1. Spring WebFlux

> 以前的web框架基于Servlet API设计运行在Servlet容器中。比如Spring Framework，Spring Web MVC。
>
> 响应式web框架，Spring WebFlux在5.0以后加入到了Spring框架中。这是完全非阻塞，支持响应式流背压，运行在Netty，Undertow以及Servlet3.1+ 的容器中

> spring-webmvc和 spring-webflux现在都是可以使用的。
>
> 现在有基于WebClient的Spring MVC controllers

1.1 概览

为什么要有Spring WebFlux

一部分答案是我们需要一个非阻塞的web技术栈，用少量线程处理大量并发，并且扩容的时候占用更少的硬件资源。Servlet3.1提供了非阻塞I/O 相关API。但是，使用这种会离rest越来越远，使用一些同步操作（Filter、Servlet）或阻塞操作（getParameter，getPart）。这是创建新的通用API的动机，能构建出非阻塞的运行时。Netty在异步、非阻塞领域做得很好，所以有必要做基于这些的非阻塞运行时

另一部分答案是函数式编程。就像Java5注解创建出新机遇（注解式REST Controller，单元测试），Java8的lambda表达式创建了函数式API的机遇。这是 非阻塞应用、连续调用风格API的新时代（CompletableFuture、ReactiveX），他们允许声明式组合异步逻辑。在编程模型层面，Java8开启了Spring WebFlux提供基于注解的函数式web端。

1.1.1、定义“Reactive”

我们经常说 非阻塞、函数式，但是 Reactive 是什么？响应式是什么？

响应式：是指一种编程模型，围绕响应式变化构建，比如：网络组建响响应 I/O事件，UI控制器响应鼠标事件，等等。这种感觉，非阻塞是响应式，因为，代替阻塞，现在随着操作完成，或者数据可用，我们需要响应式通知其他人。

在Spring团队中还有另一种重要的机制与“reactive”相关，这是非阻塞背压。在同步的命令式代码中，阻塞调用是一种自然的背压形式，迫使调用者等待。在非阻塞代码中，控制事件速率很重要，以使快速生产者不会压跨目的地。

`Reactive Stream`是一个小规范（reactive-streams-jvm/README.md at master · reactive-streams/reactive-streams-jvm · GitHub，Java9 也采用了这个东西），他定义异步组件和背压之间的关系。比如：一个数据库（Publisher）可以生产数据，HTTP Server（Subscriber）可以响应。Reactive Stream的主要目的就是让 subscriber控制 publisher 生产数据的快慢

> 问题：如果publisher 不能减速怎么办?
>
> Reactive Streams 的目标仅仅是建立这种机制和边界。如果publisher不能减速，就必须决定使用buffer，drop还是fail机制

1.1.2、 Reactive API

Reactive Streams 在 操作性 方面扮演重要角色。他很感兴趣 库 和基础设施组件但是很少用作 application API，因为太底层了。Applications 需要一个 更高层更丰富、函数式API去组合异步逻辑---就像Java8的stream API一样，但不仅仅是操作集合。这才是 响应式库需要处理的问题。

Reactor就是一个响应式库，Spring WebFlux就是用他实现的。他提供 Mono和Flux 操作类型，Mono是0..1序列，Flux是0..N序列，还有一些丰富的操作符。Reactor是响应式流的一个库，因此，所有操作符都支持非阻塞背压。Reactor 更关注服务端java，并和Spring团队紧密合作

WebFlux 需要Reactor作为核心依赖，但是也可以操作reactive streams定义的其它库。作为通用规则，WebFlux API结构一个 Publiser 作为输入，适配到 Reactor内部的类型，使用并返回Flux或者Mono作为输出。因此，你可以传递任何 Publisher 作为输入并且可以在输出上使用任何操作符，但是你如果使用其他reactor库，需要自己适配。WebFlux可以使用 RxJava或其他响应式库、参考：https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-reactive-libraries

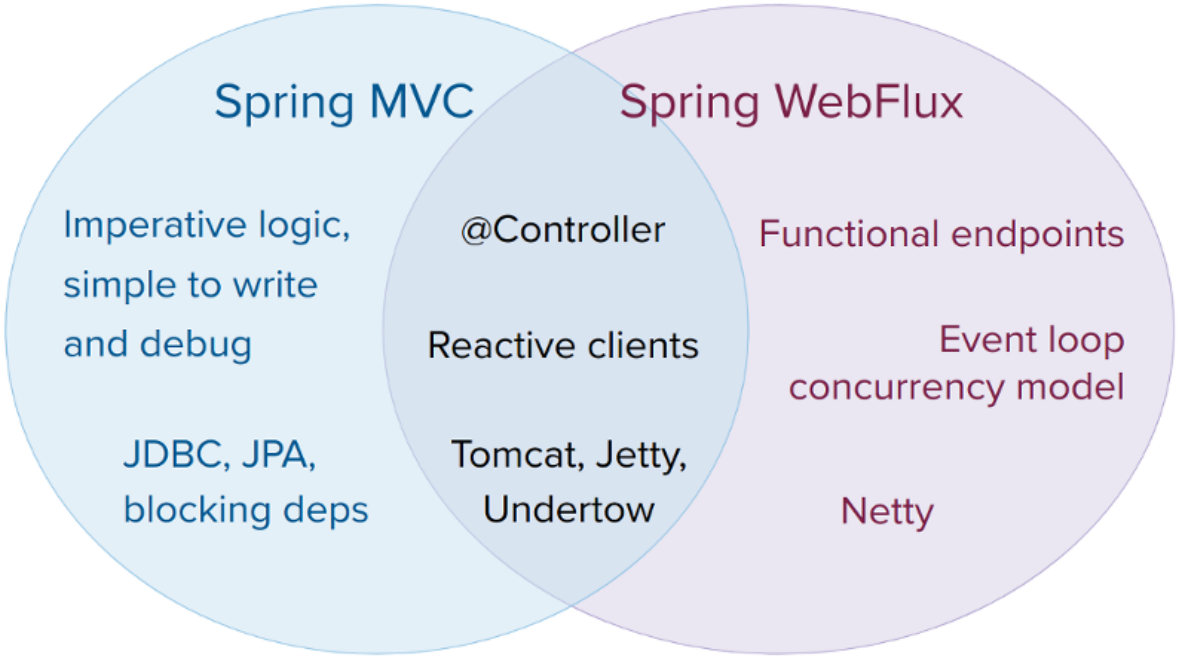
1.1.3、编程模型

`spring-web`模块包含响应式组件，如：Spring WebFlux，包含HTTP抽象，Reactive Streams 适配服务器，编码，核心 WebHandler API 就像是 Servlet API 一样，但是是非阻塞的。

这个基础上，Spring WebFlux提供两种编程模型：

- 基于注解的Controller： 以前SpringMVC的那些注解都能继续使用
- 函数式后端：基于Lambda，轻量级，函数式编程模型。可以认为这个就是一个小型库或者工具集，来让应用路由请求并处理。与注释控制器的最大区别在于，函数式负责从头到尾的请求处理，而不是通过注解来声明意图并被回调。

1.1.4、 适用性



> 推荐用的纯粹一点。要么右边，要么左边。

建议考虑如下规范：

- 如果已有的SpringMVC应用稳定工作，无需变化。就用这个算了。
- 如果准备使用非阻塞式技术栈。Spring WebFlux提供同样的执行模型。你可以选择 ServletAPI编程方式，也可以选择响应式方式（Reactor，RxJava等）
- 如果感兴趣轻量级，函数式web框架。可以使用Spring WebFlux 函数式web 后端。非常适合构建小应用或者微服务，以最小的依赖跑起来。
- 微服务架构中，可能需要混合编程方式使用MVC或者Web Flux。可以直接用以前MVC注解的方式无缝迁移
- 简单衡量，如果是 JPA，JDBC，或者网络API，使用Spring MVC就很好。响应式也可以直接改造你的同步调用，但是意义不大
- 如果你的MVC应用调用远程服务，就用 响应式的 WebClient。返回响应式对象。Spring MVC可以直接用这些响应式组件。这样还能获得更强的吞吐能力
- 你的团队够大的话，要小心 非阻塞，函数式，声明式编程的陡峭学习曲线。一个实际的场景，无需完全切换编码方式，就是使用响应是的WebClient改造以前的远程调用就好。除此之外，可以小步迭代，权衡利弊。我们期望，对于整个应用来说，这个迁移是不必要的。如果不确定你想要什么，请先学习非阻塞I/O的工作方式（比如Node.js的单线程高并发）和他产生的效果

1.1.5、服务器

Spring WebFlux 支持Tomcat，Jetty, Servlet 3.1+ containers，以及 非Servlet运行环境的 Netty，Undertow等。所有服务器都适配到底层通用API，因为只需要使用高层编程模型，就可以支持各种服务器运行环境了。

Spring WebFlux 没有内置支持服务器启动停止。然而，可以就用几行代码配置完成WebFlux基础设并运行应用。

Spring WebFlux starter自动做了如下配置。默认使用Netty，也很容易切换到Tomcat，Jetty或者Undertow等，只需要改变maven依赖就行了。默认使用Netty是因为在异步、非阻塞领域，netty很流行。

Tomcat和Jetty 可以在Spring MVC 和WebFlux中使用，记住，这两种方式完全不同。SpringMVC 是依赖Servlet阻塞式IO并且应用直接使用Servlet API的。Sring WebFlux 依赖Servlet3.1 非阻塞I/O并且绑定ServletAPI的底层。这个每个直接暴露使用。

Undertow，SpringWebFlux直接就不用ServletAPI了。

1.1.6、性能（Performance）

性能有多方面表现。响应式和非阻塞通常并不能让应用运行的更快。在某些场景下（比如：使用WebClient进行远程调用）可以更快。整体来看，他需要更多的工作去处理非阻塞的方式以至于可能会轻微增加整个处理时间。

响应式的关键提升是他以一个小的扩展，固定数量的线程和更少的内存占用。这是的应用程序在负载下更弹性，因为他们是以更平滑的方式扩展的（比如：原来1w请求进来1w线程处理，现在1w请求进来100处理，10w进来1000处理，看着更平滑，而不是一个陡峭的曲线）。但是，为了观察这些好处，您需要有一些延迟（包括缓慢且不可预测的网络I/O的混合）。这就是反应性堆栈开始表现出其优势的地方，并且差异可能是戏剧性的。

1.1.7、并发模型

SpringMVC和Spring WebFlux都支持注解Controller。并发模型和阻塞和线程的默认假设有一个关键差异。

- SpringMVC中，应用程序可以阻塞当前线程（如：远程调用）。因此，Servlet容器会使用大量线程池去吸收请求期间的阻塞处理。
- 在Spring WebFlux中，应用不阻塞。因此，非阻塞服务器使用少量，固定的线程池（event loop worker）去处理请求

> “缩放(scale)”和“少量线程”听起来可能是矛盾的，但是永远不要阻止当前线程（并且依靠回调）意味着您不需要额外的线程，因为没有阻塞式调用。

执行阻塞式API

> - 用Reactor想执行阻塞式API，可以用publishOn方法，可以继续处理使用不同线程。但是记住，阻塞式API不适合这种并发模型

Mutable State（可变状态）

> 在Reactor和Rxjava中，我们通过操作符申明逻辑。在运行时，形成响应式管道，数据有序的通过管道，经历各种不同阶段。这样做的一个关键好处是，它使应用程序不必保护**可变状态**，因为该管道中的应用代码永远不会同时调用。

Threading Model（线程模型）

> Spring WebFlux 运行的时候，能看到什么线程?
>
> - 在Spring WebFlux服务器`vanilla`（如：没有数据库访问，没有其他依赖），你能看到一个服务器线程，和其他请求处理线程（一般是CPU核心数量大小）。Servlet容器，启动的时候可能会更多（Tomcat有10个），支持servlet的阻塞式IO和servlet3.1的非阻塞式IO。
> - 响应式 WebClient 操作符是 event loop 风格的。能看到小的，固定数量的关联线程（比如：`reactor-http-nio-`是Reactor Netty 的连接器）。但是，如果Reactor Netty被用来既当做Client又当做Server，就会默认有两个event loop的线程
> - Reactor和RxJava提供线程池抽象，叫做 Schedulers，使用 publishOn操作符可以切换进程到不同线程池。schedulers 有一些不同的并发策略名字：如：`paralle`（基于CPU数量的有限线程），`elastic`（基于I/O的大量线程）。如果你看到这些线程名，代表某些代码使用了指定的Schedulers线程池策略
> - 数据访问库和其他第三方依赖也可以创建和使用他们自己的线程

1.2 Reactor 核心

> spring-web 模块，包含以下基础功能支持响应式应用

- 服务器端，请求处理的两种级别支持
  - HttpHandler： 非阻塞IO方式支持HTTP请求处理以及Reactive Stream的背压。适配了Reactor Netty，Undertow，Tomcat，Jetty和任何Servlet3.1+容器
  - WebHandler API：略微高级的API，常用的请求处理webAPI，支持更上层的编程模型，比如注解Controller和函数式后端。

- 客户端，有一个基本的 `ClientHttpConnector` 来发送HTTP请求以非阻塞的方式并支持Reactive Stream 的背压，也适配了 Reactor Netty, reactive Jetty HttpClient ，和 Apache HttpComponents。更高阶的 WebClient 也可以用来做事
- 客户端和服务器。codecs 用来序列化和反序列化 HTTP请求并响应内容。

1.2.1、HttpHandler

httphandler是一项简单的约定，具有一种处理请求和响应的单一方法，是故意降低使用难度的。它的主要目的是在不同的HTTP服务器API上成为最小的抽象。

> 支持的serve APIs

Server name	Server API used	Reactive Streams support
Netty	Netty API	Reactor Netty
Undertow	Undertow API	spring-web: Undertow to Reactive Streams bridge
Tomcat	Servlet 3.1 non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Jetty	Servlet 3.1 non-blocking I/O; Jetty API to write ByteBuffers vs byte[]	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge
Servlet 3.1 container	Servlet 3.1 non-blocking I/O	spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

> 服务器依赖； 支持的版本

Server name	Group id	Artifact name
Reactor Netty	io.projectreactor.netty	reactor-netty
Undertow	io.undertow	undertow-core
Tomcat	org.apache.tomcat.embed	tomcat-embed-core
Jetty	org.eclipse.jetty	jetty-server, jetty-servlet

参照如下代码片段

Reactor Netty

```
HttpHandler handler = ...
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create().host(host).port(port).handle(adapter).bind().block();
```

Undertow

```
HttpHandler handler = ...
UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);
Undertow server = Undertow.builder().addHttpListener(port,
host).setHandler(adapter).build();
server.start();
```

Tomcat

```
HttpHandler handler = ...
Servlet servlet = new TomcatHttpHandlerAdapter(handler);

Tomcat server = new Tomcat();
File base = new File(System.getProperty("java.io.tmpdir"));
Context rootContext = server.addContext("", base.getAbsolutePath());
Tomcat.addServlet(rootContext, "main", servlet);
rootContext.addServletMappingDecoded("/", "main");
server.setHost(host);
server.setPort(port);
server.start();
```

Servlet 3.1+ Container

`AbstractReactiveWebInitializer`

1.2.2、 WebHandler API

`org.springframework.web.server` 包建立了一个 `HttpHandler` 提供通用的web API请求处理功能，请求会通过多个 `WebExceptionHandler` 链路，多个 `WebFilter`，并且单个 `WebHandler` 组件。这个链路能在 `WebHttpHandlerBuilder` 一起构建出来，通过简单指向Spring自动注入的 `ApplicationContext` 就可以。

`HttpHandler` 的简单目标是抽象不同的HTTP servers， `WebHandler` API的目标是提供一个更广泛功能特性：

- 用户Sesssion和属性
- 请求属性
- 请求的 `Locale` 和 `Principal`
- 访问解析的、缓存形式的数据
- 抽象 多部件上传
- ...

Special bean types(特定的Bean类型)

`WebHttpHandlerBuilder` 自动探测Spring的ApplicationContext，并直接注册如下组件

Bean name	Bean type	Count	Description
	`WebExceptionHandler`	0..N	Provide handling for exceptions from the chain of `WebFilter` instances 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` instances exposed through a method on `ServerWebExchange`. `DefaultWebSessionManager` by default.
`serverCodecConfigurer`	`ServerCodecConfigurer`	0..1	For access to `HttpMessageReader` instances for parsing form data and multipart data that is 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.
`forwardedHeaderTransformer`	`ForwardedHeaderTransformer`	0..1	For processing forwarded type headers, either by extracting and removing them or by removing them only. Not used by default.

Form Data（表单数据）

`ServerWebExchange` 使用如下API访问表单数据

```
Mono<MultiValueMap<String, String>> getFormData();
```

`DefaultServerWebExchange` 使用配置好的 `HttpMessageReader` 去解析表单数据（application/x-www-form-urlencoded）为一个 `MultiValueMap`。 默认的 `FormHttpMessageReader` 会被 `ServerCodecConfigurer` 组件配置好。（参照 Web Handler API）

Multipart Data（多部件上传）

`ServerWebExchange` 使用如下API访问文件项

```
Mono<MultiValueMap<String, Part>> getMultipartData();
```

`DefaultServerWebExchange` 使用配置好的 `HttpMessageReader<MultiValueMap<String, Part>>` 去解析 `multipart/form-data` 内容为一个 `MultiValueMap`。默认的，这是 `DefaultPartHttpMessageReader` 做的事情，没有任何第三方依赖。或者，`SynchronossPartHttpMessageReader` 也可以使用，这是基于 Synchronoss NIO Multipart 的库。这两个都被 `ServerCodecConfigurer` 配置进容器了（参照 Web Handler API）

解析multipart 数据流，可以使用 `Flux<Part>` 类型的返回值，`HttpMessageReader<Part>`。比如：一个注解Controller，使用 `@RequestPart` 标注 `Map`，就是想按照name访问multipart，因此，需要需要完全解析multipart data。相比之下，可以使用 `@RequestBody` 去解码内容为 `Flux<Part>` 无需收集成一个 `MultiValueMap`。

Forwarded Headers（转发头）

请求通过代理（如负载均衡），Host，port，和scheme都可能发生变化。这将使得我们获取客户端正确的host，port,scheme等成为一个挑战。

RFC 7239 定义了一个`Forwarded`头，让代理可以使用原始请求提供的信息。也有一些非标准的请求头，比如：`X-Forwarded-Host`，`X-Forwarded-Port`,`X-Forwarded-Proto`，`X-Forwarded-Ssl`,`X-Forwarded-Prefix`。

`ForwardedHeaderTransformer`就是一个组件来修改请求的host，port,schema，基于forwarded头，然后移除这些头。如果你声明了一个名叫`forwardedHeaderTransformer`的组件，他就会被自动检查到并使用。

有针对 forwarded headers 的安全考虑，因为应用程序不知道该 header 是由代理人，还是由恶意客户端添加的。这就是为什么应该在边界代理上配置，以删除从外部传来的不信任的转发流量的原因。我们也可以配置`ForwardedHeaderTransformer`用`removeOnly=true`，这种情况下，他会删除并且不使用这些头。

> 5.1 `ForwardedHeaderFilter`过期了，并且使用`ForwardedHeaderTransformer`替代，因此forwarded header 可以被更早的处理，在exchange 创建之前。如果 这个filter以任何方式被配置了，它被从filter列表中取出，并且`ForwardedHeaderTransformer`会替代`ForwardedHeaderFilter`

1.2.3、Filters

在`WebHandler API`中，您可以使用`WebFilter`在过滤器和目标`WebHandler`的其余处理链之前和之后编写 拦截式逻辑。当使用 WebFlux Config, 注册一个`WebFilter`，和声明一个Spring Bean一样简单，也可以使用`@Order`或者实现`Ordered`接口来指定顺序。

CORS

Spring WebFlux提供CORS配置的细粒度支持，通过Controllers的注解。但是，当你使用Spring Security的时候，我们建议使用内置的`CorsFilter`，这种必须放在Spring Security 的链路Filters之前。

参照 CORS 和 CORS `WebFilter` 更多细节；

1.2.4、Exceptions

在`WebHandler API`中，可以使用`WebExceptionHandler`去处理`WebFilter`整个链路和目标处理器`WebHandler`的异常。当使用 WebFlux Config,注册一个`WebExceptionHandler`，也和注册SpringBean方式一样，依然可以用`@Order`或者实现`Ordered`来指定顺序。

`WebExceptionHandler`的一些实现如下：

Exception Handler	Description
`ResponseStatusExceptionHandler`	Provides handling for exceptions of type `ResponseStatusException` by setting the response to the HTTP status code of the exception.
`WebFluxResponseStatusExceptionHandler`	Extension of `ResponseStatusExceptionHandler` that can also determine the HTTP status code of a `@ResponseStatus` annotation on any exception.This handler is declared in the WebFlux Config.

1.2.5、Codecs

`spring-web` and `spring-core` 模块提供 序列化与反序列化功能，来让字节内容和高阶的对象通过非阻塞IO的方式进行互转。支持如下特性：

- `Encoder` and `Decoder` 是一个底层API去编解码内容，独立于HTTP
- `HttpMessageReader` and `HttpMessageWriter` 是编解码HTTP消息内容的规范
- 一个`Encoder`可以被`EncoderHttoMessageWriter`保证去在我们的web应用中使用，`Decoder`同样可以被`DecoderHttpMessageReader`包装。
- `DataBuffer` 抽象代表不同字节缓存区（如：Netty的ByteBuf，java.nio.ByteBuffer，等）并且这是所有编解码工作的地方。可以参照 Data Buffers and Codecs

`spring-core`模块提供 `byte[]`, `ByteBuffer`, `DataBuffer`, `Resource`, and `String` 的 encoding和decoder实现。 `spring-web`模块提供Jackson JSON, Jackson Smile, JAXB2, Protocol Buffers等其他encoder和decoder实现，使用HTTP message reader和writer 处理表单数据，文件上传，SSE（server-sent-events）等。

Jackson JSON

JSON和binary JSON (Smile) 在jackson库里面都支持了。

`Jackson2Decoder`工作如下：

- jackson的异步非阻塞解析器用于将一系列字节块汇总到代表json对象的tokenbuffer中
- 每个TokenBuffer都传递给Jackson的ObjectMapper，以创建一个更高的级别对象。
- 当解码到单值发布者Publisher（例如Mono）时，就只有一个`TokenBuffer`。
- 当解码到多值发布者Publisher（例如Flux）时，每个`TokenBuffer`传递给`ObjectMapper`，会使用足够形成对象的足够字节。输入内容可以是一个JSON数组，或者任何 line-delimited JSON 格式，比如（NDJSN，JSON Lines或者 JSON Text Sequeneces）

`Jackson2Encoder`工作如下：

- 单值发布者Publisher（如Mono），通过`ObjectMapper`简单序列化即可
- 多值发布者Publisher以`application/json`，默认用`Flux#collectToList()`收集所有值并且序列化这个结果集。
- 多值发布者Publisher以`application/x-ndjson` or `application/stream+x-jackson-smile`，会使用 line-delimited JSON 格式编码，写出，并且刷出每个单独值。其他流类型可能会被注册的其他encoder处理。

- SSE，`Jackson2Encoder`会执行每个事件并输出、刷出，保证派发没有延迟。

> 默认，`Jackson2Encoder` and `Jackson2Decoder`都不支持String。`CharSequenceEncoder`是处理String或者文本的。如果需要渲染一个JSON数组从`Flux<String>`中，就可以使用`Flux$collectToList()`并编码成一个`Mono<List<String>>`

Form Data

`FormHttpMessageReader` and `FormHttpMessageWriter`支持编解码`application/x-www-form-urlencoded`内容。

服务端表单内容经常会被在各种地方访问。`ServerWebExchange`提供一个`getFormData()`专用方法去解析内容通过`FormHttpMessageReader`并缓存结果为各种重复访问。参照`WebHandler` API 章节的Form Data

当`getFormData()`被使用，原始的内容就不能再被从request body中读取了。基于这个原因，应用希望通过`ServerWebExchange`一致的访问和缓存表单数据，而不是获取request body的原始数据。

Multipart

`MultipartHttpMessageReader` and `MultipartHttpMessageWriter`支持编解码`multipart/form-data`. 反过来，`MultipartHttpMessageReader`代理其他`HttpMessageReader`真正的解析数据成 一个`Flux<Part>`并且简单收集成一个`MultiValueMap`. 默认，`DefaultPartHttpMessageReader`会被使用，但是可以通过`ServerCodecConfigurer`改变。更多`DefaultPartHttpMessageReader`的信息参照javadoc of `DefaultPartHttpMessageReader`.

访问多部件表单的时候，可以额使用`ServerWebExchange`提供的`getMultipartData()`这个专门的方法。

`getMultipartData()`一旦被调用。原始内容就不可读了。基于这个原因，应用可以使用`getMultipartData()`一致性的重复访问数据，map方式的访问parts或者其他依赖SynchronossPartHttpMessageReader 第一次访问`Flux<Part>`

Limits

在服务端，`Decoder` and `HttpMessageReader`可以配置buffer在内存中的最大大小。通过他两个暴露的`maxInMemorySize`属性就可以调整。`ServerCodecConfigurer`也可以提供在一处修改所有codecs的功能。在客户端，可以使用WebClient.Builder指定大小

Multipart parsing `maxInMemorySize`属性限制非文件部件的大小。文件部件，决定于磁盘写的时候阈值。file项写到磁盘的时候，有一个额外的属性`maxDiskUsagePerPart`去限制每个part占用的磁盘总空间。`maxParts`属性限制一个文件上传请求的文件part数量。这三个在WebFLux中要配置的话，应该在`ServerCodecConfigurer`中修改`MultipartHttpMessageReader`的实例配置。

Streaming

当响应HTTP流的时候（比如：`text/event-stream`，`application/x-ndjson`），定期发送数据很重要，为了尽可能快的自动发现断开连接的客户端。可能会发送comment-only，空的SSE事件或任何其他可以作为心跳的“ NO-OP”数据。

DataBuffer

`DataBuffer`代表WebFlux中的一个字节缓存区。参照Data Buffers and Codecs.，这是Spring的核心。理解他的关键点在于，一些服务器如Netty，byte buffer是池化的并且有引用计数，并且必须在消费后释放掉，以防止内存泄漏。

WebFlux应用程序通常不需要关注此类问题，除非它们直接消费或生产数据缓冲区，而不是依靠编解码器转换和从更高级别的对象转换，或者除非他们选择创建自定义编解码器。

参考 Data Buffers and Codecs,特别是： Using DataBuffer.

1.2.6、Logging

`DEBUG`级别在Spring WebFlux中就能输出很多有用信息。

`TRACE`级别更详细。

Log Id

threadID在webflux中意义不大，因为一个请求可能会被多个线程执行，所以日志 默认是以request指定的特殊ID前缀的。

服务端。这个ID存在`ServerWebExchange`的`LOG_ID_ATTRIBUTE`属性中。完整的格式用`ServerWebExchange#getLogPrefix()`获取。

对于`WebClient`客户端，log ID存在`ClientRequest`的(`LOG_ID_ATTRIBUTE`)属性中。用`` `ClientRequest#logPrefix()`.` ``获取完成格式。

Sensitive Data

`DEBUG`和`TRACE`能打印出敏感信息。这就是为什么默认情况下将表单参数和标头掩盖的原因，您必须明确启用其日志。

服务端：

```
@Configuration
@EnableWebFlux
class MyConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true);
    }
}
```

客户端：

```
Consumer<ClientCodecConfigurer> consumer = configurer ->
        configurer.defaultCodecs().enableLoggingRequestDetails(true);

WebClient webClient = WebClient.builder()
        .exchangeStrategies(strategies -> strategies.codecs(consumer))
        .build();
```

Appenders

SLF4J和Log4j2提供异步日志记录器去避免阻塞。尽管这些缺点，例如潜在的删除消息，这些消息无法排队以伐木，但它们是当前可用于反应性，非阻滞应用程序的最佳选择。

Custom codecs

应用可以注册自定义codecs来支持更多的媒体类型，或者特殊行为是不被默认codecs支持的。

enforcing buffering limits or logging sensitive data.

客户端：

```
WebClient webClient = WebClient.builder()
        .codecs(configurer -> {
                CustomDecoder decoder = new CustomDecoder();
                configurer.customCodecs().registerWithDefaultConfig(decoder);
        })
        .build();
```

1.3 DispatcherHandler

Spring WebFlux和SpringMVC设计一样，都有一个前端控制器来处理请求。在webflux中这个是`DispatcherHandler`，这是一个中央控制器`WebHandler`。

`DispatcherHandler`从ioc容器中自动发现组件。如果`DispatcherHandler`用一个`webHandler`的名字注册到容器中，那就会发现`WebHttpHandlerBuilder`。

Spring在webflux应用中的配置包含如下：

- `DispatcherHandler`组件，名为`webHandler`
- `WebFilter`和`WebExceptionHandler`组件
- `DispatcherHandler` special beans
- 其他...

使用`WebHttpHandlerBuilder`构建请求处理链如下：

```
ApplicationContext context = ...
HttpHandler handler = WebHttpHandlerBuilder.applicationContext(context).build();
```

返回的`HttpHandler`就是用来当做服务器适配器的 server adapter.

1.3.1. Special Bean Types

`DispatcherHandler`探查的一些特殊的SpringBean如下：

Bean type	Explanation
`HandlerMapping`	Map a request to a handler. The mapping is based on some criteria, the details of which vary by `HandlerMapping` implementation — annotated controllers, simple URL pattern mappings, and others.The main `HandlerMapping` implementations are `RequestMappingHandlerMapping` for `@RequestMapping` annotated methods, `RouterFunctionMapping` for functional endpoint routes, and `SimpleUrlHandlerMapping` for explicit registrations of URI path patterns and `WebHandler` instances.
`HandlerAdapter`	Help the `DispatcherHandler` to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherHandler` from such details.
`HandlerResultHandler`	Process the result from the handler invocation and finalize the response. See Result Handling.

1.3.2. WebFlux Config

应用可以声明bean的基础设施。但是，更多情况下。`Webflux Config`是一个最佳选择去自定义配置。

1.3.3. Processing

`DispatcherHandler`处理请求如下：

- 询问`HandlerMapping`去寻找匹配的处理器，第一个找到的会被使用。
- 如果handler找到了。使用`HandlerAdpater`执行目标方法，并获取`HandlerResult`返回值。
- `HandlerResult`由`HandlerResultHandler`去处理，决定直接响应还是视图渲染。

1.3.4. Result Handling

目标方法的返回值被`HandlerAdpater`调用后封装为`HandlerResult`，并携带一些上下文信息给第一个`HandlerResultHandler`去处理请求。这个表格展示了可用的`HandlerResultHandler`实现，这些都可以在 WebFlux Config 中声明。

Result Handler Type	Return Values	Default Order
`ResponseEntityResultHandler`	`ResponseEntity`, typically from `@Controller` instances.	0
`ServerResponseResultHandler`	`ServerResponse`, typically from functional endpoints.	0
`ResponseBodyResultHandler`	Handle return values from `@ResponseBody` methods or `@RestController` classes.	100
`ViewResolutionResultHandler`	`CharSequence`, `View`, Model, `Map`, Rendering, or any other `Object` is treated as a model attribute.See also View Resolution.	`Integer.MAX_VALUE`

1.3.5. Exceptions

`HandlerAdapter`执行目标方法返回`HandlerResult`期间出现的问题可以被异常处理

- 如@Controller处理器执行失败
- 处理返回值`HandlerResultHandler`失败

错误函数可以改变响应（如：返回一个error status），只要出现错误信号，在从处理程序返回的反应类型之前就会产生任何数据项。

`@ExceptionHandler`也可以标注在@Controller类的方法上。但是在webflux中不能使用@ControllerAdvice去处理那些发生在handler被选择之前的异常。

参照 Managing Exceptions 和 Exceptions

1.3.6. View Resolution

视图解析技术不只是响应浏览器页面，还可以有各种特别的新时代技术。在Spring WebFlux中，视图解析由`HandlerResultHandler`使用`ViewResolver`的对象完成字符串（逻辑视图）到`View`对象的映射。`View`是用来渲染响应的。

Handling

`HandlerResult`会传递给`ViewResolutioinResultHandler`。各种不同的handler返回值会被做如下处理：

- String、CharSequence： 代表逻辑视图的名字，会被用`ViewResolver`解析成一个`View`的实现对象。
- void： 直接使用默认的视图名（按照请求路径，减去开头的/得到最终的默认视图名），并解析成一个`View`对象。没提供view name触发同样逻辑。
- Rendering： API，代码上直接暴露视图解析方案的一些参数设置。
- Model、Map：额外的模型属性会被放到当前request中

- 其他Any东西：其他任何返回（ BeanUtils#isSimpleProperty 判断为的简单属性）都被当成model attribute添加到model中。

model可以包含异步，响应式类型。在渲染之前, `AbstractView` 解析model中的属性为一个具体的值，并更新model。单值的响应式类型被解析为单个值，或者no value。多值响应式类型如Flux被解析成List

WebFlux Config 提供直接专门的视图解析配置。也可以直接给容器中放一个 `ViewResolutionResultHandler` 来自定义配置视图解析。

Redirecting

返回值加 `redirect:` 就可以实现重定向。

也可以返回 `RedirectView` 或者 `Rendering.redirectTo("abc").build()` 。现在controller也能处理，返回的view name如果是 `redirect:/some/resource` 代表重定向到当前应用的 /some/resource 下面。 `redirect:https://example.com/arbitrary/path` 代表重定向到其他绝对路径位置。

Content Negotiation

内容协商。 `ViewResolutionResultHandler` 支持内容协商。他对比请求的内容类型和服务器支持的媒体类型来选择一个 `View` 。第一个满足的 `View` 会被使用。

为了支持JSON、XML等这种媒体类型。Spring WebFlux 提供 `HttpMessageWriterView` ，这是一个特殊的 `View` ，他通过 `HttpMessageWriter` 进行渲染。通常，可以配置这些作为默认 `View` ，通过 WebFlux Configuration即可. 默认Views只要支持请求的媒体类型，就被选择和使用了。

1.4. Annotated Controllers

无需实现任何接口，使用 `@Controller` and `@RestController` 注解标注在任意类上，就可以处理请求

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

1.4.1. `@Controller`

```
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

1.4.2. Request Mapping

`@RequestMapping`的几种方式:标注在Controller的任何方法上就能处理请求

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

URI Patterns

Pattern	Description	Example
`?`	Matches one character	`"/pages/t?st.html"` matches `"/pages/test.html"` and `"/pages/t3st.html"`
`*`	Matches zero or more characters within a path segment	`"/resources/*.png"` matches `"/resources/file.png"``"/projects/*/versions"` matches `"/projects/spring/versions"` but does not match `"/projects/spring/boot/versions"`
`**`	Matches zero or more path segments until the end of the path	`"/resources/**"` matches `"/resources/file.png"` and `"/resources/images/file.png"``"/resources/**/file.png"` is invalid as `**` is only allowed at the end of the path.
`{name}`	Matches a path segment and captures it as a variable named "name"	`"/projects/{project}/versions"` matches `"/projects/spring/versions"` and captures `project=spring`
`{name:[a-z]+}`	Matches the regexp `"[a-z]+"` as a path variable named "name"	`"/projects/{project:[a-z]+}/versions"` matches `"/projects/spring/versions"` but not `"/projects/spring1/versions"`
`{*path}`	Matches zero or more path segments until the end of the path and captures it as a variable named "path"	`"/resources/{*file}"` matches `"/resources/images/file.png"` and captures `file=/images/file.png`

`@PathVariable`捕获路径变量

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

url path也可以有`${…}`,系统启动的时候会替换。（`PropertySourcesPlaceholderConfigurer`会看system，environment，其他property sources）

> Spring WebFlux 使用`PathPattern`和`PathPatternParser`来处理URI路径匹配。这两个类可以同时进行大量的匹配，性能非常不错。

Spring WebFlux不知会后缀匹配模式

Pattern Comparison

`PathPattern.SPECIFICITY_COMPARATOR`会在多个pattern都匹配URL的时候选择一个最佳路径。

每个pattern，都会有一个得分。如果两个pattern得分相同，选择最长的那个路径。

`**`这种catch-all的pattern没分，被排在最后一个。

Consumable Media Types

可以使用`Content-Type`缩小request mapping的范围。

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

`consumes = "application/json"`可以声明在类级别，但是方法级别的优先级会更高。

> `MediaType`类提供常见的常量，如`APPLICATION_JSON_VALUE` and `APPLICATION_XML_VALUE`.
> 不用手写值

Producible Media Types

也可以缩小request mapping的范围，基于`Accept`请求头，这个参照Controller方法的`produces`字段

```
@GetMapping(path = "/pets/{petId}", produces = "application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

类级别也可以用。

Parameters and Headers

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

```
@GetMapping(path = "/pets", headers = "myHeader=myValue")
public void findPet(@PathVariable String petId) {
    // ...
}
```

HTTP HEAD, OPTIONS

@RequestMapping = GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS

Custom Annotations

编写`RequestMappingHandlerMapping`子类，重写getCustomMethodCondition 方法，就可以检查自定义属性返回自己的`RequestCondition`

Explicit Registrations（显式注册）

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping,
UserHandler handler)
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build();

        Method method = UserHandler.class.getMethod("getUser", Long.class);

        mapping.registerMapping(info, handler, method);
    }

}
```

1.4.3 Handler Methods

```
@RequestMapping
```

Method Arguments

Controller method argument	Description
`ServerWebExchange`	Access to the full `ServerWebExchange` — container for the HTTP request and response, request and session attributes, `checkNotModified` methods, and others.
`ServerHttpRequest`, `ServerHttpResponse`	Access to the HTTP request or response.
`WebSession`	Access to the session. This does not force the start of a new session unless attributes are added. Supports reactive types.
`java.security.Principal`	The currently authenticated user — possibly a specific `Principal` implementation class if known. Supports reactive types.
`org.springframework.http.HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available — in effect, the configured `LocaleResolver`/`LocaleContextResolver`.
`java.util.TimeZone` + `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`@PathVariable`	For access to URI template variables. See URI Patterns.
`@MatrixVariable`	For access to name-value pairs in URI path segments. See Matrix Variables.
`@RequestParam`	For access to Servlet request parameters. Parameter values are converted to the declared method argument type. See `@RequestParam`.Note that use of `@RequestParam` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`@RequestHeader`	For access to request headers. Header values are converted to the declared method argument type. See `@RequestHeader`.
`@CookieValue`	For access to cookies. Cookie values are converted to the declared method argument type. See `@CookieValue`.
`@RequestBody`	For access to the HTTP request body. Body content is converted to the declared method argument type by using `HttpMessageReader` instances. Supports reactive types. See `@RequestBody`.
`HttpEntity<B>`	For access to request headers and body. The body is converted with `HttpMessageReader` instances. Supports reactive types. See `HttpEntity`.

Controller method argument	Description
`@RequestPart`	For access to a part in a `multipart/form-data` request. Supports reactive types. See Multipart Content and Multipart Data.
`java.util.Map`, `org.springframework.ui.Model`, and `org.springframework.ui.ModelMap`.	For access to the model that is used in HTML controllers and is exposed to templates as part of view rendering.
`@ModelAttribute`	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See `@ModelAttribute` as well as `Model` and `DataBinder`.Note that use of `@ModelAttribute` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`Errors`, `BindingResult`	For access to errors from validation and data binding for a command object, i.e. a `@ModelAttribute` argument. An `Errors`, or `BindingResult` argument must be declared immediately after the validated method argument.
`SessionStatus` + class-level `@SessionAttributes`	For marking form processing complete, which triggers cleanup of session attributes declared through a class-level `@SessionAttributes` annotation. See `@SessionAttributes` for more details.
`UriComponentsBuilder`	For preparing a URL relative to the current request’s host, port, scheme, and context path. See URI Links.
`@SessionAttribute`	For access to any session attribute — in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration. See `@SessionAttribute` for more details.
`@RequestAttribute`	For access to request attributes. See `@RequestAttribute` for more details.
Any other argument	If a method argument is not matched to any of the above, it is, by default, resolved as a `@RequestParam` if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as a `@ModelAttribute`, otherwise.

Return Values

Controller method return value	Description
`@ResponseBody`	The return value is encoded through `HttpMessageWriter` instances and written to the response. See `@ResponseBody`.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies the full response, including HTTP headers, and the body is encoded through `HttpMessageWriter` instances and written to the response. See `ResponseEntity`.
`HttpHeaders`	For returning a response with headers and no body.
`String`	A view name to be resolved with `ViewResolver` instances and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model, with the view name implicitly determined based on the request path.
`@ModelAttribute`	An attribute to be added to the model, with the view name implicitly determined based on the request path.Note that `@ModelAttribute` is optional. See “Any other return value” later in this table.
`Rendering`	An API for model and view rendering scenarios.
`void`	A method with a `void`, possibly asynchronous (for example, `Mono<Void>`), return type (or a `null` return value) is considered to have fully handled the response if it also has a `ServerHttpResponse`, a `ServerWebExchange` argument, or an `@ResponseStatus` annotation. The same is also true if the controller has made a positive ETag or `lastModified` timestamp check. // TODO: See Controllers for details.If none of the above is true, a `void` return type can also indicate “no response body” for REST controllers or default view name selection for HTML controllers.

Controller method return value	Description
`Flux<ServerSentEvent>`, `Observable<ServerSentEvent>`, or other reactive type	Emit server-sent events. The `ServerSentEvent` wrapper can be omitted when only data needs to be written (however, `text/event-stream` must be requested or declared in the mapping through the `produces` attribute).
Any other return value	If a return value is not matched to any of the above, it is, by default, treated as a view name, if it is `String` or `void` (default view name selection applies), or as a model attribute to be added to the model, unless it is a simple type, as determined by BeanUtils#isSimpleProperty, in which case it remains unresolved.

Type Conversion

方法参数标注 `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`，可自动转换获取字符串数据。

Matrix Variables

`"/cars;color=red,green;year=2012"` 的变量是 `"color=red;color=green;color=blue"`

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

`@RequestParam`

绑定query paramenters的。

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
```

```
    }

    // ...
}
```

`@RequestHeader`

绑定请求头的

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

`@CookieValue`

`@ModelAttribute`

`@SessionAttributes`

`@SessionAttribute`

`@RequestAttribute`

Multipart Content

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...

}
```

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        // ...
    }

}
```

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") Part metadata,
        @RequestPart("file-data") FilePart file) {
    // ...
}
```

```
@PostMapping("/")
public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) {
    // ...
}
```

```
@PostMapping("/")
public String handle(@RequestBody Flux<Part> parts) {
    // ...
}
```

`@RequestBody`

使用`HttpMessageReader`反序列化request body为一个Object。

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

```
//非阻塞方式
@PostMapping("/accounts")
public void handle(@RequestBody Mono<Account> account) {
    // ...
}
```

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Mono<Account> account) {
    // use one of the onError* operators...
}
```

`HttpEntity`

`HttpEntity`和`@RequestBody`功能或多或少是一样的，但是提供了request header和body的一个统一访问

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

`@ResponseBody`

使用`HttpMessageWriter`把返回值对象序列化成response body。

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

HTTP message codecs option of the WebFlux Config to configure or customize message writing.

`ResponseEntity`

像@ResponseBody一样，但是有status 和 header

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).build(body);
}
```

Jackson JSON

支持jackson的库

JSON Views

Spring WebFlux内置支持 Jackson's Serialization Views, 允许渲染一个Object的属性子集。配合`@ResponseBody`或者`ResponseEntity`。可以使用`@JsonView`注解去激活序列化view类。

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @JsonView(WithoutPasswordView.class)
    public String getUsername() {
        return this.username;
    }

    @JsonView(WithPasswordView.class)
```

```
    public String getPassword() {
        return this.password;
    }
}
```

> | | |
> |---|---|
> | | **`@JsonView` allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.** |
> | | |

1.4.4 Model

@ModelAttribute 注解 和 WebDataBinder 配合工作，获取 Model属性。

这个特性就别用了，麻烦。

1.4.5. `DataBinder`

Web on Reactive Stack (spring.io)

别用了，也麻烦。

1.4.6. Managing Exceptions

`@Controller`、`@ControllerAdvice`可以用`@ExceptionHandler`注解去异常处理。

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

REST API exceptions

1.4.7. Controller Advice

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class})
public class ExampleAdvice3 {}
```

1.5. Functional Endpoints

函数式端口。Spring WebFlux包括WebFlux.fn，一种轻巧的功能编程模型，其中使用功能来路由和处理请求和合同设计用于不可分割。它是基于注释的编程模型的替代方法，但在同一 Reactive Core 基础上运行。

1.5.1. Overview

WebFlux.fn 中。一个HTTP请求被 `HandlerFunction` 处理：一个function可以拿到 `ServerRequest` 并且返回一个延迟的 `ServerResponse`（`Mono<ServerRespons>`）。Request和Response都有一个不变的API（JDK8 友好）去访问HTTP请求和响应。`HandlerFunction` 是 `@RequestMapping` 的等价编程模型。

进来的请求使用 `HandlerFunction` 路由给一个请求方法（一个function可以拿到 `ServerRequest` 并且返回一个延迟的 `ServerResponse`（`Mono<ServerRespons>`））。路由匹配上后，会返回一个handler function，否则就返回一个empty Mono。`RouterFunction` 和 `@RequestMapping` 最大的区别就是 router function 不仅提供数据，也是定义一个行为。

`RouterFunctions.route()` 提供一个router的建造者模式方便创建router。例如：

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static
org.springframework.web.reactive.function.server.RequestPredicates.*;
import static
org.springframework.web.reactive.function.server.RouterFunctions.route;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

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


public class PersonHandler {
```

```
    // ...

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

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

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

运行启动 `RouterFunction` 的一种方式是转成 `HttpHandler` 并且通过一个 server adapters 来安装他。

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

大多数应用可以通过WebFlux的Java配置启动和运行。参照： Running a Server.

1.5.2. HandlerFunction

`ServerRequest` and `ServerResponse` 可以直接访问到HTTP的request和response。

request和reponse都提供了响应式流的背压方式获取Body流。request的body流是一个 `Flux` 或 `Mono`。 response body是一个 `Publisher`，包含 `Flux` 和 `Mono`。

ServerRequest

可以获取到 请求方式、URI、headers、和查询参数、请求body（通过Body方法）

```
Mono<String> string = request.bodyToMono(String.class); //访问请求体得到String
Flux<Person> people = request.bodyToFlux(Person.class); //json或者xml反序列化

//使用BodyExtractors
Mono<String> string = request.body(BodyExtractors.toMono(String.class));
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class));


Mono<MultiValueMap<String, String>> map = request.formData(); //访问表单数据
Mono<MultiValueMap<String, Part>> map = request.multipartData(); //multipart数据
Flux<Part> parts = request.body(BodyExtractors.toParts()); //还有这种Multipart方式
```

ServerResponse

ServerResponse代表一个响应，可以使用 `build` 方法创建。可以使用Builder设置response的状态码、header、body等。比如：

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person,
Person.class); //200码，响应person的json
```

```
URI location = ...
ServerResponse.created(location).build();   //201码，自定义Location头。

ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView.class).body(...); //自定义JSONView返回
```

Handler Classes

handler方法用lambda写

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().bodyValue("Hello World");
```

示例:

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static
org.springframework.web.reactive.function.server.ServerResponse.ok;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        Flux<Person> people = repository.allPeople();
        return ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person = request.bodyToMono(Person.class);
        return ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        int personId = Integer.valueOf(request.pathVariable("id"));
        return repository.getPerson(personId)
            .flatMap(person ->
ok().contentType(APPLICATION_JSON).bodyValue(person))
            .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

Validation

函数式端口也可以使用Spring的validation facilities 去校验请求body数据。

```
public class PersonHandler {

    private final Validator validator = new PersonValidator();

    // ...

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person =
request.bodyToMono(Person.class).doOnNext(this::validate);
        return ok().build(repository.savePerson(person));
    }

    private void validate(Person person) {
        Errors errors = new BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw new ServerWebInputException(errors.toString());
        }
    }
}
```

1.5.3 RouterFunction

路由函数是用来把请求路由给关联的`HandlerFunction`。通常我们不用自己写routerfuntion，而是使用`RouterFunctions`工具类创建一个。`RouterFunctions.route()`(没有参数)可以创建一个链式的builder，通过这个builder来创建router function。也可以使用`RouterFunctions.route(RequestPredicate, HandlerFunction)`直接创建一个路由规则。

通常，推荐使用`router()` builder，快速映射路由，而不用 硬编码路由发现规则。例如，function builder 提供`GET(String, HandlerFunction)`创建get请求映射，`POST(String, HandlerFunction)`创建post映射。

除了基于http请求方式的映射。还可以提供断言。所有请求方式满足这个断言`RequestPredicate`（就像之前网关的断言一样）。

Predicates

可以自己编写RequestPredicate 或者使用`RequestPredicates`工具类，基于请求路径，方式，content-type等。例如：下面根据了一个`Accept`请求头字段

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> ServerResponse.ok().bodyValue("Hello World")).build();
```

组合多个Predicates

- `RequestPredicate.and(RequestPredicate)` — both must match.
- `RequestPredicate.or(RequestPredicate)` — either can match.

Routes

路由函数的顺序如下：如果第一个没有匹配，那就依次往下测试第二个，第三个....。因此，精确匹配一定放在最前面，通用匹配放在最后面。而且路由函数，必须注册为spring bean。

当使用router function builder的时候，所有的定义的路由都被builder()组合成一个`RouterFunction`对象返回。还有一些其他的组合方式。

- `add(RouterFunction)` on the `RouterFunctions.route()` builder
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` — shortcut for `RouterFunction.and()` with nested `RouterFunctions.route()`.

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static
org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> otherRoute = ...

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

Nested Routes

就像把@RequestMapping标在类上，其他方法自动拥有这个基准路径一样。

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

`nest()`组合任何内嵌的断言

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .build();
```

1.5.4 Running a Server

如何在Http服务器运行一个路由函数。简单的一种方式是`HttpHandler`

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

一个典型的设置，就是`DispatcherHander`基于 WebFlux Config的。WebFlux 的Java配置原理如下：

- RouterFunctionMapping： 探查Spring配置中所有的`RouterFunction<?>`，给他们排序，并通过`RouterFunction.andOther`组合，并且最终返回一个`RouterFunction`
- HandlerFunctionAdapter： 简单的适配器，让`DispatcherHandler`执行映射的`HandlerFunction`。
- ServerResponseResultHandler： 处理`HandlerFunction`执行后的结果，通过执行`ServerResponse`的`writeTo()`方法。

一个实例配置

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Bean
    public RouterFunction<?> routerFunctionA() {
        // ...
    }

    @Bean
    public RouterFunction<?> routerFunctionB() {
        // ...
    }

    // ...

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // configure message conversion...
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // configure view resolution for HTML rendering...
    }
}
```

1.5.5 Filtering Handler Functions

可以过滤handler functions 通过使用 before、after、或filter方法在router function的builder中。类似我们注解的@ControllerAdvice、ServletFilter等。

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople)
            .before(request -> ServerRequest.from(request)
                .header("X-RequestHeader", "Value")
                .build()))
        .POST(handler::createPerson))
    .after((request, response) -> logResponse(response))
    .build();
```

filter方法接受ServerRequest返回ServerResponse。

```
SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    })
    .build();
```

next.handle(ServerRequest) 的执行是可选的，其实就像filter的chain.doFilter一样。

> `CorsWebFilter` 提供CORS功能

1.6. URI Links

1.6.1. UriComponents

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .build();

URI uri = uriComponents.expand("Westin", "123").toUri();
```

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

1.6.2. UriBuilder

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

使用WebClient

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new
DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

1.6.3. URI Encoding

`UriComponentsBuilder` 暴露两个方法

- UriComponentsBuilder#encode(): 在uri变量扩展之前工作。
- UriComponents#encode(): 在 uri变量扩展之后工作

> ;特殊。第一个";"会被替换成"%3B"，第二个 ";"不会被替换。

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri();

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar");
```

```
URI uri = UriComponentsBuilder.fromUriString("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar");
```

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

1.7. CORS

1.7.1 介绍

Spring WebFlux 可以处理跨域

Cross-Origin Resource Sharing (CORS) 跨域资源共享是w3c的一个规范 W3C specification，大多数浏览器都实现了。

1.7.2 处理

跨域如何工作，参照：this article,

Spring WebFlux `HandlerMapping` 内置实现跨域支持。 当请求匹配上handler以后，`HandlerMapping` 会检查跨域配置。Preflight 请求直接处理，简单请求和真实CORS请求被拦截、验证是否有相应的头。

每个`HandlerMapping`能配置基于url的`CorsConfiguration`。

可以用类或方法级别的注解`@CrossOrigin`，或者实现handler实现`CorsConfigurationSource`

> 更多自定义高级配置，参照
>
> - `CorsConfiguration`
> - `CorsProcessor` and `DefaultCorsProcessor`
> - `AbstractHandlerMapping`

1.7.3 `@CrossOrigin`

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

@CrossOrigin 默认如下效果：

- 所有 origins
- 所有 header
- 所有 methods
- `allowCredentials`没有默认开启

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com")
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }
```

```
    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

1.7.4 Global Configuration

除了在controller方法上细粒度配置，还可以定义一个全局的跨域配置。通常使用WebFlux的java配置类来做。

默认的全局跨域配置如下：

- All origins.
- All headers.
- `GET`, `HEAD`, and `POST` methods.
- `allowedCredentials` 默认也没被设置。
- `maxAge` 是30min

可以使用CorsRegistry回调来进行配置

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

1.7.5 CORS `WebFilter`

可以使用内置的 `CorsWebFilter` 很好的适配 function endpoints这种方式。

> 如果在Spring Security中想使用 CorsFilter，记住 Spring Security 内置支持跨域。

配置filter，可以使用CorsWebFilter组件并传入一个 `CorsConfigurationSource` 在构造器中。

```
@Bean
CorsWebFilter corsFilter() {

    CorsConfiguration config = new CorsConfiguration();

    // Possibly...
    // config.applyPermitDefaultValues()
```

```
    config.setAllowCredentials(true);
    config.addAllowedOrigin("https://domain1.com");
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");

    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);

    return new CorsWebFilter(source);
}
```

1.8 Web Security

Spring Security 项目提供web应用的保护，避免一些恶意请求。参照：

- WebFlux Security
- WebFlux Testing Support
- CSRF protection
- Security Response Headers

1.9. View Technologies

视图技术是Spring WebFlux中的一个可插拔功能。无论你决定用 Thymeleaf, FreeMarker或其他模板引擎。请自学Spring的视图解析原理： View Resolution.

1.9.1 Thymeleaf

Thymeleaf 是一个现代化的服务端模板引起。 Thymeleaf 官网。

Thymeleaf与Spring WebFlux的整合要使用 `SpringResourceTemplateResolver`, `SpringWebFluxTemplateEngine`, and `ThymeleafReactiveViewResolver`. 更多参照 Thymeleaf+Spring 和 WebFlux announcement.

1.9.2 FreeMarker

Apache FreeMarker 也可以整合。

View Configuration

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure FreeMarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates/freemarker");
        return configurer;
    }
}
```

上面配置，如果controller返回视图名是`welcome`，解析器就会找`classpath:/templates/freemarker/welcome.ftl`下面的模板

FreeMarker Configuration

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // ...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        Map<String, Object> variables = new HashMap<>();
        variables.put("xml_escape", new XmlEscape());

        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        configurer.setFreemarkerVariables(variables);
        return configurer;
    }
}
```

Form Handling 表单处理

Bind Macros 绑定宏

一组标准的宏指令在 spring-webflux.jar 中为FreeMarker定义的。

Form Macros 表单宏

- Input Macros
- Input Fields
- Selection Fields
- HTML Escaping

1.9.3 脚本视图

spring内置支持使用WebFlux运行任何模板库，以及java的脚本引擎。参照 JSR-223

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

> 基本的规则是，如果整合任何其他脚本引擎，必须实现 ScriptEngine 和 Invocable 接口

必须依赖

如果需要脚本引擎到雷陆军，以下细节注意

- Nashorn JavaScript engine Java8+才支持。
- 要支持Ruby就引入 JRuby 依赖
- 要支持Python 就引入 Jython 依赖
- Kotlin 案例：https://github.com/sdeleuze/kotlin-script-templating

去 WebJars 找一些依赖

Script Templates

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
```

```
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

1.9.4 JSON and XML

为了内容协商 Content Negotiation ，基于客户端的需求，响应不同内容。Spring WebFlux提供 `HttpMessageWriterView` ，可以插入个 Codecs ，比如

`Jackson2JsonEncoder` , `Jackson2SmileEncoder` , or `Jaxb2XmlEncoder` 来实现不同输出。

不像视图解析技术， `HttpMessageWriterView` 不需要一个 `ViewResolver` 。但是需要做个默认配置view。运行时匹配请求的内容类型就会用HttpMessageWriter实例或者Encoder实例来写出数据。

1.10 HTTP缓存

HTTP缓存能显著提升web性能。 `Cache-Control` 响应头以及配合一些其他额外的请求头（比如： `Last-Modified` and `ETag` ）。 `Cache-Control` 建议私有（浏览器）或者公有（代理）缓存，使得缓存可以再利用。 `ETag` 头是用来创建一个有条件的请求，如果响应内容没有变化，可能返回的结果是304（NOT_MODIFIED）并没有响应体。 `ETag` 可以在很多复杂的后续 `Last-Modified` 中看到。

1.10.1 `CacheControl`

`CacheControl` 提供 `Cache-Control` 头的定制化。

- Controllers
- Static Resources

参照 RFC 7234

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10,
TimeUnit.DAYS).noTransform().cachePublic();
```

1.10.2 Controllers

Controllers可以添加一个显式的HTTP Caching支持。推荐，一个资源的`lastModified`或`ETag`值需要在比较请求头之前进行计算。Contrllor可以给`ResponseEntity`添加`ETag`和`Cache-Control`设置。

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book);
}
```

这个例子发送一个没有响应体的304（NOT_MODIFIED）响应。如果与条件请求标头的比较表明内容没有更改。否则`ETag`和`Cache-Control`头会被添加到响应中

```
@RequestMapping
public String myHandleMethod(ServerWebExchange exchange, Model model) {

    long eTag = ...  //应用程序自己查到的etag值

    if (exchange.checkNotModified(eTag)) {
        return null;  //etag没有变化，就响应304.
    }

    model.addAttribute(...); //继续处理请求
    return "myViewName";
}
```

三种形式，`eTag`，`lastModified`或者都用，

1.10.3 Static Resources

静态资源应该使用`Cache-Control`，参照： Static Resources.

1.11 WebFlux 配置

参照`WebFluxConfigurationSupport`，和 Special Bean Types.

完全控制参照： Advanced Configuration Mode.

1.11.1. 启用WebFlux Config

`@EnableWebFlux`

```
@Configuration
@EnableWebFlux
public class WebConfig {
}
```

这种配置默认配置了一些基础设施： infrastructure beans

1.11.2 WebFlux Config API

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // Implement configuration methods...
}
```

1.11.3 Conversion, formatting

`@NumberFormat` and `@DateTimeFormat`

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }

}
```

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        DateTimeFormatterRegistrar registrar = new DateTimeFormatterRegistrar();
        registrar.setUseIsoFormat(true);
        registrar.registerFormatters(registry);
    }
}
```

> See `FormatterRegistrar` SPI and the `FormattingConversionServiceFactoryBean` for more information on when to use `FormatterRegistrar` implementations.

1.11.4 Validation

`@Valid` and `@Validated` on `@Controller`

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public Validator getValidator() {
        // ...
    }

}
```

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

1.11.5 Content Type Resolvers

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder
builder) {
        // ...
    }
}
```

1.11.6 HTTP message codecs

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().maxInMemorySize(512 * 1024);
    }
}
```

`ServerCodecConfigurer`提供默认的 readers 和 writer。

jackson JSON和XML，考虑使用 `Jackson2ObjectMapperBuilder`，自定义jackson的一些默认属性

- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

1.11.7 View Resolvers

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // ...
    }
}
```

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure Freemarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        return configurer;
    }
}
```

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        ViewResolver resolver = ... ;
        registry.viewResolver(resolver);
    }
}
```

支持内容协商，使用 `HttpMessageWriterView` 的实现，接受任何 `spring-web` 可用的 Codecs

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();

        Jackson2JsonEncoder encoder = new Jackson2JsonEncoder();
        registry.defaultViews(new HttpMessageWriterView(encoder));
    }

    // ...
}
```

参照： View Technologies

1.11.8 Static Resources

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));
    }

}
```

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new
VersionResourceResolver().addContentVersionStrategy("/**"));
    }

}
```

1.11.9 Path Matching

PathMatchConfigurer

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .setUseTrailingSlashMatch(false)
            .addPathPrefix("/api",
                    HandlerTypePredicate.forAnnotation(RestController.class));
    }
}
```

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public WebSocketService getWebSocketService() {
        TomcatRequestUpgradeStrategy strategy = new
TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

1.11.11 高级配置模型

`@EnableWebFlux` 导入 `DelegatingWebFluxConfiguration`：

- 提供默认spring配置
- `WebFluxConfigurer`

`@EnableWebFlux`

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...
}
```

1.12. HTTP/2

reactor netty,tomcat,jetty,undertow都支持 HTTP/2。 参考： HTTP/2 wiki page.

2. WebClient

webflux中的webclient拥有基于reactor的函数式、fluent API, 参照:Reactive Libraries。 可以使用异步逻辑。是一个非阻塞，支持streaming。

`WebClient`：

- Reactor Netty
- Jetty Reactive HttpClient
- Apache HttpComponents
- `ClientHttpConnector`

2.1 配置

`WebClient` 通过静态工厂方法。

- `WebClient.create()`
- `WebClient.create(String baseUrl)`

也可以使用WebClient.builder()：

- uriBuilderFactory： 自定义 UriBuilderFactory
- defaultUriVariables： 默认uri
- defaultHeader： request头
- defaultCookie： 请求cookie
- defaultRequest： Consumer自定义每个请求
- filter： `Client` filter
- exchangeStrategies： HTTP message reader/writer 自定义

- clientConnector：HTTP client配置

```
WebClient client = WebClient.builder()
        .codecs(configurer -> ... )
        .build();
```

一旦构建，`WebClient`是不可编的。但是可以克隆

```
WebClient client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build();

WebClient client2 = client1.mutate()
        .filter(filterC).filter(filterD).build();

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filterD
```

2.1.1 MaxInMemorySize

默认 256KB。

```
org.springframework.core.io.buffer.DataBufferLimitException: Exceeded limit on
max bytes to buffer
```

```
WebClient webClient = WebClient.builder()
        .codecs(configurer -> configurer.defaultCodecs().maxInMemorySize(2 *
1024 * 1024))
        .build();
```

2.1.2 Reactor Netty

自定义reactor netty设置，

```
HttpClient httpClient = HttpClient.create().secure(sslSpec -> ...);

WebClient webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

Resources

```
public ReactorResourceFactory reactorResourceFactory() {
    return new ReactorResourceFactory();
}
```

```
@Bean
public ReactorResourceFactory resourceFactory() {
    ReactorResourceFactory factory = new ReactorResourceFactory();
    factory.setUseGlobalResources(false);  //创建资源独立
    return factory;
}

@Bean
public WebClient webClient() {

    Function<HttpClient, HttpClient> mapper = client -> {
        // Further customizations...
    };

    ClientHttpConnector connector =
            new ReactorClientHttpConnector(resourceFactory(), mapper); //使用
ReactorClientHttpConnector 构造器

    return WebClient.builder().clientConnector(connector).build();
//WebClient.Builder
}
```

Timeouts

```
import io.netty.channel.ChannelOption;

HttpClient httpClient = HttpClient.create()
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000);

WebClient webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

```
import io.netty.handler.timeout.ReadTimeoutHandler;
import io.netty.handler.timeout.WriteTimeoutHandler;

HttpClient httpClient = HttpClient.create()
        .doOnConnected(conn -> conn
                .addHandlerLast(new ReadTimeoutHandler(10))
                .addHandlerLast(new WriteTimeoutHandler(10)));

// Create WebClient...
```

```
HttpClient httpClient = HttpClient.create()
        .responseTimeout(Duration.ofSeconds(2));

// Create WebClient...
```

```
WebClient.create().get()
        .uri("https://example.org/path")
        .httpRequest(httpRequest -> {
            HttpClientRequest reactorRequest = httpRequest.getNativeRequest();
            reactorRequest.responseTimeout(Duration.ofSeconds(2));
        })
        .retrieve()
        .bodyToMono(String.class);
```

2.1.3 Jetty

```
HttpClient httpClient = new HttpClient();
httpClient.setCookieStore(...);

WebClient webClient = WebClient.builder()
        .clientConnector(new JettyClientHttpConnector(httpClient))
        .build();
```

默认 HttpClient创建自己的资源（Executor，ByteBufferPool，Scheduler），直到`stop()`被调用，这些资源会销毁

```
@Bean
public JettyResourceFactory resourceFactory() {
    return new JettyResourceFactory();
}

@Bean
public WebClient webClient() {

    HttpClient httpClient = new HttpClient();
    // Further customizations...

    ClientHttpConnector connector =
            new JettyClientHttpConnector(httpClient, resourceFactory());

    return WebClient.builder().clientConnector(connector).build();
}
```

2.1.4 HttpComponents

```
HttpAsyncClientBuilder clientBuilder = HttpAsyncClients.custom();
clientBuilder.setDefaultRequestConfig(...);
CloseableHttpAsyncClient client = clientBuilder.build();
ClientHttpConnector connector = new HttpComponentsClientHttpConnector(client);

WebClient webClient = WebClient.builder().clientConnector(connector).build();
```

2.2 `retrieve()`

```
WebClient client = WebClient.create("https://example.org");

Mono<ResponseEntity<Person>> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .toEntity(Person.class);
```

```
WebClient client = WebClient.create("https://example.org");

Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToMono(Person.class);
```

获取stream

```
Flux<Quote> result = client.get()
        .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve()
        .bodyToFlux(Quote.class);
```

处理 4xx 或 5xx

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError, response -> ...)
        .onStatus(HttpStatus::is5xxServerError, response -> ...)
        .bodyToMono(Person.class);
```

2.3 Exchange

`exchangeToMono()` 和 `exchangeToFlux()`，方法（`awaitExchange { }`和`exchangeToFlow { }`）比较有用。

```
Mono<Person> entityMono = client.get()
        .uri("/persons/1")
        .accept(MediaType.APPLICATION_JSON)
        .exchangeToMono(response -> {
            if (response.statusCode().equals(HttpStatus.OK)) {
                return response.bodyToMono(Person.class);
            }
            else {
                // Turn to error
                return response.createException().flatMap(Mono::error);
            }
        });
```

2.4 Request Body

```
Mono<Person> personMono = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(personMono, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

解码对象流

```
Flux<Person> personFlux = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_STREAM_JSON)
        .body(personFlux, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

```
Person person = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .bodyValue(person)
        .retrieve()
        .bodyToMono(Void.class);
```

2.4.1 Form 数据

发送表单数据，使用`MultiValueMap<String, String>`。 会通过`FormHttpMessageWriter`自动设置`application/x-www-form-urlencoded`

```
MultiValueMap<String, String> formData = ... ;

Mono<Void> result = client.post()
        .uri("/path", id)
        .bodyValue(formData)
        .retrieve()
        .bodyToMono(Void.class);
```

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromFormData("k1", "v1").with("k2", "v2"))
        .retrieve()
        .bodyToMono(Void.class);
```

2.4.2 Multipart数据

```
MultipartBodyBuilder builder = new MultipartBodyBuilder();
builder.part("fieldPart", "fieldValue");
builder.part("filePart1", new FileSystemResource("...logo.png"));
builder.part("jsonPart", new Person("Jason"));
builder.part("myPart", part); // Part from a server request

MultiValueMap<String, HttpEntity<?>> parts = builder.build();
```

```
MultipartBodyBuilder builder = ...;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(builder.build())
        .retrieve()
        .bodyToMono(Void.class);
```

设置 Content-Type 为 multipart/form-data，使用MultipartBodyBuilder

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromMultipartData("fieldPart", "value").with("filePart",
resource))
        .retrieve()
        .bodyToMono(Void.class);
```

2.5 Filters

可以注册一个过滤器`ExchangeFilterFunction`通过`WebClient.Builder`。

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {

            ClientRequest filtered = ClientRequest.from(request)
                    .header("foo", "bar")
                    .build();

            return next.exchange(filtered);
        })
        .build();
```

认证

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAu
thentication;

WebClient client = WebClient.builder()
        .filter(basicAuthentication("user", "password"))
        .build();
```

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAu
thentication;

WebClient client = webClient.mutate()
        .filters(filterList -> {
            filterList.add(0, basicAuthentication("user", "password"));
        })
        .build();
```

```
public ExchangeFilterFunction renewTokenFilter() {
    return (request, next) -> next.exchange(request).flatMap(response -> {
        if (response.statusCode().value() == HttpStatus.UNAUTHORIZED.value()) {
```

```
            return response.releaseBody()
                    .then(renewToken())
                    .flatMap(token -> {
                        ClientRequest newRequest =
ClientRequest.from(request).build();
                        return next.exchange(newRequest);
                    });
        } else {
            return Mono.just(response);
        }
    });
}
```

2.6 Attributes

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {
            Optional<Object> usr = request.attribute("myAttribute");
            // ...
        })
        .build();

client.get().uri("https://example.org/")
        .attribute("myAttribute", "...")
        .retrieve()
        .bodyToMono(Void.class);

    }
```

2.7 Context

```
WebClient client = WebClient.builder()
        .filter((request, next) ->
                Mono.deferContextual(contextView -> {
                    String value = contextView.get("foo");
                    // ...
                }))
        .build();

client.get().uri("https://example.org/")
        .retrieve()
        .bodyToMono(String.class)
        .flatMap(body -> {
                // perform nested request (context propagates automatically)...
        })
        .contextWrite(context -> context.put("foo", ...));
```

2.8 Synchronous Use

```
Person person = client.get().uri("/person/{id}", i).retrieve()
    .bodyToMono(Person.class)
    .block();

List<Person> persons = client.get().uri("/persons").retrieve()
    .bodyToFlux(Person.class)
    .collectList()
    .block();
```

```
Mono<Person> personMono = client.get().uri("/person/{id}", personId)
        .retrieve().bodyToMono(Person.class);

Mono<List<Hobby>> hobbiesMono = client.get().uri("/person/{id}/hobbies",
personId)
        .retrieve().bodyToFlux(Hobby.class).collectList();

Map<String, Object> data = Mono.zip(personMono, hobbiesMono, (person, hobbies) -
> {
            Map<String, String> map = new LinkedHashMap<>();
            map.put("person", person);
            map.put("hobbies", hobbies);
            return map;
        })
        .block();
```

2.9 Testing

可以mock一个服务器，比如 OkHttp MockWebServer. 引入spring-test 都能测试

3. WebSockets

3.1 WebSocket简介

websocket协议规范 RFC 6455, 提供一个标准的方式，去在客户端和服务端间建立全双工，双路通信通道 通过一个单条TCP连接。它是与HTTP不同的TCP协议，但旨在使用端口80和443在HTTP上工作，并允许重新使用现有的防火墙规则。

Websocket交互始于HTTP请求，该请求使用HTTP升级标头升级或在这种情况下切换到Websocket协议。比如：

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket      //升级头
Connection: Upgrade     //使用升级连接
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

而不是通常的200状态代码，而是具有WebSocket支持的服务器返回的输出类似于以下内容：

```
HTTP/1.1 101 Switching Protocols    //协议切换
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623GznO=
Sec-WebSocket-Protocol: v10.stomp
```

成功进行握手后，HTTP升级请求下的TCP Socket仍可为客户端和服务器都打开，以继续发送和接收消息。

WebSocket的工作方式的完整介绍超出了本文档的范围。请参阅`RFC 6455`，HTML5的Websocket章节，或网络上的许多介绍和教程中的任何一个。

请注意，如果WebSocket服务器在Web服务器后面运行（例如NGINX），则可能需要将其配置为将WebSocket升级请求传递到Websocket服务器。同样，如果应用程序在云环境中运行，请检查与WebSocket相关的云提供商的说明。

3.1.1 HTTP对比WebSocket

即使Websocket被设计为与HTTP兼容，并从HTTP请求开始，但重要的是要了解这两个协议导致非常不同的架构和应用程序编程模型。

在HTTP和REST中，对应用程序进行了建模的URL模型。要与应用程序进行交互，客户端访问这些URL，请求响应样式。服务器根据HTTP URL，方法和header将请求向适当的处理程序请求。

相比之下，在Websocket中，通常只有一个用于初始连接的URL。随后，所有应用程序消息都以相同的TCP连接流动。这表明了完全不同的异步，事件驱动的消息传递体系结构。

WebSocket也是一种低级传输协议，与HTTP不同，该协议没有为消息内容规定任何语义。这意味着除非客户端和服务器就消息语义达成协议，否则无法路由或处理消息。

WebSocket客户端和服务器可以通过HTTP握手请求上的SEC-Websocket-protocol标头协商更高级别的消息传递协议（例如Stomp）的使用。在缺乏此事的情况下，他们需要提出自己的惯例。

3.1.2 什么时候用WebSocket

WebSocket可以使网页具有动态性和交互性。但是，在许多情况下，Ajax和HTTP流或长时间的轮询的组合可以提供简单有效的解决方案。

例如，新闻，邮件和社交供稿需要动态更新，但每隔几分钟就可以这样做。另一方面，协作，游戏和财务应用需要更接近实时。

仅延迟不是决定因素。如果消息量相对较低（例如，监视网络故障），HTTP流或轮询可以提供有效的解决方案。低潜伏期，高频和大数据量 是使用WebSocket的最佳案例。

还要记住，在Internet上，在您控制之外的限制性代理可能会排除Websocket交互，要么是因为它们不配置为传递升级标头，要么是因为它们关闭了出现空闲的长期连接。这意味着与公共面对应用程序相比，防火墙内的内部应用程序的使用更为直接。

3.2 WebSocket API

Same as in the Servlet stack

Spring提供WebSocketAPI

3.2.1 服务端

Same as in the Servlet stack

创建一个websocket服务器，首先应该创建一个 `WebSocketHandler`

```
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketSession;

public class MyWebSocketHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        // ...
    }
}
```

然后进行url映射

```
@Configuration
class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/path", new MyWebSocketHandler());
        int order = -1; // before annotated controllers

        return new SimpleUrlHandlerMapping(map, order);
    }
}
```

如果使用webflux，需要配置声明一个`WebSocketHandlerAdapter`。

```
@Configuration
class WebConfig {

    // ...

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter();
    }
}
```

3.2.2 `WebSocketHandler`

`WebSocketHandler` 的 `handle` 方法接受 `WebSocketSession` 参数，并返回 `Mono<Void>` 去表示应用处理会话结束。这个会话通过两个流进行处理。一个是输入，一个输出。

`WebSocketSession` method	Description
`Flux<WebSocketMessage> receive()`	Provides access to the inbound message stream and completes when the connection is closed.
`Mono<Void> send(Publisher<WebSocketMessage>)`	Takes a source for outgoing messages, writes the messages, and returns a `Mono<Void>` that completes when the source completes and writing is done.

一个`WebSocketHandler`必须统一组合输入和输出流并返回`Mono<Void>`来反应完成状态。

当将入站和出站消息流组合在一起时，无需检查连接是否打开，因为反应流信号最终活动。入站流接收到完成或错误信号，并且出站流接收一个取消信号。

处理程序的最基本实现是处理入站流的方法。以下示例显示了这样的实现：

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.receive()                //访问输入消息
                .doOnNext(message -> {
                    // ...                      //每个消息来做什么
                })
                .concatMap(message -> {
                    // ...                       //执行使用消息内容的嵌套异步操作。
                })
                .then();       //返回接收完成时完成的Mono<void>。
    }
}
```

> 内部，异步操作。需要调用`message.retain()`来读取数据。

以下实现结合了入站和出站流：

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Flux<WebSocketMessage> output = session.receive()       //    处理入站消息

                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .map(value -> session.textMessage("Echo " + value));     //创建一个出站消息

        return session.send(output);          //发送消息，并不结束，继续接受入站消息。

    }
}
```

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Mono<Void> input = session.receive()            //接收入站消息

                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .then();

        Flux<String> source = ... ;
        Mono<Void> output = session.send(source.map(session::textMessage));  //发送出站消息

        return Mono.zip(input, output).then();              //        加入流并返回单个流结束时完成的Mono<void>。
    }
}
```

3.2.3 DataBuffer

DataBuffer是WebFlux中字节缓冲区的表示形式。参考 Data Buffers and Codecs具有更多内容。要理解的关键点是，在诸如Netty，Byte Buffers之类的某些服务器上进行了汇总和参考计数，并且必须在消费时释放以避免内存泄漏。

在NetTy上运行时，应用程序必须使用databufferutils.retain（DataBuffer），如果他们希望保留输入数据缓冲区以确保它们不会被释放，并且随后在消耗缓冲区时使用DataBufferutils.Releases.Releases.Release（Databuffer）。

3.2.4 Handshake 握手

Same as in the Servlet stack

WebSockEthandlerAdapter代表WebSockEsservice。默认情况下，这是HandShakeWebSockEcketService的一个实例，该实例在WebSocket请求上执行基本检查，然后为使用的服务器使用RequestUpgradestrategy。目前，对Reactor Netty，Tomcat，Jetty和Undertow有内置的支持。

3.2.5 服务器配置

Same as in the Servlet stack

每个服务器的RequestUpgradestgy揭露了特定于基础Websocket Server Engine的配置。使用WebFlux Java配置时，您可以自定义WebFlux配置相应部分中所示的属性，或者如果不使用WebFlux配置，请使用以下内容：

```
@Configuration
class WebConfig {

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter(webSocketService());
    }

    @Bean
    public WebSocketService webSocketService() {
        TomcatRequestUpgradeStrategy strategy = new
TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

3.2.6 CORS

Same as in the Servlet stack

配置COR的最简单方法并限制对WebSocket端点的访问的方法是让您的Websockethandler实现CorsConfigurationsource，并返回带有允许的起源，标头和其他详细信息的CorsConfiguration。如果您不能这样做，则还可以在SimpleUrlhandler上设置CorsConfigurations属性，以通过URL模式指定CORS设置。如果两者都指定，则通过使用结合方法进行结合方法将它们组合在一起。

3.2.7 客户端

Spring WebFlux提供了一个WebSocketClient抽象，并提供了reactor Netty，Tomcat，Jetty，Undertow和Standard Java的实现（即JSR-356）。

要启动WebSocket会话，您可以创建客户端实例并使用其执行方法：

```
WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");
client.execute(url, session ->
        session.receive()
                .doOnNext(System.out::println)
                .then());
```

4. 测试Testing

Same in Spring MVC

spring-test 模块提供 `ServerHttpRequest`, `ServerHttpResponse`, and `ServerWebExchange`. 的mock测试

`WebTestClient` 可以用来做端到端的集成测试