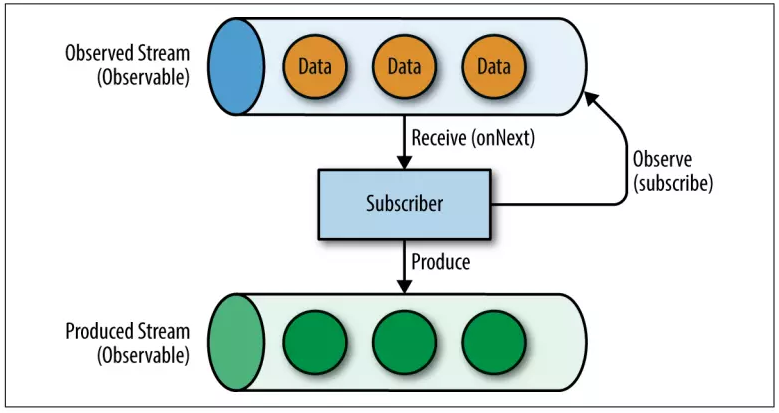
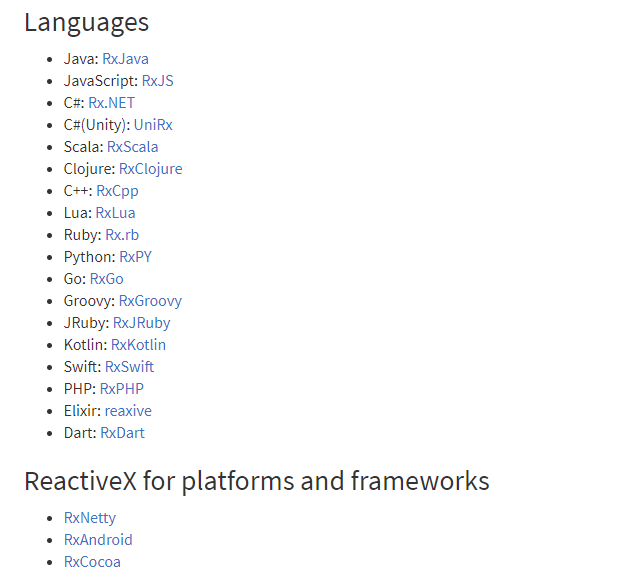
# Java Reactive 编程

Reactive定义为“给刺激信号一个反馈”，reactive程序会响应刺激信号，并根据收到的信号来调整自己的行为。Reactive编程，开发模型，其专注于数据流向、对变化的反馈以及传播它们，其开发模型如下图所示：



Reactive编程以数据流和数据传播为驱动，刺激信号（消息）为数据的转移（Streams），所有的构建通过异步消息的发送和接收来交互。为了解耦发送者和接收者，发送者将消息发送到虚拟地址，同时接收者向虚拟地址注册以接收消息。虚拟地址是消息目的地的代号，例如不透明字符串或者URL，若干者可以注册到同一个地址。目前有多种编程语言实现Reactive库，如下图：



# **RxJava的基本概念**

RxJava是一个工具库，使用通用形式的观察者模式，其基本概念：

1. Observable，被观察者，用来生产发送事件，其决定什么时候触发时间以及触发怎样的事件
2. Observer，观察者，接收被观察者传送的事件，可可以在不同的线程中执行任务。在Observer中创建了处于待命状态的观察者哨兵，可以在未来某个时刻响应Observable的通知，而不需要阻塞等待Observable发射数据
3. Subscribe，订阅，被观察者和观察者通过订阅产生关系后，才具备事件发送和接收能力
4. 事件，包装事件发送中的消息，在事件的传递过程中，可以通过操作符对事件进行各种加工（转换、过滤和组合......）

Observable和Observer通过subscribe方法实现订阅关系，从而Observable可以在需要的时候发生事件来通知Observer。与传统观察者不同，RxJava的事件回调方法除了普通的onNext之外，还定义了两个特殊的事件onCompleted和onError

* onCompleted，事件队列完结，RxJava不仅把每个事件单独处理，还会把它们看做一个队列，RxJava规定，不会再有新的onNext发生时，需要触发onCompleted方法作为标志
* OnError，事件队列异常，在事件处理过程中出异常时，onError会被触发，同时队列自动终止，不允许再有事件发出
* 在正确运行的事件序列中，onCompleted和onError有且只有一个，并且是事件序列中最后一个，这二者是互斥的，即在队列中调用其中一个，不应该再调用另一个。

1. 背压（Back Pressure）

在RxJava中，会遇到被观察者发送消息太快以至它的操作符或者订阅者不能及时处理相关消息，这就需要背压（Back Pressure）。

RxJava基于Push模型，只要生产者数据准备好了就会发射出去，如果生产者比较慢则消费者会等待新的数据到来。如果生产者较快，则会很多数据发射给消费者，而不管消费者当前有没有能力处理数据，这就会导致背压。

在RxJava 2.x中Observable不支持背压，而需要改用Flowable来专门支持背压，默认队列大小128，并且要求所有的操作符强制支持背压，目前支持5中类型的背压：

* MISSING，不指定背压策略，不会对通过onNext发射的数据做缓存或者丢弃处理
* ERROR，抛出MissingBackpressureException异常
* BUFFER，Flowable的异步缓存池没有固定大小，可以无限制添加数据直到OOM
* DROP，如果Flowable的异步缓存池满了，则会丢掉将要放入缓存池中的数据
* LATEST，缓存池满则丢掉将要放入缓存池中的数据。

# **RxJava的使用**

## 示例及执行流程分析



创建Observable和Obscriber后，通过subscribe订阅事件

1) 定义Observer

*Observer<String> observer = new Observer<String>() {*

*@Override*

*public void onSubscribe(Disposable d ) {}*

*@Override*

*public void onNext(String value) {*

*System.out.println("onNext: " + value);*

*}*

*@Override*

*public void onError(Throwable throwable) {}*

*@Override*

*public void onComplete() { }*

*}*

1. 定义Observable

*Observable observable = Observable.create(new ObservableOnSubscribe<String>() {*

*@Override*

*public void subscribe(ObservableEmitter<String> emitter) throws Exception {*

*emitter.onNext("Hello"); //定义Emitter，信息发射器*

*emitter.onNext("World");*

*Thread.sleep(1000);*

*System.out.println("Hold 1 sec");*

*emitter.onNext("RxJava");*

*emitter.onComplete();*

*}*

*}*

*);*

1. 建立订阅关系

*observable.subscribe(observer);*

输出如下：

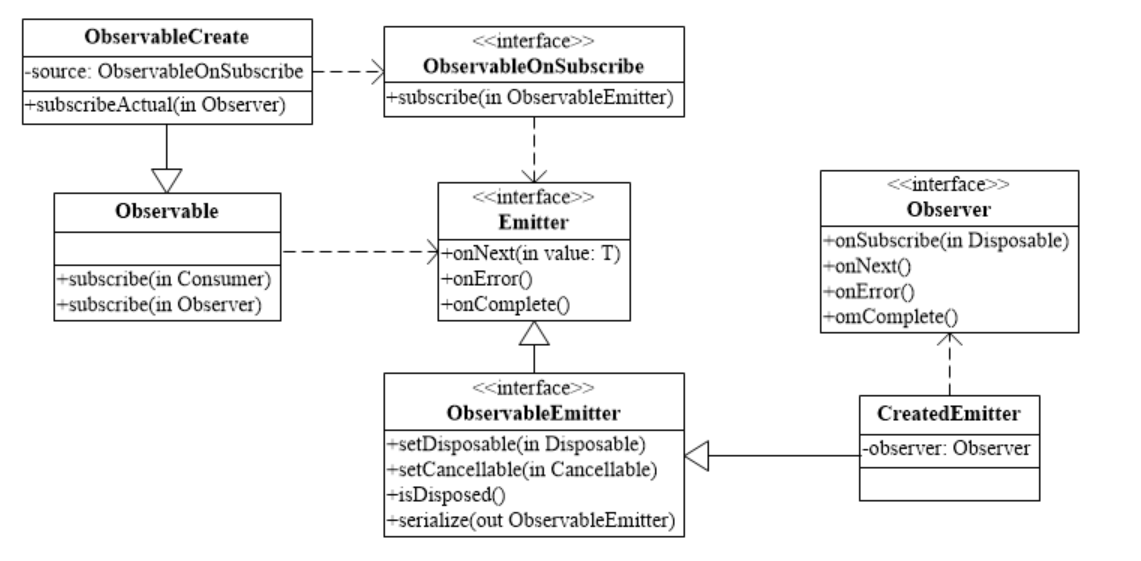
*onNext: Hello*

*onNext: World*

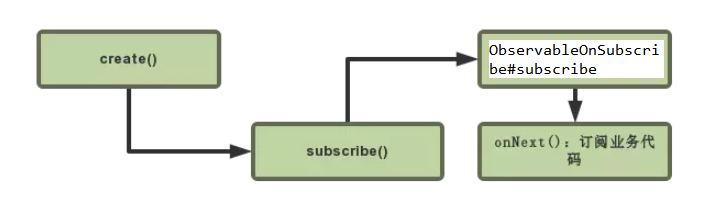
*Hold 1 sec*

*onNext: RxJava*

相关类图如下:



执行序列如下：



1. 创建Observable，其源码如下：

*Observable.create(new ObservableOnSubscribe<String>()*

*public static <T> Observable<T> create(ObservableOnSubscribe<T> source) {*

*ObjectHelper.requireNonNull(source, "source is null");*

*return RxJavaPlugins.onAssembly(new ObservableCreate<T>(source));*

*}*

*public final class ObservableCreate<T> extends Observable<T> {*

*final ObservableOnSubscribe<T> source;*

*public ObservableCreate(ObservableOnSubscribe<T> source) {*

*this.source = source;*

*}*

*@Override*

*protected void subscribeActual(Observer<? super T> observer) {*

*CreateEmitter<T> parent = new CreateEmitter<T>(observer);*

*observer.onSubscribe(parent);*

*try {*

*source.subscribe(parent);*

*} .....*

*}*

*}*

1. Observable#subscribe执行如下

*ObservableCreate#*

*protected void subscribeActual(Observer<? super T> observer) {*

*CreateEmitter<T> parent = new CreateEmitter<T>(observer);*

*observer.onSubscribe(parent);*

*try {*

*source.subscribe(parent); //ObservableOnSubscribe#subscribe,*

*} // CreateEmitter 中封装了Observer*

*}*

1. ObservableOnSubscribe#subscribe的执行，实际业务的源码如：

*@Override*

*public void subscribe(ObservableEmitter<String> emitter){*

*emitter.onNext*

*}*

其emitter的实现类为CreateEmitter，观察者模式的哨兵，其描述如下：

*static final class CreateEmitter<T> {*

*Observer<? super T> observer;*

*@Override*

*public void onNext(T t) {*

*if (!isDisposed()) { //Observer*

*observer.onNext(t);*

*}*

*}*

## **RxJava2 观察者模式**

RxJava 2.x提供五种观察者模式，如下所示：

|  |  |  |
| --- | --- | --- |
| 类型 | 描述 | 消费者 |
| Observable | 能够发射0或n个数据，并以成功或者错误事件终止 | Observer |
| Flowable | 能够发射0或n个数据，并以成功或者错误事件终止。支持背压，可以控制数据源发射的速度 | Consumer |
| Single | 只发射单个数据或错误事件 | SingleObserver |
| Completable | 从来不发射数据，只处理onComplete和onError事件，可以看成Rx的Runnable | CompletableObserver |
| Maybe | 能够发射0或者1个数据，要么成功，要么失败，类似于Optional | MaybeObserver |

### Observale

在RxJava中，Observable有Hot和Cold两种类型

1. Cold Observale

Code Observable模式下，只有观察者订阅了，才开始执行发射数据流的代码，并且Cold Observale和Observer只能一对一的关系。Observale的just、create、range和fromXXX等操作符都能生成Cold Observale。

*Consumer<Long> subscriber1 = new Consumer<Long>() {*

*@Override*

*public void accept(Long aLong) throws Exception {*

*System.out.println("Subscriber1: " + aLong);*

*}*

*};*

*Consumer<Long> subscriber2 = new Consumer<Long>() {*

*@Override*

*public void accept(Long aLong) throws Exception {*

*System.out.println("Subscribr2: " + aLong);*

*}*

*};*

*Observable<Long> observable = Observable.create(*

*new ObservableOnSubscribe<Long>() {*

*@Override*

*public void subscribe(ObservableEmitter<Long> observableEmitter) throws Exception {*

*Observable.interval(10,*

*TimeUnit.MICROSECONDS,*

*Schedulers.computation())*

*.take(Integer.MAX\_VALUE)*

*.subscribe(observableEmitter::onNext);*

*}});*

*observable.subscribe(subscriber1);*

*observable.subscribe(subscriber2);*

*try {*

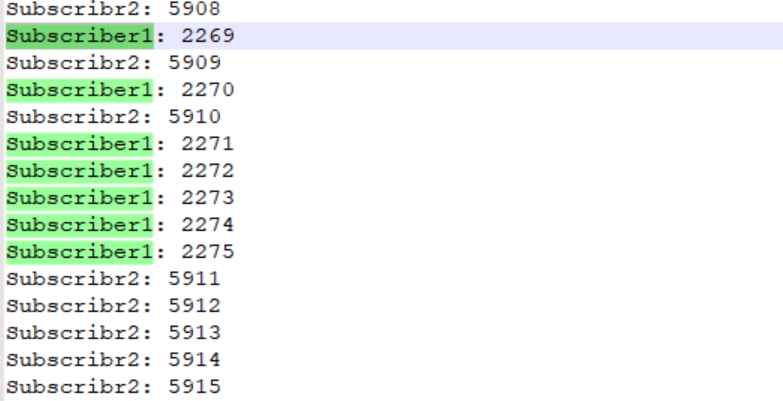
*Thread.sleep(100L);*

*} catch (InterruptedException ex) {*

*ex.printStackTrace();*

*}*

测试执行结果如下：



对于Cold Observable来说，有多个Observer的时候，其各自的事件时独立的，在输出的时候，Subscriber的结果不一定相同。

1. Hot Observale

Hot Observable模式下，无论有没有观察者进行订阅，事件始终都会发生，其与订阅者的关系是一对多的关系。

可以通过publish操作符，可以让Cold Observale转换成Hot Observale，可以将原先的Observable转换成ConnectableObservable。

*ConnectableObservable<Long> observable = Observable.create(*

*new ObservableOnSubscribe<Long>() {*

*@Override*

*public void subscribe(ObservableEmitter<Long> observableEmitter) throws Exception {*

*Observable.interval(10,*

*TimeUnit.MICROSECONDS,*

*Schedulers.computation())*

*.take(Integer.MAX\_VALUE)*

*.subscribe(observableEmitter::onNext);*

*}*

*}).observeOn(Schedulers.newThread()).publish();*

*observable.connect();*

其生成的ConnectableObservable需要调用connect才能真正的执行，执行的结果如下：



多个订阅subscriber共享同一事件，其输出相同。

### **Flowable**

Flowable是RxJava 2.x新增的被观察者，可以看成Observable新的实现，其支持背压，同时实现Reactive Streams的Publisher接口。Flowable与Observable的使用场景不同：

* Observable，处理最大不超过1000条数据，不会出现内存溢出，用于处理同步流
* Flowable，处理以某种方式生成超过10KB数据；文件读取与分析；读取数据库记录；网络I/O流；创建一个响应式非阻塞接口

Flowable使用示例如下：

*Flowable<Integer> flow = Flowable.create(*

*new FlowableOnSubscribe<Integer>() {*

*@Override*

*public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {*

*while(true) {*

*emitter.onNext(new Random().nextInt());*

*}*

*}*

*}, BackpressureStrategy.ERROR*

*);*

*flow.subscribeOn(Schedulers.newThread())*

*.observeOn(AndrioidSchedulers.newThread())*

*.subscribe(System.out::println);*

### **Single**

SinglEmitter中可以看出，Single只有onSuccess和onError事件

*public interface SingleEmitter<T> {*

*void onSuccess(@NonNull T var1);*

*void onError(@NonNull Throwable var1);*

*......*

*}*

其中,onSuccess用于发射数据，在Observable/Flowable中使用onNext来发射数据，而且只能发射一个数据。其示例如下：

*Single.create(new SingleOnSubscribe<String>() {*

*@Override*

*public void subscribe(@NonNull SingleEmitter<String> se) throws Exception {*

*se.onSuccess("test");*

*}*

*}).subscribe(*

*msg -> System.out.println(msg),*

*throwable -> throwable.printStackTrace()*

*);*

Single可以通过toXXX方法转换成Observable、Flowable、Completable及Maybe

### **Completable**

Completable创建后，不会发射任何数据，接口如下：

*public interface CompletableEmitter {*

*void onComplete();*

*void onError(@NonNull Throwable var1);*

*......*

*}*

Completable只有onComplete和onError事件，同时Completable中没有map/flatMap等操作符。可以通过fromXXX操作符来创建Completable，其示例如下：

*Completable.fromAction(*

*new Action() {*

*@Override*

*public void run() throws Exception {*

*System.out.println("Hello World");*

*}*

*}*

*).subscribe();*

### **Maybe**

Maybe可以看出是Single和Completable的结合，Maybe创建后MaybeEmitter和SingleEmitter相同，并没有onNext方法，同样通过onSuccess方法来发射数据，执行如下：

*Maybe.create( maybeEmitter -> {*

*maybeEmitter.onSuccess("testA");}*

*).subscribe(System.out::println);*

执行如下：testA

Maybe只发射0或者1个数据，即使发射多个数据，后面发射的数据也不会处理。

## **RxJava线程操作**

RxJava是一个为异步编程而实现的库，合理利用异步编程能够提高系统的处理速度。但是异步也会带来线程的安全问题，但是异步并不等于并发。

在默认情况下RxJava只在当前线程中运行，其是单线程的。此时Observable用于发射数据流，Observer用于接收和响应数据流，在数据流上可以进行各种操作符。但是函数响应式时实际应用大部分操作都在后台处理，前台响应的一个过程。因此使用RxJava编程，Observable生成发射数据流，Operators加工数据流在后台线程中进行，Observer在前台线程中接收并响应数据。多线程的操作，可以通过RxJava的调度器Scheduler来实现。

Scheduler是RxJava对线程控制器的抽象，RxJava中内置了多个Scheduler的实现，其使用示例如下：

*Observable.just("aaa", "bbb")*

*.observeOn(Schedulers.newThread())*

*.map(*

*new Function<String, String>() {*

*@Override*

*public String apply(@NonNull String s) throws Exception {*

*System.out.println("Map Thread:" + Thread.currentThread().getName());*

*return s.toUpperCase();*

*}*

*}*

*).subscribeOn(Schedulers.single())*

*.observeOn(Schedulers.io())*

*.subscribe(*

*new Consumer<String>() {*

*@Override*

*public void accept(String s) throws Exception {*

*System.out.println("Subscribe Thread:" + Thread.currentThread().getName());*

*System.out.println("Msg:" + s);*

*}*

*});*

执行输出如下：

*Map Thread: RxNewThreadScheduler-1 => Scheduler.newThread*

*Subscribe Thread: RxCachedThreadScheduler-1 => Scheduler.io*

*Msg:AAA*

map及subscribe在不同的线程中执行。

除此之外还有以下执行线程：

|  |  |
| --- | --- |
| RxJava调度器 | 说明 |
| Schedulers.immediate() | 默认线程允许立即在当前线程中执行所指定的工作 |
| Schedulers.newThread() | 新建线程，启用新线程，并在新线程中执行操作 |
| Schedulers.io() | 适用于I/O操作，根据需要增长或缩减来自适应的线程池。多数情况下 io() 比 newThread() 更有效率。不要把计算工作放在 io() 中，可以避免创建不必要的线程。 |
| Schedulers.computation() | 适用于计算工作（CPU 密集型计算），即不会被 I/O 等操作限制性能的操作。这个 Scheduler 使用的固定的线程池，大小为 CPU 核数。不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU。 |
| Schedulers.trampoline() | 当我们想在当前线程执行一个任务时，并不是立即，我们可以用.trampoline()将它入队。这个调度器将会处理它的队列并且按序运行队列中每一个任务。 |
| AndroidSchedulers.mainThread() | RxAndroid 提供的，它指定的操作将在 Android 主线程运行。 |

# **Spring WebFlux**

基于Reactive Streams规范实现的框架Reactor，提供两种响应式API:

* Mono，实现发布者，并返回0或者1个元素
* Flux，实现发布者，并返回N个元素

Spring Boot Webflux及时基于Reactor实现的，在Spring Boot 2.0中包括新的Spring-webflux模块，该模块包含对响应式HTTP、WebSocket客户端和服务器推送事件的支持，以及对REST,HTML和WebSocket交互等程序的支持。

## **WebFlux示例**

1. 创建Spring-boot应用，在pom.xml中添加依赖

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-webflux</artifactId>*

*</dependency>*

1. 添加Controller

*@RestController*

*public class HelloController {*

*@GetMapping("/hello")*

*public Mono<String> hello() {*

*return Mono.just("Welcome to reactive world ~");*

*}}*

1. 启动Spring Boot应用

*@SpringBootApplication*

*public class SpringForWebFlux {*

*public static void main(String[] args) {*

*SpringApplication.run(SpringForWebFlux.class, args);*

*}}*

启动应用，其运行在Netty上

*r.ipc.netty.tcp.BlockingNettyContext : Started HttpServer on /0:0:0:0:0:0:0:0:8081*

*o.s.b.web.embedded.netty.NettyWebServer : Netty started on port(s): 8081*

访问浏览器，输出如下：

*Welcome to reactive world From Mono ~*

Netty是异步、事件驱动的网络应用程序框架和工具，能够开发高性能、高可靠性的网络服务器和客户端程序。

## **服务器推送事件**

服务器推送事件（Server-Send Events, SSE）允许服务器端不断推送数据到客户端，相对于WebSocket而言，服务器推送事件只支持服务器端到客户端的单向数据传递。

SSE在已有的HTTP协议上使用简单易懂的文本格式来传输的数据，作为W3C的推荐规模上，其支持多种浏览器。在服务端来说，SSE是一个不断产生新数据都得流，非常适合于用反应式流来表示。下面是使用示例：

*@GetMapping("/randomNumbers")*

*public Flux<ServerSentEvent<Integer>> randomNumbers() {*

*SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");*

*return Flux.interval(Duration.ofSeconds(1))*

*.map(seq -> Tuples.of(seq, ThreadLocalRandom.current().nextInt()))*

*.map(data -> ServerSentEvent.<Integer>builder()*

*.event(dateFormat.format(System.currentTimeMillis()))*

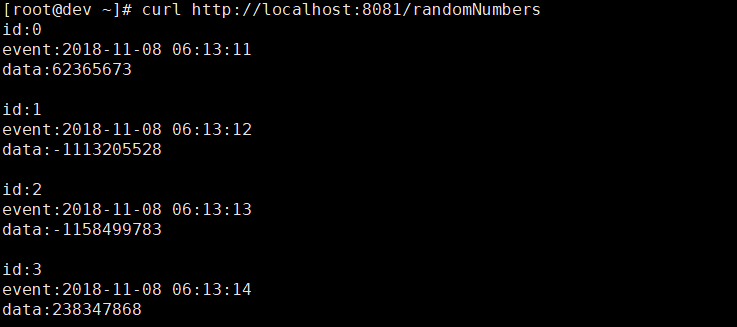
*.id(Long.toString(data.getT1()))*

*.data(data.getT2())*

*.build());*

*}*

在示例中设置返回的对象类型为Flux<ServerSentEvent>，会被自动按照SSE规范要求的格式来发送响应。启动后，调用curl命令后，输出如下：



## **WebSocket**

WebSocket支持客户端和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，其在主流浏览器上都得到支持。WebFlux也对创建WebSocket服务器端提供了支持。

在服务端，需要实现接口WebSocketHandler来处理WebSocket通讯，其方法handle的参数是接口WebSocketSession对象，可以用来获取客户端信息、接送消息和发送消息。其使用示例如下：

1. 定义Handler

*@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.getPayloadAsText())));*

*}}*

EchoHandler对于接受的消息，会发送添加“Echo ->”前缀的响应消息。WebSocketSession的receive方法返回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。

1. 注册到WebFlux中，创建WebSocketHandlerAdapter对象

*@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();*

*}*

*}*

需要创建一个类 WebSocketHandlerAdapter 的对象，该对象负责把 WebSocketHandler 关联到 WebFlux 中。代码清单中给出了相应的 Spring 配置。其中的 HandlerMapping 类型的 bean 把 EchoHandler 映射到路径 /echo。

3）WebSocketClient测试代码

*public class WSClient {*

*public static void main(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));*

*}*

*}*

执行后客户端输出信息如下：

*ECHO -> Hello*

https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html

https://blog.csdn.net/get\_set/article/details/79480233

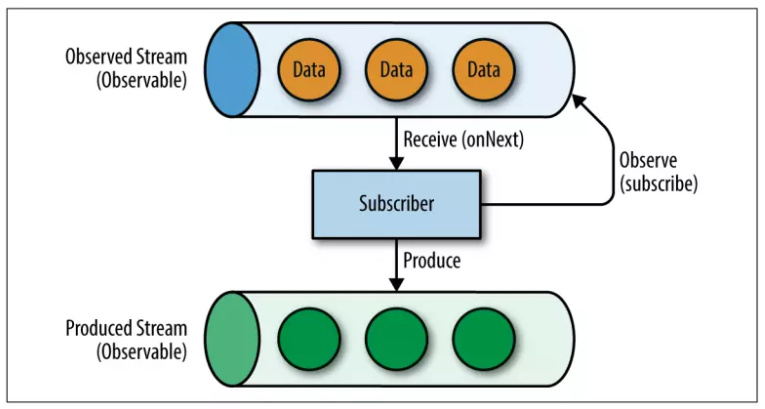
https://www.ibm.com/developerworks/cn/java/spring5-webflux-reactive/

https://blog.csdn.net/jeffli1993/article/details/79941175

MongoDb https://www.jianshu.com/p/538ab390bcfe

# **Vert.x Reactive**

Reactive编程是一种开发模型，以数据流和数据传播为驱动，在Reactive编程中刺激信号是数据转移（Streams），如下图：



其中Stream成为observables，消息者对observables进行订阅并且对其值响应。

## **RxJava示例**

1）添加依赖

在Vert.x中提供了RxJava的支持，添加依赖如下：

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-core</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-hazelcast</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

*<dependency>*

*<groupId>io.vertx</groupId>*

*<artifactId>vertx-rx-java</artifactId>*

*<version>${vertx.version}</version>*

*</dependency>*

2） 编写Receiver类

*public class Sender extends AbstractVerticle {*

*@Override*

*public void start() throws Exception {*

*EventBus eb = vertx.eventBus();*

*vertx.setPeriodic(1000,*

*v -> eb.publish("news-feed", "Some news!"));*

*}*

*}*

3）编写Sender类

*public class Receiver extends AbstractVerticle {*

*@Override*

*public void start() throws Exception {*

*EventBus eb = vertx.eventBus();*

*eb.consumer("news-feed")*

*.toObservable()*

*.subscribe( message ->*

*System.out.println("Received news: " + message.body()));*

*System.out.println("Ready!");*

*}*

*}*

4）测试

* 运行Reciver，其输出如下：

*Ready!*

* 运行Sender，在Reciver端输出如下：

*Recycler - -Dio.netty.recycler.ratio: 8*

*Received news: Some news!*

*Received news: Some news!*

## **RxJava源码分析**

RxJava MessageConsumer提供Observable<Message<T>>，

*EventBus eb = vertx.eventBus();*

*MessageConsumer<String> consumer = eb.<String>consumer("the-address");*

*Observable<Message<String>> observable = consumer.toObservable(); //Observable*

*Subscription sub = observable.subscribe(msg -> {*

*// Got message*

*});*

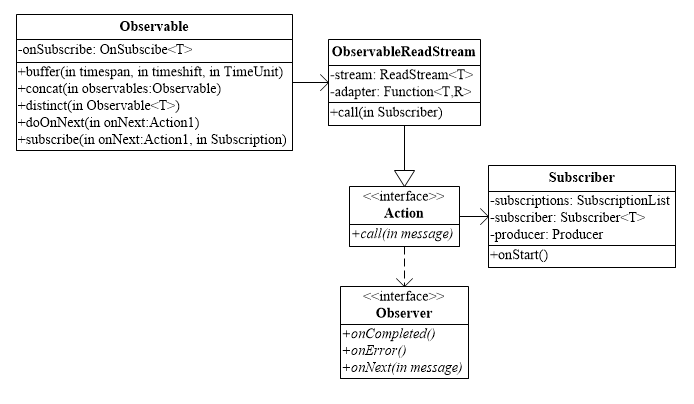
*// Unregisters the stream after 10 seconds*

*vertx.setTimer(10000, id -> {*

*sub.unsubscribe();*

*});*

Observable提供了Message流，subscribe订阅其信息流，并在其流中执行用于自定义的Action1，Observable类图如下所示：



1. Receiver -> EventBus#subscribe，将Action注册到ObservableReadStream，其执行如下：

*static <T> Subscription subscribe(Subscriber<? super T> subscriber,*

*Observable<T> observable) {*

*subscriber.onStart();*

*......*

*try {*

*RxJavaHooks.onObservableStart(observable, observable.onSubscribe).call(subscriber);*

*return RxJavaHooks.onObservableReturn(subscriber);*

*} }*

subscribe后，注册到ReadStream中。以WebSocket为例，其注册如下：

*public S handler(Handler<Buffer> handler) {*

*synchronized (conn) {*

*if (handler != null) {*

*checkClosed();*

*}*

*this.dataHandler = handler;*

*return (S) this;*

*}*

*}*

Subscriber启动后，从ObservableReadStream（EventBus）中读取数据

1. Sender，通过EventBus#pubish生成数据，将其传入到Receiver的Event Bus中，其执行过程已经分析过，这里不再介绍，其执行示例如下：

*vertx.setPeriodic(1000,*

*v -> eb.publish("news-feed", "Some news!"));*

1. Receiver获取数据流，并调用Action（封装成handler）

各数据流入WebSocket/NetSocket/HttpServer等，以WebSocketImpi为例，其通过Handler处理数据，如下所示：

*private void checkNextTick() {*

*if (!paused && queueing && !sentCheck) {*

*sentCheck = true;*

*context.runOnContext(v -> {*

*synchronized (Http1xServerConnection.this) {*

*sentCheck = false;*

*if (!paused) {*

*Object msg = pending.poll();*

*if (pending.isEmpty()) {*

*processMessage(msg);*

*checkNextTick();*

*} }}); }*

*}*

processMessage中调用handler处理message，其如下所示：

*void handleFrame(WebSocketFrameInternal frame) {*

*synchronized (conn) {*

*if (frame.type() != FrameType.CLOSE) {*

*conn.reportBytesRead(frame.length());*

*if (dataHandler != null) {*

*dataHandler.handle(frame.binaryData());} }......}*

# **Hystrix**

参考链接：

<https://www.jianshu.com/p/b469f1c7071d>

<https://blog.csdn.net/king866/article/details/55215337>

https://github.com/reactive-streams/reactive-streams-jvm/tree/v1.0.2

https://github.com/yuxingxin/RxJava-Essentials-CN

http://www.reactive-streams.org/

https://github.com/Froussios/Intro-To-RxJava/blob/master/Part%201%20-%20Getting%20Started/1.%20Why%20Rx.md

RxJava Essential

RxJava中文翻译：https://github.com/yuxingxin/RxJava-Essentials-CN

https://github.com/lzyzsd/Awesome-RxJava

RxJava系列

https://zhuanlan.zhihu.com/p/20687178