函数响应式编程（FRP）

在传统的编程范式中，一般通过迭代器（Iterator）模式来遍历一个序列，这种遍历方式是由调用者来控制节奏，采用的是拉的方式，每次由调用者通过next方法来获取序列中的下一个值。在响应式编程中采用的是推的方式，即常见的发布-订阅模式，当发布者有新的数据产生时，这些数据会被推送到订阅者来进行处理，在响应流上可以添加各种操作对数据进行处理，形成数据处理链，这种以声明式添加处理链只在订阅者进行订阅操作时才会真正执行。

响应式编程是一种编程风格，提供基于事件驱动的方式来处理异步数据流的能力，其遵循观察者设计模式，当事物发生状态变化时，其他事物将被相应地通知和更新，观察者不需要轮询事件的变化，而是异步等待事件变化的通知，观察者收到通知后处理该事件。响应式编程来源于数据流和状态变化的流转，底层执行框架通过数据流来自动传播变化，比如：

*c = a+b //简单的表达式求值*

当a或者b的值发生变化时，传统的编程范式需要对a+b进行重新计算来得到c的值。使用响应式编程，当值变化时，c的值会自动更新。使用响应式编程允许开发人员构建事件驱动、可扩展性、弹性的响应式系统。

RxJava是JVM响应式编程的库，也是反应式流规范的基础，Spring5使用Reactor作为响应式编程的基础，其基本概念是Flux和Mono

# Flux

Flux表示包含0到N个元素的异步序列，在该序列中可以包含三种不同类型的消息通知，正常包含元素的消息、序列结束的消息和序列出错的消息。当消息通知产生时，订阅者对应的方法：onNext、onComplete和onError会被调用，Flux类定义如下：

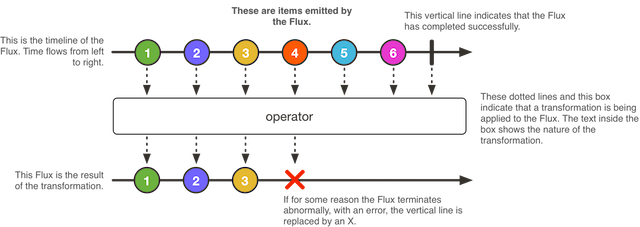
public abstract class Flux<T> implements Publisher<T>

其实现接口Publisher，用于提供数据：

public interface Publisher<T> {

public void subscribe(Subscriber<? super T> s);

}



从图中可以看出Flux是标准的异步有序序列，通过接口Publisher<T>不停产生数据，直到遇到终止及error信号。Flux的创建有多种方式：

1. Flux类的静态方法

* just，可以指定序列中包含的全部元素，创建出来的Flux序列在发布这些元素之后会自动结束
* fromArray/fromIterable/fromStream，可以从数组、Iterable对象或者Stream对象创建Flush
* empty，创建一个不包含任何元素，只发布结束消息的队列
* never，创建一个不包含任何消息通知的序列
* range（start,count）,创建从start起始count个数量的Integer对象序列
* interval(period:Duration,delay:Duration)，创建一个包含从0开始递增的Long对象的序列，其中包含的元素按照指定时间间隔来发布，通过delay指定起始元素的延迟时间
* intervalMillis(delay,period)，与interval方法相同

下面是使用示例：

Flux.just("Hello","World").subscribe(System.out::println);  
Flux.fromArray(new Integer[] {1,2,3}).subscribe(System.out::println);  
Flux.empty().subscribe(System.out::println);  
Flux.range(1,10).subscribe(System.out::println);  
Flux.interval(Duration.of(10, ChronoUnit.SECONDS)).subscribe(System.out::println);

上面这些静态方法适合简单的序列生成。

当序列的生成需要复杂的逻辑时，则应该使用generate或者create方法

1. generate方法

通过同步和逐一的方式来生成Flux序列，序列的产生是调用所提供的SynchronousSink对象的next、complete和error方法来完成，逐一生成的含义是在具体的生成逻辑中，next()方法只能最多被调用一次。

在有些情况下，序列的生成可能是有状态的，需要用到某些状态对象，此时可以使用generate方法的另外一种形式：

*generate(Callable<S> stateSupplier, BiFunction<S,SynchronousSink<T>,S> generator)*

其中StateSupplier用来提供初始的状态对象。在进行序列生成时，状态对象会作为 generator 使用的第一个参数传入，可以在对应的逻辑中对该状态对象进行修改以供下一次生成时使用

使用示例如下：

final Random random = new Random();  
Flux.generate(ArrayList::new,  
 (list,sink) -> {  
 int value = random.nextInt(100);  
 list.add(value);  
 sink.next(value);  
 if(list.size() == 10){  
 sink.complete();}  
 return list;  
 }  
).subscribe(System.out::println);

1. create

create方法与generate方法的不同之处在于所使用的FluxSink对象，其支持同步和异步的消息产生，并且可以在一次调用中产生多个元素，其使用示例如下：

Flux.*create*(sink -> {  
 for(int i = 0;i< 10;i++) {  
 sink.next(i);  
 }  
 sink.complete();  
}).subscribe(System.*out*::println);

# **Mono**

Mogo，表示包含0或者1个元素的异步序列，该序列同样可以包含与Flux相同的三种类型的消息通知。Flux和Mogo之间可以进行转换，对一个Flux序列进行计数操作，得到的结果是一个Mogo<Long>对象，把两个Mogo序列合并在一起，得到Flux对象。

Mogo的创建方式与Flux类似，Mogo也包括一些与Flux类中相同的静态方法，这些方法包括：just,empty,error及never等，下面是独有的方法：

* fromCallable,fromCompletionStage,fromFuture、fromRunnable及fromSupplier，分别从Callable,CompletionStage，CompletableFuture、Funnable和Supplier中创建Mogo
* delay(Duration duration)和delayMillis(long duration)，创建Mogo序列，在指定的延迟时间之后，产生数字0作为唯一值
* ignoreElements（Publisher<T> source），创建Mogo序列，忽略作为源Publisher中所有元素，只产生结束消息
* justOrEmpty(Optional<? extends T> data)和justOrEmpty(T data)，从一个Optional对象或可能为null对象中创建Mogo，只有Optional对象中包含值或对象不含null时，Mono序列才产生对应的元素

可以通过create方法来使用MonoSink来创建Mono。

使用示例如下：

Mono.*fromSupplier*(() -> "Hello").subscribe(  
 System.*out*::println  
);  
Mono.*justOrEmpty*(Optional.*of*("Hello")).subscribe(System.*out*::println);  
Mono.*create*(sink -> sink.success("Hello")).subscribe(System.*out*::println);

# **操作符**

reactor的强大之处在于可以在反应流上通过声明式的方式添加多种不同操作符，下面是重要的操作符

1）buffer 和 bufferTimeout

这两个操作符的作用是把当前流中的元素收集到集合中，并把集合对象作为流中的新元素。在进行收集时可以指定不同的条件：所包含的元素的最大数量或收集的时间间隔

* buffer,仅使用一个条件
* bufferTimeout，同时指定两个条件
* bufferMillis，指定时间间隔
* bufferTimeoutMillis

除了元素数量和时间间隔之外，还可以通过 bufferUntil 和 bufferWhile 操作符来进行收集。这两个操作符的参数是表示每个集合中的元素所要满足的条件的 Predicate 对象。bufferUntil 会一直收集直到 Predicate 返回为 true。使得 Predicate 返回 true 的那个元素可以选择添加到当前集合或下一个集合中；bufferWhile 则只有当 Predicate 返回 true 时才会收集。一旦值为 false，会立即开始下一次收集。

下面是buffer相关操作符的使用示例

Flux.*range*(1,100).buffer(20).subscribe(System.*out*::println);  
Flux.*range*(1,10).bufferUntil(i -> i%2 == 0).subscribe(System.*out*::println);  
Flux.*range*(1,10).bufferWhile(i -> i%2 ==0)

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

1. filter

对流中包含的元素进行过滤，只留下满足Predicate指定条件的元素，如下面示例输出1-10中所有偶数

Flux.*range*(1,10).filter(i -> i %2 == 0).subscribe(System.*out*::println);

1. window

window操作符的作用类似于buffer，所不同的是window操作符是把当前流中的元素收到另外的Flux序列中，因此返回值的类型是Flux<Flux<T>>

Flux.range(1, 100).window(20).subscribe(System.out::println);

Window操作符所产生的流中包含的是UnicastProcessor类的对象，而UnicastProcessor类的toString方法输出的就是UnicastProcessor字符。

1. merge和mergeSequential

merge 和 mergeSequential 操作符用来把多个流合并成一个 Flux 序列。不同之处在于 merge 按照所有流中元素的实际产生顺序来合并，而 mergeSequential 则按照所有流被订阅的顺序，以流为单位进行合并。

代码清单中分别使用了 merge 和 mergeSequential 操作符。进行合并的流都是每隔 100 毫秒产生一个元素，不过第二个流中的每个元素的产生都比第一个流要延迟 50 毫秒。在使用 merge 的结果流中，来自两个流的元素是按照时间顺序交织在一起；而使用 mergeSequential 的结果流则是首先产生第一个流中的全部元素，再产生第二个流中的全部元素。

Flux.merge(Flux.intervalMillis(0, 100).take(5), Flux.intervalMillis(50, 100).take(5))

        .toStream()

        .forEach(System.out::println);

Flux.mergeSequential(Flux.intervalMillis(0, 100).take(5), Flux.intervalMillis(50, 100).take(5))

        .toStream()

        .forEach(System.out::println);

5）flatMap和flatMapSequential

flatMap 和 flatMapSequential 操作符把流中的每个元素转换成一个流，再把所有流中的元素进行合并。flatMapSequential 和 flatMap 之间的区别与 mergeSequential 和 merge 之间的区别是一样的。

在代码清单中，流中的元素被转换成每隔 100 毫秒产生的数量不同的流，再进行合并。由于第一个流中包含的元素数量较少，所以在结果流中一开始是两个流的元素交织在一起，然后就只有第二个流中的元素

Flux.just(5, 10)

        .flatMap(x -> Flux.intervalMillis(x \* 10, 100).take(x))

        .toStream()

        .forEach(System.out::println);

1. concatMap

concatMap 操作符的作用也是把流中的每个元素转换成一个流，再把所有流进行合并。与 flatMap 不同的是，concatMap 会根据原始流中的元素顺序依次把转换之后的流进行合并；与 flatMapSequential 不同的是，concatMap 对转换之后的流的订阅是动态进行的，而 flatMapSequential 在合并之前就已经订阅了所有的流。

1. combineLatest

combineLatest 操作符把所有流中的最新产生的元素合并成一个新的元素，作为返回结果流中的元素。只要其中任何一个流中产生了新的元素，合并操作就会被执行一次，结果流中就会产生新的元素。在 代码清单中，流中最新产生的元素会被收集到一个数组中，通过 Arrays.toString 方法来把数组转换成 String。

Flux.combineLatest(

        Arrays::toString,

        Flux.intervalMillis(100).take(5),

        Flux.intervalMillis(50, 100).take(5)

).toStream().forEach(System.out::println);

# **消息处理**

当需要处理Flux或者Mono消息时，可以通过subscribe方法来添加相应的订阅逻辑，在调用subscribe方法时可以指定需要处理的消息类型，可以只处理其中包含的正常消息，也可以同时处理错误消息和完成消息。如下面，通过subscribe方法同时处理正常消息和错误消息：

Flux.*just*(1,2).concatWith(Mono.*error*(new IllegalStateException()))  
 .subscribe(System.*out*::println,System.*err*::println);

正常的消息处理相对简单，当出现错误时，有多种不同的处理策略

* 第一种策略是通过onErrorReturn方法返回一个默认值，例如下面代码中当出现错误时，流会产生默认值0
* 第二种策略是通过switchOnError方法来使用另外的流来产生元素，如下所示，当出现错误时，将产生Mono.just(0)对应的流，也就是数字0
* 第三种策略是通过onErrorResumeWith方法来根据不同的异常类型来选择要使用的产生元素的流，根据异常类型来返回不同的流作为出现错误时的数据来源，如下：异常类型为IllegalArgumentException，所产生的元素为-1

Flux.*just*(1,2).concatWith(Mono.*error*(new IllegalStateException()))  
 .onErrorResume( e -> {  
 if (e instanceof IllegalStateException) {  
 return Mono.*just*(1);  
 } else if(e instanceof IllegalArgumentException){  
 return Mono.*just*(0);  
 }  
 return Mono.*empty*();  
 }).subscribe(System.*out*::println);

* 当出现错误时，还可以通过retry操作符来进行重试，重试的动作是通过重新订阅序列来实现，在使用retry操作时可以指定重试的次数

Flux.*just*(1,2)  
 .concatWith(Mono.*error*(new IllegalStateException()))  
 .retry(1)  
 .subscribe(System.*out*::println);

# **调度器**

前面介绍了响应式流和在其上的可以进行的各种操作，通过调度器（Scheduler）可以指定这些操作执行的方式和所在的线程，有几种不同的调度器实现：

* 当前线程，通过 Schedulers.immediate()方法来创建。
* 单一的可复用的线程，通过 Schedulers.single()方法来创建。
* 使用弹性的线程池，通过 Schedulers.elastic()方法来创建。线程池中的线程是可以复用的。当所需要时，新的线程会被创建。如果一个线程闲置太长时间，则会被销毁。该调度器适用于 I/O 操作相关的流的处理。
* 使用对并行操作优化的线程池，通过 Schedulers.parallel()方法来创建。其中的线程数量取决于 CPU 的核的数量。该调度器适用于计算密集型的流的处理。
* 使用支持任务调度的调度器，通过 Schedulers.timer()方法来创建。
* 从已有的 ExecutorService 对象中创建调度器，通过 Schedulers.fromExecutorService()方法来创建。

某些操作符默认就是已经使用特定类型的调度器，比如intervalMillis方法创建的流就使用由Schedulers.timer创建的调度器。同时publishOn和subscribeOn方法可以切换执行操作的调度器：

* publishOn，切换操作符的执行方式
* subscribeOn，切换的是产生流中元素时的执行方式

下面的代码清单

1. 使用create方法创建一个新的Flux对象，其中包含唯一的元素是当前线程的名称
2. 接着是两对publishOn和map方法，其作用是先切换执行时的调度器，再把当前的线程名称加前缀
3. 最后通过subscribeOn方法来改变流产生时的执行方式

代码清单如下：

Flux.*create*( sink -> {  
 sink.next(Thread.*currentThread*().getName());  
 sink.complete();})  
 .publishOn(Schedulers.*single*())  
 .map(x -> String.*format*("[%s] %s", Thread.*currentThread*().getName(), x) )  
 .publishOn(Schedulers.*elastic*())  
 .map(x -> String.*format*("[%s] %s", Thread.*currentThread*().getName(), x))  
 .subscribeOn(Schedulers.*parallel*())  
 .toStream()  
 .forEach(System.*out*::println);

使用 create()方法创建一个新的 Flux 对象，其中包含唯一的元素是当前线程的名称。接着是两对 publishOn()和 map()方法，其作用是先切换执行时的调度器，再把当前的线程名称作为前缀添加。最后通过 subscribeOn()方法来改变流产生时的执行方式。运行之后的结果是[elastic-2] [single-1] parallel-1。最内层的线程名字 parallel-1 来自产生流中元素时使用的 Schedulers.parallel()调度器，中间的线程名称 single-1 来自第一个 map 操作之前的 Schedulers.single()调度器，最外层的线程名字 elastic-2 来自第二个 map 操作之前的 Schedulers.elastic()调度器。

https://www.ibm.com/developerworks/cn/java/j-cn-with-reactor-response-encode/index.html

Spring WebFlux

<https://projectreactor.io/docs/core/release/reference/#producing>

http://hao.jobbole.com/reactor/

http://projectreactor.io/docs

https://github.com/reactor/reactor-core

https://github.com/ReactiveX/RxJava

FRP(ReactiveCocoa) ,函数响应式编程

http://www.cnblogs.com/yxysuanfa/p/7348966.html

http://wiki.jikexueyuan.com/project/android-weekly/issue-145/introduction-to-RP.html

<https://blog.csdn.net/asce1885/article/details/43868133?utm_source=tuicool>

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