一文读懂 Spring的前世今生

本文转载自公众号 编程新说

情景引入

很早之前，Java就火起来了，是因为它善于开发和处理网络方面的应用。

Java有一个爱好，就是喜欢制定规范标准，但自己又不善于去实现。

反倒是一些服务提供商使用它的规范标准来制造应用服务器而赚的盆满钵满。

企业用户因要使用这些应用服务器而向提供商支付高额费用，而且也不是特别好用。

一个青年才俊为了打破这种局面而奔走呼号、奋发图强。

自我介绍

显然，这个青年才俊就是后来的Spring。

因企业应用大都和web相关，而Java的web标准中较核心的一部分其实就是JavaEE里的Servlet。

Spring和Servlet“相亲相爱”一番后，我就来到了这个世界。我的全名叫Spring MVC，这里的Spring既是我的姓也是我的“爸爸”，那Servlet就是我的“妈妈”了，大家叫我MVC就行了。

那个年代社会很落后，条件也不好，好歹我们要求也不高，求个温饱就行了。

所以我的妈妈Servlet和她的闺蜜Filter天生就是同步阻塞的，包括她们同事HttpServletRequest的getParameter，getPart等这些方法也都是阻塞的。

虽然我的爸爸Spring给了我23条染色体来进行改良，但不要忘了我还从Servlet妈妈那里继承了23条，所以我也是同步阻塞的。不过我的“长相”已经好看很多了，因为Spring爸爸知道，在以后的日子里，除了拼实力之外，颜值也是非常重要的。

因为我妈妈Servlet是一个规范，我爸爸Spring是一个框架，所以我跟他们一样，都是无法自己独立运行的。

所以在我们要运行的时候，必须要寻找一个特殊的“家”，通常称它为Servlet容器，比如tomcat就算非常知名的一个。

Servlet容器熟知我极有可能阻塞当前执行线程，所以专门量身打造。它给我准备了一个非常大的线程池，里面有好多线程。每过来一个请求，它就扔给我一个线程，说自己玩去吧，随便“折腾”。

好在那时美国那个叫乔布斯的家伙被自己的公司赶出去在外面“流浪”，Servlet容器为我量身打造的这种方法完全能够胜任日常，关键还非常的简单。

这种小富即安的日子就这样往前过着。

兄弟出生

生命不息，变化不止。随着乔布斯推出iphone，智能机瞬间大火，全民进入移动互联网时代。激增的网民数量，给现有软件架构带来极大的挑战。

一般来说，社会越发达，分工越精细，对单一工种的要求就越高。

软件也是如此，在传统“大块头”软件表现的越来越格格不入的时候，微服务就如一丝春风吹了进来。

按它的指导原则，将大软件按某种方式拆分为一个个小工程。小工程规模小，便于管理，而且机动性也好，功能聚合性更好。它承受的并发应该更高。

有人觉得与微服务比起来，过去使用的web服务器如tomcat略显笨重，不够轻量级。也有人说tomcat内部一个请求一个线程这种阻塞执行方式消耗太多线程，不太容易支撑超高并发。

无论怎么说，简而言之一句话，一个全新的时代已经到来。

此时我们需要一个更加轻量级web应用，它使用更少的硬件资源和线程，反而更容易处理高并发。那么它一定是异步非阻塞的。

这样的使命自然落到了响应式编程的范畴上了。所以我的爸爸Spring审时度势，在5.0之后就赶紧把我推出来了。

没错，我就是Spring WebFlux，这里的Spring既是我的姓也是我爸爸。大家可以叫我WebFlux。初来乍到，好多人都对我不熟悉，请容许我介绍一番。

首先这个响应式究竟是什么意思呢？响应式这个术语，指的是一个编程模型，它是围绕着对变化的反映来构建的。

如网络组件用来响应I/O事件，UI控制器用来响应鼠标事件等等。按照这种意识的话，非阻塞就是响应式的，对操作完成或数据可用通知事件的响应方式。

另外一个关于响应式的机制是非阻塞后压。在命令式代码中，同步阻塞调用带有自然的后压迫使调用者等待。

在异步代码中，它变得非常重要，用来控制事件的速率，以至于不让一个快速的事件源压垮它的响应者。就是响应者能够控制事件源发射事件的快慢。

因为响应式编程是非阻塞的，所以我也是非阻塞的，因此我通常运行在非阻塞web服务器上，如Netty，Undertow等。

因为我不会阻塞线程的执行，所以使用一个小的固定数量的线程池（event loop workers）来处理请求。典型地，线程数与CPU的核数相同。

这里还要感谢我的姥爷Java 8，他老人家引入了lambda表达式造就了函数式编程API。这对于非阻塞应用和连续式API来说是一个非常棒的东西，允许以声明的方式把异步逻辑组合起来。

我感觉我的爸爸Spring已经超越了一个框架，成为一个平台了。所以他自己并没有亲自去实现响应式处理，而是为我选择Reactor作为响应式库。

Reactor提供Flux和Mono类型，拥有丰富的操作符，支持非阻塞后压，使用函数式API来组合异步逻辑。并且Reactor强烈聚焦于Java服务器端。它在开发时就已经与爸爸Spring亲密协作了。

爸爸说，我也支持其它的库如RxJava，但看样子似乎让我更爱Reactor一些。

这就是我，WebFlux，一个集天时地利于一身的幸运儿。但你是不是已经晕晕的啦，没关系，慢慢来。

包罗万象

我想，大家都看出了我爸爸Spring的野心，他不仅要成为一个平台，还要建起自己的生态系统，竖起壁垒。

所以他的核心事业就是进行抽象，组合和装配，进而包罗万象。说的掉渣一些，就是哪个技术好，就给它整合进来。

为了抹平底层不同web服务器的差异，我爸爸抽象了一个最低级别的契约接口，HttpHandler，用于响应式HTTP请求的处理。

```
Mono<java.lang.Void> handle(ServerHttpRequest request, ServerHttpResponse response);
```

它是一个通用的接口，要横跨不同的运行时。它是有意设计成最小化的，只有一个方法，主要唯一目的就是在不同的HTTP服务器API上面成为一个最小化的抽象。

如果想用Netty服务器的话，就基于Netty实现一下，同理也可以基于Undertow实现一下，等等，只要以后有了新的服务器，都可以加进来的。

显而易见，HttpHandler的目标是抽象出来对不同HTTP服务器的使用，说白了就是为了和底层服务器对接。但由于太偏底层，不利用上层代码使用。

为此，我的爸爸又抽象出一个稍微高一点级别的契约接口，WebHandler，用于Web请求处理。很明显，WebHandler的目标是提供web应用中广泛使用的通用特性，如Session、表单数据和附件等等，也是为了更容易和上层代码对接。

很自然的，WebHandler是构建于HttpHandler之上的，换句话说WebHander的处理会通过一个适配器HttpWebHandlerAdapter最终代理给HttpHandler来执行。

WebHandler接口也只有一个方法：

```
Mono<java.lang.Void> handle(ServerWebExchange exchange);
```

参数类型是ServerWebExchange，可以这样理解，你发一个请求，给你一个响应，相当于用请求交换了一个响应，而且是在服务器端交换的。

其实，整个web请求的处理过程是一个链式的，最后才是一个WebHandler，它前面可以插入多个错误处理器，WebExceptionHandler，多个过滤器，WebFilter。

这是错误处理器接口：

```
Mono<java.lang.Void> handle(ServerWebExchange exchange, java.lang.Throwable ex);
```

这是过滤器接口：

```
Mono<java.lang.Void> filter(ServerWebExchange exchange, WebFilterChain chain);
```

可见，我的爸爸Spring的抽象能力非常强，对下抽象一个接口，抹平了不同服务器的差异。对上抽象一个接口，可以用于支撑不同的编程模型。

都有哪些编程模型呢，请继续往下看吧。

上面我在介绍自己的时候使用了美颜，所以诸位很难看清我的“真面目”，下面就来进行一下自我剖析，看看真实的我。

我包含一个轻量级函数式编程模型，函数被用来参与处理请求，它是相对于基于注解编程模型的另一种选择，这种编程模型叫做函数式端点，functional endpoints，是构建于上面提到的WebHandler之上的。

我是使用HandlerFunction来处理一个HTTP请求的，这是一个函数式接口，也称处理函数：

```
@FunctionalInterface
public interface HandlerFunction<T extends ServerResponse> {
    reactor.core.publisher.Mono<T> handle(ServerRequest request);
}
```

带有一个ServerRequest参数，返回一个Mono<ServerResponse>，其中request和response对象都是不可变的，HandlerFunction就等价于Controller中的@RequestMapping标记的方法。

实际当中，请求很多，处理函数也很多，如何知道一个请求过来后，该由哪个处理函数去处理呢？

这自然要用到我的另一个函数式接口RouterFunction来搞定，称为路由函数：

```
@FunctionalInterface
public interface RouterFunction<T extends ServerResponse> {
    reactor.core.publisher.Mono<HandlerFunction<T>> route(ServerRequest request);
}
```

带有一个ServerRequest参数，返回一个Mono<HandlerFunction>。就是它把一个请求路由到一个HandlerFunction的，当路由函数匹配时，就返回一个处理函数，否则返回一个空的Mono。RouterFunction等价于@RequestMapping注解，但主要不同的是路由函数提供的不仅是数据，还有行为。

下面通过一些示例，来更加直观的帮助大家认识这两个函数式接口。

因处理函数是函数式接口，所以可以直接用一个lambda表达式来处理请求，如下：

```
HandlerFunction<ServerResponse> handler = request -> Response.ok().body("Hello World");
```

这就表示当任何一个请求过来时，都返回Hello World作为响应。

在实际应用中，处理逻辑一般都很复杂，肯定不是一个lambda表达式能搞定的，此时希望把处理方法专门写到一个类里，就叫处理器类，和MVC里的Controller差不多一回事。

下面就是一个Person的处理器类：

```
public class PersonHandler {

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

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

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

此时就可以通过处理函数，引用这些处理器方法了，如下：

```
PersonHandler handler = new PersonHandler();

HandlerFunction<ServerResponse> list = handler::listPeople;

HandlerFunction<ServerResponse> create = handler::createPerson;

HandlerFunction<ServerResponse> get = handler::getPerson;
```

要想使请求能够正确被路由，首先要定义好路由函数，如下：

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/person/{id}", get)
    .GET("/person", list)
    .POST("/person", create)
    .build();
```

它表示当以GET方法请求/person/{id}时，最终会由getPerson方法处理。当以GET方法请求/person时，最后会由listPeople方法处理。同理，以POST方法请求/person时，会由createPerson方法处理。

可见，一个路由函数可以包含多个路由规则，实际当中，可以定义多个路由函数，这些路由函数可以组合在一起。

路由函数是按顺序计算的，如果第一个路由不匹配，计算第二个，等等。因此，把更加具体的路由放到通用路由前面是非常有意义的。注意这和基于注解的不同。

怎么样，关掉滤镜的我是不是更加真实了。我相信你也看明白了，至少要记住，这是基于函数式的一种编程模型，叫做函数式端点。

雨露均沾

像我这样的幸运儿，你们一定以为Spring爸爸对我非常溺爱吧，告诉你，确实是这样的。不过考虑到大家伙一路走来对Spring的不离不弃，爸爸也设身处地为你们着想。

为此，我除了支持函数式端点这种编程模型之外，还支持一种编程模型叫基于注解的控制器，annotated controllers，没错，就是MVC里的那个。

话说的再白一些，就是大家已经非常熟悉的Spring MVC那套东西，我百分之百的完全支持，妥妥的，放心使用。

但是，并不是所有的控制器方法参数都支持响应式类型，只有一些支持，如WebSession，java.security.Principal，@RequestBody，HttpEntity<B>，@RequestPart等。

下面看一个示例：

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

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

@PostMapping("/accounts")
public void handle(@RequestBody Mono<Account> account) {
```

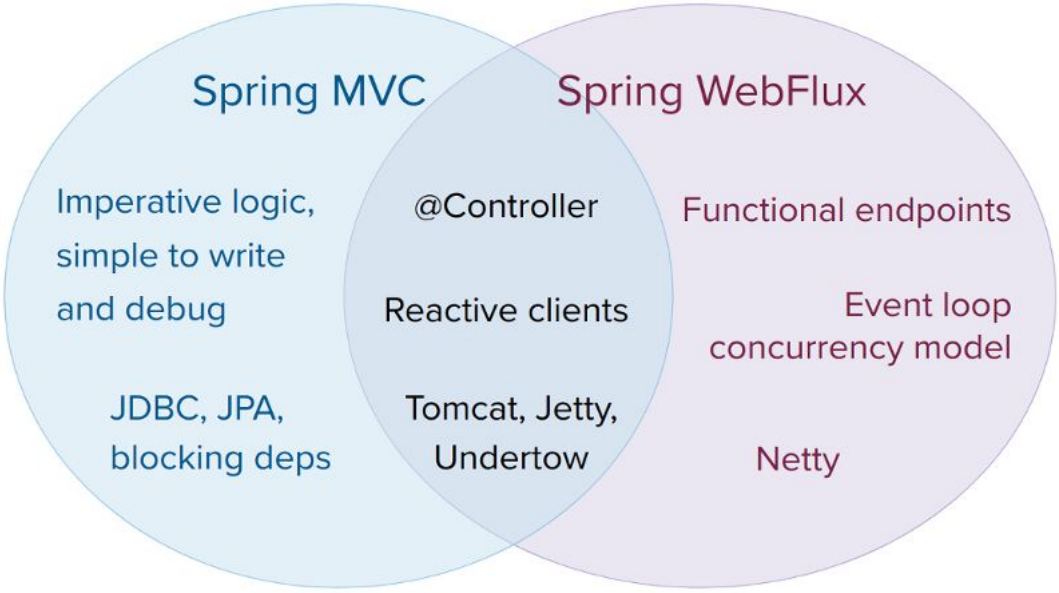
```
    // ...
}
```

不过对于控制器方法的所有返回值，都是支持响应式类型的。

各有千秋

Spring MVC和Spring WebFlux可以一起使用，从设计上讲，它们互为继续、互为一致。

它们的关系，请看下图，既有共同的部分，也有互相独立的部分。



Spring MVC的特点就是，它是命令式编程，代码非常容易写，也好理解和调试。但是它是同步的，会有人觉得它性能不好。

但是我要说的是，响应式和非阻塞通常来讲也不会使应用运行的更快。相反，非阻塞方式要求做更多的事情，而且还会稍微增加一些必要的处理时间。也就是说，还可能稍稍变慢一点，what，那为啥还要用它呢？

响应式和非阻塞的关键好处是，在使用很少固定数目的线程和较少的内存情况下的扩展能力。

这使应用在负载下更有适应能力，因为它们以一个更加具有可预见性的方式在扩展。

为了能够观察到这些好处，你需要有一些延迟才行，比如一个既不可靠且速度又慢的网络I/O，这才是响应式开始展示它强劲的地方，带来的差异（惊喜）可能是巨大的哦。

其实技术无好坏，各有各的适用场景罢了。

————————END————————