WebFlux 核心

基本介绍

Spring WebFlux 是一套全新的 Reactive Web 栈技术，实现完全非阻塞，支持 Reactive Streams 背压等特性，并且运行环境不限于 Servlet 容器（Tomcat、Jetty、Undertow），如 Netty 等。

> 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 servers such as Netty, Undertow, and Servlet 3.1+ containers.

Spring WebFlux 与 Spring MVC 可共存，在 Spring Boot 中，Spring MVC 优先级更高。

> 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 may use one or the other module, or in some cases both — e.g. Spring MVC controllers with the reactive `WebClient`.

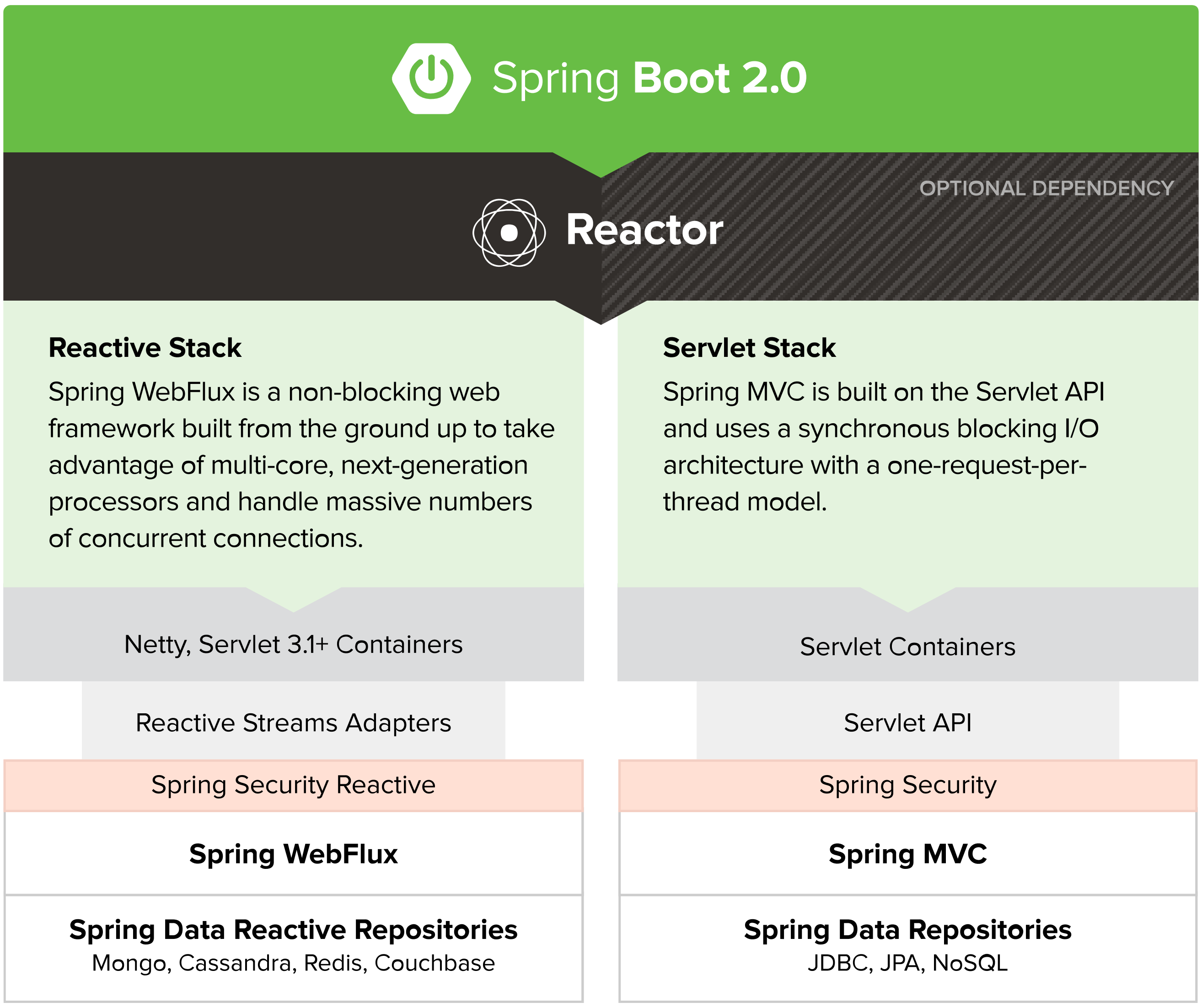
动机

Spring 官方说法

> Part of the answer is the need for a non-blocking web stack to handle concurrency with a small number of threads and scale with less hardware resources. Servlet 3.1 did provide an API for non-blocking I/O. However, using it leads away from the rest of the Servlet API where contracts are synchronous (`Filter`, `Servlet`) or blocking (`getParameter`, `getPart`). This was the motivation for a new common API to serve as a foundation across any non-blocking runtime. That is important because of servers such as Netty that are well established in the async, non-blocking space.
>
> The other part of the answer is functional programming. Much like the addition of annotations in Java 5 created opportunities — e.g. annotated REST controllers or unit tests, the addition of lambda expressions in Java 8 created opportunities for functional APIs in Java. This is a boon for non-blocking applications and continuation style APIs — as popularized by `CompletableFuture` and ReactiveX, that allow declarative composition of asynchronous logic. At the programming model level Java 8 enabled Spring WebFlux to offer functional web endpoints alongside with annotated controllers.
>
> > 原文链接：https://docs.spring.io/spring/docs/5.0.x/spring-framework-reference/web-reactive.html#webflux-new-framework

实际动机

从 Spring MVC 注解驱动的时代开始，Spring 官方有意识地**去 Servlet 化**。不过在 Spring MVC 的时代，Spring 扔拜托不了 Servlet 容器的依赖，然而 Spring 借助 Reactive Programming 的势头，WebFlux 将 Servlet 容器从必须项变为可选项，并且默认采用 Netty Web Server 作为基础，从而组件地形成 Spring 全新技术体系，包括数据存储等技术栈：


Spring Boot 2.0
Reactor
OPTIONAL DEPENDENCY
Reactive Stack
Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.
Servlet Stack
Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.
Netty, Servlet 3.1+ Containers
Servlet Containers
Reactive Streams Adapters
Servlet API
Spring Security Reactive
Spring Security
Spring WebFlux
Spring MVC
Spring Data Reactive Repositories
Mongo, Cassandra, Redis, Couchbase
Spring Data Repositories
JDBC, JPA, NoSQL

定义 Reactive

Spring Framework

> The term "reactive" refers to programming models that are built around **reacting to change** — network component reacting to I/O events, UI controller reacting to mouse events, etc. In that sense **non-blocking** is reactive because instead of being blocked we are now in the mode of reacting to notifications as operations complete or data becomes available.
>
> > https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-why-reactive

关键字：

- 变化响应（reacting to change ）
- 非阻塞（non-blocking）

侧重点：

- 响应通知
  - 操作完成（operations complete）
  - 数据可用（data becomes available）

Reactor API

`Mono`

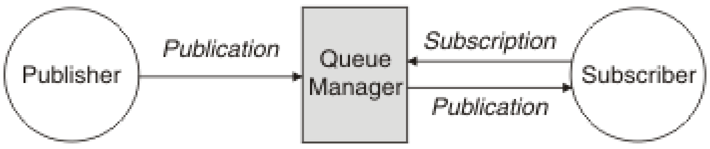
定义：0-1 的非阻塞结果

实现：Reactive Streams JVM API `Publisher`

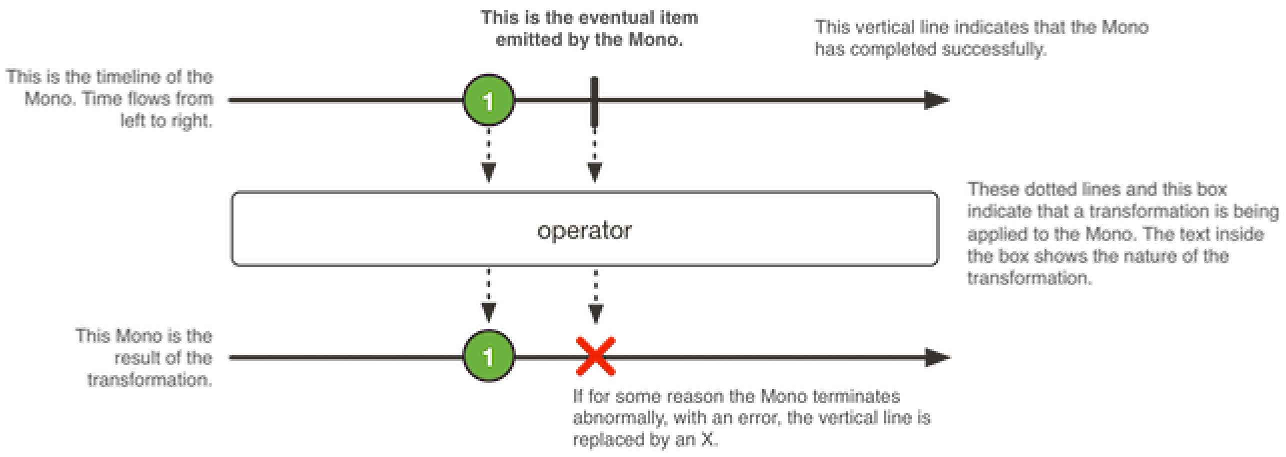
类比：非阻塞 `Optional`

类似模式

点对点模式



图解


This is the eventual item emitted by the Mono.
This vertical line indicates that the Mono has completed successfully.
This is the timeline of the Mono. Time flows from left to right.
1
operator
These dotted lines and this box indicate that a transformation is being applied to the Mono. The text inside the box shows the nature of the transformation.
This Mono is the result of the transformation.
1
If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.

`Flux`

定义：0-N 的非阻塞序列

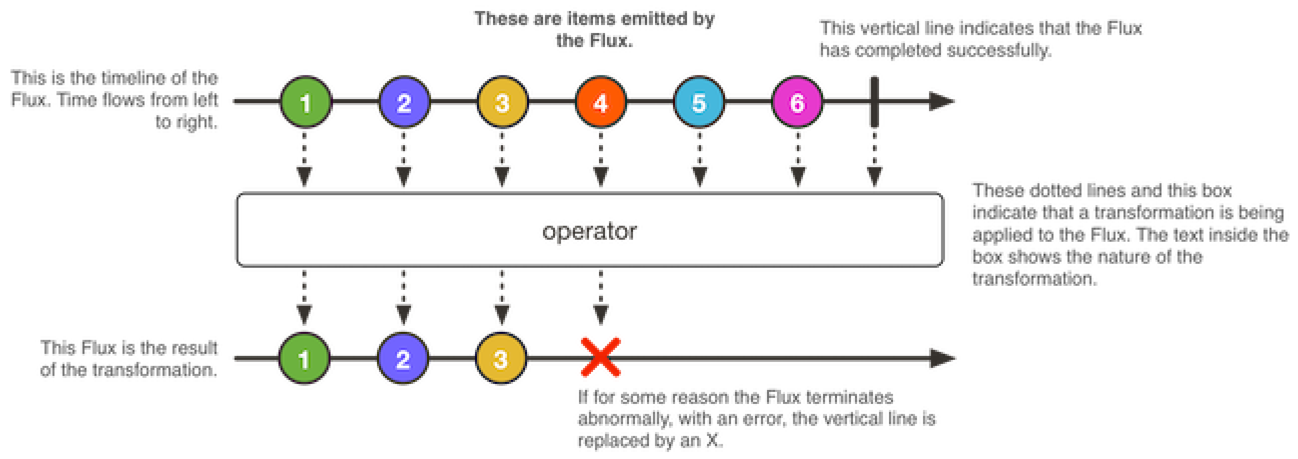
实现：Reactive Streams JVM API `Publisher`

类比：非阻塞 `Stream`

类似模式

发布者/订阅者模式

图解


These are items emitted by the Flux.
This vertical line indicates that the Flux has completed successfully.
This is the timeline of the Flux. Time flows from left to right.
1
2
3
4
5
6
operator
These dotted lines and this box indicate that a transformation is being applied to the Flux. The text inside the box shows the nature of the transformation.
This Flux is the result of the transformation.
1
2
3
If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X.

编程模型

注解驱动（Annotated Controllers）

定义

特性	Spring Web MVC	Spring WebFlux
应用控制器注解声明	`@Controller`	相同
应用 REST 控制器注解声明	`@RestController`	相同

映射

特性	Spring Web MVC	Spring WebFlux
请求映射注解声明	`@RequestMapping`	相同
GET 方法映射	`@GetMapping`	相同
POST 方法映射	`@PostMapping`	相同
PUT 方法映射	`@PutMapping`	相同
DELETE 方法映射	`@DeleteMapping`	相同
PATCH 方法映射	`@PatchMapping`	相同

请求

特性	Spring Web MVC	Spring WebFlux
获取请求参数	`@RequestParam`	相同
获取请求头	`@RequestHeader`	相同
获取Cookie值	`@CookieValue`	相同
获取完整请求主体内容	`@RequestBody`	相同
获取请求路径变量	`@PathVariable`	相同
获取请求内容（包括请求主体和请求头）	`RequestEntity`	相同

响应

特性	Spring Web MVC	Spring WebFlux
响应主体注解声明	`@ResponseBody`	相同
响应内容（包括响应主体和响应头）	`ResponseEntity`	相同
`ResponseCookie`	响应 Cookie 内容	相同

拦截

特性	Spring Web MVC	Spring WebFlux
`@Controller` 注解切面通知	`@ControllerAdvice`	相同
`@RestController` 注解切面通知	`@RestControllerAdvice`	相同

跨域

特性	Spring Web MVC	Spring WebFlux
资源跨域声明注解	`@CrossOrigin`	相同
资源跨域拦截器	`CorsFilter`	`CorsWebFilter`
注册资源跨域信息	`WebMvcConfigurer#addCorsMappings`	`WebFluxConfigurer#addCorsMappings`

函数式端点（Functional Endpoints）

Java 函数编程基础

函数式接口 - `@FunctionInterface`

用于函数式接口类型声明的信息注解类型，这些接口的实例被 Lambda 表示式、方法引用或构造器引用创建。函数式接口只能有一个抽象方法，并排除接口默认方法以及声明中覆盖 Object 的公开方法的统计。同时，`@FunctionalInterface` 不能标注在注解、类以及枚举上。如果违背以上规则，那么接口不能视为函数式接口，当标注 `@FunctionalInterface` 后，会引起编译错误。

不过，如果任一接口满足以上函数式接口的要求，无论接口声明中是否标注 `@FunctionalInterface`，均能被编译器视作函数式接口。

消费函数 - `Consumer`

只有输入，没有输出

生产函数 - `Supplier`

没有输入，只有输出

处理函数 - `Function`

有输出，有输出

判定函数 - `Predicate`

判定输入真伪性

映射路由接口 - `RouterFunction`

函数映射

```
RouterFunction<ServerResponse> route =
    route(GET("/person/{id}").and(accept(APPLICATION_JSON)), handler::getPerson)
        .andRoute(GET("/person").and(accept(APPLICATION_JSON)), handler::listPeople)
        .andRoute(POST("/person"), handler::createPerson);
```

注解映射

```
@GetMapping(value="/person/{id}",consumes=APPLICATION_JSON)
public void getPerson(HttpServletRequest request , HttpServletResponse) {

}

@GetMapping(value="/person",consumes=APPLICATION_JSON)
public void listPeople(HttpServletRequest request , HttpServletResponse) {

}

@PostMapping(value="/person")
public void createPerson(HttpServletRequest request , HttpServletResponse) {

}
```

函数处理接口

`Function` 接口

```
public class PersonHandler {

    // ...

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        // ...
    }
}
```

路由方法 - `RouteFunctions#route`

请求判定 - `RequestPredicate`

类似于 `Predicate` ，判断 `ServerRequest`

处理器函数 - `HandlerFunction`

HTTP 请求接口 - `ServerRequest`

HTTP 响应接口- `ServerResponse`

并发模型

Spring 官方说明

Spring MVC 和 Spring WebFlux 均能使用注解驱动 Controller，然而不同点在于并发模型和阻塞特性。

> Both Spring MVC and Spring WebFlux support annotated controllers, but there is a key difference in the concurrency model and default assumptions for blocking and threads.

Spring MVC 通常是 Servlet 应用，因此，可能被当前线程阻塞。以远程调用为例，由于阻塞的缘故，导致 Servlet 容器使用较大的线程池处理请求。

> In Spring MVC, and servlet applications in general, it is assumed that applications *may block* the current thread, e.g. for remote calls, and for this reason servlet containers use a large thread pool, to absorb potential blocking during request handling.

Spring WebFlux 通常是非阻塞服务，不会发生阻塞，因此该阻塞服务器可使用少量、固定大小的线程池处理请求。

> In Spring WebFlux, and non-blocking servers in general, it is assumed that applications *will not block*, and therefore non-blocking servers use a small, fixed-size thread pool (event loop workers) to handle requests.

核心组件

`HttpHandler` API

基本概念

`HttpHandler` 是一种带有处理 HTTP 请求和响应方法的简单契约。

> `HttpHandler` is a simple contract with a single method to handle a request and response.

> Lowest level contract for reactive HTTP request handling that serves as a common denominator across different runtimes.

接口定义

```java
public interface HttpHandler {

   /**
    * Handle the given request and write to the response.
    * @param request current request
    * @param response current response
    * @return indicates completion of request handling
    */
   Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response);

}
```

`WebHandler` API

Bean 类型

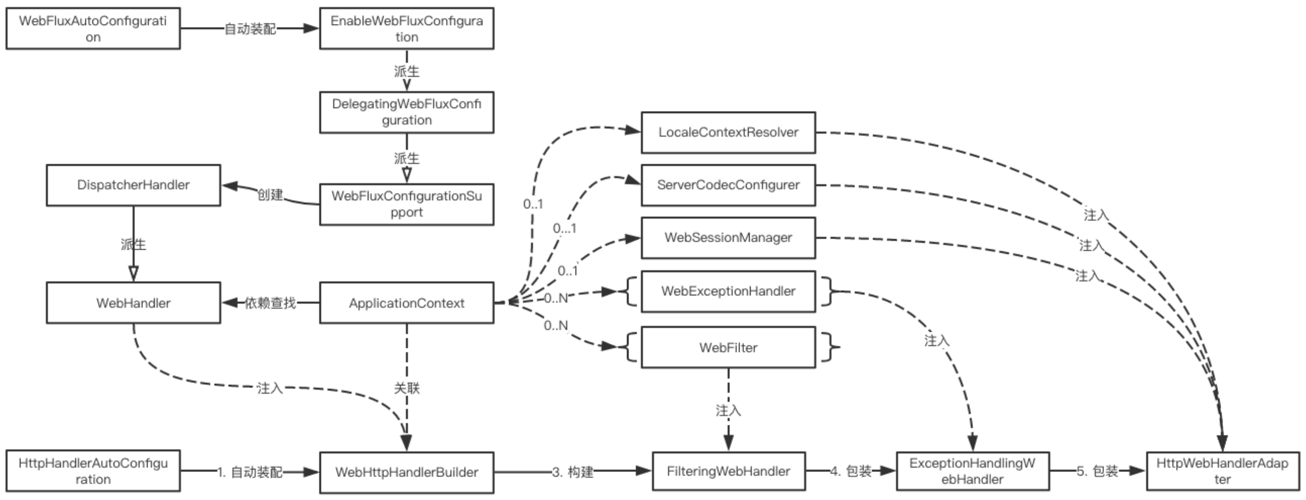
Bean 名称	Bean 类型	数量	描述
	`WebExceptionHandler`	0..N	Provide handling for exceptions from the chain of `WebFilter`'s and the target `WebHandler`. For more details, see Exceptions.
	`WebFilter`	0..N	Apply interception style logic to before and after the rest of the filter chain and the target `WebHandler`. For more details, see Filters.
"webHandler"	`WebHandler`	1	The handler for the request.
"webSessionManager"	`WebSessionManager`	0..1	The manager for `WebSession`'s exposed through a method on `ServerWebExchange`. `DefaultWebSessionManager` by default.
"serverCodecConfigurer"	`ServerCodecConfigurer`	0..1	For access to `HttpMessageReader`'s for parsing form data and multipart data that's then exposed through methods on `ServerWebExchange`. `ServerCodecConfigurer.create()` by default.
"localeContextResolver"	`LocaleContextResolver`	0..1	The resolver for `LocaleContext` exposed through a method on `ServerWebExchange`. `AcceptHeaderLocaleContextResolver` by default.

Web MVC VS. WebFlux

核心组件	Spring Web MVC	Spring WebFlux
前端控制器(Front Controller)	`DispatcherServlet`	`DispatcherHandler`
Handler 请求映射	`o.s.w.servlet.HandlerMapping`	`o.s.w.reactive.HandlerMapping`
Handler 请求适配器	`o.s.w.servlet.HandlerAdapter`	`o.s.w.reactive.HandlerAdapter`
Handler 异常处理器	`o.s.w.servlet.HandlerExceptionResolver`	`HandlerResult.exceptionHandler`
视图处理器	`o.s.w.servlet.ViewResolver`	`o.s.w.reactive.r.v.ViewResolver`
Locale 解析器	`o.s.w.servlet.LocaleResolver` `LocaleContextResolver`	`LocaleContextResolver`
`@Enable` 模块注解	`@EnableWebMvc`	`@EnableWebFlux`
自定义配置器	`WebMvcConfigurer`	`WebFluxConfigurer`
内容协商配置器	`ContentNegotiationConfigurer`	`RequestedContentTypeResolverBuilder`
内容协商管理器	`ContentNegotiationManager`	无
内容协商策略	`ContentNegotiationStrategy`	`RequestedContentTypeResolver`
资源跨域注册器	`o.s.w.servlet.c.a.CorsRegistry`	`o.s.w.reactive.c.CorsRegistry`
`HanderMethod` 参数解析器	`o.s.w.m.s.HandlerMethodArgumentResolver`	`o.s.w.reactive.r.m.HandlerMethodArgumentResolver`
`HanderMethod` 返回值解析器	`HandlerMethodReturnValueHandler`	`HandlerResultHandler`

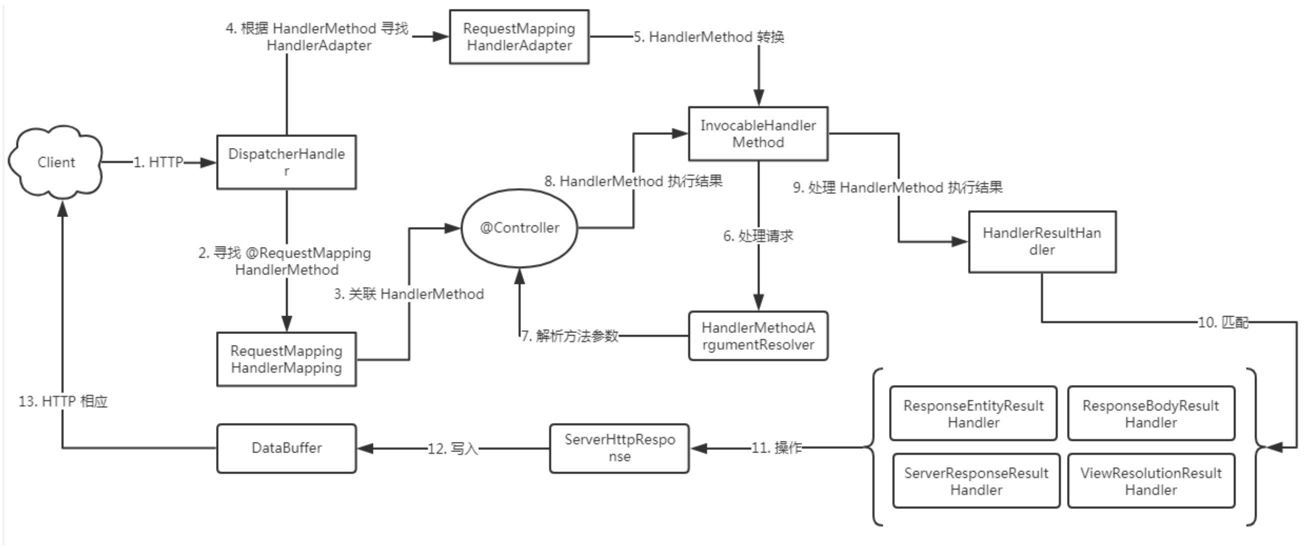
执行流程

核心组件初始化流程



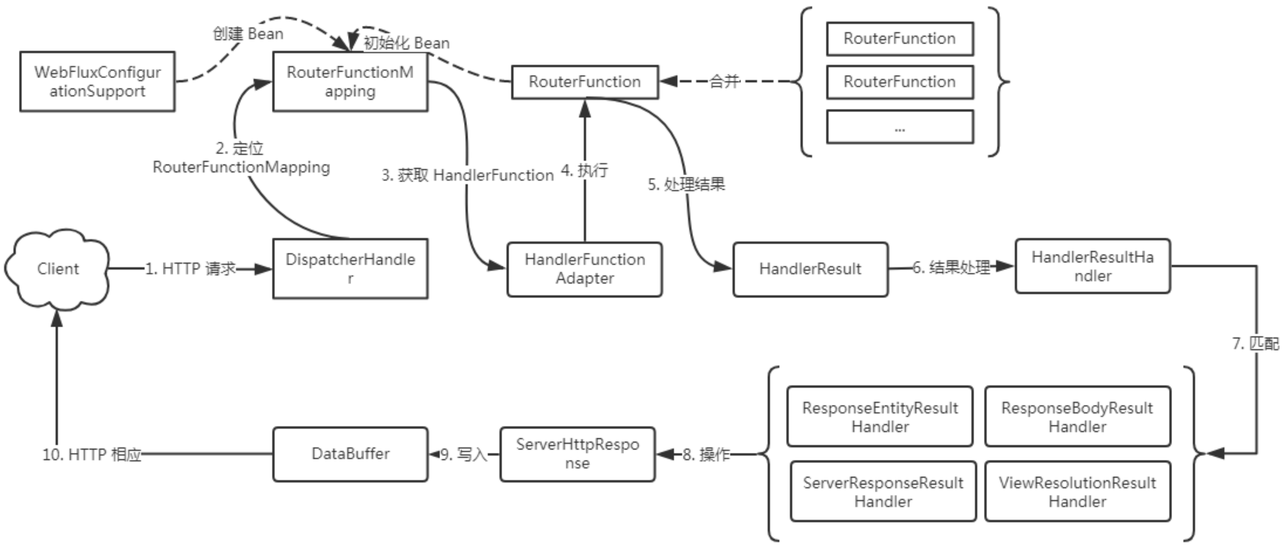
核心组件请求处理流程

注解驱动（ Annotated Controllers ）组件请求处理流程


Client
1. HTTP
DispatcherHandler
2. 寻找 @RequestMapping HandlerMethod
RequestMapping HandlerMapping
3. 关联 HandlerMethod
4. 根据 HandlerMethod 寻找 HandlerAdapter
RequestMapping HandlerAdapter
5. HandlerMethod 转换
InvocableHandler Method
@Controller
6. 处理请求
HandlerMethodArgumentResolver
7. 解析方法参数
8. HandlerMethod 执行结果
9. 处理 HandlerMethod 执行结果
HandlerResultHandler
10. 匹配
ResponseEntityResult Handler
ResponseBodyResult Handler
ServerResponseResult Handler
ViewResolutionResult Handler
11. 操作
ServerHttpResponse
12. 写入
DataBuffer
13. HTTP 相应

> 与 Spring Web MVC 流程类似

函数式端点（ Functional Endpoints ）组件请求处理流程



使用场景

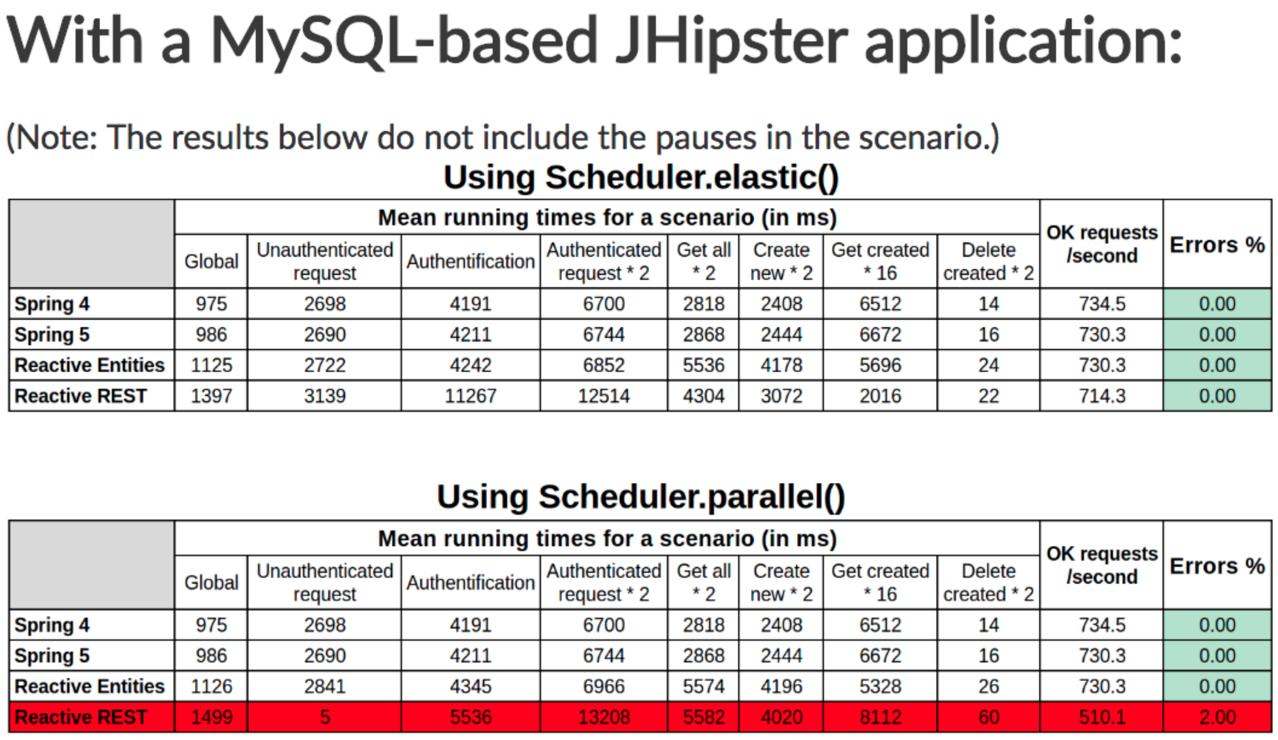
性能考虑

Spring 官方说明

> Performance has many characteristics and meanings. Reactive and non-blocking generally do not make applications run faster. They can, in some cases, for example if using the `WebClient` to execute remote calls in parallel. On the whole it requires more work to do things the non-blocking way and that can increase slightly the required processing time.
>
> The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load because they scale in a more predictable way. In order to observe those benefits however you need to have some latency including a mix of slow and unpredictable network I/O. That's where the reactive stack begins to show its strengths and the differences can be dramatic.

JHipster WebFlux 性能测试报告

基于 MySQL 的 JHipster 应用



With a MySQL-based JHipster application:

(Note: The results below do not include the pauses in the scenario.)

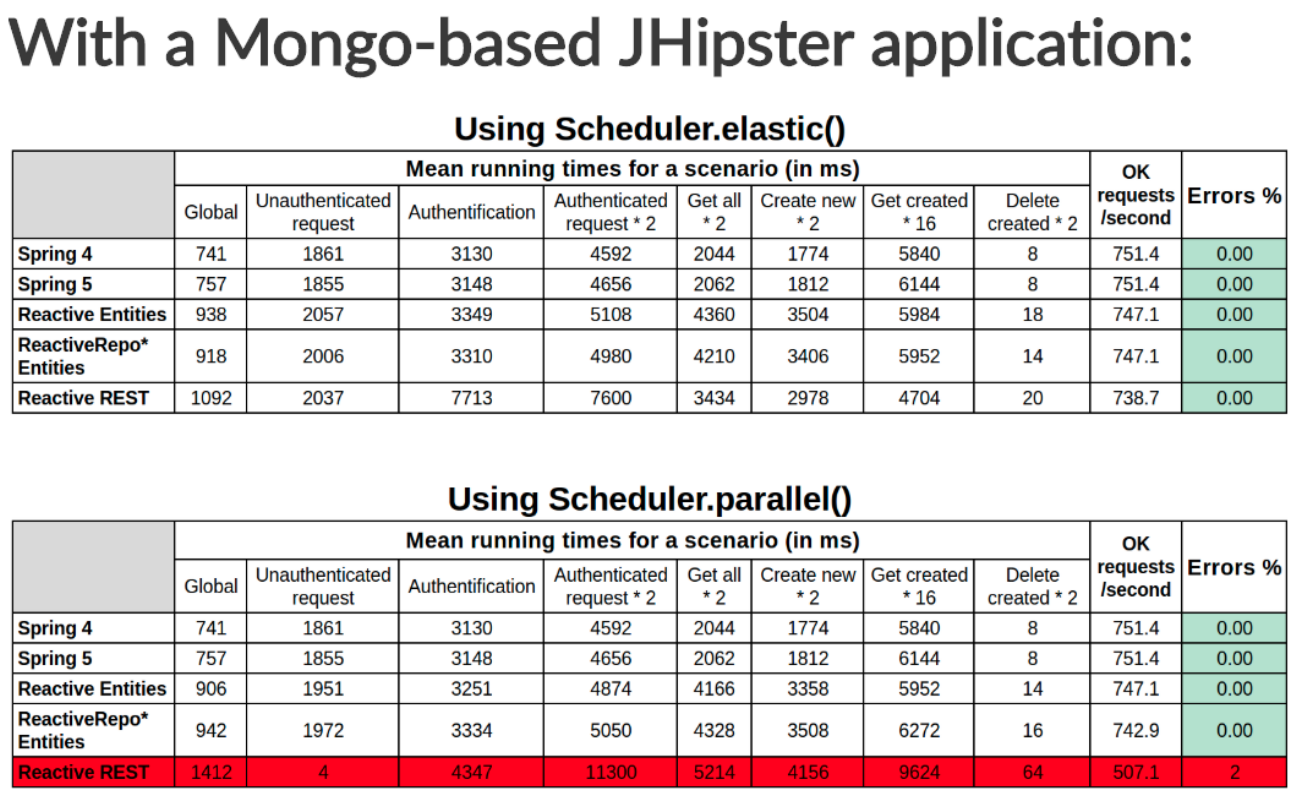
Using Scheduler.elastic()

	Mean running times for a scenario (in ms)								OK requests /second	Errors %
	Global	Unauthenticated request	Authentification	Authenticated request * 2	Get all * 2	Create new * 2	Get created * 16	Delete created * 2		
Spring 4	975	2698	4191	6700	2818	2408	6512	14	734.5	0.00
Spring 5	986	2690	4211	6744	2868	2444	6672	16	730.3	0.00
Reactive Entities	1125	2722	4242	6852	5536	4178	5696	24	730.3	0.00
Reactive REST	1397	3139	11267	12514	4304	3072	2016	22	714.3	0.00

Using Scheduler.parallel()

	Mean running times for a scenario (in ms)								OK requests /second	Errors %
	Global	Unauthenticated request	Authentification	Authenticated request * 2	Get all * 2	Create new * 2	Get created * 16	Delete created * 2		
Spring 4	975	2698	4191	6700	2818	2408	6512	14	734.5	0.00
Spring 5	986	2690	4211	6744	2868	2444	6672	16	730.3	0.00
Reactive Entities	1126	2841	4345	6966	5574	4196	5328	26	730.3	0.00
Reactive REST	1499	5	5536	13208	5582	4020	8112	60	510.1	2.00

基于 Mongo 的 JHipster 应用



With a Mongo-based JHipster application:

Using Scheduler.elastic()

	Mean running times for a scenario (in ms)								OK requests /second	Errors %
	Global	Unauthenticated request	Authentification	Authenticated request * 2	Get all * 2	Create new * 2	Get created * 16	Delete created * 2		
Spring 4	741	1861	3130	4592	2044	1774	5840	8	751.4	0.00
Spring 5	757	1855	3148	4656	2062	1812	6144	8	751.4	0.00
Reactive Entities	938	2057	3349	5108	4360	3504	5984	18	747.1	0.00
ReactiveRepo* Entities	918	2006	3310	4980	4210	3406	5952	14	747.1	0.00
Reactive REST	1092	2037	7713	7600	3434	2978	4704	20	738.7	0.00

Using Scheduler.parallel()

	Mean running times for a scenario (in ms)								OK requests /second	Errors %
	Global	Unauthenticated request	Authentification	Authenticated request * 2	Get all * 2	Create new * 2	Get created * 16	Delete created * 2		
Spring 4	741	1861	3130	4592	2044	1774	5840	8	751.4	0.00
Spring 5	757	1855	3148	4656	2062	1812	6144	8	751.4	0.00
Reactive Entities	906	1951	3251	4874	4166	3358	5952	14	747.1	0.00
ReactiveRepo* Entities	942	1972	3334	5050	4328	3508	6272	16	742.9	0.00
Reactive REST	1412	4	4347	11300	5214	4156	9624	64	507.1	2

结论

- 没有明显的速度提升（甚至性能结果稍微更恶劣）

> *No improvement in speed was observed with our reactive apps (the Gatling results are even slightly worse).*

- 关注编程用户友好性，Reactive 编程尽管没有新增大量的代码，然而编码（和调试）却是变得更为复杂

> *Concerning user-friendliness, reactive programming does not add a lot of new code, but it certainly is a more complex way of coding (and debugging...). A quick Java 8 refresher might be required.*

- 现在面临的最大问题是缺少文档。在生成测试应用中，它已经给我们造成了最大障碍，并使得我们可能已经缺少了关键点。因此，我们并不会太快地投入 Reactive 编程，同时等待关于它的更多反馈。因此，Spring WebFlux 尚未证明自身明显地优于 Spring MVC。

> *The main problem right now is the lack of documentation. It has been our greatest obstacle in generating test apps, and we may have missed a crucial point because of that.We therefore advise not to jump too quickly on reactive programming and wait for more feedback. Spring WebFlux has not yet proved its superiority over Spring MVC.*

编程模型考虑

注解驱动编程模型

函数式编程模型

并发模型考虑