使用 Spring 5 的 WebFlux 开发反应

成 富
2017 年 10 月 2 日发布 / *更新: 2019 年 5 月 05 日*

Spring 5 是流行的 Spring 框架的下一个重大的版本升级。Spring 5 中最重要改动是把反应式编程 面，Spring 5 的反应式编程以 Reactor 库为基础。在之前的文章《使用 Reactor 进行反应式编程 详细的介绍。读者如果需要了解 Reactor，可以参考之 篇文章。Spring 5 框架所包含的内 增的 WebFlux 模块。开发人员可以使用 WebFlux 创建 的 Web 应用和客户端。本文对 WebFlux 模块进行了详细介绍，包括其中的 HTTP、服务器推送事件和 WebSocket 支持。

WebFlux 简介

WebFlux 模块的名称是 spring-webflux，名称中的 Flux Reactor 中的类 Flux。该模块中包含了对反应式 HTTP、服务器推送事件和 WebSocket 的客户端和服务器端的支持。对于 员来说，比较重要的是服务器端的开发，这也是本文的重点。在服务器端，WebFlux 支持两种不同的编程模型：第一种 ng MVC 中使用的基于 Java 注解的方式；第二种是基于 Java 8 的 lambda 表达式的函数式编程模型。这两种编程模型只是 码编写方式上存在不同。它们运行在同样的反应式底层架构之上，因此在运行时是相同的。WebFlux 需要底层提供运行时 ，WebFlux 可以运行在支持 Servlet 3.1 非阻塞 IO API 的 Servlet 容器上，或是其他异步运行时环境，如 Netty 和 Undertow。

最方便的创建 WebFlux 应用的方式是使用 Spring Boot 提供的应用模板。直接访问 Spring Initializ 网站（http://start.spring.io/），选择创建一个 Maven 或 Gradle 项目。Spring Boot 的版本选择 2.0.0 M2。在添加的依赖中，选择 Reactive Web。最后输入应用所在的分组和名称，点击进行下载即可。需要注意的是，只有在选择了 Spring Boot 2.0.0 M2 之后，依赖中才可以选择 Reactive Web。下载完成之后可以导入到 IDE 中进行编辑。本文的示例代码使用 Intellij IDEA 2017.2 进行编写。

本文从三个方面对 WebFlux 进行介绍。首先是使用经典的基于 Java 注解的编程模型来进行开发，其次是使用 WebFlux 新增的函数式编程模型来进行开发，最后介绍 WebFlux 应用的测试。通过这样循序渐进的方式让读者了解 WebFlux 应用开发的细节。

Java 注解编程模型

基于 Java 注解的编程模型，对于使用过 Spring MVC 的开发人员来说是再熟悉不过的。在 WebFlux 应用中使用同样的模式，容易理解和上手。我们先从最经典的 Hello World 的示例开始说明。代码清单 1 中的 BasicController 是 REST API 的控制器，通过 @RestController 注解来声明。在 BasicController 中声明了一个 URI 为/hello_world 的映射。其对应的方法 sayHelloWorld()的返回值是 Mono<String>类型，其中包含的字符串"Hello World"会作为 HTTP 的响应内容。

清单 1. Hello World 示例

```
@RestController
public class BasicController {
    @GetMapping("/hello_world")
    public Mono<String> sayHelloWorld() {
        return Mono.just("Hello World");
    }
}
```

从代码清单 1 中可以看到，使用 WebFlux 与 Spring MVC 的不同在于，WebFlux 所使用的类型是与反应式编程相关的 Flux 和 Mono 等，而不是简单的对象。对于简单的 Hello World 示例来说，这两者之间并没有什么太大

REST API

简单的 Hello World 示例并不足以说明 WebFlux 的用法。在下面的小节中，本文将介绍其他具体
起。REST API 在 Web 服务器端应用中占据了很大的一部分。我们通过一个具体的实例来说明如
API。

该 REST API 用来对用户数据进行基本的 CRUD 操作。作为领域对象的 User 类中包含了 id、na
为了对 User 类进行操作，我们需要提供服务类 UserService，如代码清单 2 所示。类 UserServ
户的信息，并不是一个持久化的实现。这对于示例应用来说已经足够了。类 UserService 中的方
回值，这也是 WebFlux 应用的特征。在方法 getById()中，如果找不到 ID 对应的 User 对象，会
ResourceNotFoundException 异常通知的 Mono 对象。方法 getById()和 createOrUpdate()都可
数。Flux 类型的参数表示的是有多个对象需要处理。这里使用 doOnNext()来对其中的每个对象

清单 2. UserService

```
@Service
class UserService {
    private final Map<String, User> data = new ConcurrentHashMap<>();

    Flux<User> list() {
        return Flux.fromIterable(this.data.values());
    }

    Flux<User> getById(final Flux<String> ids) {
        return ids.flatMap(id -> Mono.justOrEmpty(this.data.get(id)));
    }

    Mono<User> getById(final String id) {
        return Mono.justOrEmpty(this.data.get(id))
                .switchIfEmpty(Mono.error(new ResourceNotFoundException()));
    }

    Mono<User> createOrUpdate(final User user) {
        this.data.put(user.getId(), user);
        return Mono.just(user);
    }

    Mono<User> delete(final String id) {
        return Mono.justOrEmpty(this.data.remove(id));
    }
}
```

代码清单 3 中的类 UserController 是具体的 Spring MVC 控制器类。它使用类 UserService 来完成具体的功能。类 UserController 中使用了注解@ExceptionHandler 来添加了 ResourceNotFoundException 异常的处理方法，并返回 404 错误。类 UserController 中的方法都很简单，只是简单地代理给 UserService 中的对应方法。

清单 3. UserController

```
@RestController
@RequestMapping("/user")
public class UserController {
    private final UserService userService;

    @Autowired
    public UserController(final UserService userService) {
        this.userService = userService;
    }

    @ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Resource not found")
    @ExceptionHandler(ResourceNotFoundException.class)
    public void notFound() {
    }

    @GetMapping("")
    public Flux<User> list() {
        return this.userService.list();
    }

    @GetMapping("/{id}")
    public Mono<User>getById(@PathVariable("id") final String id) {
        return this.userService.getById(id);
    }

    @PostMapping("")
    public Mono<User> create(@RequestBody final User user) {
        return this.userService.createOrUpdate(user);
```

```
29      }
30
31      @PutMapping("/{id}")
35          return this.userService.createOrUpdate(user);
36      }
37
38      @DeleteMapping("/{id}")
39      public Mono<User>  delete(@PathVariable("id") final String id) {
40          return this.userService.delete(id);
41      }
42  }
```

Java 注解编程模型

服务器推送事件

客户端

服务器推送事件（Server-Sent Events，SSE）允许服务器端不断地推送数据到客户端。相对于 WebSocket 等其他技术，SSE 天生支持服务器端到客户端的单向数据传递。虽然功能较弱，但优势在于 SSE 在已有的 HTTP 协议上就可以使用，可以表示传输的数据。作为 W3C 的推荐规范，SSE 在浏览器端的支持也比较广泛，除了 IE 之外的其他浏览器都支持。IE 浏览器也可以使用 polyfill 库来提供支持。在服务器端来说，SSE 是一个不断产生新数据的流，非常适合于响应式编程。在 Spring WebFlux 中创建 SSE 的服务器端是非常简单的。只需要返回的对象的类型是 Flux<ServerSentEvent>即可。Spring 会自动根据客户端要求的格式来发送响应。

代码清单 4 中的 SseController 是一个使用 SSE 的控制器的一个例。其中的方法 randomNumbers 会产生随机数的 SSE 端点。我们可以使用类 ServerSentEvent.Builder 来创建 ServerSentEvent 对象。这里我们设置了事件的名称及每个事件的标识符和数据。事件的标识符是一个递增的整数，而数据则是产生的随机数。

清单 4. 服务器推送事件示例

```
1   @RestController
2   @RequestMapping("/sse")
3   public class SseController {
4       @GetMapping("/randomNumbers")
5       public Flux<ServerSentEvent<Integer>> randomNumbers() {
6           return Flux.interval(Duration.ofSeconds(1))
7                   .map(seq -> Tuples.of(seq, ThreadLocalRandom.current().nextInt()))
8                   .map(data -> ServerSentEvent.<Integer>builder()
9                           .event("random")
10                          .id(Long.toString(data.getT1()))
11                          .data(data.getT2())
12                          .build());
13      }
14  }
```

在测试 SSE 时，我们只需要使用 curl 来访问即可。代码清单 5 给出了调用 curl http://localhost:8080/sse/randomNumbers 的结果。

清单 5. SSE 服务器端发送的响应

```
1   id:0
2   event:random
3   data:751025208
4
5   id:1
6   event:random
7   data:-1591883873
8
9   id:2
10  event:random
11  data:-1899224227
```

WebSocket

WebSocket 支持客户端与服务器端的双向通讯。当客户端与服务器端之间的交互方式比较复杂时，可以使用 WebSocket。WebSocket 在主流的浏览器上都得到了支持。WebFlux 也对创建 WebSocket 服务器端提供了支持。在服务器端，我们需要实现接口 org.springframework.web.reactive.socket.WebSocketHandler 来处理 WebSocket 通讯。接口 WebSocketHandler 的方法 handle 的参数是接口 WebSocketSession 的对象，可以用来获取客户端信息、接送消息和发送消息。代码清单 6 中的 EchoHandler 对于每个接收的消息，会发送一个添加了"ECHO -> "前缀的响应消息。WebSocketSession 的 receive 方法的返回值是一个 Flux<WebSocketMessage>对象，表示的是接收到的消息流。而 send 方法的参数是一个

Publisher<WebSocketMessage>对象，表示要发送的消息流。在 handle 方法，使用 map 操作对 receive 方法得到的 Flux<WebSocketMessage>中包含的消息继续处理，然后直接由 send 方法来发送。


```
@Component
public class EchoHandler implements WebSocketHandler {
    @Override
    public Mono<Void> handle(final WebSocketSession session) {
        return session.send(
                session.receive()
                        .map(msg -> session.textMessage("ECHO -> " + msg.getPay
    }
}
```


函数式编程模型
客户端
测试
结束语

在创建了 WebSocket 的处理器 EchoHandler 之后，下一步需要把它注册到 WebFlux 中。我们首 WebSocketHandlerAdapter 的对象，该对象负责把 WebSocketHandler 关联到 WebFlux 中。代 配置。其中的 HandlerMapping 类型的 bean 把 EchoHandler 映射到路径 /echo。

清单 7. 注册 EchoHandler

```
@Configuration
public class WebSocketConfiguration {

    @Autowired
    @Bean
    public HandlerMapping webSocketMapping(final EchoHandler echoHandler) {
        final Map<String, WebSocketHandler> map = new HashMap<>(1);
        map.put("/echo", echoHandler);

        final SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
        mapping.setOrder(Ordered.HIGHEST_PRECEDENCE);
        mapping.setUrlMap(map);
        return mapping;
    }

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

运行应用之后，可以使用工具来测试该 WebSocket 服务 F工具页面 https://www.websocket.org/echo.html，然后连接到 ws://localhost:8080/echo，可以发送消息并查看服务器 回的结果。

函数式编程模型

在上节中介绍了基于 Java 注解的编程模型，WebFlux 还支持基于 lambda 表达式的函数式编程模型。与基于 Java 注解的编程模型相比，函数式编程模型的抽象层次更低，代码编写更灵活，可以满足一些对动态性要求更高的场景。不过在编写时的代码复杂度也较高，学习曲线也较陡。开发人员可以根据实际的需要来选择合适的编程模型。目前 Spring Boot 不支持在一个应用中同时使用两种不同的编程模式。

为了说明函数式编程模型的用法，我们使用 Spring Initializ 来创建一个新的 WebFlux 项目。在函数式编程模型中，每个请求是由一个函数来处理的，通过接口 org.springframework.web.reactive.function.server.HandlerFunction 来表示。HandlerFunction 是一个函数式接口，其中只有一个方法 Mono<T extends ServerResponse> handle(ServerRequest request)，因此可以用 labmda 表达式来实现该接口。接口 ServerRequest 表示的是一个 HTTP 请求。通过该接口可以获取到请求的相关信息，如请求路径、HTTP 头、查询参数和请求内容等。方法 handle 的返回值是一个 Mono<T extends ServerResponse>对象。接口 ServerResponse 用来表示 HTTP 响应。ServerResponse 中包含了很多静态方法来创建不同 HTTP 状态码的响应对象。本节中通过一个简单的计算器来展示函数式编程模型的用法。代码清单 8 中给出了处理不同请求的类 CalculatorHandler，其中包含的方法 add、subtract、multiply 和 divide 都是接口 HandlerFunction 的实现。这些方法分别对应加、减、乘、除四种运算。每种运算都是从 HTTP 请求中获取到两个作为操作数的整数，再把运算的结果返回。

清单 8. 处理请求的类 CalculatorHandler

```
@Component
public class CalculatorHandler {

    public Mono<ServerResponse> add(final ServerRequest request) {
        return calculate(request, (v1, v2) -> v1 + v2);
    }

    public Mono<ServerResponse> subtract(final ServerRequest request) {
        return calculate(request, (v1, v2) -> v1 - v2);
    }
```

```
    public Mono<ServerResponse>  multiply(final ServerRequest request) {
        return calculate(request, (v1, v2) -> v1 * v2);
```



```
        return calculate(request, (v1, v2) -> v1 / v2);
    }

    private Mono<ServerResponse> calculate(final ServerRequest request,
                                           final BiFunction<Integer, Integer,
        final Tuple2<Integer, Integer> operands = extractOperands(request);
        return ServerResponse
                .ok()
                .body(Mono.just(calculateFunc.apply(operands.getT1(), operands
    }

    private Tuple2<Integer, Integer> extractOperands(final ServerRequest reque
        return Tuples.of(parseOperand(request, "v1"), parseOperand(request, "v
    }

    private int parseOperand(final ServerRequest request, final String param)
        try {
            return Integer.parseInt(request.queryParam(param).orElse("0"));
        } catch (final NumberFormatException e) {
            return 0;
        }
    }
}
```


评论

在创建了处理请求的 HandlerFunction 之后，下一步是为这些 HandlerFunction 提供路由信息，调用的条件。这是通过函数式接口 org.springframewor reactive.function.server.RouterFu RouterFunction 的方法 Mono<HandlerFunction<T ext rverResponse>> route(ServerRe ServerRequest，都返回对应的 0 个或 1 个 HandlerFu 对象，以 Mono<HandlerFunction>来表示。当找到对应的 HandlerFunction 时，该 HandlerFunction 被调用来处 rverRequest，并把得到的 ServerResponse 返回。在使用 WebFlux 的 Spring Boot 应用中，只需要创建 RouterFunction 类 ean，就会被自动注册来处理请求并调用相应的 HandlerFunction。

代码清单 9 给了示例相关的配置类 Config。方法 Route ons.route 用来根据 Predicate 是否匹配来确定 HandlerFunction 是否被应用。RequestPredicates 中包含了很多静态方法 常用的基于不同匹配规则的 Predicate。如 RequestPredicates.path 用来根据 HTTP 请求的路径来进行匹配。此处我们检查 路径是/calculator。在清单 9 中，我们首先使用 ServerRequest 的 queryParam 方法来获取到查询参数 operator 的值，然 反射 API 在类 CalculatorHandler 中找到与查询参数 operator 的值名称相同的方法来确定要调用的 HandlerFunction 的实 后调用查找到的方法来处理该请求。如果找不到查询参数 operator 或是 operator 的值不在识别的列表中，服务器端返回 4 是；如果反射 API 的方法调用中出现错误，服务器端返回 500 错误。

清单 9. 注册 RouterFunction

```
@Configuration
public class Config {

    @Bean
    @Autowired
    public RouterFunction<ServerResponse>routerFunction(final CalculatorHandler calculatorHandler) {
        return RouterFunctions.route(RequestPredicates.path("/calculator"), request ->
            request.queryParam("operator").map(operator ->
                    Mono.justOrEmpty(ReflectionUtils.findMethod(CalculatorHandler.class, operator, Serve
                            .flatMap(method -> (Mono<ServerResponse>) ReflectionUtils.invokeMethod(metho
                            .switchIfEmpty(ServerResponse.badRequest().build())
                            .onErrorResume(ex -> ServerResponse.status(HttpStatus.INTERNAL_SERVER_ERROR)
                    .orElse(ServerResponse.badRequest().build()));
    }
}
```

客户端

除了服务器端实现之外，WebFlux 也提供了反应式客户端，可以访问 HTTP、SSE 和 WebSocket 服务器端。

HTTP

对于 HTTP 和 SSE，可以使用 WebFlux 模块中的类 org.springframework.web.reactive.function.client.WebClient。代码清单 10 中的 RESTClient 用来访问前面小节中创建的 REST API。首先使用 WebClient.create 方法来创建一个新的 WebClient 对象，然后使用方法 post 来创建一个 POST 请求，并使用方法 body 来设置 POST 请求的内容。方法 exchange 的作用是发送请求并得到以 Mono<ServerResponse>表示的 HTTP 响应。最后对得到的响应进行处理并输出结果。ServerResponse 的 bodyToMono 方法把

响应内容转换成类 User 的对象，最终得到的结果是 Mono<User>对象。调用 createdUser.block 方法的作用是等待请求完成并得到所产生的类 User 的对象。


```
public class RESTClient {
    public static void main(final String[] args) {
        final User user = new User();
        user.setName("Test");
        user.setEmail("test@example.org");
        final WebClient client = WebClient.create("http://localhost:8080/user"
        final Monol<User> createdUser = client.post()
                .uri("")
                .accept(MediaType.APPLICATION_JSON)
                .body(Mono.just(user), User.class)
                .exchange()
                .flatMap(response -> response.bodyToMono(User.class));
        System.out.println(createdUser.block());
    }
}
```

测试

结束语

SSE

参考资源

WebClient 还可以用同样的方式来访问 SSE 服务，如代码清单 11 所示。这里我们访问的是在之前创建的 SSE 服务。使用 WebClient 访问 SSE 在发送请求部分与访问 REST API 是相同的，所不同的地方在于 SSE 服务的响应是一个消息流，我们需要使用 flatMapMany 方法将 Mono<ClientResponse>转换成 Flux<ServerSentEvent>对象，这是通过方法 BodyExtractors.toFlux 来完成的，其参数 new ParameterizedTypeReference<ServerSentEvent<String>>()表明了响应消息流中的内容是 ServerSentEvent 对象。由于 SSE 服务器会不断地发送消息，这里我们只是通过 buffer 方法获取前 10 条消息并输出。

清单 11. 使用 WebClient 访问 SSE 服务

```
public class SSEClient {
    public static void main(final String[] args) {
        final WebClient client = WebClient.create();
        client.get()
                .uri("http://localhost:8080/sse/randomNumbers")
                .accept(MediaType.TEXT_EVENT_STREAM)
                .exchange()
                .flatMapMany(response -> response.body(BodyExtractors.toFlux(new ParameterizedTypeReference<
                }))
                .filter(sse -> Objects.nonNull(sse.data()))
                .map(ServerSentEvent::data)
                .buffer(10)
                .doOnNext(System.out::println)
                .blockFirst();
    }
}
```

WebSocket

访问 WebSocket 不能使用 WebClient，而应该使用专门的 WebSocketClient 客户端。Spring Boot 的 WebFlux 模板中默认使用的是 Reactor Netty 库。Reactor Netty 库提供了 WebSocketClient 的实现。在代码清单 12 中，我们访问的是上面小节中创建的 WebSocket 服务。WebSocketClient 的 execute 方法与 WebSocket 服务器建立连接，并执行给定的 WebSocketHandler 对象。该 WebSocketHandler 对象与代码清单 6 中的作用是一样的，只不过它是工作于客户端，而不是服务器端。在 WebSocketHandler 的实现中，首先通过 WebSocketSession 的 send 方法来发送字符串 Hello 到服务器端，然后通过 receive 方法来等待服务器端的响应并输出。方法 take(1)的作用是表明客户端只获取服务器端发送的第一条消息。

清单 12. 使用 WebSocketClient 访问 WebSocket

```
public class WSClient {
    public static void main(final String[] args) {
        final WebSocketClient client = new ReactorNettyWebSocketClient();
        client.execute(URI.create("ws://localhost:8080/echo"), session ->
                session.send(Flux.just(session.textMessage("Hello")))
                        .thenMany(session.receive().take(1).map(WebSocketMessage::getPayloadAsText))
                        .doOnNext(System.out::println)
                        .then())
                .block(Duration.ofMillis(5000));
    }
}
```

测试

IBM Developer 学习 开发 社区

试 WebFlux 服务器。进行测试时既可以通过 mock 的方式来进行，也可以对实际运行的服务器进

内容 集成测试来测试 UserController 中的创建用户的功能。方法 WebTestClient.bindToServer 给

概览 基础 URL。发送 HTTP 请求的方式与代码清单 10 相同，不同的是 exchange 方法的返回值是 Re

WebFlux 简介 expectStatus 和 expectBody 等方法来验证 HTTP 响应的状态码和内容。方法 jsonPath 可以根据

证。

清单 13. 测试 UserController

```
public class UserControllerTest {
    private final WebTestClient client = WebTestClient.bindToServer().baseUrl(

    @Test
    public void testCreateUser() throws Exception {
        final User user = new User();
        user.setName("Test");
        user.setEmail("test@example.org");
        client.post().uri("/user")
                .contentType(MediaType.APPLICATION_JSON)
                .body(Mono.just(user), User.class)
                .exchange()
                .expectStatus().isOk()
                .expectBody().jsonPath("name").isEqualTo("Test");
    }
}
```


结束语

反应式编程范式为开发高性能 Web 应用带来了新的机会。Spring 5 中的 WebFlux 模块可以作为开发反应式 Web 应用的基础。由于 Spring 框架的流行，WebFlux 会成为开发 Web 的重要趋势之一。本文对 Spring 5 中的 WebFlux 模块进行了详细的介绍，包括如何用 WebFlux 开发 HTTP、SSE 和 WebSocket 服务器端应用，以及作为客户端来访问 HTTP、SSE 和 WebSocket 服务。对于 WebFlux 的基于 Java 注解和函数式编程等两种模型都进行了介绍。最后介绍了如何测试 WebFlux 应用。

参考资源

- 参考 WebFlux 的参考指南，了解 WebFlux 的更多内容。
- 查看使用 Reactor 进行反应式编程一文，了解 Reactor 项目的更多内容。
- 了解 Reactor Netty 的更多内容。

内容