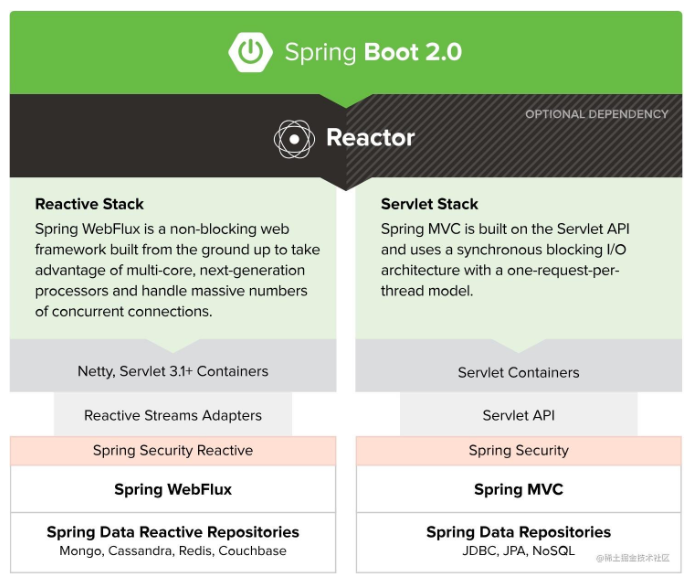
# 基本概念



**Spring MVC 构建于 Servlet API 之上，使用的是同步阻塞式 I/O 模型，什么是同步阻塞式 I/O 模型呢？就是说，每一个请求对应一个线程去处理。Spring WebFlux 是一个异步非阻塞式的 Web 框架，它能够充分利用多核 CPU 的硬件资源去处理大量的并发请求。**

# ****优势****

WebFlux 内部使用的是响应式编程（Reactive Programming），以 Reactor 库为基础, 基于异步和事件驱动，可以让我们在不扩充硬件资源的前提下，提升系统的吞吐量和伸缩性。WebFlux 并不能使接口的响应时间缩短，它仅仅能够提升吞吐量和伸缩性。

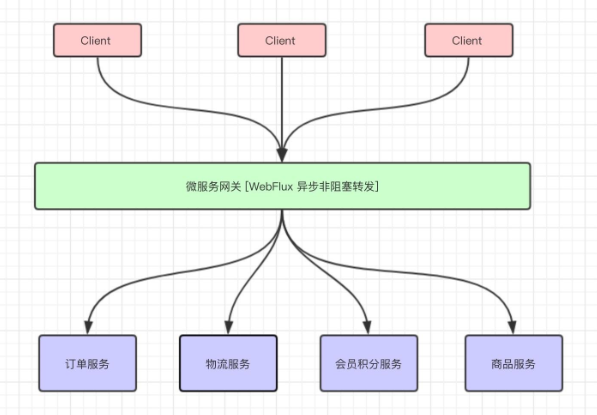
# 特点

1、非阻塞式：在有限资源下，提高系统吞吐量和伸缩性，以Reactor为基础实现响应式编程。

2、函数式编程：Spring5框架基于java8，Webflux使用java7函数式编程方式实现路由请求。

# 应用场景

 Spring WebFlux 是一个异步非阻塞式的 Web 框架，所以，它特别适合应用在 IO 密集型的服务中，比如微服务网关这样的应用中。IO 密集型包括：**磁盘IO密集型**, **网络IO密集型**，微服务网关就属于网络 IO 密集型，使用异步非阻塞式编程模型，能够显著地提升网关对下游服务转发的吞吐量。

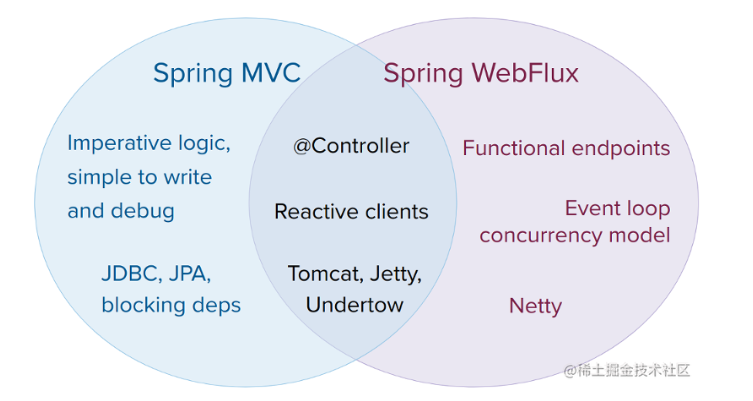


# 选 WebFlux 还是 Spring MVC

首先你需要明确一点就是：**WebFlux 不是 Spring MVC 的替代方案**！，虽然 WebFlux 也可以被运行在 Servlet 容器上（需是 Servlet 3.1+ 以上的容器），但是 WebFlux 主要还是应用在异步非阻塞编程模型，而 Spring MVC 是同步阻塞的，如果你目前在 Spring MVC 框架中大量使用非同步方案，那么，WebFlux 才是你想要的，否则，使用 Spring MVC 才是你的首选。

在微服务架构中，Spring MVC 和 WebFlux 可以混合使用，比如已经提到的，对于那些 IO 密集型服务(如网关)，我们就可以使用 WebFlux 来实现。总之一句话，在合适的场景中，选型最合适的技术。

# 异同点



## 相同点

1、都可以使用 Spring MVC 注解，如 @Controller, 方便我们在两个 Web 框架中自由转换；

2、均可以使用 Tomcat, Jetty, Undertow Servlet 容器（Servlet 3.1+）

## 注意点

1、Spring MVC 因为是使用的同步阻塞式，更方便开发人员编写功能代码，Debug 测试等，一般来说，如果 Spring MVC 能够满足的场景，就尽量不要用 WebFlux;

2、WebFlux 默认情况下使用 Netty 作为服务器;

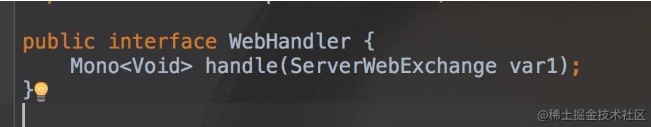
3、WebFlux 不支持 MySql。

# 响应式编程

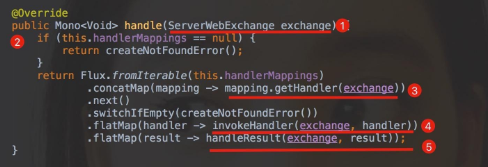
响应式编程是一种面向数据流和变化传播的编程范式。这意味着可以在编程语言中很方便地表达静态或动态的数据流。而相关的计算模型会自动将变化的值通过数据流进行传播。例如，在命令式编程环境中，a=b+c表示将表达式的结果赋给a，而之后改变b或c的值不会影响a。但在响应式编程中，a的值会随着b或c的更新而更新。

# WebFlux执行流程

使用过 Spring MVC 的小伙伴们，应该到知道 Spring MVC 的前端控制器是 DispatcherServlet, 而 WebFlux 是 DispatcherHandler，它实现了 WebHandler 接口：



来看看DispatcherHandler类中处理请求的 handle 方法：



1、ServerWebExchange 对象中放置每一次 HTTP 请求响应信息，包括参数等。

2、判断整个接口映射 mappings 集合是否为空，空则创建一个 Not Found 的错误。

3、根据具体的请求地址获取对应的 handlerMapping。

4、调用具体业务方法，也就是我们定义的接口方法。

5、处理返回的结果。

# 入门教程

## 1、添加 webflux 依赖



## 2、定义接口

新建一个 controller 包，用来放置对外的接口类，再创建一个 HelloWebFluxController.class 类，定义两个接口：



User类



以上控制器类中，我们使用的全都是 Spring MVC 的注解，分别定义了两个接口：

1、一个 GET 请求的 /hello 接口，返回 Hello, WebFlux !字符串。

2、又定义了一个 GET 请求的 /user方法，返回的是 JSON 格式 User 对象。

这里注意，User 对象是通过 Mono 对象包装的，你可能会问，为啥不直接返回呢？在 WebFlux 中，Mono 是非阻塞的写法，只有这样，你才能发挥 WebFlux **非阻塞 + 异步**的特性。

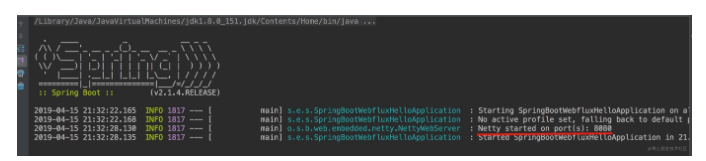
补充：在 WebFlux 中，除了 Mono 外，还有一个 Flux，这哥俩均能充当响应式编程中发布者的角色，不同的是：

1、Mono：返回 0 或 1 个元素，即单个对象。

2、Flux：返回 N 个元素，即 List 列表对象。

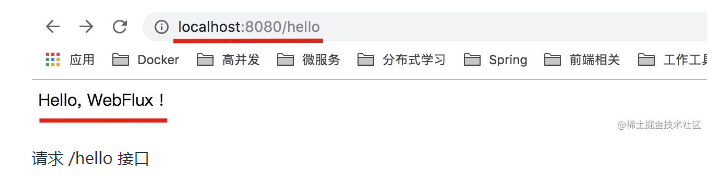
## 3、测试接口

启动项目，查看控制台输出：



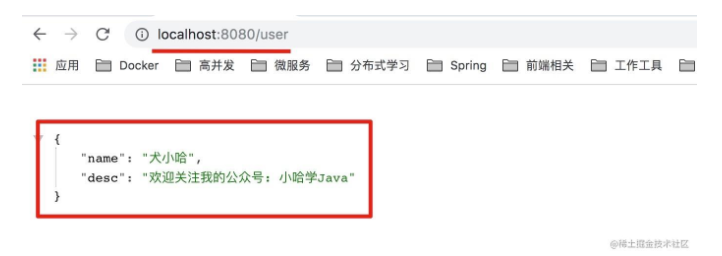
当控制台中输出中包含 Netty started on port(s): 8080 语句时，说明默认使用 Netty 服务已经启动了。

打开浏览器，先对 /hello 接口发起调用：



返回成功

再来对 /user 接口测试一下:



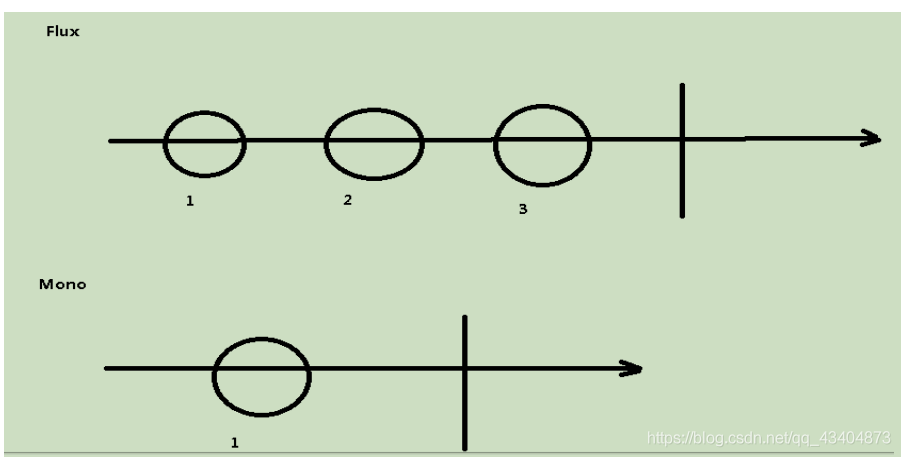
返回 JSON 格式的 User 实体也是 OK 的！

# 响应式编程(Reactor实现)

## 基本说明

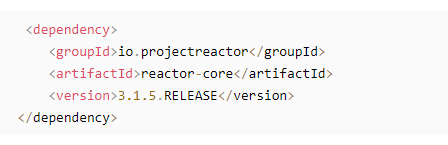
响应式编程操作中，Reactor是满足Reactive规范框架。Reactor有两个核心类，Momo和Flux，这两个类实现接口Publisher，提供丰富操作符。Flux对象实现发布者，返回N个元素；Mono实现发布者，返回0或者1个元素。

Flux和Mono都是数据流的发布者，使用Flux和Mono都可以发出三种数据信号：元素值、错误信号、完成信号，错误信号和完成信号都代表终止信号，终止信号用于告诉订阅者数据流结束了，错误信号终止数据流同时把错误信息传递给订阅者。

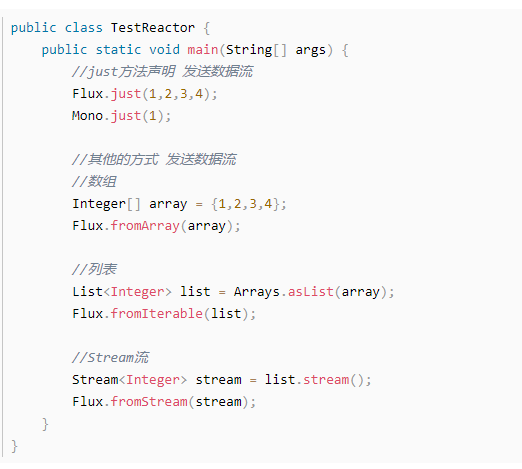


## 示例

### 1、引入依赖



### 2、编写代码



#### 三种信号的特点

1、错误信号和完成信号都是终止信号，不能共存的。

2、如果没有发送任何元素值，而是直接发送错误或者完成信号，表示是空数据流。

3、如果没有错误信号，没有完成信号，表示是无限数据流。

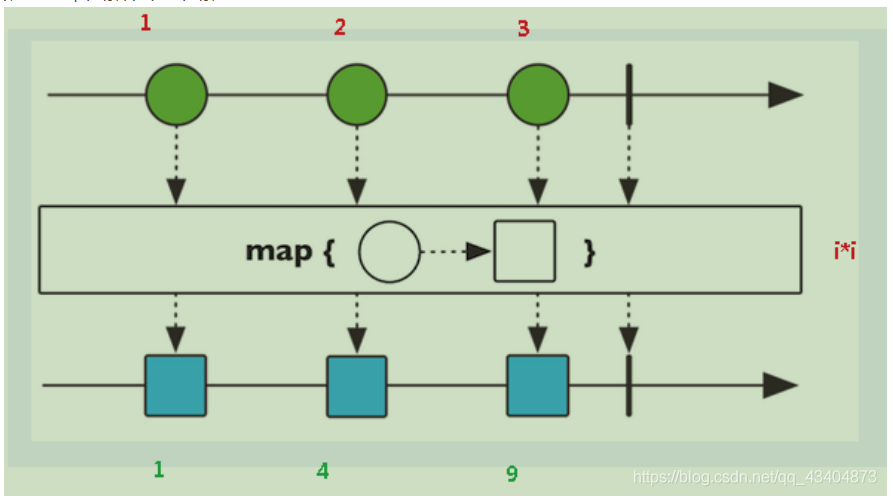
调用just或者其他方法只是声明数据流，数据流并没有发出，只有进行订阅之后才会触发数据流，不订阅什么都不会发生的。



#### 操作符

对数据流进行一道道操作，成为操作符，比如工厂流水线。

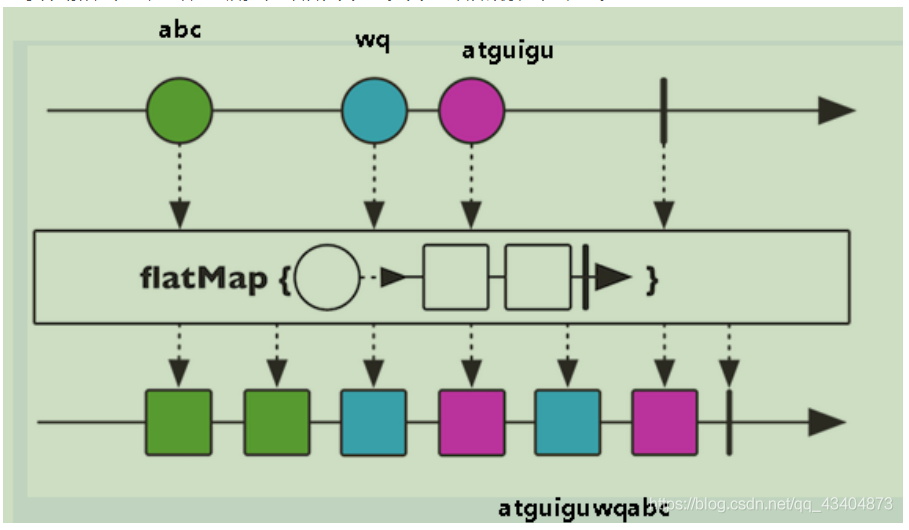
第一：map 元素映射为新元素



比如在上图中，绿色圆圈的数值代表1，2，3，经过map操作符映射后，映射的规则是平方，最后变成蓝色方块1，4，9。

第二： flatMap元素映射为流

把每个元素转换为流，把转换之后多个流合并为大的流。 大的流合并顺序是不确定的。



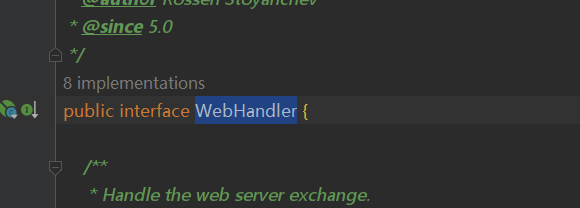
# SpringWebFlux(基于注解编程模型)

# 接口

## WebHandler

### 基本概念

### 源码



### 方法

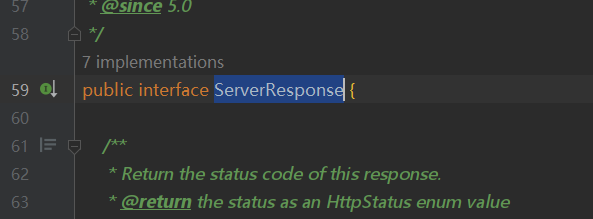
#### Mono<Void> handle(ServerWebExchange exchange);

## ServerHttpRequest

## ServerResponse

### 基本概念

### 源码



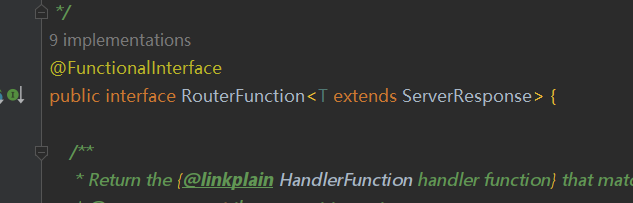
### 方法

## RouterFunction

### 基本概念

RouterFunction 接口是一个非常重要的接口，用于处理响应式Web请求路由和处理函数的映射。

### 源码



## WebClient

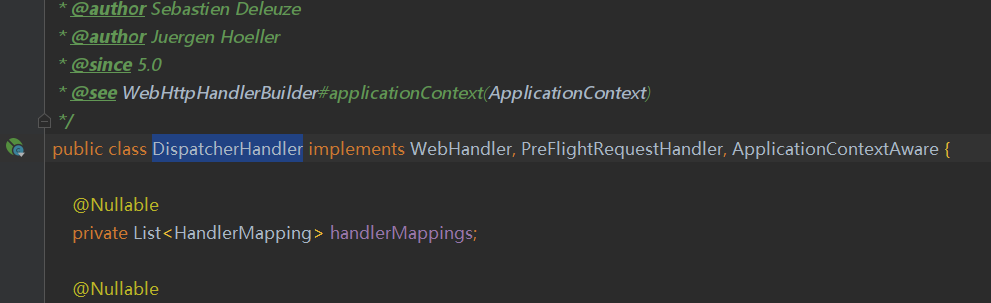
## HttpHandler

# 类

## DispatcherHandler

### 基本概念

### 源码



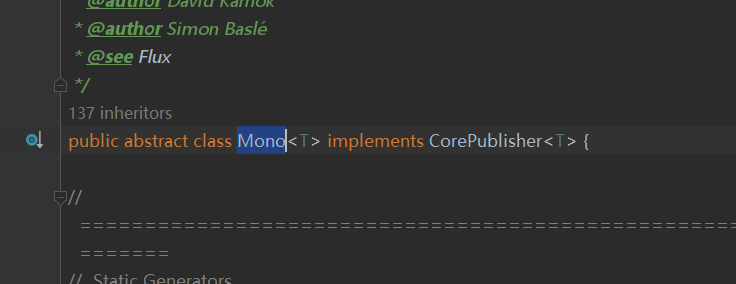
### 方法

#### public Mono<Void> handle(ServerWebExchange exchange)

## Mono

### 基本概念

### 源码



### 方法

#### public static <T> Mono<T> just(T data)

#### public static <T> Mono<T> justOrEmpty(@Nullable T data)

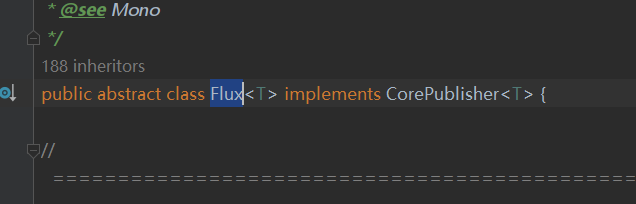
#### public final Mono<T> doOnNext(Consumer<? super T> onNext)

#### public final Disposable subscribe()

## Flux

### 基本概念

### 源码



### 方法

#### public static <T> Flux<T> fromIterable(Iterable<? extends T> it)

## RouterFunctions