

谷粒商城

WebFlux



一、Reactive Programming（响应式编程）

响应式编程(reactive programming)是一种基于数据流(data stream)和变化传递(propagation of change)的声明式(declarative)的编程范式。

推荐博文: <https://blog.51cto.com/liukang/2090163>

1、变化传递(propagation of change)

举个简单的例子, 某电商网站正在搞促销活动, 任何单品都可以参加“满 199 减 40”的活动, 而且“满 500 包邮”。

| 购物车 | | | | | 付款 | |
|------|--------|----|--------|---------|------------|--------|
| 商品 | 单价 | 数量 | 商品金额 | 满199减40 | 订单总金额 | 521.20 |
| 饼干 | 8.50 | 5 | 42.50 | 42.50 | 邮费(满500包邮) | 0.00 |
| 干果 | 39.90 | 2 | 79.80 | 79.80 | 最终应付款 | 521.20 |
| 玉米油 | 42.90 | 1 | 42.90 | 42.90 | | |
| 牛肉干 | 59.00 | 1 | 59.00 | 59.00 | | |
| 长粒香米 | 39.00 | 2 | 78.00 | 78.00 | | |
| 剃须刀 | 259.00 | 1 | 259.00 | 219.00 | | |

| B | C | D | E | F | G | H | I |
|------|------|----|--------|----------------------|---|------------|------------------|
| 购物车 | | | | | | 付款 | |
| 商品 | 单价 | 数量 | 商品金额 | 满199减40 | | 订单总金额 | =SUM(F:F) |
| 饼干 | 8.5 | 5 | =C3*D3 | =IF(E3>199,E3-40,E3) | | 邮费(满500包邮) | =IF(I2>500,0,50) |
| 干果 | 39.9 | 2 | =C4*D4 | =IF(E4>199,E4-40,E4) | | 最终应付款 | =I2+I3 |
| 玉米油 | 42.9 | 1 | =C5*D5 | =IF(E5>199,E5-40,E5) | | | |
| 牛肉干 | 59 | 1 | =C6*D6 | =IF(E6>199,E6-40,E6) | | | |
| 长粒香米 | 39 | 2 | =C7*D7 | =IF(E7>199,E7-40,E7) | | | |
| 剃须刀 | 259 | 1 | =C8*D8 | =IF(E8>199,E8-40,E8) | | | |

“商品金额”是通过“单价 x 数量”得到的, “满 199 减 40”会判断该商品金额是否满 199 并根据情况减掉 40, 右侧“订单总金额”是“满 199 减 40”这一列的和, “邮费”会根据订单总金额计算, “最终应付款”就是订单总金额加上邮费。

响应式的核心特点之一: 变化传递(propagation of change)。一个单元格变化之后, 会像多米诺骨牌一样, 导致直接和间接引用它的其他单元格均发生相应变化。



生产者只负责生成并发出数据/事件, 消费者来监听并负责定义如何处理数据/事件的变化传

递方式。

2、数据流（data stream）

数据/事件在响应式编程里会以数据流的形式发出。

小明选购商品的过程，为了既不超预算，又能省邮费，有时加有时减



这一次一次的操作就构成了一串数据流，如果我们能够及时对数据流的每一个事件做出响应，会有效提高系统的响应水平。这是响应式的另一个核心特点：**基于数据流（data stream）**。

```
public Invoice(Cart cart) {
    ...
    this.listenOn(cart.eventStream()); // 1
    ...
}
```

`cart.eventStream()`是要监听的购物车的操作事件数据流，`listenOn`方法能够对数据流中到来的元素依次进行处理。

3、声明式（declarative）

我们再到 `listenOn` 方法去看一下：

`Invoice` 模块中，上边的一串公式被组装成如下的伪代码：

```
public void listenOn(DataStream<CartEvent> cartEventStream) {
    double sum = 0;
    double total = cartEventStream
        // 分别计算商品金额
        .map(cartEvent -> cartEvent.getProduct().getPrice() * cartEvent.getQuantity())
        // 计算满减后的商品金额
        .map(v -> (v > 199) ? (v - 40) : v)
        // 将金额的变化累加到 sum
        .map(v -> {sum += v; return sum;})
        // 根据 sum 判断是否免邮，得到最终总付款金额
        .map(sum -> (sum > 500) ? sum : (sum + 50));
    ...
}
```

这是一种“**声明式（declarative）**”的编程范式。通过四个串起来的 `map` 调用，我们先声明好了对于数据流“将会”进行什么样的处理，当有数据流过来时，就会按照声明好的处理流程逐个进行处理。



命令式是面向过程的，声明式是面向结构的。

不过命令式和声明式本身并无高低之分，只是声明式比较适合基于流的处理方式。这是响应式的第三个核心特点：**声明式（declarative）**。结合“变化传递”的特点，声明式能够让基于数据流的开发更加友好。

再举个简单的例子方便理解：

```
a = 1;
```

```
b = a + 1;
```

```
a = 2;
```

这个时候，**b** 是多少呢？在 Java 以及多数语言中，**b** 的结果是 2，第二次对 **a** 的赋值并不会影响 **b** 的值。

假设 Java 引入了一种新的赋值方式 **:=**，表示一种对 **a** 的绑定关系，如

```
a = 1;
```

```
b := a + 1;
```

```
a = 2;
```

由于 **b** 保存的不是某次计算的值，而是针对 **a** 的一种绑定关系，所以 **b** 能够随时根据 **a** 的值的变化而变化

4、总结

响应式编程的“变化传递”就相当于果汁流水线的管道；在入口放进橙子，出来的就是橙汁；放西瓜，出来的就是西瓜汁，橙子和西瓜、以及机器中的果肉果汁以及残渣等，都是流动的“数据流”；管道的图纸是用“声明式”的语言表示的。

这种编程范式如何让 Web 应用更加“reactive”呢？

我们设想这样一种场景，我们从底层数据库驱动，经过持久层、服务层、MVC 层中的 **model**，到用户的前端界面的元素，全部都采用声明式的编程范式，从而搭建一条能够传递变化的管道，这样我们只要更新一下数据库中的数据，用户的界面上就相应的发生变化，岂不美哉？尤其重要的是，一处发生变化，我们不需要各种命令式的调用来传递这种变化，而是由搭建好的“流水线”自动传递。

这种场景用在哪呢？比如一个日志监控系统，我们的前端页面将不再需要通过“命令式”的轮询的方式不断向服务器请求数据然后进行更新，而是在建立好通道之后，数据流从系统源源不断流向页面，从而展现实时的指标变化曲线；再比如一个社交平台，朋友的动态、点赞和留言不是手动刷出来的，而是当后台数据变化的时候自动体现到界面上的。

二、Reactive Stream（响应式流）

为啥不用 Java Stream 来进行数据流的操作？

- Web 应用具有 **I/O 密集** 的特点，**I/O 阻塞** 会带来比较大的性能损失或资源浪费，我们需要一种 **异步非阻塞** 的响应式的库，而 Java Stream 是一种同步 API。
- 假设我们要搭建从数据层到前端的一个变化传递管道，可能会遇到数据层每秒上千次的数据更新，而显然不需要向前端传递每一次更新，这时候就需要一种流量控制能力，就像我们家里的水龙头，可以控制开关流速，而 Java Stream 不具备完善的对数据流的 **流量控制** 的能力。

具备“**异步非阻塞**”特性和“**流量控制**”能力的**数据流**，我们称之为**响应式流（Reactive Stream）**。

目前有几个实现了响应式流规范的 Java 库，这里简单介绍两个：RxJava 和 Reactor。

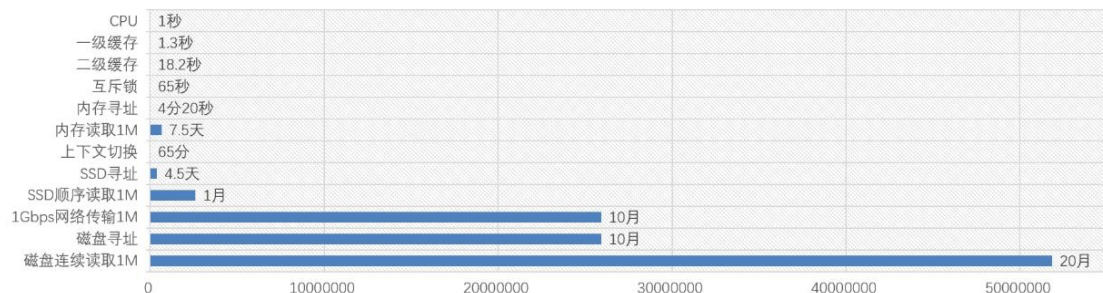
Reactor 和 Sprig 都是同一个公司 Pivotal 旗下的项目。也是 Spring5 响应式编程的底层框架在 Java 9 版本中，响应式流的规范被纳入到了 JDK 中，相应的 API 接口是 `java.util.concurrent.Flow`。

1、阻塞、非阻塞以及同步、异步

- 阻塞和非阻塞反映的是调用者的状态，当调用者调用了服务提供者的方法后，如果一直在等待结果返回，否则无法执行后续的操作，那就是阻塞状态；如果调用之后直接返回，从而可以继续执行后续的操作，那可以理解为非阻塞的。
- 同步和异步反映的是服务提供者的能力，当调用者调用了服务提供者的方法后，如果服务提供者能够立马返回，并在处理完成后通过某种方式通知到调用者，那可以理解为异步的；否则，如果只是在处理完成后才返回，或者需要调用者再去主动查询处理是否完成，就可以理解为是同步的。

互联网时代的大背景下，Web 应用通常要面对高并发、海量数据的挑战，性能从来都是必须要考量的核心因素。**阻塞**便是性能杀手之一。

CPU 眼中其他人的速度



对于阻塞造成的性能损失，我们通常有两种思路来解决：

- **并行化**：使用更多的线程和硬件资源；
 - “多线程并非银弹”，存在一些固有的弊端，但是多线程在高并发方面发挥了重要作用。况且，多线程仍然是目前主流的高并发方案。
 - 高并发环境下，多线程的切换会消耗 CPU 资源
 - 应对高并发环境的多线程开发相对比较难（需要掌握线程同步的原理与工具、ExecutorService、Fork/Join 框架、并发集合和原子类等的使用），并且有些问题难以发现或重现（比如指令重排）；
 - 高并发环境下，更多的线程意味着更多的内存占用（JVM 默认为每个线程分配 1M 的线程栈空间）
- **异步化**：基于现有的资源来提高执行效率。
 - 异步非阻塞
 - ◆ 回调。如 ajax 的 callback
 - ◆ 异步的 CompletableFuture。

2、流量控制—回压

在响应式流中，数据流的发出者叫做 Publisher，监听者叫做 Subscriber。“发布者”和“订阅者”。



问题来了，假如发布者发出数据的速度和订阅者处理数据的速度不同的时候，怎么办呢？订阅者处理速度快的话，那还好说，但是如果处理速度跟不上数据发出的速度



如果没有流量控制，那么订阅者会被发布者快速产生的数据流淹没。就像在一个流水线上，

如果某个工位处理比较慢，而上游下料比较快的话，这个工位的工人师傅就吃不消了，这个时候他需要一种途径来告诉上游下料慢一些。

同样的，订阅者也需要有一种能够向上游反馈流量需求的机制：

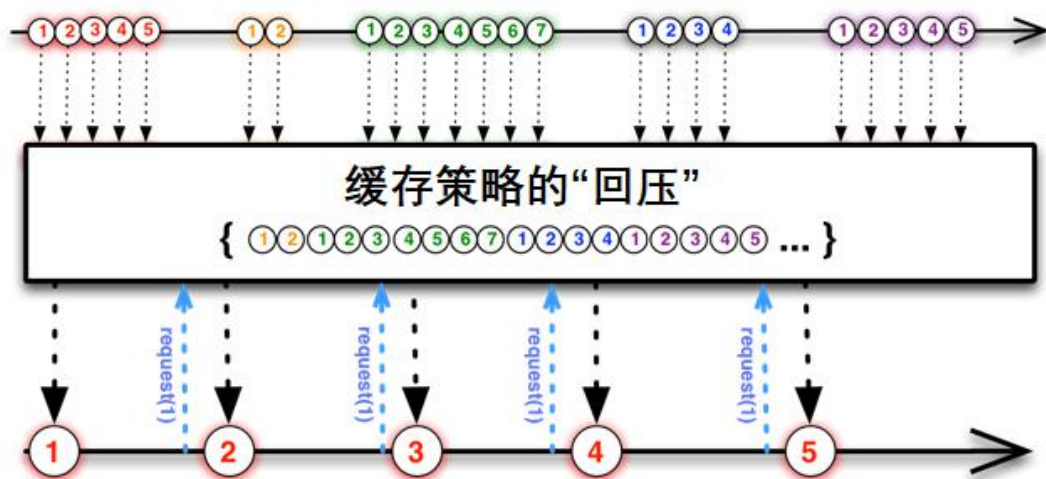


这种能够向上游反馈流量请求的机制就叫做回压（backpressure，也有翻译为“背压”的）。

在具体的使用过程中，回压的处理会涉及不同的策略。

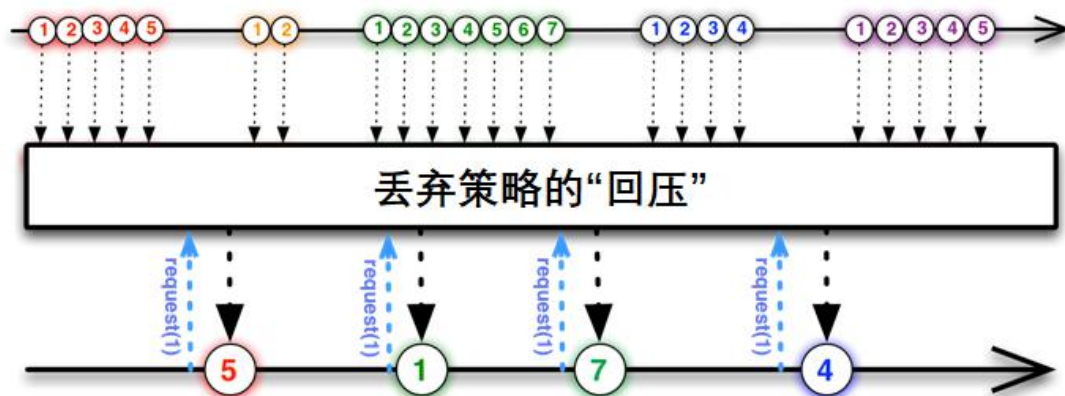
缓存的策略

发布者会将未处理的数据元素缓存起来



丢弃的策略

发布者不需要缓存来不及处理的数据，而是直接丢弃，当订阅者请求数据的时候，会拿到发布者那里最近的一个数据元素



响应式流的两个核心特点：异步非阻塞，以及基于“回压”机制的流量控制。



三、Reactor

推荐阅读：

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

1、Mono 与 Flux

1)、数据流三种信号

Reactor 两个核心概念 **Mono**、**Flux**。

Reactor 中的发布者（**Publisher**）由 **Flux** 和 **Mono** 两个类定义

既然是“数据流”的发布者，**Flux** 和 **Mono** 都可以发出三种“数据信号”：

- 元素值
- 错误信号
- 完成信号，

错误信号和完成信号都是**终止信号**，完成信号用于告知下游订阅者该数据流正常结束，错误信号终止数据流的同时将错误传递给下游订阅者。当消息通知产生时，订阅者中对应的方法 **onNext()**、**onComplete()** 和 **onError()** 会被调用。

Flux 表示的是包含 0 到 N 个元素的异步序列。

Mono 表示的是包含 0 或者 1 个元素的异步序列。

下图所示就是一个 **Flux** 类型的数据流，黑色箭头是时间轴。它连续发出“1” - “6”共 6 个元素值，以及一个完成信号（图中⑥后边的加粗竖线来表示），完成信号告知订阅者数据流已经结束。



下图所示是一个 **Mono** 类型的数据流，它发出一个元素值后，又发出一个完成信号。



上两图表示为如下代码

```
Flux.just(1, 2, 3, 4, 5, 6);
```

```
Mono.just(1);
```

Flux 和 **Mono** 提供了多种创建数据流的方法，**just** 就是一种比较直接的声明数据流的方式，

其参数就是数据元素。

```
● $ fromArray(T[]) <T> : Flux<T>  
● $ fromIterable(Iterable<? extends T>) <T> : Flux<T>  
● $ fromStream(Stream<? extends T>) <T> : Flux<T>
```

```
Integer[] array = new Integer[]{1,2,3,4,5,6};  
Flux.fromArray(array);  
List<Integer> list = Arrays.asList(array);  
Flux.fromIterable(list);  
Stream<Integer> stream = list.stream();  
Flux.fromStream(stream);
```

不过，这三种信号都不是一定要具备的：

- 首先，错误信号和完成信号都是终止信号，二者不可能同时共存；
- 如果没有发出任何一个元素值，而是直接发出完成/错误信号，表示这是一个空数据流；
- 如果没有错误信号和完成信号，那么就是一个无限数据流。

2)、创建 Flux、Mono

1、通过 Flux 类的静态方法，快速创建

- `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()`和 `fromStream()`：可以从一个数组、`Iterable` 对象或 `Stream` 对象中创建 `Flux` 对象。
- `empty()`：创建一个不包含任何元素，只发布结束消息的序列。
- `error(Throwable error)`：创建一个只包含错误消息的序列。
- `never()`：创建一个不包含任何消息通知的序列。
- `range(int start, int count)`：创建包含从 `start` 起始的 `count` 个数量的 `Integer` 对象的序列。
- `interval(Duration period)`和 `interval(Duration delay, Duration period)`：创建一个包含了从 0 开始递增的 `Long` 对象的序列。其中包含的元素按照指定的间隔来发布。除了间隔时间之外，还可以指定起始元素发布之前的延迟时间。
- `intervalMillis(long period)`和 `intervalMillis(long delay, long period)`：与 `interval()`方法的作用相同，只不过该方法通过毫秒数来指定时间间隔和延迟时间。

```
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);  
Flux.intervalMillis(1000).subscribe(System.out::println);
```

2、generate() 或 create()

generate() 序列的产生是通过调用所提供的 SynchronousSink 对象的 next(), complete() 和 error(Throwable)方法来完成。

generate() 只提供序列中单个消息的产生逻辑(同步通知), 其中的 sink.next()最多只能调用一次

```
Flux.generate(sink -> {
    sink.next("Hello");
    sink.complete();
}).subscribe(System.out::println);

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);
```

create()方法与 generate()方法的不同之处在于所使用的是 FluxSink 对象。FluxSink 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。generate 的 next 只能调用一次。

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

3、创建 Mono

Mono 的创建方式与 Flux 是很相似的。除了 Flux 所拥有的构造方式之外, 还可以支持与 Callable、Runnable、Supplier 等接口集成。

```
// 只有完成信号的空数据流
Flux.just();
Flux.empty();
Mono.empty();
Mono.justOrEmpty(Optional.empty());
```

```
// 只有错误信号的数据流
Flux.error(new Exception("some error"));
Mono.error(new Exception("some error"));
```

空的数据流有什么用？举个例子，当我们从响应式的 DB 中获取结果的时候，就有可能为空：

```
Mono<User> findById(long id);
```

```
Flux<User> findAll();
```

无论是空还是发生异常，都需要通过完成/错误信号告知订阅者，已经查询完毕，但是抱歉没有得到值。

2、subscribe；订阅前什么都不会发生

数据流有了，假设我们想把每个数据元素原封不动地打印出来：

```
Flux.just(1, 2, 3, 4, 5, 6).subscribe(System.out::print);//123456
Mono.just(1).subscribe(System.out::println);//1
```

```
// 订阅并触发数据流
```

```
subscribe();
```

```
// 订阅并指定对正常数据元素如何处理
```

```
subscribe(Consumer<? super T> consumer);
```

```
// 订阅并定义对正常数据元素和错误信号的处理
```

```
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer);
```

```
// 订阅并定义对正常数据元素、错误信号和完成信号的处理
```

```
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer);
```

```
// 订阅并定义对正常数据元素、错误信号和完成信号的处理，以及订阅发生时的处理逻辑
```

```
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer);
```

```
Flux.just(1, 2, 3, 4, 5, 6).subscribe(
    System.out::println,
    System.err::println,
    () -> System.out.println("Completed!"));
//1
//2
//3
//4
//5
//6
//Completed!
```

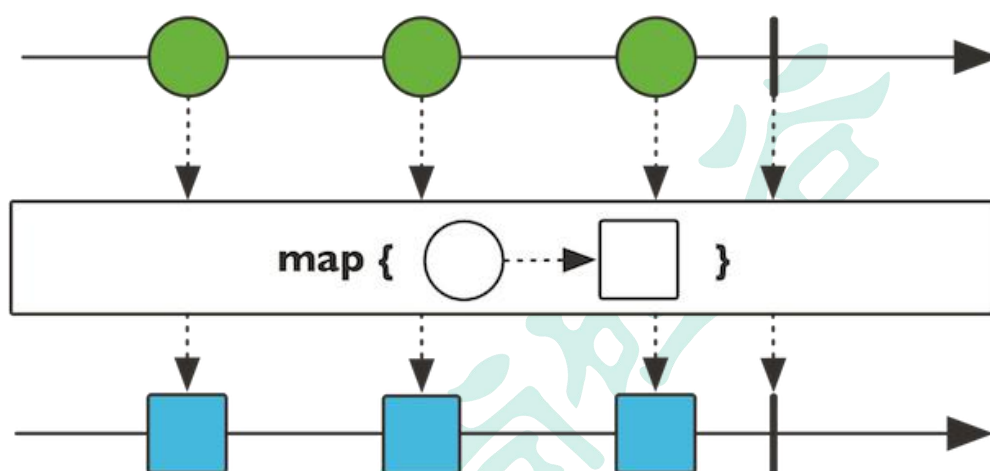
```
Mono.error(new Exception("some error")).subscribe(
    System.out::println,
```

```
System.err::println,
() -> System.out.println("Completed!")
);
//java.lang.Exception: some error
```

这里需要注意的一点是，`Flux.just(1, 2, 3, 4, 5, 6)`仅仅声明了这个数据流，此时数据元素并未发出，只有 `subscribe()`方法调用的时候才会触发数据流。所以，订阅前什么都不会发生。

3、操作符

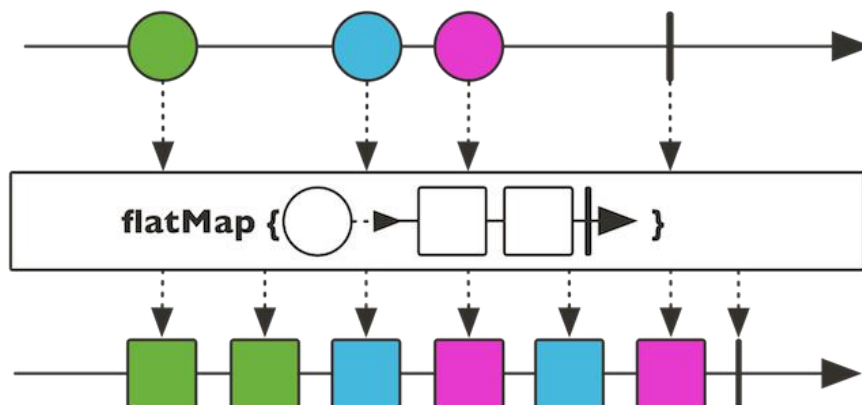
1)、map - 元素映射为新元素



```
Flux.range(1, 4).map(item->{
    return item*2;
}).subscribe(System.out::println);
//2 4 6 8
```

2)、flatMap - 元素映射为流

`flatMap` 操作可以将每个数据元素转换/映射为一个流，然后将这些流合并为一个大的数据流。

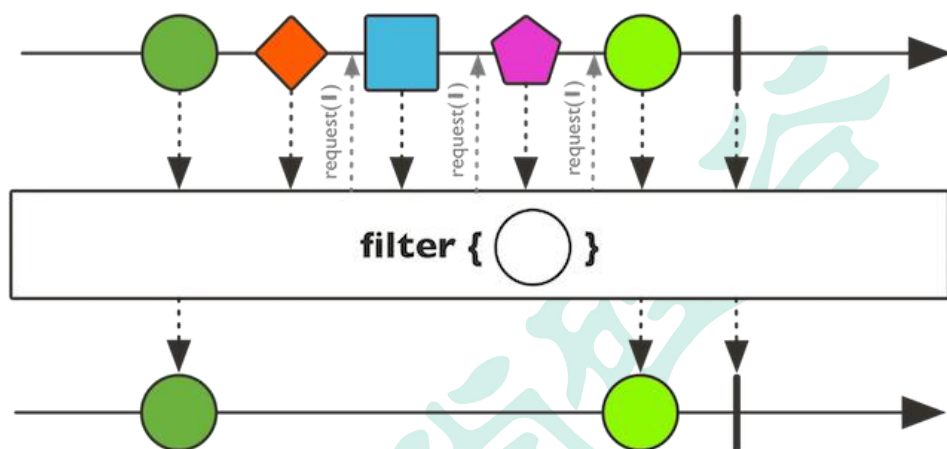


注意到，流的合并是异步的，先来先到，并非是严格按照原始序列的顺序（如图蓝色和红色方块是交叉的）。

```
Flux.just("Hello", "World")
    .flatMap(item -> Flux.fromArray(item.split("\\s*")))
// .doOnNext(System.out::println) //doOnNext 方法是“偷窥式”的方法，不会消费数据流
    .subscribe(System.out::println);
```

3）、filter - 过滤

filter 操作可以对数据元素进行筛选。

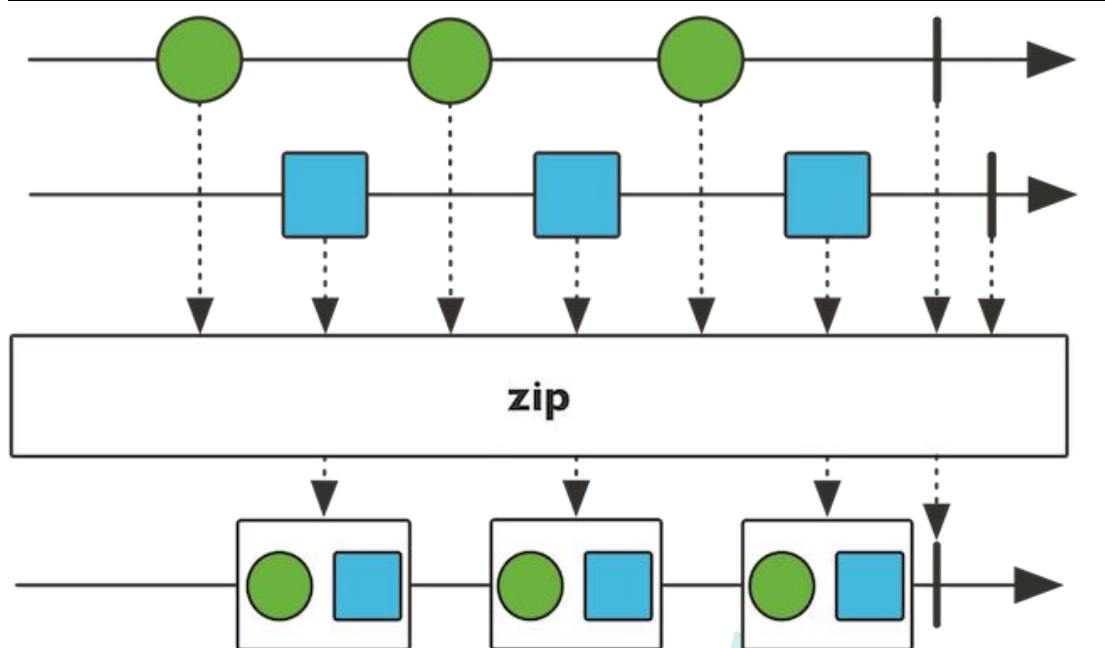


```
Flux.range(1, 10)
    .filter(p -> p % 2 == 0)
    .subscribe(System.out::println); //2,4,6,8,10
```

4）、zip - 一对一合并

将多个流一对一的合并起来。zip 有多个方法变体，我们介绍一个最常见的二合一的。

```
Flux.zip(Flux.range(1, 10), Flux.range(100, 10))
    .concatMap(item -> {
        System.out.println(item);
        Optional<Object> reduce = item.toList().stream().reduce((a, b) ->
            Integer.parseInt(a.toString()) + Integer.parseInt(b.toString()));
        return Flux.just(reduce.get());
    })
    .subscribe(System.out::println, System.err::println);
```

5)、更多

Reactor 中提供了非常丰富的操作符，除了以上几个常见的，还有：

- 用于编程方式自定义生成数据流的 `create` 和 `generate` 等及其变体方法；
- 用于“无副作用的 `peek`”场景的 `doOnNext`、`doOnError`、`doOnComplete`、`doOnSubscribe`、`doOnCancel` 等及其变体方法；
- 用于数据流转换的 `when`、`and/or`、`merge`、`concat`、`collect`、`count`、`repeat` 等及其变体方法；
- 用于过滤/拣选的 `take`、`first`、`last`、`sample`、`skip`、`limitRequest` 等及其变体方法；
- 用于错误处理的 `timeout`、`onErrorReturn`、`onErrorResume`、`doFinally`、`retryWhen` 等及其变体方法；
- 用于分批的 `window`、`buffer`、`group` 等及其变体方法；
- 用于线程调度的 `publishOn` 和 `subscribeOn` 方法。

详细：

<https://htmlpreview.github.io/?https://github.com/get-set/reactor-core/blob/master-zh/src/docs/index.html#which-operator>

4、异常处理

在前面所提及的这些功能基本都属于正常的流处理，然而对于异常的捕获以及采取一些修正

手段也是同样重要的。

利用 Flux/Mono 框架可以很方便的做到这点。

将正常消息和错误消息分别打印

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

当产生错误时默认返回 0

```
Flux.just(1, 2)
    .concatWith(Mono.error(new IllegalStateException()))
    .onErrorReturn(0)
    .subscribe(System.out::println);
```

自定义异常时的处理

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

当产生错误时重试

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

这里的 `retry(1)` 表示最多重试 1 次，而且重试将从订阅的位置开始重新发送流事件

5、调度器与线程模型

在 Reactor 中，对于多线程并发调度的处理变得异常简单。

在以往的多线程开发场景中，我们通常使用 `Executors` 工具类来创建线程池，通常有如下四种类型：

- `newCachedThreadPool` 创建一个弹性大小缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程；
- `newFixedThreadPool` 创建一个大小固定的线程池，可控制线程最大并发数，超出的线程

会在队列中等待；

- `newScheduledThreadPool` 创建一个大小固定的线程池，支持定时及周期性的任务执行；
- `newSingleThreadExecutor` 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

我们说过，响应式是异步化的，那么就会涉及到多线程的调度。

Reactor 提供了非常方便的调度器(**Scheduler**)工具方法，可以指定流的产生以及转换(计算)发布所采用的线程调度方式。

这些方式包括：

| 类别 | 描述 |
|---------------------|--------------------|
| immediate | 采用当前线程 |
| single | 单一可复用的线程 |
| elastic | 弹性可复用的线程池(IO型) |
| parallel | 并行操作优化的线程池(CPU计算型) |
| timer | 支持任务调度的线程池 |
| fromExecutorService | 自定义线程池 |

```
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 `publishOn` 指定了流发布的调度器，`subscribeOn` 则指定的是流订阅的调度器。

首先是 `parallel` 调度器进行流数据的生成，接着使用一个 `single` 单线程调度器进行发布，此时经过第一个 `map` 转换为另一个 `Flux` 流，其中的消息叠加了当前线程的名称。最后进入的是一个 `elastic` 弹性调度器，再次进行一次同样的 `map` 转换。

最终，经过多层转换后的输出如下：

`[elastic-2] [single-1] parallel-1`

四、WebFlux

1、Pom

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

2、Controller

```
@RestController
public class HelloFluxController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello");
    }
}
```

3、Service

```
@Service
public class UserService {

    public List<User> getUsers() throws Exception{
        long random = (long) ((Math.random()*50L) + 100L);
        TimeUnit.MILLISECONDS.sleep(random);
        return Arrays.asList(new User(1L, "zhangsan", 18), new
User(2L, "lisi", 19));
    }
}
```

4、压力测试效果

ab 压测, gatling 压测, jmeter 压测

从 java1.4 开始 NIO，引入 Webflux->Reactor->Reactive Streams API，如果是 CPU 密集型，reactive 没用。4 核心 8 线程，一个核心绑定 2 线程，最好，不用进行线程切换。Reactive 是一种观察模式的扩展，Future 是阻塞式的

Supplier 只出（返回）不进（参数）

Consumer 只进（参数）不出（返回）

Function 又进（参数）又出（返回）

BiFunction 二元操作 func(a,b)

为什么 Reactive？传统 Tomcat 400 最大线程，多了就阻塞，线程多切换多。
Netty，Reactor 方式。少量线程几乎不切换，基于事件方式。处理大量并发。

基于异步非阻塞的响应式应用或驱动能够以少量且固定的线程应对高并发的请求或调用，对于存在阻塞的场景，能够比多线程的并发方案提供更高的性能。

响应式和非阻塞并不是总能让应用跑的更快，况且将代码构建为非阻塞的执行方式本身还会带来少量的成本。但是在类似于 WEB 应用这样的高并发、少计算且 I/O 密集的应用中，响应式和非阻塞往往能够发挥出价值。尤其是微服务应用中，网络 I/O 比较多的情况下，效果会更加惊人。