
        // 设置返回类型为普通文本
        response.getHeaders().setContentType(MediaType.TEXT_PLAIN);
        // 获取异常信息
        String message = getExceptionMessage(ex);
        // 获取数据缓存
        DataBuffer buffer = response.bufferFactory().wrap(message.getBytes());
        // 以Mono的形式给出响应
        return response.writeWith(Mono.just(buffer));
    }
}
```

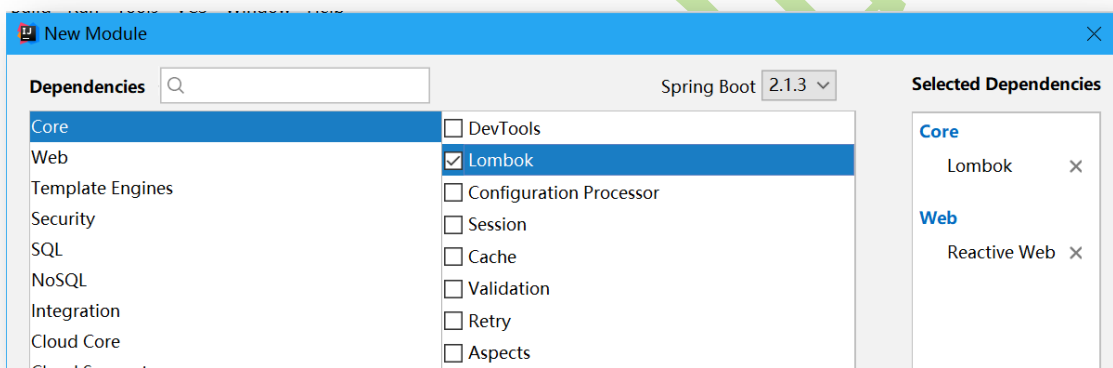
```
private String getExceptionMessage(Throwable ex) {
    // 设置一般异常信息
    String exMsg = "发生异常";
    // 设置指定异常信息
    if(ex instanceof StudentException) {
        StudentException e = (StudentException) ex;
        exMsg = e.getMessage() + "【" + e.getErrField() + e.getErrValue() + "】";
    }
    return exMsg;
}
```

第3章 WebFlux 客户端开发

前面我们通过两种方式开发了 WebFlux 服务器端，下面我们来开发 WebFlux 客户端，消费 WebFlux 服务端提供的服务。WebFlux 官方推荐我们使用 WebClient 客户端，其是随 WebFlux 一起推出的。

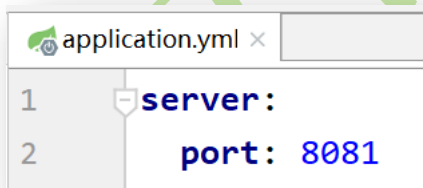
3.1 定义工程 08-webclient

创建一个 spring boot 工程，并命名为 08-webclient。无需 MongoDB 的依赖了，因为客户端无需访问 DB。



3.2 修改端口号

由于服务端使用的端口号为 8080，所以这里要修改客户端的端口号。



3.3 定义实体类

```
@Data
public class Student {
    private String id;
    private String name;
    private int age;
}
```

3.4 定义处理器

3.4.1 插入

```
@RestController
public class StudentController {

    // 创建WebClient客户端
    private WebClient client = WebClient.create("http://localhost:8080/student");

    @PostMapping("/save")
    public String saveStudentHandle(@RequestBody Student student) {
        Mono<Student> studentMono = client.post() RequestBodyUriSpec
            .uri("/save") RequestBodySpec
            .body(Mono.just(student), Student.class) RequestHeadersSpec<capture
            .retrieve() ResponseSpec
            .bodyToMono(Student.class);
        // 输出每mono中的元素
        studentMono.subscribe(System.out::println);
        return "插入完毕";
    }
}
```

3.4.2 删除

```
@DeleteMapping("/del/{id}")
public String deleteStudentHandle(@PathVariable("id") String id) {
    Mono<Void> voidMono = client.delete() RequestHeadersUriSpec<capture of ?>
        .uri("/delstat/{id}", id)
        .retrieve() ResponseSpec
        .bodyToMono(Void.class);
    voidMono.subscribe();
    return "删除完毕";
}
```

3.4.3 修改

```
@PutMapping("/update/{id}")
public String updateStudentHandle(@PathVariable("id") String id,
    @RequestBody Student student) {
    Mono<ResponseEntity> responseEntityMono = client.put() RequestB
        .uri("/update/{id}", id) RequestBodySpec
        .body(Mono.just(student), Student.class) RequestHeaders
        .retrieve() ResponseSpec
        .bodyToMono(ResponseEntity.class);
    responseEntityMono.subscribe();
    return "修改完成";
}
```

3.4.4 查询所有

```
@GetMapping("/list")
public Flux<Student> listAllHandle() {
    Flux<Student> studentFlux = client.get() Request
        .uri("/all") capture of ?
        .retrieve() ResponseSpec
        .bodyToFlux(Student.class);
    studentFlux.subscribe(System.out::println);
    return studentFlux;
}
```

3.4.5 根据 id 查询

```
@GetMapping("/get/{id}")
public Mono<Student> getStudentHandle(@PathVariable("id") String id) {
    Mono<Student> studentMono = client.get() RequestHeadersUriSpec<capture
        .uri("/find/{id}", id) capture of ?
        .retrieve() ResponseSpec
        .bodyToMono(Student.class);
    studentMono.subscribe(System.out::println);
    return studentMono;
}
```