Reactor 指南中文版 V2.0

说明

简介

Reactor 是一个基于 JVM 之上的异步应用基础库。为 Java 、Groovy 和其他 JVM 语言提供了构建基于事件和数据驱动应用的抽象库。Reactor 性能相当高，在最新的硬件平台上，使用无堵塞分发器每秒钟可处理 1500 万事件。

《Reactor 指南中文版》原文出自《Reactor Guide(http://projectreactor.io/docs/reference/)》，由多位网友在开源中国众包平台协作翻译完成，其中：

《Reactor 介绍》、《reactor 核心》由 @viemacs(http://my.oschina.net/u/2500465) 翻译， @静怡芸香(http://my.oschina.net/u/2485518) 校对；

《reactor-数据流》由 @leoxu(http://my.oschina.net/xuleo?fromerr=R4dx9orJ) 、@K6F(http://my.oschina.net/Khiyuan) 翻译及校对；

《reactor－总线》、《reactor－网络》、《扩展》由@K6F(http://my.oschina.net/Khiyuan) 翻译，@暖冰(http://my.oschina.net/u/2276921) 校对。

反馈

对《Reactor 指南中文版》有任何反馈，欢迎在以下网址评论：

http://www.oschina.net/news/68445(http://www.oschina.net/news/68445)

特别说明

《Reactor 指南中文版》由开源中国组织翻译，转载请注明出处，未经许可不得为其它商业目的使用。

Reactor 介绍

Reactor 介绍

Reactor 指南

作者：

Stephane Maldini，Jon Brisbin

2.0.7.BUILD-SNAPSHOT

翻译：

OSC众包翻译社

Reactor 介绍

Reactor 是一个基础库，可用它构建时效性**流式数据**应用，或者有**低延迟**和**容错性**要求的微/纳/皮级服务。

— 前言Preface

TL;DR

什么是 Reactor?

现在你要了解下 Reactor，不妨在你喜欢的搜索引擎里输入 *Reactive，Spring+Reactive，Asynchronous+Java* 之类的关键词，或者直接输入 *Reactor是什么货？*。简单说，Reactor 是一个轻量级 JVM 基础库，帮助你的服务或应用**高效，异步**地传递消息。

"高效"是指什么?

*消息从A传递到B时，产生很少的* *内存* \*垃圾，甚至不产生。

*解决消费者处理消息的效率低于生产者时带来的* *溢出* \*问题。

*尽可能提供非阻塞* *异步流* \*。

从经验可知(主要是 *#rage* 和 *#drunk* 的推特)，异步编程很难，而像 JVM 这类提供众多可选参数的平台则尤其困难。 Reactor 旨在帮助大多数用例真正非阻塞地运行。我们提供的 API 比 JDK 的 *java.util.concurrent* 库低级原语更高效。Reactor 提供了下列功能的替代函数 (并建议不使用 JDK 原生语句)：

• 阻塞等待：如 Future.get()

• 不安全的数据访问：如 ReentrantLock.lock()

• 异常冒泡：如 try…​catch…​finally

• 同步阻塞：如 synchronized{ }

• Wrapper分配(GC 压力)：如 new Wrapper<T>(event)

当消息传递效率成为系统性能瓶颈的时候(10k msg/s，100k msg/s，1M...)，非阻塞机制就显得尤为重要。

虽然这个有理论支持 (参见 Amdahl’s Law(http://en.wikipedia.org/wiki/Amdahl%27s\_law))，但读起来太无聊了。我们举例说明，比如你用了个 Executor(https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html) 方法：

private ExecutorService threadPool = Executors.newFixedThreadPool(8);  
  
final List<T> batches = new ArrayList<T>();  
  
Callable<T> t = new Callable<T>() { // \*1  
  
 public T run() {  
 synchronized(batches) { // \*2  
 T result = callDatabase(msg); // \*3  
 batches.add(result);  
 return result;  
 }  
 }  
};  
  
Future<T> f = threadPool.submit(t); // \*4  
T result = f.get() // \*5

1. Callable 分配 -- 可能导致 GC 压力。

2. 同步过程强制每个线程执行停-检查操作。

3. 消息的消费可能比生产慢。

4. 使用线程池(ThreadPool)将任务传递给目标线程 -- 通过 FutureTask 方式肯定会产生 GC 压力。

5. 阻塞直至 callDatabase() 回调。

在这个简单的例子中，很容易指出为什么扩容是很有限的：

• 分配对象可能产生**GC压力**，特别是当任务运行时间过长。

• 每次 GC 暂停都会影响全局性能。

• 默认，队列是**无界**的，任务会因为数据库调用而堆积。

• 积压虽然不会直接导致内存泄漏，但会带来严重副作用：GC 暂停时要扫描更多的对象；有丢失重要数据位的风险；等等 …

• 典型链式队列节点分配时会产生大量内存压力。

• **阻塞回调**容易产生恶性循环。

• 阻塞回调会降低消息生产者的效率。在实践中，任务提交后需要等待结果返回，此时流式过程几乎演变为**同步**的了。

• 会话过程抛出的任何带数据存储的异常都会以不受控的方式被传递给生产者，否定了任何通常在线程边界附近可用的容错性。

要实现完全非阻塞是很难办到的，尤其是在有着类似**微服务架构**这样时髦绰号的分布式系统的世界里。因此 Reactor 做了部分妥协，尝试利用最优的可用模式，使开发者觉得他们是在写异步纳米服务，而不是什么数学论文。

没有什么传播得比光快(除了绯闻和网红猫的视频)，正如到了某个阶段，延迟是每一个系统到都要面对的实实在在的问题。为此：

Reactor 提供的框架可以帮助减轻应用中由延迟产生的副作用，只需要增加一点点开销：

*使用了一些聪明的结构，通过启动预分配策略解决运行时* *分配问题* \*；

*通过确定信息传递主结构的* *边界* \*，避免任务的无限堆叠；

*采用主流的* *响应与事件驱动构架* *模式，提供包含反馈在内的* *非阻塞端对端流* \*；

\* 引入新的 Reactive Streams(http://projectreactor.io/docs/reference/#reactivestreams) 标准，拒绝超过当前容量请求，从而保证限制结构的有效性；

*在* *IPC(http://projectreactor.io/docs/reference/#net-overview)* *上也使用了类似理念，提供对流控制友好的* *非阻塞 IO 驱动* \*；

*开放了帮助开发者们以* *零副作用* \*方式组织他们代码的函数接口，借助这些函数来处理容错性和线程安全。

关于该项目

该项目始于 2012 年。 经过长时间的内部孵化，于 2013 年发布 Reactor 1.x 版本。 Reactor 1 在各种架构下都能成功部署，包括开源的(如 Meltdown)和商业的(如 Pivotal RTI)。2014年，我们开始与一些新兴的响应式数据流规范(http://projectreactor.io/docs/reference/gettingstarted..html#reactivestreams)合作，重新设计并于 2015年4月发布 Reactor 2.0版本。响应式数据流规范填补了*指派*机制的最后一个缺口：传输过程中，数据设置多大，才不会触发线程边界问题

同时，随着响应式扩展(http://projectreactor.io/docs/reference/gettingstarted..html#rx)日益普及，文档逐渐完善，我们也不断调整部分事件驱动和任务协作类 API。

Pivotal(http://pivotal.io/) 不仅是 Spring 框架发起者，许多员工都曾是各种 Spring 代码核心贡献者；Pivotal 也资助着 Reactor，两名 Reactor 核心开发人员在 Pitoval 工作。我们提供 Reactor 到 Spring 的整合支持，以及部分 Spring 框架重要功能的支持，如 *spring-messaging* 的 STOMP 中继代理。尽管如此，我们并不强求使用 Reactor 用户必须采用 Spring。我们为"Reactive 大众"保留可嵌入工具箱。实际上，Reactor 仅仅致力于解决异步和函数调用问题。

Reactor 遵循 Apache 2.0 许可(http://www.apache.org/licenses/LICENSE-2.0.html)，可在 GitHub(https://github.com/reactor/reactor) 上获取。

使用前提

• 使用 Reactor，Java 版本最低需要 Java 7。

• 若要充分发挥函数组件潜力，需要 Java 8 Lambdas 支持。

• 采用 Spring，Clojure 和 Groovy 扩展作为后备。

• JVM 支持**非安全**访问时 Reactor 可满负荷运行 (比如不是Android的情况)。

• 若没有非安全访问策略，基于**环形缓冲区**的所有特性都不起作用。

• Reactor 在 Maven Central 中以传统 JAR 格式打包，你可以用你所习惯的构建工具在任何 JVM 工程中安装该依赖。

构架总览

![Figure 1. The main modules present in Reactor 2.0](http://projectreactor.io/docs/reference/images/modules.png

"图1. Reactor 2.0 主要模块")

图1. Reactor 2.0 主要模块

Reactor 代码库拆分成多个子模块，便于选择所需功能，不受其他功能代码块干扰。

下面举例说明，为实现异步目标，响应式技术和 Reactor 模块该如何搭配：

• Spring XD + Reactor-Net (Core/Stream)： 使用 Reactor 作为 Sink/Source IO 驱动。

|  |  |
| --- | --- |
| **\* Grails** | **Spring + Reactor-Stream (Core)： 用 Stream 和 Promise 做后台处理。** |

• Spring Data + Reactor-Bus (Core)： 发射数据库事件 (保存/删除/…​)。

• Spring Integration Java DSL + Reactor Stream (Core)： Spring 集成的微批量信息通道。

• RxJavaReactiveStreams + RxJava + Reactor-Core： 融合富结构与高效异步 IO 处理者

• RxJavaReactiveStreams + RxJava + Reactor-Net (Core/Stream)： 用 RxJava 做数据输入，异步 IO 驱动做传输。

图片地址：http://projectreactor.io/docs/reference/images/overview.png

图2. Reactor 模块相互依赖关系快速概览

响应式数据流

响应式数据流(http://www.reactive-streams.org/) 作为一种新的数据流规范应用于 Java 9 及其后续版本，并被多个供应商和技术企业采纳，包括包括 Netflix，Oracle，Pivotal 或 Typesafe。

这一规范的定位非常清晰，旨在提供同/异步数据序列流式控制机制，并在 JVM 上首先推广。该规范由 4 个 Java 接口，1 个 TCK 和一些样例组成。在实现所需的 4 个接口之外，该规范的实质在于经由 TCK 验证的行为状态。能成功通过 TCK 实现类检测意味着满足 *Reactive Streams Ready* 状态。

图片地址：http://projectreactor.io/docs/reference/images/rs.png

图3. 响应式数据流约定

,

响应式数据流接口

\* org.reactivestreams.Pubslisher(https://github.com/reactive-streams/reactive-streams-jvm/blob/master/api/src/main/java/org/reactivestreams/Publisher.java) ：数据流发布者(信号从 0 到 N，N 可为无穷)。提供两个可选终端事件：错误和完成。

*org.reactivestreams.Subscriber(https://github.com/reactive-streams/reactive-streams-jvm/blob/master/api/src/main/java/org/reactivestreams/Subscriber.java)* *：数据流消费者(信号从 0 到 N，N 可为无穷)。消费者初始化过程中，会* 请求\*生产者当前需要订阅多少数据。其他情况，通过接口回调与数据生产方交互: 下一条(新消息)和状态。状态包括：完成/错误，可选。

\* org.reactivestreams.Subscription(https://github.com/reactive-streams/reactive-streams-jvm/blob/master/api/src/main/java/org/reactivestreams/Subscription.java) ：初始化阶段将一个小追踪器传递给订阅者。它控制着我们准备好来消费多少数据，以及我们想要什么时候停止消费(取消)。

\* org.reactivestreams.Processor(https://github.com/reactive-streams/reactive-streams-jvm/blob/master/api/src/main/java/org/reactivestreams/Processor.java) ：同时作为发布者和订阅者的组件的标记。

图片地址：http://projectreactor.io/docs/reference/images/signals.png

图4. 响应式数据流发布协议

订阅者有两种方式向发布者请求数据，如下所示：

*无界的* *：订阅者只需要调用*  Subscription#request(Long.MAX\_VALUE)\* 即可。

*有界的* *：订阅者保留数据引用，调用* request(long)\* 方法消费。

- 通常订阅者在订阅时会请求一个初始数据集或者一个数据

- 在 onNext 成功后(如 Commit，Flush 等…​ 之后)，请求更多数据

- 建议请求数量呈线性，尽量避免请求叠加， 如每下一个信号请求 10 个数据

表1. 迄今为止，Reactor 可直接使用的接口有：

|  |  |  |  |
| --- | --- | --- | --- |
| **响应式数据流** | **Reactor 模块** | **实现** | **描述** |
| Processor | reactor-core, reactor-stream | reactor.core.processor.\*, reactor.rx.\* | Reactor-core：提供背压就绪的RingBuffer\*Processor 等多种Processor reactor-streeam：提供全套的操作语句和广播机制 |
| Publisher | reactor-core, reactor-bus, reactor-stream, reactor-net | reactor.core.processor.\*, reactor.rx.stream.\*, reactor.rx.action.\*, reactor.io.net.\* | Reactor-core：Processors里植入Published Reactor-Bus： 提供无界的路由事件的发射发布机制 Reactor-stream：Publisher中直接引入数据流扩展 Reactor-net：提供2中发布方式：一种是Channel，另一种是flush和闭环回调 |
| Subscriber | reactor-core, reactor-bus, reactor-stream, reactor-net | reactor.core.processor.\*, reactor.bus.EventBus.\*, reactor.rx.action.\*, reactor.io.net.impl.\* | Reactor-core：Processors植入 Subscriber Reactor-bus：通过无界Publisher/Subcriber机制扩展总线能力 Reactor-stream：订阅行为通过计算指定回调实现 Reactor-net： IO 层实现处理写入，关闭和 flush |
| Subscription | reactor-stream, reactor-net | reactor.rx.subscription.\*, reactor.io.net.impl.\* | Reactor-Stream：提供了最优化过 PushSubscriptions 和 缓冲就绪的 ReactiveSubscription； Reactor-net：通过定制Subscription 异步 IO 用户端实现背压 |

我们从 Reactor 2 开始就严格遵守这个规范直到 **1.0.0** 发布，在 *Maven Central* 和其他镜像上都可使用。当然它也是 **reactor-core 的传递依赖**。

响应式扩展

响应式扩展，就是通常所说的 Rx(https://msdn.microsoft.com/en-gb/data/gg577609.aspx)，是一组定义良好的函数式 API，大规模扩展了观察者模式。

Rx 模式支持响应式数据序列处理，主要的设计要点有：

• 使用回调链分离时间/延迟：仅当数据可用时才会回调

• 分离线程模型：用 *Observable / Stream* 来处理同步或异步

• 控制错误链/终止：数据载荷信号以及错误与完成信号都传递给回调链

• 解决各种预定义API中多重分散-聚合和构造问题

JVM 中响应式扩展的标准实现是 RxJava(https://github.com/ReactiveX/RxJava)。它提供了强大的函数式 API，并将原始微软库中几乎全部的概念移植了过来。

Reactor 2 提供了一个实现了响应式扩展子集的特定模块(http://projectreactor.io/docs/reference/#streams)，并在少量场合下调整了名称来匹配我们的特定行为。这种关注以数据为中心的问题(微批处理，构造…​)的方法依赖于 Reactor 函数式单元，指派者和响应式数据流约定. 我们提倡需要各种 响应式扩展的用户尝试使用 RxJava(https://github.com/ReactiveX/RxJavaReactiveStreams) 并和我们沟通。最终，用户能受益于 Reactor 以及与 RxJava 生态系统组合所提供的强大的异步和 IO 能力。

&iexcl; 现阶段一些操作，行为和响应式数据流的概念仍然是 Reactor 所特有的。我们将会在合适的章节(http://projectreactor.io/docs/reference/#streams)充实这些特性。

! 因为背压和自动冲洗(flush)选项和原因， 异步 IO(http://projectreactor.io/docs/reference/#net-overview) 能力仍依赖于 *数据流能力* 。

表2. Rx 与 Reactor 数据流之间的错位

|  |  |  |
| --- | --- | --- |
| **rx** | **reactor-数据流** | **注解** |
| 观测 | reactor.rx.Stream | 反映了响应式数据流发布者的实现 |
| 操作者 | reactor.rx.action.Action | 反映了响应式数据流处理者的实现 |
| 带有不多于1个数据的观测 | reactor.rx.Promise | 给出唯一结果，反映了响应式数据流处理者的实现，并提供可选的异步指派 |
| 工厂 API (just，from，merge…​.) | reactor.rx.Streams | 与核心中以数据为中心的子集对齐，返回数据流 |
| 函数式 API (map，filter，take…​.) | reactor.rx.Stream | 与核心中以数据为中心的子集对齐，返回数据流 |
| 调度者 | reactor.core.Dispatcher, org.reactivestreams.Processor | Reactor 数据流使用无界共享式调度者或有界处理者来计算操作 |
| Observable.observeOn() | Stream.dispatchOn() | 仅对调度者参数调整下名称 |

reactor-核心

reactor-核心

永远别独自展开异步工作。

— Jon Brisbin

*在写 Reactor 1 之后*

永远别独自展开异步工作。

— Stephane Maldini

*在写 Reactor 2 之后*

先来看看，某项目是如何使用 Groovy 的：

// 初始化上下文，获取默认调度者  
Environment.initialize()  
  
// RingBufferDispatcher，默认带 8192 槽容量  
def dispatcher = Environment.sharedDispatcher()  
  
// 创建回调  
Consumer<Integer> c = { data ->  
 println "some data arrived: $data"  
 }  
  
// 创建 error 回调  
  
Consumer<Throwable errorHandler = { it.printStackTrace }  
  
// 异步分发数据  
dispatcher.dispatch(1234, c, errorHandler)  
  
Environment.terminate()

然后，再看看响应式数据流例子

// 独立异步处理者  
def processor = RingBufferProcessor.<Integer>create()  
  
// 发送数据，确保数据的安全性，直到订阅成功  
processor.onNext(1234)  
processor.onNext(5678)  
  
// 消费整型数据  
processor.subscribe(new Subscriber<Integer>(){  
  
 void onSubscribe(Subscription s){  
 //unbounded subscriber  
 s.request Long.MAX  
 }  
  
 void onNext(Integer data){  
 println data  
 }  
  
 void onError(Throwable err){  
 err.printStackTrace()  
 }  
  
 void onComplete(){  
 println 'done!'  
 }  
}  
  
// 完全关闭内部线程和调用  
processor.onComplete()

核心概述

图片地址：http://projectreactor.io/docs/reference/images/core-overview.png

图5. Doge 如何使用 Reactor-核心

**Reactor 核心**含有如下特性：

 *通用 IO & 函数式类型*\*，一些 Java 8 接口的反向移植

- 函数，提供者，消费者，谓词，双向消费者，双向函数

- 元组

- 资源池、暂停器、定时器

- 缓冲器，编解码和少量预定义的编解码器

*环境* \*上下文

*调度者* \*约定和几个预定义调度者

*预定义* *响应式数据流处理者* \*

Reactor-核心自身可替代其它消息传递机制，完成时序任务调度，或者帮你将代码组织为函数块，实现 Java 8 的反向移植接口。这种拆分便于同其他的响应式库配合使用，而没耐心的开发者也不用再去费劲弄懂环形缓冲区了。

&iexcl; Reactor-核心隐含覆盖 LMAX Disruptor，所以它不和已有的 Disruptor 依赖共存或碰撞。

函数式功能

可重用函数块基本是你一开始使用 Reactor 就需要的核心功能。[1] 那么函数式编程酷在哪里呢? 其核心理念之一将可执行代码当作另一种数据来处理。[2]业务逻辑由原始调用者决定，这与闭包和匿名函数的理念不谋而合。函数式编程还避免了 IF/SWITCH 语句块的包袱，并清晰地分离了功能：每个代码块只负责一个独立功能，而不共享内容。

1. 除非你只想用核心处理功能，而这些功能在这一阶段是基本独立的。我们打算逐步将调度器与核心调整到一致。

2. 有人也许要说这观点过于简化了，不过我们这里先讲求实用 :)

规划函数块

每个函数组件有明确的功能：

• 消费者(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/Consumer.java)：使用回调函数 — 登记后就不用管了

• 双向消费者(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/BiConsumer.java)：带双参数的简单回调 (通常用于序列比较，如前后参数比较)

• 函数(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/Function.java)：转换逻辑 - 请求/回应

• 双向函数(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/Consumer.java)：带双参数的转换逻辑 (通常用于累加器，比较前后参数并返回一个新值)

• 供给者(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/Supplier.java)：工厂逻辑 - 轮询

• 谓词(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/java/reactor/fn/Predicate.java)：测试逻辑 - 过滤

&iexcl; 我们将发布者和订阅者接口也作为**函数块**处理，我们称之为*响应式函数块*。它们是基本的组件，在 Reactor 和 Beyond 中到处都有用到。通常可以直接调用数据流 API 来创建恰当的订阅者，你只需要向 API 传入 **reactor.fn** 参数。

好消息是：封装在函数功能中的可执行指令，可以像乐高积木一样重用。

Consumer<String> consumer = new Consumer<String>(){  
 @Override  
 void accept(String value){  
 System.out.println(value);  
 }  
};  
  
// 为了简约，现在用 Java 8 风格  
Function<Integer, String> transformation = integer -> ""+integer;  
  
Supplier<Integer> supplier = () -> 123;  
  
BiConsumer<Consumer<String>, String> biConsumer = (callback, value) -> {  
 for(int i = 0; i < 10; i++){  
 // 对要运行的最后逻辑运行做惰性求值  
 callback.accept(value);  
 }  
};  
  
// 注意生产者到双向消费者执行过程  
biConsumer.accept(  
 consumer,  
 transformation.apply(  
 supplier.get()  
 )  
);

乍一看，你可能会觉得这个革新并不特别，但是这种编程理念的变化，对后续我们构建分层可组合代码却尤其重要。调度者通过消费者处理类型化的数据和错误的回调。Reactor Stream 模块也基于该理念实现优雅编码。

&spades; 使用 Spring 这样的 IoC 容器的良好实践是利用 Java 配置(http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-java)特性返回无状态函数式 Beans。然后就可以从容地将代码块注入数据流管道，或者指派代码块的执行。

元组

或许你已经注意到：Reactor 提供的接口都是强类型、带有泛型支持和少量确定数目的参数。那如果形参个数大于1或者2呢，又该怎么办呢？此时，需要使用一个类：**元组**。元组像是单对象实例中的带类型 CSV 行，在函数式编程中，就是通过元组保证类型安全呢和可变参数。

让我们用双参数双向消费者代替单参数消费者实现上例的过程：

Consumer<Tuple2<Consumer<String>, String>> biConsumer = tuple -> {  
 for(int i = 0; i < 10; i++){   
 // 类型正确，开启编译器  
 tuple.getT1().accept(tuple.getT2());  
 }  
};  
  
biConsumer.accept(  
 Tuple.of(  
 consumer,  
 transformation.apply(supplier.get())  
 )  
);

&iexcl; 元组涉及到更多的资源分配，因此，通常键值对比较和键值信号量更倾向使用 Bi 类型接口。

环境与调度者

函数式功能块到位后，接下来开启异步之旅。第一步我们来看调度者的部分。

使用调度者前，先要确认我们能快速地创建它们。通常创建调度者开销较大，因为它们要预分配一个内存段以确保高度信号，实际上这是序言中所阐述的非常著名的运行时 VS 启动时平衡问题。 Reactor 引入了名为 **环境** 的特定共享上下文来管理各种调度者，来避免不恰当的创建。

环境

环境由 Reactor 用户(或可用的扩展库，如 *@Spring*)来创建和终止。它们自动读取位于 META\_INF/reactor/reactor-environment.properties(https://github.com/reactor/reactor/blob/master/reactor-core/src/main/resources/META-INF/reactor/reactor-environment.properties) 的配置文件。

&spades; 属性文件可在运行时通过 *META-INF/reactor* 类路径下所需的新属性配置进行调校。

运行时替换默认配置的操作可通过传递如下环境变量来实现：*reactor.profiles.active.*

java - jar reactor-app.jar -Dreactor.profiles.active=turbo

环境初始化与终止示例

Environment env = Environment.initialize();  
  
// 判断系统环境与初始化环境是否相同  
Assert.isTrue(Environment.get() == env);  
  
// 找到名为"共享"的调度者  
Dispatcher d = Environment.dispatcher("shared");  
  
// 计时器与环境计时器绑定  
Timer timer = Environment.timer();  
  
// 关闭可能运行非守护进程的寄存调度者和计时器  
Environment.terminate();  
// 备选方案：通过注册shutdownHook实现终止自动调用

&spades; 对一个给定的 JVM 应用，最佳实践是：保持单一的运行环境。大多数情况首选 *Environment.initializeIfEmpty()*。

调度者

Reactor 1 就提及过调度者，它用类似 Java Executor 的通用约定抽象出信息传递的方法。它实际上扩展了 Executor。

调度者约定提供了一个传递信号的强类型方式，相应的数据及错误消费者可以同(异)步执行。我们用这种方法解决了经典的 Executor 首要面临的问题：错误隔离。调用 Error 消费者比中断已分配资源过程效果更好。如果没有可调用的，调度者将尝试找一个现存的环境并使用其 *errorJournalConsumer*。

异步调度者提供的第二个特性是使用*尾递归*策略的可重入调度。尾递归的使用情形是，调度检测到 调度者classLoader 已被分配到运行中的线程，此时将当前消费者返回时将要执行的任务入队。

使用同步多线程调度者，比如这个 Groovy Spock 测试(https://github.com/reactor/reactor/blob/master/reactor-core/src/test/groovy/reactor/core/dispatch/DispatcherSpec.groovy)：

import reactor.core.dispatch.\*  
  
//...  
  
given:  
 def sameThread = new SynchronousDispatcher()  
 def diffThread = new ThreadPoolExecutorDispatcher(1, 128)  
 def currentThread = Thread.currentThread()  
 Thread taskThread = null  
  
 def consumer = { ev ->  
 taskThread = Thread.currentThread()  
 }  
  
 def errorConsumer = { error ->  
 error.printStackTrace()  
 }  
  
when: "a task is submitted"  
 sameThread.dispatch('test', consumer, errorConsumer)  
  
then: "the task thread should be the current thread"  
 currentThread == taskThread  
  
when: "a task is submitted to the thread pool dispatcher"  
 def latch = new CountDownLatch(1)  
 diffThread.dispatch('test', { ev -> consumer(ev); latch.countDown() }, errorConsumer)  
  
 latch.await(5, TimeUnit.SECONDS) // 等待任务执行  
  
then: "the task thread should be different when the current thread"  
 taskThread != currentThread

! 我们将在随 2.x 发布计划加入 Executor 等没有的特性：响应式数据流协议。它们是 Reactor 没有直接绑定到响应式数据流标准的剩余部分之一。然而，它们可以与 Reactor 数据流结合并快速绑定，正如我们在数据流章节(http://projectreactor.io/docs/reference/#streams)中将要探索的一样。本质而言，它意味着用户可以直接使用，直到最终或暂时遇到大部分调度者实现的容量界限。

表3. 调度者族介绍

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **调度者** | **源环境** | **描述** | **强项** | **弱项** |
| 环形缓冲区 | sharedDispatcher() | An LMAX Disruptor 基于环形缓冲区的指派者. | 较小的容许延迟峰值 最快的异步指派者，在商用硬件上 10-15M+ 指派/秒 支持预订 | 满容量时获取下一槽位会出现'打滑'循环 单线程，不能并发指派 |
| Mpsc | sharedDispatcher() 如果不安全选项可用 | 信息传递结构的另一种优化. | 容许延迟峰值 在商用硬件上 5-10M+ 指派/秒 支持预订 | 无边界，并可能会使用尽可能多的可用内存堆 单线程，不能并发指派 |
| 工作队列 | workDispatcher() | An LMAX Disruptor 基于环形缓冲区的指派者. | 有限时间内的容许延迟峰值 最快的多线程指派者，在商用硬件上 5-10M+ 指派/秒 | 满容量时获取下一槽位会出现'打滑'循环 并发指派 不支持预定 |
| 同步 | dispatcher("sync") or SynchronousDispatcher. INSTANCE | 在当前线程上运行. | 上游和消费者同位执行 对测试支持很有用 在当前线程上折返指派时支持预定 | 不支持尾递归 阻塞 |
| 尾递归 | tailRecurse() or TailRecurse Dispatcher. INSTANCE | 同步折返指派者在正在指派时将指派入队. | 上游和消费者同位执行 减少了执行栈，极大地扩展了函数调用链 | 无边界尾递归深度 阻塞 支持预定 (窃取线程) |
| 线程执行者 | newDispatcher(int, int, DispatcherType. THREAD\_POOL\_EXECUTOR) | 使用底层线程池执行者的信息传递 | 多线程 阻塞消费者，容许永久延迟 在商用硬件上 1-5M+ 指派/秒 | 对指定消费者的并发执行两次或以上 默认无边界 default 不支持预定 |
| 可追踪代理 | N/A | 使用追踪级别日志来修饰现在指派者. | 指派窃听 比单独的代理指派者运行更慢 | 日志开销 (运行时，磁盘) |

图片地址：http://projectreactor.io/docs/reference/images/rbd2.png

图6. 给定时刻 T 的 RingBufferDispatcher

调度供给者

你也许注意到了一些调度者是间线程的，特别是 **RingBufferDispatcher** 和 **MpscDispatcher**。更进一步，根据响应式数据流规范，订阅者/处理者的实现中不应允许并发通知。这特别影响到了 Reactor **数据流**。如果尝试带着调度者使用 **Stream.dispatchOn(Dispatcher)** 会引入并发信号，很明显会失败。

不过，可以用调度者池或**调度供给者**来绕过这个限制。作为**供给者**工厂，**Supplier.get()** 提供的间接取回调度者功能容许了一些有趣的缓冲池策略：循环，最少使用， …​

**环境**提供了静态助手来对当前活动的调度者**环境**池进行创建，并最终反寄存：循环所返回的调度者组。一旦就绪，供给者将提供可控数量的调度者。

**环境**对调度者通常的一站式管理：

Environment.initialize();  
//....  
  
// 创建有2个调度者的匿名池，采用自动的默认设定(与默认调度者相同的类型，默认的积压大小...)  
DispatcherSupplier supplier = Environment.newCachedDispatchers(2);  
  
Dispatcher d1 = supplier.get();  
Dispatcher d2 = supplier.get();  
Dispatcher d3 = supplier.get();  
Dispatcher d4 = supplier.get();  
  
Assert.isTrue( d1 == d3 && d2 == d4);  
supplier.shutdown();  
  
// 创建并注册带3个调度者的新池  
DispatcherSupplier supplier1 = Environment.newCachedDispatchers(3，"myPool");  
DispatcherSupplier supplier2 = Environment.cachedDispatchers("myPool");  
  
Assert.isTrue( supplier1 == supplier2 );  
supplier1.shutdown();

计时器

调度者以尽可能快的速度运算传入的任务，而计时器提供了定期的一次性调度 API。Reactor 核心默认提供了 **HashWheelTimer**，并自动绑定到任一新建的环境上。HashWheelTimer 可以完美处理内存中大量并发任务，是 Java **TaskScheduler** 的强大替代选择。

! 虽然它适用于窗口技术 (分钟级以下的迷你任务)，但因所有任务会随应用关闭而丢失，它并不用于弹性调度。

&spades; 计时器在下次发布时会得到更多关注，比如我们想对 Redis 加入持续/共享调度支持。请在此表达你的意见，或提供一些贡献!

在我们的 Groovy Spock 测试(https://github.com/reactor/reactor/blob/master/reactor-core/src/test/groovy/reactor/fn/timer/HashWheelTimerYieldingStrategy.groovy) 中创建的一个简单计时器：

import reactor.fn.timer.Timer  
  
//...  
  
given: "a new timer"  
 Environment.initializeIfEmpty()  
 Timer timer = Environment.timer()  
 def latch = new CountDownLatch(10)  
  
when: "a task is submitted"  
 timer.schedule(  
 { Long now -> latch.countDown() } as Consumer<Long>,  
 period,  
 TimeUnit.MILLISECONDS  
 )  
  
then: "the latch was counted down"  
 latch.await(1, TimeUnit.SECONDS)  
 timer.cancel()  
 Environment.terminate()

核心处理者

核心处理者的工作比调度者更加专一：计算**支持背压的**异步任务。

同时它直接实现了org.reactivestreams.Processor(https://github.com/reactive-streams/reactive-streams-jvm/blob/master/api/src/main/java/org/reactivestreams/Processor.java) 接口，可以良好地与其它响应式数据流提供方合作。比如同时作为订阅者*和*发布者的处理者。你可以将它插入到响应式数据流链中你想要的地方(源，处理过程，槽)。

! 规范并不明确推荐直接使用 **Processor.onNext(d)**。我们技术上支持那种做法，但背压并不会传播，除非最终出现阻塞。你可以明确地用匿名订阅来做传递，先是用 Processor.onSubscribe 传递给处理者，用所实现的请求方式来获取背压反馈。

! OnNext 必须要序列化，例如每次来自一个单线程(不允许并发的 onXXX 信息)。不过 Reactor 可以支持这种方法，如果使用常规的 **Processor.share()** 方式创建处理者的话，比如 **RingBufferProcessor.share()** 。在创建时就要决定用哪种方法，以便在实现中使用正确的协调逻辑，所以要谨慎选择：是要做一个标准发布序列(无并发)，还是要使用多线程。

Reactor 在处理特定的某某 **工作** 处理者功能时有一个特例：

\* 通常响应式数据流处理者会将相同的数据序列异步调度至所有在给定时刻 T 订阅的订阅者。这与发布/订阅模式类似。

\* 工作处理者以方便的方式来分发数据，尽可能利用所有的订阅者。这意味着在给定时刻 T 的订阅者将一直看到不同的数据。这与工作队模式类似。

我们计划随着 2.x 的发布计划增加我们的核心处理者集。

环形缓冲区处理者

基于环形缓冲区的响应式数据流处理者有一些很棒的特性：

• 高吞吐量

• 重现最新未消费数据

• 如果没有订阅者在监听，数据不会丢失（不同于Reactor-数据流的**广播者**）。

• 如果订阅者在处理过程中取消，信号可以安全重现，该功能用环形缓冲区工作处理者(http://projectreactor.io/docs/reference/#rbwp-note) 时也可以良好工作

• 智能背压，允许任意时刻动态分配大小，方便订阅者负责消费和请求更多数据

• 传播背压，作为处理者，可被订阅并传递信息

• 多线程入站/出站处理能力

实际上环形缓冲区处理者就是有类型的 **MicroMessageBroker**！

它仅有的缺陷在于运行时创建的成本较高，也不像**环形缓冲区指派者**那样容易共享。因而它适用于高吞吐量预定义数据管线。

环形缓冲区处理者

Reactor 中 [RingBufferProcessor](http://projectreactor.io/docs/api/index.html?reactor/core/processor/RingBufferProcessor.html

) 组件本质上是适应响应式数据流 API 的 Disruptor RingBuffer (https://github.com/LMAX-Exchange/disruptor)。它的目的在于提供与尽可能接近裸机的效率。它的适用场合是将任务以极低的开销，极高的吞吐量指派到其它线程上，并在你的工作流中管理背压.

我用环形缓冲区处理者来异步计算各种远程产出调用：AMQP，SSD 存储和内存中的储存，处理者完全覆盖了多变的延迟，我们的每秒百万信息的数据源从未阻塞过！

— Reactor 用户

*环形缓冲区处理者用例*

![Figure 7. RingBufferProcessor at a given time T, with 2 Subscribers, all consuming the same sequence, but delta consuming rate is allowed until the ring buffer is full. This will happen when blue cube is colliding with its next clockwise yellow cube.](http://projectreactor.io/docs/reference/images/RBP.png "

图7. 给定时刻 T 带2个订阅者的环形缓冲区处理者。订阅者消费相同的序列，但在环形缓冲区满之前可以允许 delta 消费率存在。这在蓝方块与顺时针方向的下一个黄方块发生碰撞时会发生。")

图7. 给定时刻 T 带2个订阅者的环形缓冲区处理者。订阅者消费相同的序列，但在环形缓冲区满之前可以允许 delta 消费率存在。这在蓝方块与顺时针方向的下一个黄方块发生碰撞时会发生。

你需要用静态**创建**帮助者的方法来创建 **环形缓冲区处理者**。

Processor<Integer, Integer> p = RingBufferProcessor.create("test", 32); //\*1  
Stream<Integer> s = Streams.wrap(p); //\*2  
  
s.consume(i -> System.out.println(Thread.currentThread() + " data=" + i)); //\*3  
s.consume(i -> System.out.println(Thread.currentThread() + " data=" + i)); //\*4  
s.consume(i -> System.out.println(Thread.currentThread() + " data=" + i)); //\*5  
  
input.subscribe(p); //\*6

1. 创建带有 32 个槽容量的内部环形缓冲区的**处理者**。

2. 从响应式数据流**处理者**创建 **Reactor 数据流**。

3. 每个对**消费**的调用在其自身**线程**上创建一个 **Disruptor**。

4. 每个对**消费**的调用在其自身**线程**上创建一个 **Disruptor**。

5. 每个对**消费**的调用在其自身**线程**上创建一个 **Disruptor**。

6. 将这个**处理者**订阅至一个响应式数据流**发布者**。

每个传递给处理者的 **Subscribe.onNext(Buffer)** 方法的数据元素都会被"广播"给所有消费者。在**处理者**中没有循环指派，因为循环指派位于**环形缓冲区工作处理者**中，在下面会讨论这点。如果向**处理者**传递整数1，2，3，你在控制台看到的输入会像下面这样：

Thread[test-2,5,main] data=1  
Thread[test-1,5,main] data=1  
Thread[test-3,5,main] data=1  
Thread[test-1,5,main] data=2  
Thread[test-2,5,main] data=2  
Thread[test-1,5,main] data=3  
Thread[test-3,5,main] data=2  
Thread[test-2,5,main] data=3  
Thread[test-3,5,main] data=3

每个线程收到传递给**处理者**的所有值，因为内部使用**环形缓冲区**来管理发布值的可用槽，所以每个线程以有序的方式接收值。

! RingBufferProcessor 可以向任何潜在的订阅者重现因无订阅者而丢失的信号。如果满缓冲区未被订阅者耗尽，这会使处理者等待 onNext()。从 subsUp 接收的上一序列，到环形缓冲区的配置大小，都保持就绪以便为每个新的订阅者重现信号，即便事件已发出(分列)也是这样。

环形缓冲区工作处理者(RingBufferWorkProcessor)

与标准的环形缓冲区处理者向所有消费者广播数值的做法不同，环形缓冲区工作处理者根据消费者数量将传入的数值分区。进入处理者的数值以循环的方式被发送到各种线程(因为每个消费者都有自己的线程)，同时这些数值通过适当地向生产者提供背压来使用内部环形缓冲区有效地管理数值的发布。

我们实现了环形缓冲区工作处理者来对各种 HTTP 微服务调用扩大规模和平衡负载。我说的也许不对，不过看起来它比光还快(！)，而且垃圾回收压力完全得到控制。

— 开心的 Reactor 用户

*环形缓冲区工作处理者用例*

图片地址：http://projectreactor.io/docs/reference/images/RBWP.png，但在环形缓冲区满之前可以允许 delta 消费率存在。这在蓝方块与顺时针方向的下一个黄方块发生碰撞时会发生。")

图8. 在给定时刻 T 的 RingBufferWorkProcessor，带2个订阅者，消费各自独有的序列(可用 FIFO)，但在环形缓冲区满之前可以允许 delta 消费率存在。这在蓝方块与顺时针方向的下一个黄方块发生碰撞时会发生。

要使用**环形缓冲区工作处理者**，上述例子中你唯一需要修改的是静态**创建**方法的引用，你需要在**环形缓冲区工作处理者**类自身上使用。其余代码都一致。

Processor<Integer, Integer> p = RingBufferWorkProcessor.create("test", 32); //\*1

1. 创建带有32槽容量的内部环形缓冲区**处理者**。

现在当值被发布到**处理者**时，并不会广播给每个消费者，而是根据消费者数量分区。现在当我们运行这个例子时，可以看到像下面这样的输出：

Thread[test-2,5,main] data=3  
Thread[test-3,5,main] data=2  
Thread[test-1,5,main] data=1

! RingBufferWorkProcessor 能够重现被中断的信号，从终止的订阅者那里检测 **CancelException**。当信号最终被其它订阅者实际播放时，这是唯一会出现的情况。**我们可以保证任何事件都至少被交付一次。** 如果你熟悉语义的话，你可能会说 "嗯~，这个环形缓冲区工作处理者用起来像 Message Broker?"，的确是这样。

编解码器与缓冲区

字节操作是许多数据管线配置中用到的核心概念。从 reactor-net(http://projectreactor.io/docs/reference/#net-overview) 到通过 IO 接收发送编组和解组字节中都得到广泛运用。

**reactor.io.buffer.Buffer** 是 Java **ByteBuffer** 操作的修饰符，提供了一系列操作，目的是通过调整字节缓冲区大小以及读取或覆盖预分配字节来最小化字节拷贝。在字节缓冲区中追踪定位可以让开发者快速进入脑痛期，至少对我们是这样。因而我们决定向我们的用户推荐这个小工具。

**Groovy Spock 测试(https://github.com/reactor/reactor/blob/master/reactor-core/src/test/groovy/reactor/io/buffer/BufferSpec.groovy)**中简单的缓冲区操作代码如下所示：

import reactor.io.buffer.Buffer  
  
//...  
  
given: "an empty Buffer and a full Buffer"  
 def buff = new Buffer()  
 def fullBuff = Buffer.wrap("Hello World!")  
  
when: "a Buffer is appended"  
 buff.append(fullBuff)  
  
then: "the Buffer was added"  
 buff.position() == 12  
 buff.flip().asString() == "Hello World!"

一个有效的缓存应用是 **Buffer.View** ，它可由像 **split()** 这样的多路操作返回。它以免拷贝的方式来描述和内观字节缓冲区中的字节。Buffer.View 也是一种缓冲区，开放了相同的操作。

使用定界符和 **Buffer.View** 重用相同的字节来分块读取：

byte delimiter = (byte) ';';  
byte innerDelimiter = (byte) ',';  
  
Buffer buffer = Buffer.wrap("a;b-1,b-2;c;d;");  
  
List<Buffer.View> views = buffer.split(delimiter);  
  
int viewCount = views.size();  
Assert.isTrue(viewCount == 4);  
  
for (Buffer.View view : views) {  
 System.out.println(view.get().asString()); //prints "a" then "b-1,b-2", then "c" and finally "d"  
  
 if(view.indexOf(innerDelimiter) != -1){  
 for(Buffer.View innerView : view.get().split(innerDelimiter)){  
 System.out.println(innerView.get().asString()); //prints "b-1" and "b-2"  
 }  
 }  
}

对通常的编组/解组用例而言调整**缓冲区**有点低层次的感觉。Reactor 带有一系列的预定义转换器，叫做 **Codec**。一些 **Codec** 需要类路径中的适当额外依赖，如 JSON 操作需要 Jackson(http://jackson.codehaus.org/)。

**Codec** 有两种工作方式，第一种方式中它实现了将任意内容直接编码并返回已编码数据的**函数**，通常以缓冲的形式完成。这种很棒的方式**只适用于无状态编码**。另一种是使用 **Codec.encoder()** 返回的编码函数。

Codec.encoder() vs Codec.apply(Source)

*Codec.encoder()* \* 返回唯一编码函数，在不同线程间不能共享。

*Codec.apply()* \* 直接编码(保存分配的编码器)，在此情况下 Codec 需要在线程间共享。

&spades; Reactor Net(http://projectreactor.io/docs/reference/#net-overview) 实际上为每个新的连接都调用了 **Codec.encoder** 来处理这种区别。

**Codec** 也能从源类型解码数据，在大多数 Codec 实现上通常解码至缓冲区。要解码源数据，我们要从 **Codec.decoder()** 检索解码函数。与编码过程不同，解码没有像编码那样已经为编码目的而重载的便捷方法。与编码过程相同的是，解码函数在不同线程间不应共享。

**Codec.decoder()** 函数有两种形式，一种是直接返回解码数据，另一种 **Codec.decoder(Consumer)** 为每个解码事件调用已传递消费者。

Codec.decoder() vs Codec.decoder(Consumer)

• **Codec.decoder()** 是阻塞式解码函数，直接在传入源码数据中返回解码数据。

• **Codec.decoder(Consumer)** 可用于非阻塞解码，它返回 null，仅在解码后调用已传递消费者，可与任意异步方式结合。

Groovy Spock 测试(https://github.com/reactor/reactor/blob/master/reactor-core/src/test/groovy/reactor/io/codec/json/JsonCodecSpec.groovy) 中使用预定义编/解码器示例代码：

import reactor.io.json.JsonCodec  
  
//...  
  
given: 'A JSON codec'  
 def codec = new JsonCodec<Map<String, Object>, Object>(Map);  
 def latch = new CountDownLatch(1)  
  
when: 'The decoder is passed some JSON'  
 Map<String, Object> decoded;  
 def callbackDecoder = codec.decoder{  
 decoded = it  
 latch.countDown()  
 }  
 def blockingDecoder = codec.decoder()  
  
 // 这个异步策略太简单，不过先不用管它 :)  
 Thread.start{  
 callbackDecoder.apply(Buffer.wrap("{\"a\": \"alpha\"}"))  
 }  
  
 def decodedMap = blockingDecoder.apply(Buffer.wrap("{\"a\": \"beta\"}")  
  
then: 'The decoded maps have the expected entries'  
 latch.await()  
 decoded.size() == 1  
 decoded['a'] == 'alpha'  
 decodedMap['a'] == 'beta'

表4. 可用的核心编解码器：

|  |  |  |
| --- | --- | --- |
| **名称** | **描述** | **需求依赖** |
| ByteArrayCodec | (从缓冲区)编码/解码字节数组(至缓冲区) | N/A |
| DelimitedCodec | 分解/聚合缓冲区，委派给已传递编解码器作单元编组 | N/A |
| FrameCodec | 根据连续前缀长度分解/聚合缓冲至帧缓冲区 | N/A |
| JavaSerializationCodec | 用 Java Serialization 解序列/序列化缓冲区 | N/A |
| PassThroughCodec | 保持缓冲区不变 | N/A |
| StringCodec | 从缓冲区转换字符串/转换字符串至缓冲区 | N/A |
| LengthFieldCodec | 找寻长度， (从缓冲区)解码/编码适量的字节(至缓冲区) | N/A |
| KryoCodec | 用带缓冲的 Kryo 将缓冲转换为 Java 对象 | com.esotericsoftware.kryo:kryo |
| JsonCodec,JacksonJsonCodec | 用带缓冲的 Jackson 将缓冲转换为 Java 对象 | com.fasterxml.jackson.core:jackson-databind |
| SnappyCodec | 压缩编解码器，在对缓冲解包后/打包前申请代理编解码器 | org.xerial.snappy:snappy-java |
| GZipCodec | 压缩编解码器，在对缓冲解包后/打包前申请代理编解码器 | N/A |

reactor-数据流

reactor-stream

注意，你应该再也别去使用 Future.get() 了。

— Stephane Maldini

与一个银行业的客户

首先来看看一个 Java 8 示例中流 (Stream) 的运作方式

import static reactor.Environment.\*;  
import reactor.rx.Streams;  
import reactor.rx.BiStreams;  
  
//...  
  
Environment.initialize()  
  
//找到一个 String 列表中开头的 10 个词  
Streams.from(aListOfString)  
 .dispatchOn(sharedDispatcher())  
 .flatMap(sentence ->  
 Streams  
 .from(sentence.split(" "))  
 .dispatchOn(cachedDispatcher())  
 .filter(word -> !word.trim().isEmpty())  
 .observe(word -> doSomething(word))  
 )  
 .map(word -> Tuple.of(word, 1))  
 .window(1, TimeUnit.SECONDS)  
 .flatMap(words ->  
 BiStreams.reduceByKey(words, (prev, next) -> prev + next)  
 .sort((wordWithCountA, wordWithCountB) -> -wordWithCountA.t2.compareTo(wordWithCountB.t2))  
 .take(10)  
 .finallyDo(event -> LOG.info("---- window complete! ----"))  
 )  
 .consume(  
 wordWithCount -> LOG.info(wordWithCount.t1 + ": " + wordWithCount.t2),  
 error -> LOG.error("", error)  
 );

使用 Stream 和 Promise（约定） 协调任务

![图 9. 程序员狗如何使用 Reactor-Stream][1]

图 9. 程序员狗如何使用 Reactor-Stream

**Reactor Streams** 拥有下面这些功能部件:

+ **Stream** 及其直接实现。

+ 包含 **reactive extensions** 和其它一些 API 组件。

+ 带有一套特殊 [A+ 风格][2] API 的 **Promise**。

+ 可以使用 **Promise.stream()** 再转换回到 Stream。

+ 静态工厂，可以一站式的创建出相关的组件。

+ **Streams** 用来从定义良好的数据源(Iterable, 什么都没有的对象, Future, Publisher…)创建出 **Stream** 。

+ **BiStreams** 用于处理键值对型 Stream Stream<Tuple2> (reduceByKey…)。

+ **IOStreams** 用于对**Streams** 进行 [持久化（Persisting）][3] 和 [解码（Decoding）][4]。

+ **Promises** 用于单一数据的 **Promise**。

+ 由**Stream**提供的**Action** 及其每一个操作的直接实现都遵循响应式数据流处理器（Reactive Streams Processor）的规范。

+ 并不直接创建，而是使用 Stream 的 API (**Stream.map()** → MapAction, **Stream.filter()** → FilterAction…)。

+ **Broadcaster**, 一个为了实现动态数据分配而暴露 onXXXX 接口的明确的 Action。

+ 不像 [Core Processors][5], 如果没有订阅者访问，它们一般不会去接触缓存数据。

+ 不过 **BehaviorBroadcaster** 会向新的订阅者 （Subscriber） 回放最近一次的信号。

[1]: http://projectreactor.io/docs/reference/images/streams-overview.png

[2]: https://promisesaplus.com/

[3]: http://projectreactor.io/docs/reference/#streams-persistent

[4]: http://projectreactor.io/docs/reference/#core-codecs

[5]: http://projectreactor.io/docs/reference/#core-processor

不要把 **reactor.rx.Stream** 同 JDK 8 中新的 **java.util.stream.Stream** 混淆了. 后者不提供一个基于 API 的 Reactive Streams ，也没有 Reactive Extensions。不过，当用于原生类型和集合时，JDK 8 的 **Stream** API 还是相当完备的。 事实上对于能使用 JDK 8 的应用程序而言将 JDK 的支持和 Reactive Streams 混在一起用相当有趣。

Streams 的基础知识

Reactor 提供了基于 Reactive Streams 标准的 **Stream** 或者 **Promise** 来组成静态类型的数据管道。

它是一个非常实用且灵活的组件。它在被用于像 RxJava 的 **Observable** 那样仅仅只对异步操作进行组合时是足够灵活的。而它也足够强大，可以像一个异步工作队列那样运行，取出或者加入任意的组件，或者来自于其它标准实现之一的 Reactive Streams 组件。[&#91;3&#93;][1]。

粗略来分有两类基础的流

+ 一个是热 **Stream** ，它是没有界限的，像一个接收器那样接受输入数据。

+ 可以回想一下诸如鼠标点击的UI事件，或者诸如传感器的实时反馈，交易位置或者推特（Twitter）。

+ 适用于使用 Reactive Streams 协议的适配了的背压策略。

+ 还有一个冷 **Stream** 则是有界限的，其创建一般是来自于固定的数据集合，像 **List** 或者或者 **Iterable** 对象。

+ 可以回想一下诸如 IO 读取，数据库查询这样的 **游标型读取操作（Cursored Read）** ，

+ 适用于自动的 Reactive Streams 背压策略。

[1]: http://projectreactor.io/docs/reference/#*footnote*3

如我们之前(http://projectreactor.io/docs/reference/#core-dispatchers)所看到的, Reactor 使用一个 **Environment** 实体来持有 **Dispatcher** 实体，用于在一个给定的 JVM（和类装载器）中进行共享使用。一个 **Environment** 实体可以在一个应用程序中被创建出来并在各处进行传递，以避免类装载各自隔离的问题，或者其静态的辅助方法也可以被使用到。这个站点上的所有示例中，我们都将使用到静态其辅助方法，也鼓励你这样做。 为此，你需要在应用程序的某个地方初始化静态的 **Environment**。

`java

static {

Environment.initialize();

}

`

创建 Stream 和 Promise

创建 Stream 和 Promise

如果你是数据源的拥有者，并且想要使其 Reactive 化，带有能直接访问各种 *Reactive Extension* 和 *Reactive Stream* 的能力，这里就是起点。

有时的情况是要使用 **Stream** 的 API 来扩展已经存在的 **Reactive Stream Publisher** ，而幸运的是我们也提供了一站式的静态 API 来处理这种情况。

就像我们利用 **IterableStream**，**SingleValueStream** 等等所做的那样创建使用 Reactor API 注入的 **Publisher** 源来扩展现有的 Reactor **Stream** 也是一种受到鼓励的选择

Stream 和 Promise 的花费相对也不大，我们的基准测试套件在商用硬件上成功创造了超过 150M/s 的记录。大多数的 Stream 都坚持使用 **无分享模式（Share-Nothing ）**，只在需要时才创建新的不可变对象。

每一个操作都将返回新的实体：

`java

Stream<A> stream = Streams.just(a);

Stream<B> transformedStream = stream.map(transformationToB);

Assert.isTrue(transformationStream != stream);

stream.subscribe(subscriber1); //subscriber1 将会看到数据 A 保持不变

transformedStream.subscribe(subscriber2); //subscriber2 将会看到转换自 A 之后的数据 B。

//注意这两个订阅者将实现独立的流管道，这是一个我们称之为提升的过程

`

从冷数据源创建

你可以从各种来源创建 Stream，包括一个已知值的 Iterable 对象，一个用来作为基础任务流的单一值，或者甚至是来自于诸如 Future 或者 Supplier 这样的块结构。

**Streams.just()**

Stream<String> st = Streams.just("Hello ", "World", "!"); (1)  
  
st.dispatchOn(Environment.cachedDispatcher()) (2)  
 .map(String::toUpperCase) (3)  
 .consume(s -> System.out.printf("%s greeting = %s%n", Thread.currentThread(), s)); (4)

1. 从一个已知值创建 Stream，但并不指定一个默认的 调度器（Dispatcher）。

2. .dispatchOn(Dispatcher) 告诉 Stream 在哪一个线程上执行任务。用这来将任务的执行从一个线程转移到另外一个线程。

3. 使用常见的约定 —— map() 方法 —— 来对输入进行转化。

4. 在管道上产生需求，这意味着“处理开始了”。这是一个对 subscribe(Subscriber) 进行了优化后的快捷方式，默认只请求 Long.MAX\_VALUE 次。

冷数据源从一开始就会因为每一个传入 Stream.subscribe(Subscriber) 的新的 Subscriber 而被重新发放而因此就可能发生重复的消耗。

|  |  |  |
| --- | --- | --- |
| **工厂方法** | **数据类型** | **作用** |
| Streams.<T>empty() | T | 只在被订阅者请求时，发出一次 `onComplete()`。 |
| Streams.<T>never() | T | 从不发出任何东西。 对于保持活动状态的行为很有用。 |
| Streams.<T, Throwable>fail(\*\*Throwable\*\*) | T | 只发出 `onError(Throwable)`。 |
| Streams.from(\*\*Future<T>\*\*) | T | 在传入的可能会发出 `onNext(T)` 和 `onComplete()` ，或者异常时发出 `onError(Throwable)` 的 `Future.get()` 上阻止 `Subscription.request(long)`。 |
| Streams.from(\*\*T[]\*\*) | T | 每次 `Subscription.request(N)` 被调用到时发出 N 个 onNext(T) 元素。 如果 N == Long.MAX\_VALUE, 就发出所有数据。 一旦整个数组都已经被读取了一遍，就发出 \*\*onComplete()\*\*。 |
| Streams.from(\*\*Iterable<T>\*\*) | T | 每次 `Subscription.request(N)` 被调用到时发出 N 个 onNext(T) 元素。 如果 N == Long.MAX\_VALUE, 就发出所有数据。一旦整个数组都已经被读取了一遍，就发出 `onComplete()`。 |
| Streams.range(\*\*long\*\*, \*long\*) | Long | 每次 `Subscription.request(N)` 被调用到时就发出有 N 个 `onNext(Long)` 的一个序列。如果 N == Long.MAX\_VALUE, 就发出所有东西。 一旦读取达到所能包容的上限，就发出 `onComplete()`。 |
| Streams.just(T, \*T\*, \*T\*, \*T\*, \*T\*, \*T\*, \*T\*, \*T\*) | T | 在只是行为相似的 `Streams.from(Iterable)` 的一种优化。 用来发送没有和 Streams.from() 签名相冲突的 Iterable, Array 或者 Future 也很有用。 |
| Streams.generate(\*\*Supplier<T>\*\*) | T | 每当 `Subscription.request(N)` 被调用时就发送从 `Supplier.get()` 工厂产出的 `onNext(T)`。忽略掉要求的数量 N，因为只有一个数据会被发送。当返回一个 null 值时，就发送 `onComplete()`。 |
| Promises.syncTask(Supplier<T>), Promises.task(, Supplier<T>) | T | 当 `Subscription.request(N)` 第一次接收时发送一个从 `Supplier.get()` 产生的 `onNext(T)` 和 `onComplete()` 。忽略掉数量 N。 |
| Promises.success(\*\*T\*\*) | T | 无论一个订阅者何时被提供给 `Promise.subscribe(Subscriber)`，都发送 `onNext(T)` 和 `onComplete()`。 |
| Promises.<T>error(\*\*Throwable\*\*) | T | 无论一个被订阅了的订阅者何时被提供给 `Promise.subscribe(Subscriber)`，都发送 `onError(Throwable)`。 |

从现有的 Reactive Publisher 创建

现有的 Reactive Stream 的 Publishers 能很好的从包括用户的其它实现那里，或者从 Reactor 本身创建出来。

用例包括：

• 用来协调各种数据源的可以组合使用的 API(http://projectreactor.mydoc.io?v=10626&t=44496)

• 懒惰的资源访问，在订阅和请求上读取一个数据源，例如远程HTTP调用。

• 面向数据的操作，比如Key/Value的元组流（Tuple Stream），持续流（Persistent Stream）或解码。

• 使用 Stream API 的原生发布器装饰

Processor<String,String> processor = RingBufferProcessor.create();  
  
Stream<String> st1 = Streams.just("Hello "); (1)  
Stream<String> st2 = Streams.just("World "); (1)  
Stream<String> st3 = Streams.wrap(processor); (2)  
  
Streams.concat(st1, st2, st3) (3)  
 .reduce( (prev, next) -> prev + next ) (4)  
 .consume(s -> System.out.printf("%s greeting = %s%n", Thread.currentThread(), s)); (5)  
  
processor.onNext("!");  
processor.onComplete();

1. 从一个已知的值创建 Stream。

2. 使用 Stream API 装饰核心处理器。注意：Streams.concat() 会将处理器直接作为一个可用的 Publisher 参数来接收。

3. 连接 3 个上行流的来源 (所有的 st1，然后是所有的 st2 ， 再然后是所有的 st3 ） 。

4. 两个两个的对输入进行累积，并在最后从 st3 完成之后，在完成的上行流上发送结果。

5. 在管道上产生需求，意味着“现在就开始执行”。

|  |  |  |
| --- | --- | --- |
| **工厂方法** | **数据类型** |  |
| 作用 |  |  |
| Streams.create(Publisher<T>) | T | 当第一次 `Subscription.request(N)` 命中返回的 `Stream` 时只订阅传入的 `Publisher` 。因而它支持不会按每个规范所需要的那样调用 `Subscriber.onSubscribe(Subscription)` 的格式不正确的 `Publisher` 。 |
| Streams.wrap(Publisher<T>) | T | 一个针对传入 `Publisher.subscribe(Subscriber<T>) ` 参数的下发 `Stream`。只支持正确使用 Reactive Stream 协议的\*格式正确\*的 Publisher： onSubscribe > onNext\\* > (onError | onComplete) |
| Streams.defer(Supplier<Publisher<T>>) | T | 每次 `Stream.subscribe(Subscriber)` 被调用时，会使用由 `Supplier.get()` 提供的间接界别的惰性 Publisher 访问。 |
| Streams.createWith(BiConsumer<Long,SubscriberWithContext<T, C>, Function<Subscriber<T>,C>, Consumer<C>) | T | 一个针对每个 `Subscriber` 请求，启动和停止事件都有明确回调的 Stream 生成器。类似于去掉了通用模板的 `Streams.create(Publisher)`。 |
| Streams.switchOnNext(Publisher<Publisher<T>>) | T | 一个 Stream 会以 FIFO（先进先出）的顺序在从传入的 Publisher 发送 `onNext(Publisher<T>)` 间隙变更。 信号会导致在下行数据流 Subscriber<T> 中接收下一个 `onNext(T)` 的 Publisher 序列。当 `onNext(Publisher<T>)` 信号被接收到时它可能会打断正在上行的流的发布。 |
| Streams.concat(Publisher<T>, Publisher<T>\*) Streams.concat(Publisher<Publisher<T>>) | T | 如果一个 Publisher<T> 已经发送了，在处理下一个等待处理的 Publisher<T> 之前要等待这个 `onComplete()`。 其名称就暗示它对于[串联不同的数据源](http://rxmarbles.com/#concat)并保持顺序正确，这些方面的作用。 |
| Streams.merge(Publisher<T>, Publisher<T>, Publisher<T>\*) Streams.merge(Publisher<Publisher<T>>) | T | [接收多种数据源](http://rxmarbles.com/#merge)并将它们各自的序列\*\*交替插入\*\*。顺序不会像使用 `concat` 那样被保持。 对于 Subscriber 的需求将会在不同来源之间被分隔开，最小的数量是 1，以确保每个都能有发送一些东西的机会。 |
| Streams.combineLatest(Publisher<T1>, Publisher<T2>, Publisher<T3-N> x6, Function<Tuple2-N, C>) | C | 使用给定的累积 `Function` 来[整合最近发出的](http://rxmarbles.com/#combineLatest)来自于传入来源的元素。 |
| Streams.combineLatest(Publisher<T1>, Publisher<T2>, Publisher<T3-N> x6, Function<Tuple2-N, C>) | C | 每次有来源发送了信号，应用给定的 `Function`，并清理临时累积结果，[将最近的元素整合一次](http://rxmarbles.com/#zip)。 事实上这是一种针对多类型来源的灵活的联合机制。 |
| Streams.join(Publisher<T>, Publisher<T>, Publisher<T> x6) | List<T> | 一个压缩的快捷操作，只用来累加一个匹配传入的参数源的顺序的列表中完成了的累积结果。 |
| Streams.await(Publisher<>, long, unit, boolean) | void | 阻塞调用的线程，直到传入的 `Publisher`发出 `onComplete` 。可选的参数用来协调处理超时，以及需要有请求数据传入的情况。 如果最终状态时 `onError` 它就会抛出一个异常。 |
| IOStreams.<K,V>persistentMap(String, deleteOnExit) | V | [ChronicleStream 构造器的一个简单的快捷操作](${domain}?v=10626&t=44498) ，它是一个基于磁盘的附加/尾部（appender/tailer）日志记录器。命名参数必须匹配 /tmp/persistent-queue\[name\] 下面一个已经存在的队列。 |
| IOStreams.<K,V>persistentMapReader(String) | V | [ChronicleReaderStream 构造器的一个简单的快捷操作](${domain}?v=10626&t=44498) ，它是一个基于磁盘的尾部日志记录器（tailer）。命名参数必须匹配 /tmp/persistent-queue\[name\] 下的一个已经存在的队列。 |
|  |  |  |
| IOStreams.decode(Codec<SRC, IN, ?>, Publisher<SRC>) | IN | 使用 [Codec 解码器](${domain}?v=10626&t=44486) 可以将源数据类型解码成 \*\*IN\*\* 类型。 |
|  | Tuple2<KEY,VALUE> | 一种键值对操作，他会累加按次序两个两个的传入 `BiFunction` 参数的 `onNext(VALUE)` 所计算出来的结果。结果只会在 `onComplete()` 中释放。可选项让你可以使用现有的一个 map 来存储和侦听其事件。 |
|  | Tuple2<KEY,VALUE> | 一种键值对操作，它会累加按次序两个两个的传入 `BiFunction` 参数的 `onNext(VALUE)` 所计算出来的结果。结果在每次存储好之后就被释放了。可选项让你可以使用现有的一个 map 来存储和侦听其事件。 |
| Promises.when(Promise<T1>, Promise<T2>, Promise<T3-N> x6) | TupleN<T1,T2,\\*?> | 将所有来自于 `Promise` 的但个结果合到一起，并为新的 `Promise` 提供汇总的 `Tuple` 。 |
| Promises.any(Promise<T>, Promise<T>, Promise<T> x6) | T | 选取传入的约定（promise）中第一个信号可用的，然后对结果中返回的 `Promise` 发出 `onNext(T)` 。 |
| Promises.multiWhen(Promise<T>, Promise<T>, Promise<T> x6) | List<T> | 将所有来自于 `Promise` 参数的单个结果都合到一起，提供一个汇总的 `List`。同 `when` 方法的不同之处在于约定（promise） 的类型必须匹配。 |

从自定义 Reactive Publisher 处创建

随着时间的推移，Reactor的用户会愈加熟悉 **Reactive Streams**。 那就会是开始创建自定义响应式数据源的完美时分! 一般实现者会遵循规范的要求并使用 **reactive-streams-tck** 依赖来验证它的成果。对于约定的遵行需要一个 **Subscription** 还有在发送任何数据之前对于 **onSubscribe** 加一个 **request(long)** 的一次调用。

不过 Reactor 允许在只处理消息传递部分有一些灵活性，并将会明确的提供自动缓存的 **Subscription**，不同之处在下面的 代码示例中有所展示。

**Streams.create 和 Streams.defer 实战**

final Stream<String> stream1 = Streams.create(new Publisher<String>() {  
 @Override  
 public void subscribe(Subscriber<? super String> sub) {  
 sub.onSubscribe(new Subscription() { //1  
 @Override  
 public void request(long demand) {  
 if(demand == 2L){  
 sub.onNext("1");  
 sub.onNext("2");  
 sub.onComplete();  
 }  
 }  
  
 @Override  
 public void cancel() {  
 System.out.println("Cancelled!");  
 }  
 });  
 }  
});  
  
final Stream<String> stream2 = Streams.create(sub -> {  
 sub.onNext("3"); //2  
 sub.onNext("4");  
 sub.onComplete();  
});  
  
final AtomicInteger counterSubscriber = new AtomicInteger();  
  
Stream<String> deferred = Streams.defer(() -> {  
 if (counterSubscriber.incrementAndGet() == 1) { //3  
 return stream1;  
 }  
 else {  
 return stream2;  
 }  
});  
  
deferred  
 .consume(s -> System.out.printf("%s First subscription = %s%n", Thread.currentThread(), s));  
deferred  
 .consume(s -> System.out.printf("%s Second subscription = %s%n", Thread.currentThread(), s));

1. 从一个可用的首先会调用 onSubscribe(Subscription) 的自定义 Publisher 那里创建一个 Stream。

2. 从格式难看的跳过了 onSubscribe(Subscription) 并立即调用了 onNext(T) 的自定义 Publisher 那里创建一个 Stream。

3. 创建一个 DeferredStream，它将会在每一次 Stream.subscribe 调用上替换原来的 Publisher<T>，计算 Subscriber 的总数。

从这儿到哪儿去呢? 有大量的用例可以从自定义的 Publisher 那里获益：

+ 响应式外观可以将任何IO调用转换成一个配对的需求和组合： HTTP 调用（读取N次）， SQL 查询（选择最大的 N），文件读取（读取 N 行）……

+ 异步外观可以将任何热数据调用转换成一个组合式的 API： AMQP Consumer, Spring MessageChannel 端点……

Reactor 提供了一些可重用的组件来避免你不得不去做的样板检查，无需扩展现有的 Stream 或者 PushSubscription。

+ 扩展 PushSubscription 而不是直接实现 Subscription ，从而来获得终端状态 (PushSubscription.isComplete()) 的好处。

+ 使用 PublisherFactory.create(args) 或者 Streams.createWith(args) 来为每个生命周期的 step (requested，stopped，started) 使用函数式消耗者。

+ 扩展 Stream 而不是直接实现 Publisher 来获得组合式 API 的好处。

Streams.createWith, 一个去掉了一些样板的 create() 的替代

final Stream<String> stream = Streams.createWith(  
 (demand, sub) -> { //1  
 sub.context(); //2  
 if (demand >= 2L &amp;&amp; !sub.isCancelled()) {  
 sub.onNext("1");  
 sub.onNext("2");  
 sub.onComplete();  
 }  
 },  
 sub -> 0, //3  
 sub -> System.out.println("Cancelled!") //4  
);  
  
stream.consume(s -> System.out.printf("%s greeting = %s%n", Thread.currentThread(), s));

1. 在 Subscriber 的请求之上附上一个请求消耗者响应，并传入要求和请求的订阅者。

2. *sub* 参数实际上是一个可能带上由所有请求回调共享的 初始状态的 SubscriberWithContext。

3. 启动时候执行一次，这里也是我们初始化可选的上下文的地方；每次请求回调都会从 context() 上接收到一个 0。

4. 在任何终端事件上执行一次：cancel()，onComplete() 或者 onError(e)。

开始去为响应式数据流方式编写代码的一个好地方就是简单的去看看更加详细，背压可用的文件流(http://projectreactor.mydoc.io?v=10626&t=46166)。

从热数据源创建

如果你正在处理的是一种数组的无边界的流，就好像一个通过REST接口接收用户输入的WEB应用程序，我也许想要使用 Reactor 中我们称作 Broadcaster(http://projectreactor.io/docs/api/index.html?reactor/rx/stream/Broadcaster.html) 的“热”类型的 Stream 。

为了使用它，你得在 Broadcaster 上声明可组合的，函数式任务的管道并在稍后调用 Broadcaster.onNext(T) 来将值发布到管道上。

Broadcaster 是一个有效的 Processor 和 Consumer。要 onSubscribe 一个 Broadcaster 是可能的，因为要将其作为 Consumer 将 Consumer.accept(T) 分配给 Broadcaster.onNext(T) 是可能的。

Broadcaster.create()

Broadcaster<String> sink = Broadcaster.create(Environment.get());//1   
  
sink.map(String::toUpperCase) //2  
 .consume(s -> System.out.printf("%s greeting = %s%n", Thread.currentThread(), s)); //3  
  
sink.onNext("Hello World!"); //4

1. 使用默认的方式创建一个 Broadcaster， 将 RingBufferDispatcher 作为 Dispatcher 来进行共享。

2. 使用常规的约定 —— map () 方法 —— 对输入进行转换。

3. .consume() 是一个 "终端" 操作 ，这意味着它使用 Reactive 流的语法来产生需求。

4. 将值发布到管道中，这将导致要对任务的调用。

热数据源永远不会被重播。用户在它们已经被传入 Stream.subscribe(Subscriber) 的那一刻看到数据。一个例外就是 BehaviorBroadcaster（会重播上次发射的单元)；Streams.timer() 和 Streams.period() 也会保持独立的时控游标， 但还是会忽略掉背压。

用户会看到在他们于时间 T 进行了订阅之后，每过 t+1[[N]] 的时间就会有新的数据 N 流经 Broadcaster 。

|  |  |  |  |
| --- | --- | --- | --- |
| **工厂** | **输入** | **输出** | **作用** |
| Streams.timer(delay, unit, timer) | N/A | Long | 在 `Stream.subscribe(Subscriber)` 启动一个定时器调用和发送一个单独的 `onNext(0L)` 然后在延迟时间一过就 `onComplete()`。要确保如何当前没有活跃的 `Environment` 就传入可选的 `Timer` 参数。 `Subscription.request(long)` 会因为没有可以应用到一个计划的发布的背压策略而被忽略。 |
| Streams.period(period, unit, timer) | N/A | Long | 在 `Stream.subscribe(Subscriber)` 上启动一个定时器调用并在每个时间段发送 `onNext(N)` ， 这里的 N 是一个从 0 开始的增量计数器。要确保如果当前没有活跃的 `Environment` 就传入可选的 `Timer` 参数。 `Subscription.request(long)` 会因为没有可以应用到一个计划的发布的背压策略而被忽略。 |
| Streams.<T>switchOnNext() | Publisher<T> | T | 一个 `Action`，它对于记录而言也是一个 `Processor`。`onNext(Publisher<T>)` 信号会导致下行流 `Subscriber<T>` 接收下一个`onNext(T)` 的 `Publisher` 序列。它可能会在 `onNext(Publisher<T>)` 信号被接收到时打断当前上行流的发布。 |
| Broadcaster.<T>create(Environment, Dispatcher) | T | T | 在任何被允许调用 `onSubscribe`、 `onNext`、`onComplete` 或者 `onError` 的上下文，和一个Stream下的这些信号的可组合序列。 如果没有订阅者在进行活跃的注册，下一个信号可能会触发一个 `CancelException`。可选的 `Dispatcher` 和 `Environment` 参数定义了去哪里发送每个信号。最后，一个 Broadcaster 对于一个 `Publisher` 而言任何时候都可以被订阅，就像是一个 `Stream`。 |
| SerializedBroadcaster.create(Environment, Dispatcher) | T | T | 类似于 `Broadcaster.create()` 不过添加了对于来自于可能会调用相同的广播器 `onXXX` 方法的并行上下文的并发 `onNext` 的支持。 |
| BehaviorBroadcaster.create(Environment, Dispatcher) | T | T | 类似于 `Broadcaster.create()` 不过总是会重播最后一条数据信号 (如果有的话 ) 以及针对新的 `Subscriber` 的最后的终端信号 （`onComplete()`，`onError(Throwable)`)。 |
| BehaviorBroadcaster.first(T, Environment, Dispatcher) | T | T | 类似于 `BehaviorBroadcaster` 不过是以一个默认值 T 开始的。 |
|  |  |  |  |
| Streams.wrap(Processor<I, O>) | I | O | 一个针对传入的`Publisher.subscribe(Subscriber<O>)` 参数的分配 `Stream`。只能正确的支持使用了Reactive Streams协议的格式良好的 Publisher： onSubscribe > onNext\* > (onError ¦ onComplete) |
| Promises.<T>prepare(Environment, Dispatcher) Promises.ready() | T | T | 准备一个 `Promise`，用来应对任何外部的上下文通过 `onNext` \*\*仅有的一次\*\*调用。因为它是一个有状态的持有满足要求的约定的结果的容器，所有新的订阅者会立即在当前的线程上运行。 |

对于异步的广播，要总是考虑将 Core Processor(http://projectreactor.mydoc.io?v=10626&t=44484) 作为 Broadcaster 的替代：

> + Broadcaster 在没有订阅者时会触发一个 CancelException(http://projectreactor.io/docs/api/reactor/core/processor/CancelException.html) 。一个核心 RingBuffer\*Processor 将总是将缓存的数据传递到第一个订阅者。

> + 有些可以被赋值给 Broadcaster 的 Dispatcher 类型可能不支持并发的 onNext 。使用 RingBuffer\*Processor.share() 作为线程安全，并发 **onNext** 的替代。

> + RingBuffer\*Processor 支持通过一个下行流订阅者回放取消正在进行的处理这一事件，如果这个订阅者仍然在处理器线程上运行的话。Broadcaster 不会支持回放。

> + RingBuffer\*Processor 比它们的带有一个 RingBufferDispatcher 的 Broadcaster 替代要更快。

> + RingBufferWorkProcessor 支持扩大附上的订阅者的数量。

> + Broadcaster 在 2.1 中可能会被发展为一个 Processor , 这样就达成同样了的目的，并且没有让 Reactor 用户为 Processor 和 Broadcaster 之间的选择犯难。

绑定一个 Stream

Streams 操作 — 除了一些像终端动作和 broadcast() 的异常 — 将永远不会直接订阅，而是将懒惰地预备自己被订阅。这在函数式编程中常常被称为提升。

基本的意思就是 Reactor Stream 用户会明确的调用 Stream.subscribe(Subscriber) 或者可选的终端动作，比如 Stream.consume(Consumer) 来实现所有注册了的操作。在那之前 Actions 并不是真的存在的。我们使用 Stream.lift(Supplier) 来将这些 Action 的创建延迟到 Stream.subscribe(Subscriber) 被明确调用。

当所有的东西都绑定好了，每个动作都会维持一个上行流 Subscription 和一个下行流 Subscription 而 Reactive Streams 所有的约定都会应用到管道。

通常终端的动作会返回一个 Control 对象而不是 Stream。这是一个你可以用来请求或者取消一个管道的组件，不用再一个 Subscriber 上下文里面或者是实现整个 Subscriber 约定。

绑定两个管道

import static reactor.Environment.\*;  
import reactor.rx.Streams;  
import reactor.rx.Stream;  
//...  
  
Stream<String> stream = Streams.just("a","b","c","d","e","f","g","h");  
  
//prepare two unique pipelines  
Stream<String> actionChain1 = stream.map(String::toUpperCase).filter(w -> w.equals("C"));  
Stream<Long> actionChain2 = stream.dispatchOn(sharedDispatcher()).take(5).count();  
  
actionChain1.consume(System.out::println); //start chain1  
Control c = actionChain2.consume(System.out::println); //start chain2  
//...  
c.cancel(); //force this consumer to stop receiving data

图片地址：http://projectreactor.io/docs/reference/images/wiringup.png

要从一个统一管道向订阅者输出, 可以使用 Stream.process(Processor)，Stream.broadcast()，Stream.broadcastOn() 和 Stream.broadcastTo()。

共享一个上行流管道，并绑定两个下行流管道

import static reactor.Environment.\*;  
import reactor.rx.Streams;  
import reactor.rx.Stream;  
//...  
  
Stream<String> stream = Streams.just("a","b","c","d","e","f","g","h");  
  
//prepare a shared pipeline  
Stream<String> sharedStream = stream.observe(System.out::println).broadcast();  
  
//prepare two unique pipelines  
Stream<String> actionChain1 = sharedStream.map(String::toUpperCase).filter(w -> w.equals("C"));  
Stream<Long> actionChain2 = sharedStream.take(5).count();  
  
actionChain1.consume(System.out::println); //start chain1  
actionChain2.consume(System.out::println); //start chain2

图片地址：http://projectreactor.io/docs/reference/images/broadcast.png

|  |  |  |
| --- | --- | --- |
| **Stream&lt;T&gt; 方法** | **返回类型** | **作用** |
| subscribe(Subscriber<T>) subscribeOn | void | 订阅传入的 `Subscriber<T>` 并将任何附带的上行流实例化, 进行懒绑定(针对非终端操作的非明确的提升)。注意一个 `Subscriber` 必须请求数据，如果它期望一些数据的话。`dispatchOn` 和 `subscribeOn` 是提供给用传入的 `Dispatcher` 发送 `onSubscribe` 信号的可选项。 |
| consume(Consumer<T>,Consumer<T>,Consumer<T>) consumeOn | Control | 每当有相关的信号被侦测到，就会使用一个同每个传入的 `Consumer` 交互的 `ConsumerAction` 来调用 `subscribe`。它将会向接收的 Subscription 请求 `request(Streams.capacity())` 获取容量限制， 默认是 `Long.MAX\_VALUE`, 这会导致无节制的使用。`subscribeOn` 和 `consumeOn` 是提供给传入的 `Dispatcher` 发送 `onSubscribe` 信号的可选项。如有必要，就返回一个 `Control` 组件来取消 实例化的 `Stream`。 注意如果 `onNext(T)` 信号触发了一个阻塞请求， `ConsumeAction` 就会维持一个无限的递归。 |
| consumeLater() | Control | 类似于 `consume`，但并不会启动一个初始的 `Subscription.request(long)`。 返回的 `Control` 可以被用来在任何时候执行 `request(long)`。 |
| tap() | TapAndControls | 类似于 `consume` 但返回的是一个 `TapAndControls`，它将会在每次 `onNext(T)` 接收到信号或者被取消时动态的更新。 |
| batchConsume(Consumer<T>, Consumer<T>, Consumer<T>, Function<Long,Long>) batchConsumeOn | Control | 类似于 `consume` 但将会请求映射的 `Long` 需求，给定了之前的需求并且以默认的 `Stream.capacity()` 开始。 对于要动态适配各种因素的需求很有用。 |
|  | Control | 类似于 `batchConsume`，但是将会请求需要的 `Long` 值计算出来的序列。它可以被用来插入流程控制，比如延迟需求的 `Streams.timer()`。 |
| next() | Promise<T> | 返回一个 `Promise<T>` ，它会积极的向 `Stream` 发出订阅，对其进行实例化，并在解除注册之前请求一个信号数据。最近的 onNext(T), onComplete() or onError(Throwable)信号 将会满足约定。 |
| toList() | Promise<List<T>> | 类似于 `next() `，但是会等待直到整个序列已经产生 (`onComplete()`) 并且会在一个单独的 `List<T>` 中传递累积的 `onNext(T)` 以满足返回的约定。 |
| Stream.toBlockingQueue() | CompletableBlockingQueue<T> | 向 `Stream` 发出信号并返回一个累积所有 `onNext` 信号的阻塞 `Queue<T>`。 `CompletableBlockingQueue.isTerminated()` 可以被用来作为一个阻塞 `poll()` 循环退出的条件。 |
| cache() | Stream<T> | 将任何 Stream 转成\*\*冷\*\* Stream，能够针对每个 Subscriber 单独回放所有的信号序列。因为动作的无边界属性，所以你可能应该只将它用于小的 (ish) 序列上。 |
| broadcast() broadcastOn(Environment, Dispatcher) | Stream<T> | 将任何 Stream 转成\*\*热\*\* Stream。这将会阻止管道通过当前的 `Stream` 的实例化发生重复，并准备向 N 个 Subscriber 下行流发布该信号。 需求将会从所有的子 Subscriber 处聚集。 |
| broadcastTo(Subscriber<T>) | Subscriber<T> | 一个 `Stream.subscribe` 替代，因为返回的实体就是传入的参数，所以允许方法链。 |
| process(Processor<T, O>) | Stream<O> | 类似于 broadcast() 但接受任何给定的 `Processor<T, O>`。这里有一个对于 [Core Processor](${domain}?v=10626&t=44484) 的完美介绍！ |

设定容量

Reactive Streams 标准鼓励应用程序开发者在处理中的数据上设定合理的限制。这会防止组件变得太臃肿，拥有超过它们处理能力的数据量, 这样会导致整个应用程序发生不可预测的问题。Reactive Streams 的一个核心理念之一就是 "背压", 或者说是管道同一次只能处理固定数量数据项的上行流组件通信的能力。 一个用来描述使用队列来请求小批量大容量数据的有用的技术项目就是 "微批量"。

在一个 Reactor 的 Stream 中, 将数据项微批量化来限制要处理的数据的数量是可能的。这在许多方法中拥有不同的好处，并不仅仅是如果系统要崩溃了，它就能通过阻止系统接收更多它负担不起的数据，来限制数据丢失的发生。

为了要限制 Stream 中要处理的数据量，就使用 .capacity(long) 方法。

Streams.just()

Stream<String> st;  
  
st  
 .dispatchOn(sharedDispatcher())  
 .capacity(256) //1  
 .consume(s -> service.doWork(s)); //2

1. 限制要处理的数据数量到每次不超过 256 个元素。

2. 在请求接下来的 256 个数据元素时，产生上行流需求。

如果当前用 dispatchOn 设置的 Stream 调度器时一个 SynchronousDispatcher.INSTANCE (如果没有设置，这就是默认的) ，capacity 就不会影响到 consume。

我们留下这里作为 Reactor 使用者的练习，用来了解设置容量同使用 Stream.adaptiveConsume 或者一个定制的 Subscriber 计算动态需求的对比。

函数式组合

类似于许多其它的函数式的库，Reactor 提供了许多实用的方法，用来在一个 Stream 上对函数进行组合。 你可以被动地对值进行观察，将它们转换成另外一种类型, 过滤掉你不想要的值，一直达到一个大小或者时间，还有许多其它实用的操作。

这些操作被叫做 动作（Action）, 而他们将不会直接绑定 `Stream`(http://projectreactor.mydoc.io?v=10626&t=44505) 。它们在任何 Stream 实体中都是可用的，这意味着[在这一阶段你应该有这样一个东西] (http://projectreactor.mydoc.io?v=10626&t=44489) 。

+ Action 是以声明的顺序（左或者右）进行 onSubscribe() 的, 因此 stream.actionA().actionB() 将会首先执行 actionA 然后才是 actionB。

+ onSubscribe() 运行于父级 Publisher 线程上下文之上，因而其可以被 subscribeOn(Dispatcher) 变更。

+ Actions subscribe() 声明的是反向顺序（从右至左）。无论何时 subscribe 在管道的结尾处被调用到， subscribe() 都会向后增加。

+ subscribe() 同步的向后增长可能会影响到栈大小的使用。如果这成为了一个问题，就使用一个指派的 Processor 在一个 Environment.tailRecurse() 分配器上运行 subscribe() 。然后在调用链的任何一个点上对它进行 process() 处理。

如果你想要被动的观察传入管道的数据，那就使用 .observe(Consumer) 方法和其它的 reactor.rx.action.passive 动作。观察值就使用 .observe(Consumer<? super T>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#observe-reactor.fn.Consumer-)。观察不用在最后处理的错误，就使用 .observe(Class<? extends Throwable>, BiConsumer<Object,? extends Throwable>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#observeError-java.lang.Class-reactor.fn.BiConsumer-)。要观察 Reactive Streams 完成的信号，就使用 .observeComplete(Consumer<Void>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#observeComplete-reactor.fn.Consumer-)。为了观察取消的信号，就使用 .observeCancel(Consumer<Void>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#observeCancel-reactor.fn.Consumer-)。为了观察 Reactive Streams 订阅的信号，就使用 observeSubscribe(Consumer<? super Subscription<T>>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#observeSubscribe-reactor.fn.Consumer-)。

observe(Consumer<T>)

Stream<String> st;  
  
st.observe(s -> LOG.info("Got input [{}] on thread [{}}]", s, Thread.currentThread())) //1  
 .observeComplete(v -> LOG.info("Stream is complete")) //2  
 .observeError(Throwable.class, (o, t) -> LOG.error("{} caused an error: {}", o, t)) //3  
 .consume(s -> service.doWork(s)); //4

1. 不产生需求就可以被动的观察传入的值。

2. 所有的值已经处理之后就运行一次，而且 Stream 已经完成了标记。

3. 冒出一个错误时就运行.

4. 在管道上产生一个需求并且用到任何的值。

过滤传入一个 Stream 的值是可能的，因此下行流的动作值会看到你想要它们看到的数据。 过滤动作可以在 **reactor.rx.action.filter** 包下面看到。 最常用的一个就是 **.filter(Predicate<T>)** 方法。

不能匹配的数据会触发 **Subscription.request(1)**，如果数据流实际上是无界限的，没有预先要求的 Long.MAX\_VALUE。

filter(Predicate<T>)

Stream<String> st;  
  
st.filter(s -> s.startsWith("Hello")) //1  
 .consume(s -> service.doWork(s)); //2

1. 这将会值允许有以 'Hello' 开头的字符串传到下行流。

2. 在管道上产生需求并且用到任何的值。

过滤器的特殊应用程序就是给 Stream 设定限制。限制的动作可以在 reactor.rx.action.filter 包下面找到。有各种方式来告诉 Stream<T> 其在事件，大小和/或特殊条件上是有边界的。最常用的一个就是 .take(long) 方法。

Stream.take(long)

Streams  
 .range(1, 100)  
 .take(50) //1  
 .consume(  
 System.out::println,  
 Throwable::printStackTrace,  
 avoid -> System.out.println("--complete--")  
 );

1. 只取首先的 50 个元素，然后取消上行流，并且完成下行流。

转换

如果你想要主动地将管道中传递的数据进行转换，那就使用 .map(Function) 和其它的 reactor.x.action.transformation 动作。最常用的转换动作就是 .map(Function<? super I, ? extends O>))(http://projectreactor.io/docs/api/reactor/rx/Stream.html#map-reactor.fn.Function-)。一些其它的动作依赖于要转换的数据，特别是像 flatMap 或者 concatMap 这样的 组合操作(http://projectreactor.mydoc.io?v=10626&t=44496) 。

Stream.map(Function<T,V>)

Streams  
 .range(1, 100)  
 .map(number -> ""+number) //1  
 .consume(System.out::println);

1. 将每一个 Long 都转换成 String。

给定一个实际的输入数据，如果你想要执行一个明确的管道 Stream<V> 或者 Publisher<V>，你可以使用组合操作，比如 .flatMap(Function) 和其他 reactor.rx.action.combination 动作。

要将值转换为一个明确的，异步的 Publisher<V>，就使用 .flatMap(Function<? super I, ? extends Publisher<? extends O>)(http://projectreactor.io/docs/api/reactor/rx/Stream.html#map-reactor.fn.Function-)。返回的 Publisher<V> 之后将会**合并**到主要的流程中，发送信号 onNext(V)。它们完成以后会适时从混合的动作中移除。flatMap, concatMap 和 switchOnMap 之间的区别在于混合的策略。对应是 **Interleave**，**Fully Sequential** 和 **Partially Sequential** (会被 onNext(Publisher<T>)打断)。

下行流请求是分开的 (最小是 1，通过混合的 Publisher)

Stream.flatMap(Function)

Streams  
 .range(1, 100)  
 .flatMap(number -> Streams.range(1, number).subscribeOn(Environment.workDispatcher()) ) //1  
 .consume(  
 System.out::println, //2  
 Throwable::printStackTrace,  
 avoid -> System.out.println("--complete--")  
 );

1. 将进来的任何数据转换成范围从 1 到 N 的数字混合回去，并在给定的 Dispatcher 上执行。

阻塞可以被看做是 **Reactor** 中的一种反模式。也就是说，我们确实提供了一种相近的API（啊哈），用于集成遗留的操作，并支持测试。

Stream.toList()

Promise<List<Long>> result = Streams  
 .range(1, 100)  
 .subscribeOn(Environment.workDispatcher())  
 .toList(); //1  
  
System.out.println(result.await()); //2  
result.onSuccess(System.out::println); //3

1. 在分配器线程上使用 subscribeOn(Dispatcher) 操作中给定的整个序列。

2. 阻塞 (默认是 30 秒钟 ) 直到 onComplete() 并且只在 onNext(List<Long>) 中打印，或者，如果是 onError(e)，就封装成 RuntimeException 并重新抛出来.

3. 因为约定已经满足条件， System.out.println() 会立即在当前上下文上面执行。

|  |  |
| --- | --- |
| **函数式API或者工厂方法** | **作用** |
| Streams.await(Publisher<?>>) | 阻塞直到传入的 Publisher `onComplete()` 或者 `onError(e)`，冒出可能的异常。 |
| Stream.next() with Promise.await(), Promise.get()… | 如果信号是一个信号的话，就只在一个 Promise 中获取最近的下一个信号和 `onComplete()`。 `get()` 可以被用来跟约定接触，但并不等待约定满足。 |
| Stream.toList() with Promise.await(), Promise.get()… | 类似于 `next()`，但只获取 List<T> 中全部的序列来满足 `Promise<List<T>>` 的返回。 |
| Stream.toBlockingQueue() | 订阅 `Stream `并返回一个可迭代的阻塞 `Queue<T>`，它累积了所有的 `onNext` 信号。 `CompletableBlockingQueue.isTerminated()` 可以被用作一个退出阻塞 `poll()` 循环的条件。 |
| 绑定同步的 Stream | 它不特定针对任何API, 但是如果当前`Stream` 是在 `SynchronousDispatcher` 上面分配的, 实际上就会在\*\*终结\*\*动作时阻塞操作，例如`consume()`。 |

理解线程模型

**有赖于信号回调**，**Reactive Streams** 和 **Reactive Extensions** 通常目标就是不干预线程行为。 Stream 整个就只关心**它将会在某个从现在到某个时间 T 之间执行**。 非并行的信号可能也会禁止 Subscriber 的无共享(share-nothing)并发访问, 不过信号和请求可以在两个非对称的线程上运行。

Stream 默认被分配了一个SynchronousDispatcher 并将通过 Stream.getDispatcher() 来通知它当前的子 Actions。

各种 Stream 工厂、 Broadcaster、 Stream.dispatchOn 还有终端的 xxxOn 方法可能会变更默认的 SynchronousDispatcher。



+ Stream.dispatchOn 是 Stream 下唯一能在给定的 Dispatcher 上派发 **onError** 、 **onComplete** 和 **onNext** 信号的动作（action）。

+ 如果其上下文中已经就绪， request 和 cancel 也将会运行在调配器上面。否则它将会在当前的派发结束时执行。

+ Stream.subscribeOn 动作只会在传入的调配器上执行 **onSubscribe** 。

+ 由于传入的 Dispatcher 被调用的唯一时机是 **onSubscribe** , 任何的调配器，包括诸如 WorkQueueDispatcher 这样并发的都可以被使用。

+ 第一个 request 可能仍在 **onSubscribe** 线程中执行，例如使用 Stream.consume() 动作。

+ 通过 Stream.process 附上一个 处理器（Processor） ，例如也可以影响到线程。诸如 Processor 这样的线程可以在其管辖的线程上运行 Subscriber

+ request 和 cancel 将会运行在处理器上，如果其上下文已经准备就绪也是这样。

+ RingBufferWorkProcessor 将只会将 **onNext** 信号分派到一个 Subscriber ，最多除非是它已经取消了正在进行的处理 (向一个新的 Subscriber 进行回放)。

最常用来开请求数据的 onSubscribe, subscribeOn 是一类高效的用来扩大数据流的工具，特别是那些没有的边界的。 如果一个 Subscriber 在 onSubscribe 中请求 Long.MAX\_VALUE，然后它将会成为唯一要执行的请求，并且会运行于 subscribeOn中分配的调度器上。这是针对无边界的 `Stream.consume` 动作的默认行为。

无界需求在线程间的跳动

Streams  
 .range(1, 100)  
 .dispatchOn(Environment.sharedDispatcher()) //2  
 .subscribeOn(Environment.workDispatcher()) //1  
 .consume(); //3

1. 分配一个 onSubscribe 工作队列调配器。

2. 分配一个信号 **onNext**、**onError**、**onComplete** 调配器。

3. 使用 Subscription.request(Long.MAX) 来处理 Stream 的 **onSubscribe**。

图片地址：http://projectreactor.io/docs/reference/images/longMaxThreading.png

不过，在涉及到超过 1 个请求的时候，subscribeOn 就很少有用了, 就像使用 Stream.capacity(n) 按步骤处理一样。唯一的请求执行可能运行于在 subscribeOn 中分配的**第一个**调度器之上。

有界需求 1 在线程间的跳动

Streams  
 .range(1, 100)  
 .process(RingBufferProcessor.create()) //2  
 .subscribeOn(Environment.workDispatcher()) //1  
 .capacity(1); //3  
 .consume(); //4

1. 分配一个 **onSubscribe** 工作队列调配器。要注意它是被放置在了 处理（process） 之后， 因为 subscribeOn 将会运行订阅器的 上的 ringBuffer 现成之上，而我们想将其变更为工作调配器。

2. 分配一个异步信号 **onNext**, **onError**, **onComplete** 处理器。类似于 dispatchOn 的行为。

3. 将 Stream 的容量指定为 1，得以让下行流动作可以适配。

4. 使用 Subscription.request(1) 处理 Stream 的 **onSubscribe**，并且是在每次的 **onNext** 之后。

图片地址：http://projectreactor.io/docs/reference/images/nThreading.png

微批处理

微批处理

搾取系统所有的的CPU 和 内存，减少使用过度导致的延迟！

- 克林贡谚语

在读过一俩遍 101 种数据流崩溃说明(#streams-basics) 之后，你的黑客之心蠢蠢欲动，已经想*尽快*实验一下了。事实上，有效分派并非 **处理中每秒百万级数据待办列表** 中的唯一事项，这还远远不够。

在**分布式系统**中，独立部分及缓冲 IO 写入的延迟成本依然是一个常见问题。当遇到这种问题，**微批处理 (MicroBatching)** 或*小分块处理*是一种组织独立数据操作的方法。而术语中所谓的 **微** ，更具体的说来，就是指**在内存中**。尽管当今系统的限制已经是光速，内存读取依然比**硬盘**读取更加快捷。

延迟时间比较

>|||||

>|:------------------------------ | ----------------:| --------:|:---------------------- |

>| L1 缓存参考 | 0.5 ns | | |

>| 分支错误预测 | 6 ns | | |

>| L2 缓存参考 | 7 ns | | 14x L1 缓存 |

>| 互斥锁 加锁/解锁 | 25 ns | | |

>| 内存参考 | 100 ns | | |

>| 使用 Zippy 算法压缩 1K 字节 | 3,000 ns | | |

>| 在 1Gbps 网络上发送 1K 字节 | 10,000 ns | 0.01 ms | |

>| SSD 硬盘中随机读取 4K 字节 \* | 150,000 ns | 0.15 ms | |

>| 内存中按顺序读取 1MB 数据 | 250,000 ns | 0.25 ms | |

>| 在同一个数据中心中往返 | 500,000 ns | 0.5 ms | |

>| SSD 硬盘中顺序读取 1MB \* | 1,000,000 ns | 1 ms | 4x 内存 |

>| 机械硬盘寻道 | 10,000,000 ns | 10 ms | 20x 数据中心往返 |

>| 机械硬盘顺序读取 1MB | 20,000,000 ns | 20 ms | 80x 内存，20x SSD 硬盘 |

>| 发送数据包 CA->Netherlands->CA | 150,000,000 ns | 150 ms | |

>

>|备注 |

>|:-------------------------------------- |

>| 1 ns = 10^(-9) 秒 |

>| 1 ms = 10^(-3) 秒 |

>| \* 假设为读取速度约每秒 1GB 的 SSD 硬盘 |

>

>|致谢 |

>|:--------------------------------------------------------------------- |

>| 作者： Jeff Dean(http://research.google.com/people/jeff/) |

|  |  |
| --- | --- |
| **>>** | **[最初版本](http://norvig.com/21-days.html#answers)作者： Peter Norvig** |

**数据流**是序列化的数据，因此，寻找边界切分聚合的缓冲区是一种灵巧的API。



*缓冲区 (Buffer) ：合并边界，将*  *onNext<T>*  汇集\*在分组的 List<T> 中，然后给子 订阅者（Subscriber） 。

\* 最好同要求 Lterable<T> 作为输入参数的外部 API 一同使用。

*窗口 (Window) ：分离边界，将*  *onNext<T>*  *转发* \*给不同的 Stream<T> ，然后传递给子订阅者。

\* 最好同累加器一同使用，例如 reduce 或任何响应 onComplete() 的订阅者/ Action 。

\* 和可以结合 flatMap 或 concatMap 使用，这两个函数可以将独立的窗口合并回一个普通的 Stream<T> 。

使用缓冲区

将数据 T 按照序列分组为列表 List<T> 的主要目的有二：

\* 将匹配分界条件的序列暴露给一个 JVM API 常用的Iterable 结构体。

\* 减少 onNext(T) 的信号量，类如 buffer(5) 会将一个有10元素的序列转换成2个列表（每个列表有5个元素）。

收集数据将会产生内存甚或 CPU 的开销，应当适当的调整大小。建议使用小巧且定时的分界，以避免任何长时间的聚合。

如果一个 buffer() 被标记为定时的，却并未提供 Timer 参数时，必须先为其初始化一个环境（Environment）。

long timeout = 100;  
final int batchsize = 4;  
CountDownLatch latch = new CountDownLatch(1);  
  
final Broadcaster<Integer> streamBatcher = Broadcaster.<Integer>create(env);  
streamBatcher  
 .buffer(batchsize, timeout, TimeUnit.MILLISECONDS)  
 .consume(i -> latch.countDown());  
  
  
streamBatcher.onNext(12);  
streamBatcher.onNext(123);  
Thread.sleep(200);  
streamBatcher.onNext(42);  
streamBatcher.onNext(666);  
  
latch.await(2, TimeUnit.SECONDS);

|  |  |
| --- | --- |
| **Stream&lt;T&gt; API** | **作用** |
| buffer(int) | 聚合数据，直到 `onComplete()` 被调用；或是到达给定 `int` 参数的值，然后开始一个新的聚合。 |
| buffer(Publisher<?>, Supplier<? extends Publisher<?>>) | 聚合数据，直到 `onComplete()` 被调用，或是第一个 `Publisher<?>` 参数发出信号。可选参数 `Supplier<? extends Publisher<?>>` 提供了一个序列，它的第一个信号将终止其所链接的聚合。这意味着重叠的（位移的缓冲区）或脱节的聚合数据可以发送给子订阅者 `Subscriber<List<T>>`。 |
| buffer(Supplier<? extends Publisher<?>>) | 聚合数据，直到 `onComplete()` 被调用；或是与提供的 `Publisher<?>` 协调。`Supplier<? extends Publisher<?>>` 提供了一个序列，它的第一个信号将终止其所链接的聚合并立即开始一个新的聚合。 |
| buffer(int, int) | 聚合数据，直到 `onComplete()` 被调用；或是到达给定的\*\*忽略值\*\*（第二个参数），然后开始一个新的聚合。第一个尺寸参数`int` 将界定缓冲区聚合元素的最大数量。这意味着重叠的（位移的缓冲区）或脱节的聚合数据可以发送给子订阅者 `Subscriber<List<T>>`。 |
| buffer(long, TimeUnit, Timer\\_) | 聚合数据，直到 `onComplete()` 被调用；或是到达等待\*\*时长\*\*（第一个长整型参数），然后开始一个新的聚合。 |
| buffer(long, long, TimeUnit, Timer\\_) | 聚合数据，直到 `onComplete()` 被调用；或是到达给定的\*\*时移\*\*（第二个长整型参数），然后开始一个新的聚合。\*\*时间跨度\*\*（第一个长整型参数）将界定缓冲区聚合元素的最大数量。这意味着重叠的（位移的缓冲区）或脱节的聚合数据可以发送给子订阅者 `Subscriber<List<T>>`。 |
| buffer(int, long, TimeUnit, Timer) | `buffer(int)` \*\*或\*\* `buffer(long, TimeUnit, Timer)` 条件的组合。数据聚合的过程在到达给定\*\*大小\*\*或\*\*时间跨度\*\*耗尽时完成。 |

使用窗口

将数据 T 按照序列分发给 Stream<T> 的主要目的有三：

将数据 T 的一个序列暴露给一系列有限且分组的观察和统计：取和计算、平均值计算或灵活的聚合（Map、Tuple……）。

将分组数列同 dispatchOn 结合，并为生成的每个 Stream<T> 进行并行化处理。

对每个独立的分组序列重复 onComplete() 调用，例如，在异步 IO 模块中界定一次数据冲洗。

如果是同聚合所有的 Stream.buffer() 方法相结合，Stream<T> 窗口等效于聚合生产者，较之缓冲 API稍欠优化：

`

stream.buffer(10, 1, TimeUnit.SECONDS);

//equivalent to

stream.window(10, 1, TimeUnit.SECONDS).flatMap( window -> window.buffer() )

`

如果一个 window() 被标记为定时的，却并未提供 Timer 参数时，必须先为其初始化一个 环境（Environment）。

//create a list of 1000 numbers and prepare a Stream to read it  
Stream<Integer> sensorDataStream = Streams.from(createTestDataset(1000));  
  
//wait for all windows of 100 to finish  
CountDownLatch endLatch = new CountDownLatch(1000 / 100);  
  
Control controls = sensorDataStream  
 .window(100)  
 .consume(window -> {  
 System.out.println("New window starting");  
 window  
 .reduce(Integer.MAX\_VALUE, (acc, next) -> Math.min(acc, next))  
 .finallyDo(o -> endLatch.countDown())  
 .consume(i -> System.out.println("Minimum " + i));  
 });  
  
endLatch.await(10, TimeUnit.SECONDS);  
System.out.println(controls.debug());  
  
Assert.assertEquals(0, endLatch.getCount());

|  |  |
| --- | --- |
| **Stream&lt;T&gt; API** | **作用** |
| window(int) | 分发数据到一个生成的 `Stream<T>`，直到 `onComplete()` 被调用；或是到达给定 `int` 参数的值，然后开始一个新的聚合。 |
| window(Publisher<?>, Supplier<? extends Publisher<?>>) |  |
| window(Supplier<? extends Publisher<?>>) |  |
| window(int, int) | 分发数据到一个生成的 `Stream<T>`，直到 `onComplete()` 被调用；或是到达给定的\*\*忽略值\*\*（第二个参数），然后开始一个新的 `Stream<T>`。第一个尺寸参数`int` 将界定缓冲区聚合元素的最大数量。这意味着重叠的（位移的缓冲区）或脱节的聚合数据可以发送给子订阅者 `Subscriber<Stream<T>>`。 |
| window(long, TimeUnit, Timer\\_) | 分发数据到一个生成的 `Stream<T>`，直到 `onComplete()` 被调用；或是到达等待\*\*时长\*\*（长整型参数），然后开始一个新的 `Stream<T>`。 |
| window(long, long, TimeUnit, Timer\_) | 分发数据到一个生成的 `Stream<T>`，直到 `onComplete()` 被调用；或是到达给定的`时移`（第二个长整型参数），然后开始一个新的聚合。\*\*时间跨度\*\*（第一个长整型参数）将界定缓冲区聚合元素的最大数量。这意味着重叠的（位移的缓冲区）或脱节的聚合数据可以发送给子订阅者 `Subscriber<Stream <T>>`。 |
| window(int, long, TimeUnit, Timer) | `buffer(int)` 和 `buffer(long, TimeUnit, Timer)` 条件的组合。对生成的 `Stream<T>` 分发数据的过程在到达给定\*\*大小\*\*或`时间跨度`耗尽时完成。 |

背压和溢出

多数情况下，依照 **Reactor Stream** 的协定，背压可以被自动处理。如果订阅者（Subscriber）请求的数据并没有超过其处理能力（例如类似 Long.MAX\_VALUE 的东西），数据源上游可以避免发送过多数据。如果你想在使用一个 “冷”的`发布者（Publisher）` 时享受这种便利，你必须可以在任何时候关闭数据源的读取操作：*从 socket 中读取多少、SQL 查询指针中有多少行、文件中有多少行、迭代构造体中有多少元素……*

如果是 **热** 数据源，例如定时器或 **UI** 事件，或是一个可能从大型数据集上请求 Long.Max\_VALUE 大小数据的`订阅者（Subscriber）`，开发者必须针对**背压**制定明确的策略。



\* 非控（热）序列应当主动管理。

**减少** 序列的信号量，例如“取样”。

当需求超过容量时， **忽略** 数据。

当需求超过容量时， **缓冲** 数据。

\* 受控（冷）序列应当被动管理。

通过 **降低** 来自 订阅者（Subscriber） 或 Stream 上任意点的 **需求** 。

通过延迟请求 **断歇需求** 。

**Reactor 扩展**文档中应用最广泛的示例就是 **Marble Diagram**，双时间线帮助我们更直观的了解发布者（Publisher）、Stream以及订阅者（Subscriber） （如Action）在何时被观察，观察的内容又是什么。我们将使用这些图表来强调需求流，表明例如 *Map* 和 *filter* 这样的变换的本质。

图片地址：http://projectreactor.io/docs/reference/images/marble/marble-101.png

当两个 Action 的调度器或容量不同时，**Reactor** 将自动提供一个内存溢出缓冲区。这不适用于核心处理器，它有自己的溢出处理机制。调度器可以重复使用，且 **Reactor** 必须限制调度器的数量，因此 Action 的调度器不同时，将添加内存缓冲区。

Streams.just(1,2,3,4,5)  
 .buffer(3) //1  
 //onOverflowBuffer()  
 .capacity(2) //2  
 .consume()  
  
  
Streams.just(1,2,3,4,5)  
 .dispatchOn(dispatcher1) //3  
 //onOverflowBuffer()  
 .dispatchOn(dispatcher2) //4  
 .consume()

1. buffer 操作设定容量为 3。

2. consume() 或任何下游动作都被设定为 capacity(2)，隐式的添加了一个 onOverflowBuffer()。

3. 在调度器 1 上执行第一个动作。

4. 在调度器 2 上执行第二个动作，隐式的添加了一个 onOverflowBuffer()。

最终Subscriber可以逐一的请求数据，限制管道中传输的数据为一个元素，并在每次成功调用 onNext(T) 后请求下一个元素。这种行为也可以通过 capacity(1).consume(...) 获得。

Streams.range(1,1000000)  
 .subscribe(new DefaultSubscriber<Long>(){   
 Subscription sub;  
  
 @Override  
 void onSubscribe(Subscription sub){  
 this.sub = sub;  
 sub.request(1);   
 }  
  
 @Override  
 void onNext(Long n){  
 httpClient.get("localhost/"+n).onSuccess(rep -> sub.request(1));   
 }  
 );

1. 使用 DefaultSubscriber 以避免逐个实现订阅者的所有方法。

2. 持有订阅的指针后安排第一次需求请求。

3. 在成功的 **GET** 请求后，使用 异步 HTTP API(http://projectreactor.mydoc.io?v=10626&t=44516) 再次请求。延迟信息自然将被传递给 RangeStreamPublisher。你可以想到，通过计算两次请求的时间间隔，我们将能够深入的了解执行过程及 IO 操作所产生的延迟。

|  |  |
| --- | --- |
| **Stream&lt;T&gt; API** | **作用** |
| subscribe(Subscriber<T>) | 自定义 `Subscriber<T>` 的请求时机很灵活。如果`订阅者`使用阻塞操作，最好改变这些请求的大小。 |
| capacity(long) | 对当前 `Stream<T>` 的[设定容量](${domain}?v=10626&t=44506) |
| onOverflowBuffer(CompletableQueue) | 创建或使用给定的 `CompletableQueue` 储存溢出元素。当`订阅者`请求的数据量小于 `Publisher` 送出的数据量时，将产生溢出。溢出的数据将在下一次 `request(long)` 调用时被消耗。 ![marble-overflowbuffer](http://projectreactor.io/docs/reference/images/marble/marble-overflowbuffer.png) |
| onOverflowDrop() | 忽略溢出的元素。当 `Subscriber` 请求的数据量小于 `Publisher ` 送出的数据量时，将产生溢出。溢出的数据将在下一次 `request(long)` 调用时被消耗。 ![marble-overflowdrop](http://projectreactor.io/docs/reference/images/marble/marble-overflowdrop.png) |
| throttle(long) | 延迟下游的 `request(long)` ，并定期削减积累的对上游请求的需求。 ![marble-throttle.png](http://projectreactor.io/docs/reference/images/marble/marble-throttle.png) |
| requestWhen(Function<Stream<Long>, Publisher<Long>>) | 将所有下游的 `request(long)` 传递给 `Stream<Long>` 的请求序列，此序列可以变更，并返回任意形式的`Publisher<Long>`。`RequestWhenAction` 将订阅生产序列，并立即对上游的 `request(long)` 分配 `onNext(Long)`。它的行为类似于 `adaptiveConsume`，但可以插入到`数据流`管道的任意位置。 |
| batchConsume(Consumer<T>, Consumer<T>, Consumer<T>, Function<Long,Long>) batchConsumeOn | 类似于 `consume`，但将使用默认的 `Stream.capacity()` 启动，请求映射的 `Long` 需求，并给出前一次的需求。对适配来源各异的需求非常有用。 |
|  | 类似于 `batchConsume`, 但将请求 `Long` 需求计算后的序列。它可以用于插入流程控制，例如延迟需求的 `Stream.timer()`。`AdaptiveConsumerAction` 将订阅生产序列，将订阅生产序列，并立即对上游的 `request(long)` 分配 `onNext(Long)`。 |
| process(Processor<T, ?>) | 任何`处理器（Processor ）`都可以对需求或缓冲数据进行改造。对使用中的特定处理器实现进行检查是值得的。 |
| All filter(arguments), take(arguments), takeWhile(arguments)… | 所有的限定操作都可以用于主动限制 `Stream` 的信号量。 |
| buffer(arguments), reduce(arguments), count(arguments)…​ | 所有的聚合操作及度量操作都可以用于主动限制 `Stream` 的信号量。 |
| All sample(arguments), sampleFirst(arguments) | 通过对最后一个（或首个） `onNext(T)` 信号进行条件匹配，减少 `Stream<T>` 的信号量。这些条件可以被计数、计量、计数或计量，甚可以互动（事件驱动）。 |
| zip(arguments), zipWith(arguments) | 将 N `Stream<T>` 的信号量减少至压缩`发布者`生产的最后一次信号。每个`发布者`的聚合信号都可以用于从 N 最近的上游 `onNext(T)` 产生单独的值。 |

组合操作

为了协调数据的**并行序列**，我们可以组合发布者。由于生成序列是合并的结果，它们也可以用于数据的[异步转换][异步转换](http://projectreactor.mydoc.io?v=10626&t=44507)。

通过非阻塞的协调方式可以避免开发者使用 Future.get() 或 Promise.await()，这两个方法在多信号存在是容易引发问题。非阻塞意味着管道除了订阅者的需求，不会做任何等待。订阅者的请求将被切分至最小，然后分配给已经组合的发布者。

合并行为在 FanInAction 中建模，并通过一个订阅者委托的**线程偷取型** SerializedSubscriber代理处理并行信号。它将对校验每个信号，查看对应的委托订阅者是否已经运行，如果没有运行，则重新分配信号。当繁忙的线程关闭订阅者代码时，信号将被轮询，处理信号的线程很可能已经不再是生产它的那个了。

在使用 flatMap 之间就削减需求信号量(http://projectreactor.mydoc.io?v=10626&t=44493) 没法说是好主意还是坏主意。实际上，如果无法处理所有的数据，是没有必要订阅多个并行发布者并合并操作的。然而它对并行发布者数据量的限制，也不会给予高速发布者挂起请求的机会。

Stream.zipWith(Function)

Streams  
 .range(1, 100)  
 .zipWith( Streams.generate(System::currentTimeMillis), tuple -> tuple ) //1  
 .consume(  
 tuple -> System.out.println("number: "+tuple.getT1()+" time: "+tuple.getT2()) , //2  
 Throwable::printStackTrace,  
 avoid -> System.out.println("--complete--")  
 );

1. “Zip” 或聚合来自 RangeStream 的最新的信号，传递 SupplierStream 以提供当前时间。

2. 通过 “Zip” 操作，压缩发布者按照声明的顺序（自左及右，stream1.zipWith(stream2)）生成数据元组。

|  |  |
| --- | --- |
| **功能 API 或工厂方法** | **角色** |
| Stream.flatMap(Function<T, Publisher<V>>) | 这是一个[异步转换][异步转换](${domain}?v=10626&t=44507)，是一个 `map(Function<T, Publisher<V>>).merge()` 的类型化快捷方式。 映射部分使用传递来的数据 `T` 生成一个 `Publisher<V>`，这是[微服务构架](http://projectreactor.io/docs/reference/#streams-microservice)中常用的一种模式。 |
| Streams.switchOnNext(Publisher<Publisher<T>>) | 在传入的发布者发出 `onNext(Publisher<T>)` 信号的间隙， Stream 会以 FIFO（先进先出）的顺序轮替。这个信号将使下游 Subscriber<T> 接收到下一个 `onNext(T)` 的发布者序列。当 ` onNext(Publisher<T>)`信号被接收时，可能会打断上游正在进行的发布。所有的 `Publisher<T>` 都必须在 `Subscriber<T>` 完结之前完结。 |
| Streams.merge(Publisher<T>, Publisher<T> x7) Streams.merge(Publisher<Publisher<T>>) Stream.mergeWith(Publisher<T>) Stream.merge() |  |
| Streams.concat(Publisher<T>, Publisher<T> x7) Streams.concat(Publisher<Publisher<T>>) Stream.concatWith(Publisher) Stream.startWith(Publisher) | 类似 `merge()` ，但如果一个 Publisher<T> 已经在发布中，则等待它的 `onComplete()`，然后再接处理下一个挂起的 Publisher<T>。序列将以声明的顺序被订阅，自左及右，例如 `stream1.concatWith(stream2)`，或是使用参数—— `stream2.startWith(stream1)`。 |
|  | 合并来自独立`Publisher<T>`的最新 `onNext(T)` 信号。合并状态持续到信号源 `Publisher<T>` 发出新的 `onNext(T)` 信号并取代它。在所有 `Publisher<T>` 都发出至少一个信号\*\*之后\*\*，给定的组合器函数将接收所有合并的信号，并生成期望的合并对象。 |
|  | 合并来自独立`Publisher<T>`的最新 `onNext(T)` 信号。合并状态持续到信号源 `Publisher<T>` 发出新的 `onNext(T)` 信号并取代它。\*\*每当\*\* 所有 `Publisher<T>` 都发出一个信号时，给定的压缩函数将接收所有的信号，并生成期望的压缩对象。如果过一个 `Publisher<T>` 完结，下游 `Subscriber<T>` 也将完结。 |
| Streams.join(Publisher<T>, Publisher<T> x7) Streams.join(Publisher<Publisher<T>>) Stream.joinWith(Publisher<T>) | 功能预设配置的 `zip`命令的快捷方法，可以将最新的 `Tuple` 转化成 `List<?>`。 |

微服务

微服务

微服务(http://martinfowler.com/articles/microservices.html) 这个概念近年来愈发流行了起来。简单来说，我们编写软件组件的一个首要目标就是鼓励*独立性*、*缩放*适应性以及*重用性*。事实上，我们使用微服务已经超过 30 年了。

Unix 下一个微服务的使用示例

history | grep password

甚至在应用程序内部，我们也可以找到类似概念的功能粒度。

*命令式* Java 代码中微服务的示例

User rick = userService.get("Rick");  
User morty = userService.get("Morty");  
List<Mission> assigned = missionService.findAllByUserAndUser(rick, morty);

当然，在分布式系统和云就绪构架(http://12factor.net)中，这种应用程序的使用已经相当广泛了。只要功能足够独立，它将只取决于 N， 其它部分则可以:访问数据、通过网络调用子程序、发送数据到消息总线、查询 HTTP REST 端点等等。这也导致了一些麻烦：**执行流跨越了数个上下文边界**。随着系统数据量和访问的增长，相对的延迟和失败也将被放大。

在这点上，我们可以选择向外扩展，多数平台，例如 CloudFoundry(https://www.cloudfoundry.org)都允许对 JVM 程序进行弹性增减。但看看我们对 CPU 和内存的使用，在压力之下并不很好。当然不好，每个远程调用都阻塞了整个服务，阻止了并发用户的请求接入：它们只是停留在了某个线程池队列中。同时，主动的请求很愉快的的停留了数个毫秒，或是更久，等待远程 HTTP 请求 socket 并实际进行写入操作，我们把这些延误称为**延时**。

这也同样适用于错误，我们可以让应用更具弹性（回退机制、超时机制、重试机制……），独立性是最主要的，不对外依赖。一个经典的愿景是微服务处理请求，负载均衡器监测失败：

Load Balancer: "are you dead ?"  
30 sec later  
Load Balancer: "are you dead ?"  
30 sec later  
Load Balancer: "you're dead !"  
MicroService "I'am alive !"

类似 Stream 或 Promise 的 Publisher 对处理**微服务**的延迟和错误非常理想。想要更好的进行错误隔离，非阻塞服务调用，你的代码必须符合这两者的约束。为了能够成功的迁移到 Reactive 构架，你最好能够迭代开发，不断的调整，测试。

本节中，我们将在基础知识之上创建一个响应式外观，处理每个代价高昂的远程调用，构建功能性服务，并使它们应对延迟。



1.将目标服务调用转化成 *无界限* 的 Stream 或 Promise 类型返回

- 异步切换阻塞→非阻塞开关

- 错误隔离

2. 使用 **Reactor Stream** API 组合成服务

- 阻塞→非阻塞协作

- 并行处理阻塞调用

3. 将转化后的服务升级为背压就绪的 Stream

- 使用有界限的访问分块进行处理/读取操作。

- 使用微批处理优化 IO 操作

|  |  |
| --- | --- |
| **功能 API 或工厂方法** | **作用** |
| Streams.create(Publisher), Streams.defer(Supplier), Streams.wrap(Publisher), Streams.generate(Supplier) | 我们鼓励使用`发布者`保护资源访问。一些 [Stream 工厂](${domain}?v=10626&t=44489)很有用处。创建`发布者`的关键点是在数据准备好（例如 IO 回调）后的 `onNext(T)`。如果可以实现某种形式的背压，读取操作只应由`订阅者`的请求触发。 |
| Stream.timeout(arguments) | 访问外部数据，特别是远程访问，一定要限制时长，这样可以更具弹性的应对外部条件，例如网络环境。超时操作可以回退到另一个`发布者`以接替服务调用，或者单纯调用 `onError(TimeoutException)`。计时器在每次观测到新的 `onNext(T)` 调用时将被重置。 |
| Stream.take(arguments) | 类似于 `timeout()`，限定外部资源的需求大小范围也是一种常见方式。它也可以用于触发一个含有 `onComplete()` 操作的管道。 |
| Stream.flatMap(Function<T,Publisher<V>) | 这个[异步转换](${domain}?v=10626&t=44507) 可以使用传递的数据 `T` 创建 `Publisher<V>`。可以在恢复当前执行过程之前，将调用挂载在另一个服务商。 通过\*安全的\*并行订阅，产生的 `Publisher<V>` 将以 `V` 的序列流入`订阅者`。 |
| Stream.subscribeOn(Dispatcher), Stream.dispatchOn(Dispatcher), \*核心处理器\* | [线程控制](${domain}?v=10626&t=44490) 的策略： > \* 低速微服务，低容量，或低吞吐量，例如 HTTP GET 请求 → `subscribeOn(workQueueDispatcher())` ，将请求扩展到并发服务调用。 > \* 高速微服务，高容量，或高吞吐量，例如 消息总线 → `dispatchOn(sharedDispatcher())` 或 `RingBufferXXXProcessor.create()` ，进行消息分派扩展。 |

创建非阻塞服务

首先要做的就是隔离微服务的访问。现在，我们将不再返回 T 或 Future<T>，取而代之的将是 Publisher<T> 以及特定的 Stream<T> 或 Promise<T>。立竿见影的好处就是我们可以不必费心考虑错误处理或是线程（仅是现在)了：错误将在 onError 调用（非冒泡）中传递，而线程可以在后续过程中使用类似 dispatchOn 的方法进行调整。另一个好处就是它能让我们的代码更加实用。Java 8 Lambdas 也可以正常使用。我们的目标是减少控制语句中的括号噪音（if、for、while 等），并限制上下文中分享的需求。我们的最终设计目标就是鼓励在大型数据集中流式轮询：函数将适用于序列，一个一个返回结果，避免循环重复。

我们提倡使用实现的部件且非 Publisher<T> 在编译时访问所有的 Reactor Stream API，除非你希望你的 API 隐藏（图书馆的开发者可能有此需求）。 Streams.wrap(Publisher<T>) 将会使尽解数将这种通用的返回类型转化成一个合适的 Stream<T>`。

|  |  |
| --- | --- |
| **不是很好的** | **很好的** |
| ```java public User get(String name) throws Exception { Result r = userDB.findByName(name); return convert(r); } public List<User> allFriends(User user) throws Exception { ResultSet rs = userDB.findAllFriends(user); return convertToList(r); } public Future<List<User>> filteredFind(String name) throws Exception { User user = get(name); if(user == null || !user.isAdmin()){ return CompletedFuture.completedFuture(null); } else { //could be in an async thread if wanted return CompletedFuture.completedFuture(allFriends(user)); } } ``` | ```java public Promise<User> get(final String name) { return Promises .task( () -> userDB.findByName(name)) .timeout(3, TimeUnit.Seconds) .map(this::convert) .subscribeOn(workDispatcher()); } public Stream<User> allFriends(final User user) { return Streams .defer(() -> Streams.just(userDB.findAllFriends(user))) .timeout(3, TimeUnit.Seconds) .map(this::convertToList) .flatMap(Streams::from) .dispatchOn(cachedDispatcher()); .subscribeOn(workDispatcher()); } public Stream<User> filteredFind(String name){ return get(name) .stream() .filter(User::isAdmin) .flatMap(this::allFriends); } ``` |

|  |
| --- |
| **结果** |
| \* \*\*所有的查询方法\*\* 时： \* 不再有`异常抛出`，异常都在管道中传递。 \* 不再有控制逻辑，我们使用例如 map 或 filter 这样的预定义操作。 \* 只返回`发布者`(Stream 或 Promise)。 \* 通过超时机制限制阻塞的查询（之后可以进行重试或者回退等操作）。 \* 在订阅时使用一个分配好的 workDispatcher。 \* \*\*get(name):\*\* 时： \* 使用类型定义的\*\*单一数据\*\*发布者，或是`Promise`。 \* 在订阅时，调用\*\*task\*\*回调函数。 \* \*\*allFriends(user):\*\* 时： \* 在 \*\*onSubscribe\*\* 线程中使用 `defer()` 惰性的调用数据库查询。 \* 现在还没有背压策略，我们在一个阻塞（但异步）的调用中读取所有的结果。 \* 在 FlatMap 中，我们将返回的列表转化为一个数据流。 \* 在一个异步调度器中分派数据，这样下游处理不会影响读取操作。 \* \*\*filteredFind(name):\*\* 时： \* 我们使用 `stream()` 将第一个 get 操作获取到的 `Promise` 转化为 Stream。 \* 如果用户可用，我们只需要调用 allFriends() sub-stream。 \* 返回的 Stream<User> 将在第一次 allFriends() 信号发出后恢复。 |

组合多个服务调用

第二步，我们将思维扩大到消费方面。在过渡阶段，请牢记我们的 Stream 可以使用运算符来阻塞(http://projectreactor.mydoc.io?v=10626&t=44507) 。 我们有两个问题亟需处理：一个是鲁棒性（网络分裂问题等），一个是如何避免在新服务执行前等待前一服务：

|  |  |
| --- | --- |
| **不是很好的** | **很好的** |
| ```java int tries = 0; while(tries < 3){ try{ Future<List<User>> rickFriends = userService.fitleredFind("Rick"); Future<List<User>> mortyFriends = userService.fitleredFind("Morty"); System.out.println( rickFriends.get(3, TimeUnit.SECONDS) .addAll( mortyFriends.get(3, TimeUnit.SECONDS)) ); }catch(Exception e){ if(tries++ >= 3) throw e; Thread.sleep(tries\*1000); } } ``` | ```java return Streams.merge( userService.filteredFind("Rick"), userService.filteredFind("Morty") ) .buffer() .retryWhen( errors -> errors .zipWith(Streams.range(1,3), t -> t.getT2()) .flatMap( tries -> Streams.timer(tries) ) ) .consume(System.out::println); ``` |

|  |
| --- |
| **结果** |
| \* `Streams.merge()` 将两个查询合并，是一个非阻塞的协调操作。 \* `buffer()` 将聚合所有结果，直到运行完结或失败（在之前计数的）。 \* `retryWhen(Function<Stream<Throwable>, Publisher<?>>` 将在出错时保证重新订阅。 \* `zipWith` 将合并错误，并进行至多 3 次的重试。 \* `zipWith` 只返回元组重试的数量。 \* `flatMap` + `Streams.timer(long)` 将每次重试转化为延迟信号（使用默认时间）。 \* 每当此出错信号返回到`发布者`时，取消并重新订阅，直到一个 `onComplete` 信号或 `onError`信号被发出。 \* `flatMap` 只在内部计数器和上游都完结时结束，也就是在 3次重试之后，或是在错误序列之后，它才会总结。 |

支持响应式背压

最后一步，我们将探访 *UserService.allFriends 查询*，它将从数据中获取整个数据集。

|  |  |
| --- | --- |
| **好的代码** | **超赞的代码** |
| ```java return Streams .defer(() -> Streams.just(userDB.findAllFriends(user))) .timeout(3, TimeUnit.Seconds) .map(this::convertToList) .flatMap(Streams::from) .dispatchOn(cachedDispatcher()); .subscribeOn(workDispatcher()); stream .buffer() .consume(System.out::println); ``` | ```java return Streams .createWith( (demand, sub) -> { ResultSet rs = sub.context(); long cursor = 0l; while(rs.hasNext() && cursor++ < demand && !sub.isCancelled()){ sub.onNext(rs.next()); } if(!rs.hasNext()){ sub.onComplete(); } }, sub -> userDB.findAllFriends(user), resultSet -> resultSet.close() ) .timeout(3, TimeUnit.Seconds) .map(this::convert) .dispatchOn(cachedDispatcher()); .subscribeOn(workDispatcher()); stream .buffer(5, 200, TimeUnit.MILLISECONDS) .consume(System.out::println); ``` |

|  |
| --- |
| **结果** |
| \* 是的，这很啰嗦。 \* …但现在，我们将查询的结果依次流式处理（可能已经通过 SQL 限定分页）。 \* `Streams.createWith` 是一个 `PublisherFactory`，它将截断请求，执行启动或停止操作。 \* 请求的消费者给出了订阅者准备接收的元素数量。 \* 请求的消费者接收一个 `SubscriberWithContext`，它实现了实际`订阅者`的委托，提供了对共享上下文和取消状态的访问。 \* 我们发送了至多同需求相当的独立结果。 \* 在读取充分执行后，我们的操作将完结。 \* 由于现在数据是独立的，`convertToList`不再需要了，用 `convert` 替换它。 \* 数据消费方面，可以使用一些诸如 `capacity(long)` 或 `buffer(int)` 的工具 5 个一组的批量消费请求。 \* 这样做的结果是流动可以明显加快，因为不需要在每行读取后打印它。 \* 由于批处理可能无法匹配大小，我们还要加上一个时间限制。 |

对类似 List<T> 这样的状态化 Iterable<T> 的使用和对独立流化的 T 的使用之间的平衡非常重要。由于创建需要一定的时间，List<T> 可能会导致一些延迟。而且它的恢复性也不好，一旦出错，我们会丢失一整批的数据。最后，流化的 T 数据将使限定尺寸的需求更可预测，因为我们能够发出单个的信号，而不是批量的信号。

错误处理

由于错误隔离是 **Reactive** 协定中一个很重要的部分，Stream API 已经实现了容错管道和相关的服务调用。

错误隔离可以防止 onNext、onSubscribe 和 onComplete 回调函数弹出任何异常。作为替代，这些异常被传递给 onError 回调函数，并传播给下游。一些 Action 可以积极或消极的对待这些信号，类如 when() 仅仅观察错误，而 onErrorResumeNext() 则可以切换至备用发布者。

将传播过程逆向至消费侧而非生产侧是将数据生产者同管道错误隔离，保证生产者快活力的一种响应式的模式。

最后，错误将通过 onError() 回调函数通知给链中的最后一个订阅者。假设 订阅者 是一个 ConsumerActon，如果没有一个通过 Stream.consume(dataConsumer, errorConsumer) 注册的 *errorConsumer* 回调函数存在，Reactor 将重新路由错误信号。路由将触发当前环境的错误日志（如果有设定），默认使用 SLF4J 记录错误。

**Reactor** 对**致命异常**的处理也不同常规，特别是在 onSubscribe 执行过程中。这些异常将不被孤立，且不会传递给下游 subscriber(s):

\* CancelException

\* 如果 onNext 信号传播时没有可用的订阅者，此异常将被触发，例如在 onNext 信号传输时，订阅者被异步的取消了。

*使用 JVM 属性*  *-Dreactor.trace.cancel=true* \* 可以开启 CancelException 的详细模式，并将其记录在默认的环境日志中。如果不设置，环境日志中不会记录异常以及相关的错误堆栈。

\* ReactorFatalException

\* 此异常在 Reactor 遇到不可恢复的情况时触发，例如在 Timer 的调配不能匹配条件时。

\* JVM unsafe exceptions:

\* StackOverflowError

\* VirtualMachineError

\* ThreadDeath

\* LinkageError

很多章节中都可以看到明确设定时间限制的好习惯，timeout() + retry() 将是你对付网络分裂问题的最好伴侣。流向 Stream 的数据越多，它就越应具有自愈性和良好的服务可用性。

理论上，在Reactive Streams中最多有一个错误能够穿过通道，因此你实不必在一个订阅者 上两次重复 onError(e)。而实践中，我们实现 Rx 的 retry() 和 retryWhen() 操作符将在 onError 时进行取消和重订阅操作。就是说，在新的通道，带着新的操作示例，被同名的物化时，我们依然遵循着协定。这也意味着在这种情形下，像 buffer() 这样状态化的 Action 应当谨慎使用，因为我们只是取消了对它们的引用，它们的状态可能会丢失。我们正在研究替代方案，一个想法就是为安全的状态化 Action 引入外部持久化。你可以在相关章节(http://projectreactor.mydoc.io?v=10626&t=44498) 窥见一斑。**回退流很有趣**

Broadcaster<String> broadcaster = Broadcaster.create();  
  
Promise<List<String>> promise =  
 broadcaster  
 .timeout(1, TimeUnit.SECONDS, Streams.fail(new Exception("another one!")))   
 .onErrorResumeNext(Streams.just("Alternative Message"))   
 .toList();  
  
broadcaster.onNext("test1");  
broadcaster.onNext("test2");  
Thread.sleep(1500);  
  
try {  
 broadcaster.onNext("test3");  
} catch (CancelException ce) {  
 //Broadcaster has no subscriber, timeout disconnected the pipeline  
}  
  
promise.await();  
  
assertEquals(promise.get().get(0), "test1");  
assertEquals(promise.get().get(1), "test2");  
assertEquals(promise.get().get(2), "Alternative Message");

1. 当给定的时间段内没有数据发出时，TimeoutAction 可以提供回滚，但这个示例中它仅仅发出了另一个异常……

2. ……不过，我们很幸运有可以捕捉此异常的 onErrorResumeNext(Publisher) ，它传递了一些有效的字符串负荷。

另一个经典的管道容错示例在手册(http://projectreactor.mydoc.io?v=10626&t=44523)一节

|  |  |
| --- | --- |
| **Stream&lt;T&gt; API** | **作用** |
| when(Class<Throwable>, Consumer<Throwable>) | 观察来自`onError(Throwable)` 信号的特定的异常类型（以及它们的子类） |
| oberveError(Class<Throwable>, BiConsumer<Object,Throwable>) | 同 `when()` 相似，但在异常初次触发时可以对失败的 `onNext(Object)` 内省。 |
| onErrorReturn(Class<Throwable, Function<Throwable,T>) | 提供一个信号 `T`的回退，给定一个匹配传递类型的异常。通常用于自愈服务。 |
| onErrorResume(Class<Throwable, Publisher<T>) | 提供一个信号 `T`的回退序列，给定一个匹配传递类型的异常。通常用于自愈服务。 |
| materialize() dematerialize() | 将上游信号转换为 `Signal<T>` ,并将其视为 `onNext(Signal<T>)` 信号进行处理。直接效果就是：它将接收错误信号和完结信号，因此可以用来高效的处理错误。一旦有错误产生，我们可以将 `dematerialize()` 回调中的 `Signal<T>` 转化到 Reactive Streams，保证服务的运行。 |
| retry(int, Predicate<Throwable) | 对父流进行取消操作和重订阅操作，次数由可选的 \*tries\* 限制，如果提供了 `Predicate` ，也需要进行匹配。 |
| retryWhen(Function<Stream<Throwable>,Publisher<?>>) | 当传递的 `Function` 中的 `Publisher` 发出 `onNext(Object)` 信号时，对父流进行取消操作和重订阅操作。函数将在生成的`Publisher`被订阅时被调用。如果 `Publisher` 发出 `onError(e)` 或 `onComplete()` 信号，它们将被发送到下游。`Function` 接收一个可能在任一管道中出错的 `Stream`。可以同\*\*计数\*\*和\*\*延迟\*\*配合，提供有界并高速的重试策略。 |
| recover(Class<Throwable>, Subscriber<Object>) | 只是一个 `retryWhen()` 的快捷方式，当`onError(Throwable)` 匹配给定类型时对父流进行取消操作和重订阅操作。当恢复成功时，传递的 `Subscriber` 参数将接受到同异常关联的最初的 `onNext(Object)` 信号。 |
| ignoreError(Predicate<Throwable>) | 将匹配的 `onError(Throwable)` 信号转换为 `onComplete()` 信号。如果没有给出参数，那么所有的异常都将被转化。 |
| 抛出 CancelException | 这可能是我们唯一依次提及异常冒泡。在所有 `onNext(T)` 回调中抛出 `CancelException.INSTANCE` 的方法很简单，可以对传入值进行 \*\*no-ack\*\* 处理，并通知同位的（同一线程栈中的）发布者，例如\*\*核心处理器\*\*，在之后重新分配这个数据。 |

Stream 数据持久化

并不是所有的数据都要待在内存里，Reactor已经开始整合（可选依赖）Java Chronicle(https://github.com/OpenHFT/Chronicle-Queue)。

return Streams.merge(  
 userService.filteredFind("Rick"),  
 userService.filteredFind("Morty")  
)  
.buffer()  
.retryWhen( errors ->  
 errors  
 .zipWith(Streams.range(1,3), t -> t.getT2())  
 .flatMap( tries -> Streams.timer(tries) )  
)  
.consume(System.out::println);

|  |  |
| --- | --- |
| **功能 API 或 工厂方法** | **作用** |
| Stream.onOverflowBuffer(CompletableQueue) |  |
| IOStreams.persistentMapReader() |  |
| IOStreams.persistentMap() |  |

分析

度量操作和其它状态化操作一样，都是 Stream API 的一部分。实际上，熟悉 Spark 的用户能够认出一些方法。ScanAction 也提供了一些常用的同 reduce() 和 scan() 相关的累积功能。

Broadcaster<Integer> source = Broadcaster.<Integer> create(Environment.get());  
long avgTime = 50l;  
  
Promise<Long> result = source  
 .throttle(avgTime) (1)  
 .elapsed() (2)  
 .nest() (3)  
 .flatMap(self ->  
 BiStreams.reduceByKey(self, (prev, next) -> prev + 1) (4)  
 )  
 .sort((a,b) -> a.t1.compareTo(b.t1)) (5)  
 .log("elapsed")  
 .reduce(-1L, (acc, next) ->  
 acc > 0l ? ((next.t1 + acc) / 2) : next.t1 (6)  
 )  
 .next(); (7)  
  
for (int i = 0; i < 10; i++) {  
 source.onNext(1);  
}  
source.onComplete();

1. 将传入的订阅者（Publisher）减速至每 50 毫秒一次，逐个等待数据发出。

2. 在 onSubscribe 和 第一个信号之间，或是在两个信号之间产生一个拥有**时间增量**和**有效载荷**的 Tuple2。

3. 使当前流可以接收 onNext 信号，以便我们将其同 flatMap组合。

4. 累积所有数据，直到以 Tuple2.t1 和 Tuple2.t2 为键值对的内部 Map 发出 onComplete() 信号。下一个匹配的主键将为累加器 BiFunction 提供前一次的值和新发出的 onNext 信号。这样我们就可以每个键增加一个有效载荷。

5. 累积所有数据，直到内部 PriorityQueue 发出 onComplete() 信号，并使用给定比较器对流逝的时间 *t1* 进行排序。在 onComplete() 之后，所有的数据都会按顺序发出，然后就完成了。

6. 累积所有数据，直到 onComplete() 信号的平均传送时间为默认的首次被接收的时间。

7. 发出下一个信号，并且只计算平均值。

输出

03:14:42.013 [main] INFO elapsed - subscribe: ScanAction  
03:14:42.021 [main] INFO elapsed - onSubscribe: {push}  
03:14:42.022 [main] INFO elapsed - request: 9223372036854775807  
03:14:42.517 [hash-wheel-timer-run-3] INFO elapsed - onNext: 44,1  
03:14:42.518 [hash-wheel-timer-run-3] INFO elapsed - onNext: 48,1  
03:14:42.518 [hash-wheel-timer-run-3] INFO elapsed - onNext: 49,2  
03:14:42.518 [hash-wheel-timer-run-3] INFO elapsed - onNext: 50,3  
03:14:42.518 [hash-wheel-timer-run-3] INFO elapsed - onNext: 51,3  
03:14:42.519 [hash-wheel-timer-run-3] INFO elapsed - complete: SortAction  
03:14:42.520 [hash-wheel-timer-run-3] INFO elapsed - cancel: SortAction

|  |  |  |
| --- | --- | --- |
| **Stream&lt;T&gt; API 或工厂函数** | **输出值** | **作用** |
| count() | Long | 在观测到 `onComplete()` 信号后，产生观测到的 `onNext(T)` 信号的总量。对定时的`窗体（Windows）`很有用，对限制大小的`窗体（Windows）`意义不大。比如说 ` stream.window(5).flatMap(w -> w.count()) ` 的结果是 5，棒棒的。 |
| scan(BiFunction<T,T>) | T |  |
| scan(A, BiFunction<A,T>) | A |  |
| reduce(BiFunction<T,T>) | T |  |
| reduce(A, BiFunction<A,T>) | A |  |
| BiStreams.reduceByKey() |  |  |
| BiStreams.scanByKey() |  |  |
| timestamp() | Tuple2<Long,T> |  |
| elapsed() | Tuple2<Long,T> |  |
| materialize() dematerialize() | Signal<T> | 将上游信号转换为 `Signal<T>` ,并将其视为 `onNext(Signal<T>)` 信号进行处理。直接效果就是：它将接收错误信号和完结信号，因此可以用来高效的处理错误。一旦有错误产生，我们可以将 `dematerialize()` 回调中的 `Signal<T>` 转化到 Reactive Streams，保证服务的运行。 |

分区

分区（Partition）是一种针对并行、并发作业的 Stream。

以响应式编程的方式编写的功能组件，有一个重要的方面就是它的工作可以审慎的切块，交由任意调度器完成。这意味着你可以很轻松的将输入值组合冲一个工作流——在另一个线程执行操作，然后当结果可用时将其交给子序列，完成转化。这是 Reactor 很常见的使用模式。

DispatcherSupplier supplier1 = Environment.newCachedDispatchers(2, "groupByPool");  
DispatcherSupplier supplier2 = Environment.newCachedDispatchers(5, "partitionPool");  
  
Streams  
 .range(1, 10)  
 .groupBy(n -> n % 2 == 0) //1  
 .flatMap(stream -> stream  
 .dispatchOn(supplier1.get()) //2  
 .log("groupBy")  
 )  
 .partition(5) //3  
 .flatMap(stream -> stream  
 .dispatchOn(supplier2.get()) //4  
 .log("partition")  
 )  
 .dispatchOn(Environment.sharedDispatcher()) //5  
 .log("join")  
 .consume();

1. 创建至多两个（奇／偶）数据流，以键值 0 或 1 标记，并将 onNext(T) 信号分发给匹配的数据流。

2. 使用前面的 GroupByAction，为两个正在发送的 Stream 添加一个已经生成好的调度器。通过像这样使用分配于各自调度器的两个分区，数据流得到了有效的扩充。FlatMap 将合并两个分区的返回值，这个过程运行在两个线程之一，但绝不会并行处理。

3. 创建 5 个分区，并将 onNext(T) 信号以循环的方式分发给它们。

4. 使用第二个调度器分配新生成的数据流。返回的序列将被合并。

5. 使用 Environment.sharedDispatcher() 而不是前两个线程池分派数据。 五个线程将在 Dispatcher 线程合并。

提取输出

03:53:42.060 [groupByPool-3] INFO groupBy - onNext: 4  
03:53:42.060 [partitionPool-8] INFO partition - onNext: 9  
03:53:42.061 [groupByPool-3] INFO groupBy - onNext: 6  
03:53:42.061 [partitionPool-8] INFO partition - onNext: 4  
03:53:42.061 [shared-1] INFO join - onNext: 9  
03:53:42.061 [groupByPool-3] INFO groupBy - onNext: 8  
03:53:42.061 [partitionPool-4] INFO partition - onNext: 6  
03:53:42.061 [shared-1] INFO join - onNext: 4  
03:53:42.061 [groupByPool-3] INFO groupBy - onNext: 10  
03:53:42.061 [shared-1] INFO join - onNext: 6  
03:53:42.061 [groupByPool-3] INFO groupBy - complete: DispatcherAction

|  |  |  |
| --- | --- | --- |
| **Stream&lt;T&gt; API** | **输出类型** | **角色** |
| groupBy(Function<T,K>) | GroupedStream<K,T> |  |
| partition(int) | GroupedStream<K,T> |  |
| \*所有的\* window(arguments) | Stream<T> | 分区是依照时间、大小，或为了同外部信号协调而进行切分的，[窗口](${domain}?v=10626&t=44491)其就是是为此而存在。 |
| process(XXXWorkProcessor) | T | 因为 `RingBufferWorkProcessor` 可以将信号分发给每个订阅，如果只是为了扩充，而不是路由，它可以很好的替换 `partition()`。 |

Rx之外的其它 API

除了 Reactive Stream 直接实现的方法之外，还有一些 Stream 方法并没有被涉及，或是没有录入 Reactive 扩展文档之中。

|  |  |  |  |
| --- | --- | --- | --- |
| **Stream&lt;T&gt; API** | **输入类型** | **输出类型** | **角色** |
| after() | T | Void | 只消费 `onComplete()` 和 `onError()` 信号 |
| log(String) | T | T | 使用 SLF4J 和 给定类别记录每个信号。 |
| split | Iterable<T> | T | 将 `Iterable<T>` 阻塞转化成尽可能多的 `onNext(T)`。 |
| sort(int, Comparator<T>) | T | T | 将给定尺寸的数据存入存于内存的 `PriorityQueue`中，使用 ` Comparator<T>`进行排序，并将所有挂起的 `onNext(T)` 信号发出。 |
| combine() | T | O | 自右向左扫描最上层的对象或 `Action`。它将创建一个新的`处理器（Processor）`作为返回结果，处理器使用分发给旧 action 的 `onXXXX` 信号作为输入值，并出处代理当前 action 的`订阅者（Subscriber）`。 示例 ```java Action<Integer, String> processor = stream .filter( i -> i<2 ) .map(Object::toString) .combine(); processor.consume(System.out::println); processor.onNext(1); processor.onNext(3); ``` |
| keepAlive() | T | T | 阻止来自于`订阅者（Subscriber）`的 `Subscription.cancel()` 信号的传播。 |

reactor-总线

数据路由

图片地址：http://projectreactor.io/docs/reference/images/bus-overview.png

图片地址：null

发布订阅模型

发布使用的是 EventBus ，而响应事件则使用 发布订阅模型

Reactor 的 EventBus 可以让你在通知的主键匹配特定条件时创建一个处理事件的消费者(Consumer)。这项任务是在 选择器(Selector) 中被执行的。这跟订阅主题很像,但 Reactor 选择器 的实现可以匹配各种标准，从 Class<?> 类型到 JsonPath 表达式。这种抽象化非常的灵活且强大，它为匹配提供了广泛的可能性。

在创建多个使用相同 Selector 的消费者，也可以创建多个匹配同一主键的 Selector 。这种方式使得聚合和广播更加容易：你只需要将多个消费者订阅到同一主体选择器下即可。

如果你刚从 Reactor 1.1 升级的话，你会发现 Reactor 类消失了。它被重命名为 EventBus 类，这是为了更精准的反应此类在框架中的角色。

使用选择器处理事件

EventBus bus = EventBus.create(Environment.get());   
  
bus.on($("topic"), (Event<String> ev) -> {  
 String s = ev.getData();  
 System.out.printf("Got %s on thread %s%n", s, Thread.currentThread());  
});   
  
bus.notify("topic", Event.wrap("Hello World!"));

1. 使用默认设置创建一个 EventBus 类，从静态的 Environment 中分享 RingBufferDispatcher

2. 注册一个 Customer，当主键匹配 Selector 时通知 EventBus 调用。

3. 使用给定主题发布 Event 到 EventBus。

静态方法 $ 是方便调用 Selectors.object() 的缩写。有的人不喜欢使用缩写方法，例如**对象选择器**的 $() *,正则选择器的 R()，类选择器*\*的 T() 等等。选择器(http://projectreactor.io/docs/api/reactor/bus/selector/Selectors.html#method.summary)也为这些缩写方法提供了长命名的替代方法。这些缩写方法只是为了减少代码噪音，提高代码可阅读性而的设定的。

请求应答模式

EventBus 发布和响应事件使用的是请求应答模式。

常见的情景是，你希望能够从运行在 EventBus 配置好的调度器(Dispatcher)中的任务里获取应答。Reactor 的 EventBus 提供了比简单的发布订阅模型更全面的事件处理模型。除了 Cunsumer，你也可以同样注册一个函数，EventBus 会自动将 Function 的返回值推送给 replyTo 主键中的主题。在这里，推荐使用 .receive() 和 .send() 方法，而不是 .on() 和 .notify() 方法。

请求应答

EventBus bus;  
  
bus.on($("reply.sink"), ev -> {  
 System.out.printf("Got %s on thread %s%n", ev, Thread.currentThread())  
});   
  
bus.receive($("job.sink"), ev -> {  
 return doWork(ev);  
});   
  
bus.send("job.sink", Event.wrap("Hello World!", "reply.sink"));

1. 分配一个处理所有应答的 consumer，不进行任何分析。

2. 分配一个工作在 Dispatcher 线程的 Function,完成工作并返回结果。

3.使用给定的 replyTo 主键在总线中发布 Event。

如果没有一个发布应答的通用主题，你可以将请求和应答的操作绑定到一个单独的对 .sendAndReceive(Object, Event<?>, Consumer<Event<?>>) 方法的调用中。此方法将调用 .send() ，并在函数被调用时在 Dispatcher 线程调用给定的 replyTo 回调函数。

sendAndReceive()

EventBus bus;  
  
bus.receive($("job.sink"), (Event<String> ev) -> {  
 return ev.getData().toUpperCase();  
});   
  
bus.sendAndReceive(  
 "job.sink",  
 Event.wrap("Hello World!"),  
 s -> System.out.printf("Got %s on thread %s%n", s, Thread.currentThread())  
);

1. 分配一个在 Dispatcher 线程完成工作并返回结果的 Function。

2. 在总线中发布一个 Event，并在 Dispatcher中安排给定的 replyTo Consumer，将接收事件的函数的返回值作为输入传递给它。

有时候你希望取消一个任务，停止响应事件通知。注册函数 .on() 和 .receive() 将返回一个 Registration 对象,如果持有该对象的引用，你可以用它取消给定 Selector 的 Consumer 或 Function。

EventBus bus;  
  
Registration reg = bus.on($("topic"),  
 s -> System.out.printf("Got %s on thread %s%n", s, Thread.currentThread()));  
  
bus.notify("topic", Event.wrap("Hello World!"));   
  
// ...some time later...  
reg.cancel();   
  
// ...some time later...  
bus.notify("topic", Event.wrap("Hello World!"));

1. 对给定主题发布一个事件，应当在控制台中打印 Event.toString()。

2. 取消 Registration 对象的注册，组织消息抵达 Consumer。

3. 这个通知不应当有任何结果。

牢记，取消一个 Registration 的注册将对内部注册表进行原子访问。当系统中存在大量流向消费者的时间时，有时在你的 .cancel() 调用完成后 注册表（Registry） 清理缓存并移除 Registration 前，你的 Consumer 或Function依然会接收到一些事件。.cancel() 方法可以被称为："请求尽快的取消"。

在测试类中你能够察觉这一行为特征，测试类中在 .on() 、 .notify() 和 .cancel() 的调用之间没有任何时间延迟。

注册表

使用注册表（Registry）缓存内存中的值。

reactor-网络

异步 TCP、UDP 及 HTTP

异步 TCP、UDP 及 HTTP

谁都不会比光的速度还要快，唯有坏消息例外，它们遵循自己的一套规律。

道格拉斯∙诺尔∙亚当斯

《基本无害》（1992）

先来一些网络方面的 Java 8 代码

import reactor.io.net.NetStreams;  
import reactor.io.net.tcp.TcpServer;  
import reactor.io.net.tcp.TcpClient;  
  
//...  
  
CountDownLatch latch = new CountDownLatch(10);  
  
TcpServer<Buffer, Buffer> server = NetStreams.tcpServer(port);  
TcpClient<Buffer, Buffer> client = NetStreams.tcpClient("localhost", port);  
  
final JsonCodec<Pojo, Pojo> codec = new JsonCodec<Pojo, Pojo>(Pojo.class);  
  
//the client/server are prepared  
server.start( input ->  
  
 //for each connection echo any incoming data  
  
 //return the write confirm publisher from writeWith  
 // >>> close when the write confirm completed  
  
 input.writeWith(  
  
 //read incoming data  
 input  
 .decode(codec) //transform Buffer into Pojo  
 .log("serve")  
 .map(codec) //transform Pojo into Buffer  
 .capacity(5l) //auto-flush every 5 elements  
 )  
).await();  
  
client.start( input -> {  
  
 //read 10 replies and close  
 input  
 .take(10)  
 .decode(codec)  
 .log("receive")  
 .consume( data -> latch.countDown() );  
  
 //write data  
 input.writeWith(  
 Streams.range(1, 10)  
 .map( it -> new Pojo("test" + it) )  
 .log("send")  
 .map(codec)  
 );  
  
 //keep-alive, until 10 data have been read  
 return Streams.never();  
  
}).await();  
  
latch.await(10, TimeUnit.SECONDS);  
  
client.shutdown().await();  
server.shutdown().await();

概述

图片地址：http://projectreactor.io/docs/reference/images/net-overview.png

图片地址：null

所以，为什么你会关心一个进行网络操作的异步运行时？正与章节⎡Microservice with Streams⎦ 一节所示，它更倾向于不阻塞系统应答。从资源方面来说，网络中的非阻塞写操作会比可阻塞的更耗资源，但非阻塞对生产者来说**更灵敏**。对请求流的响应影响系统的方方面面，最终，会有 1 个或 N 个用户等待它们的时机来推送新消息。

图片地址：http://projectreactor.io/docs/reference/images/net-latency.png

图片地址：null

当并发服务使用类似 TCP 或 WebSocket 的长链接时，阻塞的读写操作将变成噩梦。在IO 读写方法锁定线程的应用中，阻塞 Socket 的作用很有限。当然，会因为链接时间过长而超时的网络路由组件除外。

当然，对一个线程池或任一类似**核心处理器(Core Processor)**的异步封装，总有能够减轻读写冲突阻塞的选择。问题是，在非阻塞调度的 **Reactor** 世界中，这类线程并不多，因此，在 4/8/16 异步封装中阻塞是一个不得已的选择。再者，维护大量队列甚至线程的线程池也无需处理这种情况。

**Reactor Net** 的目标是提供一个 **异步IO** 运行时，支持 **Reactor Streams** 抵御由于客户端和服务器对一众协议及驱动的需求所产生的背压。多数驱动并不会实现所有的协议，但至少这么一个，**Netty**，实现了现有的全部协议。当前，Reactor Net 支持  **Netty 4.x** 以及 **ZeroMQ** 或 **jeroMQ0.3.+**，你必须将其中之一明确的添加到应用的类路径中。

**Reactor Net** 拥有下列组件：

\* ReactorChannel 协议及直接实现它的 ChannelStream 和 HttpChannel

\* 远程主机同应用程序之间直接连接的代表

\* 拥有非阻塞IO读写操作

\* 对于读取操作，Reactor 驱动程序会直接暴露 ChannelStream 对象，以便读取操作访问 Stream 的功能性API。

\* ReactorPeer 和 ReactorChannelHandler 描述了通用网络组件（客户端/服务端）协定

\* 提供了 启动(start) 及 关闭(shutdown) 操作

\* 在 启动 时绑定一个 ReactorChannelHandler 方法来监听请求中的 ChannelStream 。

\* 函数 ReactorChannelHandler 接受 ChannelStream 请求并返回一个管理链接关闭操作的 Publisher .

\* ReactorClient 是一个通用客户端协定

*扩展了*  *ReactorPeer*  *并提供了一个* 断线重连\*更友好的启动操作

\* NetStreams 和 Spec 可以用来创建任意客户端或服务端。

\* 类似于 Streams 、 BiStreams 或其它 Reactor Stream 工厂类

\* NetStreams 工厂类会在创建时接受一次 Function<Spec,Spec> 调用，自定义网络组件的配置

*HTTP/WS/UDP/TCP* \* 协议 ReactorPeer 的实现

\* HttpServer 和 HttpClient 将提供路由扩展

\* DatagramServer 将提供广播扩展

\* TcpServer 和 TcpClient 将提供额外的 TCP/IP 上下分信息。

*Netty*  *和*  *ZeroMQ* \* 驱动

**Reactor Net** 实现了一个在 Reactive IPC(https://github.com/reactive-ipc/reactive-ipc-jvm) 倡议下讨论的模型。在今后，我们会继续调整它，并在2016年后确定出具体的部件。现在，您将有机会尝试这些理论，以让我们能够为用户提供更好的下一代标准。

通道

通道处理程序

规范

客户端规范

服务器规范

背压

使用 Reactor 和 Reactor Stream 标准对 TCP 网络端点进行流程控制

最后的最后

结合 Reactor Net、Stream 以及 Core 组件，创造一个独立的云存储服务或（小型）数据服务。

TCP 101

TCP 101

利用 Reactor 对 TCP 的支持来创建高性能的 TCP 客户端和服务器端。

启动和关闭

数据写入

缓冲区刷新策略

消费数据

背压策略

关闭通道

HTTP 101

HTTP 101

利用 Reactor 对 HTTP 的支持来创建高性能的 HTTP 客户端和服务器端

HTTP 路由解析

数据写入

---

---

---

缓冲区刷新策略

消费数据

---

---

背压策略

关闭通道

扩展

Spring 框架支持

在 Spring 的 **ApplicationContext** 中编写 Reactor 应用。

在 Spring 的 **ApplicationContext** 中编写 Reactor 应用。TODO:

Reactor 是基于 Spring Messaging 模块的 TCP 支持(http://docs.spring.io/spring/docs/current/spring-framework-reference/html/websocket.html)的。请确保 Reactor Net 模块 **io.projectreactor:reactor-net::2.0.7.BUILD-SNAPSHOT** 在 Spring 应用的 classpath 中。

Groovy 语言扩展

在 Grails 3.0 发行版本中，Reactor 作为事件支持(http://grails.github.io/grails-doc/3.0.x/guide/single.html#events) 的一部分被核心组件支持。

在 Spring XD 系统中使用 Reactor

Clojure 语言支持

Meltdown 是 Clojure 语言对 Reactor 的接口。Reactor 是一个为 JVM 设计的异步编程、事件传递和数据流处理的工具包。Meltdown 是 Romulan 项目的继承者，Romulan 则是 ClojureWerkz 组织过去的一个已经被舍弃的基于 LMAX Disruptor 的项目。

- README文件

*项目页面*

ClojureWerkz Meltdown(https://github.com/clojurewerkz/meltdown)

简明手册

制作一个简单的文件流

让我们从实现一个简单的 publisher 开始，我们将使用 Reactor API 来简化后面的示例代码。作为一个 Publisher 你不得不顾及很多细节，因为这些细节将在 **Reactive Streams** 中的 TCK 模块中被检测。这样做的目的是让你能够更好的理解 Reactor 在特定条件下能够完成的工作，从而避免重造轮子。

理论上说，在单线程、简单循环阻塞的文件读取消费中，**Reactor Streams** 并不能为你提供什么帮助。如果接收端点阻塞，它将发送多少读取多少，这已经是某种形式上的背压。**Reactor 文件流** 的优势在于，当流与消费者之间有一个或多个边界需要跨越时，可以通过采用队列或环形缓冲器来解耦合。你不妨想象一下：当你在向一个消费者发送数据时，同时进行数据的读取，那当它下次请求数据的时候（在前一次数据发送完成后），被请求的数据**已经保存在内存中**了。换句话说，就是预读取。

创建一个匹配订阅者请求的文件惰性读取发布者

Publisher<String> fileStream = new Publisher<String>() {   
 @Override  
 public void subscribe(final Subscriber<? super String> subscriber) {  
 final File file = new File("settings.gradle");   
   
 try {  
 final BufferedReader is = new BufferedReader(new FileReader(file));   
   
 subscriber.onSubscribe(new Subscription() {  
   
 final AtomicBoolean terminated = new AtomicBoolean(false);  
   
 @Override  
 public void request(long n) {  
 long requestCursor = 0l;  
 try {  
 String line;  
 while ((requestCursor++ < n || n == Long.MAX\_VALUE)   
 &amp;&amp; !terminated.get()) {   
   
 line = is.readLine();  
 if (line != null) {  
 subscriber.onNext(line);  
 } else {  
 if (terminate()) {  
 subscriber.onComplete();   
 }  
 return;  
 }  
 }  
 } catch (IOException e) {  
 if (terminate()) {  
 subscriber.onError(e);   
 }  
 }  
 }  
   
 @Override  
 public void cancel() {  
 terminate();  
 }  
   
 private boolean terminate() {  
 if (terminated.compareAndSet(false, true)) {  
 try {  
 is.close();   
 } catch (Exception t) {  
 subscriber.onError(t);  
 }  
 return true;  
 }  
 return false;  
 }  
 });  
   
 } catch (FileNotFoundException e) {  
 Streams.<String, FileNotFoundException> fail(e)  
 .subscribe(subscriber);   
 }  
 }  
 };  
   
 Streams.wrap(fileStream)  
 .capacity(4L)   
 .consumeOn(   
 Environment.sharedDispatcher(),  
 System.out::println,  
 Throwable::printStackTrace,  
 nothing -> System.out.println("## EOF ##")  
 );

1. 实现一个 Publisher。下一个列子中你将看到依靠核心和数据流，发布者能够多么灵巧。

2. 创建供一个订阅者读取的 File 指针，来展示如何玩这个：这是一个 Cold Stream。

3. 根据传参匹配要读取的行数，若传参为 Long.MAX\_VALUE 则忽略行数限制。

4. 在每次调用 onNext() 之前检查一下数据流是否**已取消**。

5. 调用 **onComplete()**，它将把**订阅**状态标记为*已取消*，并忽略以后所有可能出现的终端信号。

6. 调用 **onError(e)**，它将把**订阅**状态标记为*已取消*，并忽略以后所有可能出现的终端信号。

7. 在订阅者不再关注的时候关闭文件（因为出现错误、读取完成或被取消时）。

8. 当失败时创建一个流，将订阅者传递给 onSubscribe() 并调用 onError(e)。

9. capacity 将会提示下游操作（这里是 consumeOn) 把请求按照 4 字节的大小分块。

10. consumeOn 要在分配器中执行请求，另需要3个额外的参数，以便其它 3 种可能存在的 Consumer 对其信号作出反应。

使用核心的发布者类工厂（2.0.2 后支持）创建一个文件惰性读取发布者，并与 Stream API 组合

`

final String filename = "settings.gradle";

Publisher<String> fileStream = PublisherFactory.create(

(n, sub) -> {   
 String line;  
 final BufferedReader inputStream = sub.context()   
 long requestCursor = 0l;

|  |  |  |
| --- | --- | --- |
| **while ((requestCursor++ < n** |  | **n == Long.MAX\_VALUE) &amp;&amp; !sub.isCancelled()) {** |

try {  
 line = inputStream.readLine();  
 if (line != null) {  
 sub.onNext(line);  
 } else {  
 sub.onComplete();   
 return;  
 }  
 }  
 catch (IOException exc) {  
 sub.onError(exc);  
 }  
 }  
 },  
 sub -> new BufferedReader(new FileReader(filename)),   
 inputStream -> inputStream.close()

);

Streams

.wrap(fileStream)  
 .process(RingBufferProcessor.create())  
 .capacity(4L)  
 .consume(  
 System.out::println,  
 Throwable::printStackTrace,  
 nothing -> System.out.println("## EOF ##")

);

`

1. 实现一个 BIConsumer，以响应每个 Subscriber发出请求，请求的长度为 Long 型 n。任何未检查的意外都将触发终止回调函数并调用 Subscriber.onError(e)。

2. 回调函数中传递的 Subscriber 是一个 SubscriberWithContext 装饰器，用它可以访问在开始时填充好的 context()

3. 根据传参匹配要读取的行数，若传参为 Long.MAX\_VALUE 则忽略行数限制。同时在每次读取前使用 SubscriberWithContext.isCancelled() 检查 Subscriber是否异步取消了请求。

4. 调用 onComplete()，它将把 Subscriber 状态标记为取消，并忽略以后所有可能出现的终端信号。

5. 为之后新 Subscriber 的每次 SubscriberWithContext.context() 请求 创建一个上下文环境。

6. 当拦截到 cancel()、onComplete() 或 onError(e) 信号时，定义一个终止回调函数。

我们可以利用 PublisherFactory 工厂，或 Streams 工厂(http://projectreactor.mydoc.io?v=10626&t=44489)（例如 Streams.createWith()）来完成常见的任务：

• 打开一次 IO 操作。

• 响应请求。

• 更优雅的处理关闭操作。

制作一个快速断路器

在本节练习中，我们将更关注于如何更高效的利用你手中的 **Reactor Stream 模块**。下面是一个典型用例：使用断路器模式(http://martinfowler.com/bliki/CircuitBreaker.html)创建一个自愈型数据管道（也许很快就能够在 **Stream API**中实现，也许哦～）。

本用例中，我们希望在哪怕出错的时候，Stream 依然可以存活。当错误积累到一定程度，我们将希望能够关闭来自主*断路器*(实际上是 Stream)的消费。在一个很短的时间段中，我们将断开断路器，启用一个备用发布者 Stream。任何类型的发布者都可以作为备份使用，它只是用来发布替代信息。关键是在一段时间内阻止对出错 Stream 的新访问，给予它恢复的机会。

快速（简单）断路器测试

final Broadcaster<String> closeCircuit = Broadcaster.create();   
final Stream<String> openCircuit = Streams.just("Alternative Message");   
  
final Action<Publisher<? extends String>, String> circuitSwitcher = Streams.switchOnNext();   
  
final AtomicInteger successes = new AtomicInteger();   
final AtomicInteger failures = new AtomicInteger();  
  
final int maxErrors = 3;  
  
Promise<List<String>> promise =   
 circuitSwitcher   
 .observe(d -> successes.incrementAndGet())   
 .when(Throwable.class, error -> failures.incrementAndGet())  
 .observeStart(s -> {   
  
 System.out.println("failures: " + failures +  
 " successes:" + successes);  
  
 if (failures.compareAndSet(maxErrors, 0)) {  
 circuitSwitcher.onNext(openCircuit);   
 successes.set(0);  
  
 Streams  
 .timer(1)   
 .consume(ignore -> circuitSwitcher.onNext(closeCircuit));  
 }  
 })  
 .retry()   
 .toList();   
  
circuitSwitcher.onNext(closeCircuit);   
  
closeCircuit.onNext("test1");  
closeCircuit.onNext("test2");  
closeCircuit.onNext("test3");  
closeCircuit.onError(new Exception("test4"));  
closeCircuit.onError(new Exception("test5"));  
closeCircuit.onError(new Exception("test6"));  
Thread.sleep(1500);   
closeCircuit.onNext("test7");  
closeCircuit.onNext("test8");  
closeCircuit.onComplete();   
circuitSwitcher.onComplete();  
  
System.out.println(promise.await());  
Assert.assertEquals(promise.get().get(0), "test1");  
Assert.assertEquals(promise.get().get(1), "test2");  
Assert.assertEquals(promise.get().get(2), "test3");  
Assert.assertEquals(promise.get().get(3), "Alternative Message");  
Assert.assertEquals(promise.get().get(4), "test7");  
Assert.assertEquals(promise.get().get(5), "test8");

1. 创建一个主要的**活跃** Broadcaster，用来发送数据。

2. 创建一个简单的回退流，以防意外。

3. 创建一个 SwitchAction，以便 Processor 接受新的数据的 Publisher。

4. 准备共享的成功失败计数器。

5. Stream.toList() 返回一个 Promise 对象，将数据流转换为实际意义的 List 列表

6. circuitSwitcher Processor 代理中的数据消费将视失败数量而定。

7. 对每一次成功的 **onNext(String)** 调用和每一次可抛出（Throwable）异常产生分别计数。

8. 监视 onSubscribe(Subscription) 的调用，它在每次成功启动数据流后将被调用。

9. 如果错误数达到了 *maxErrors*，将 circuitSwitcher 对象当前的数据源切换到备用的数据源，这也将触发断路器。

10. 一秒钟后，将主数据流传递给 circuitSwitcher 对象，恢复其对主数据流的数据消费。

11. 在任何异常（即取消和重订阅）后都尝试重新运行。这也是我们使用 observeStart() 的原因——任何错误都将触发它。

12. 使用主 Stream 启动 circuitSwitcher 对象。

13. 人为延迟，留出断路器关闭的时间。

14. 同时在主数据流和 circuitSwitcher 对象上调用 onComplete()（否则它们会因为 onComplete() 信号的缺失而在后台挂起。

制作一个高效的数据管道

制作一个非阻塞型微服务

制作一个类 CQRS 应用

其他示例程序

快速入门(https://github.com/reactor/reactor-quickstart) 一个展示了Rector 基本组件数种使用方法的应用。

示例代码(https://github.com/reactor/reactor-samples) 一个通过 JUnit 测试演示 Reactor 功能的简单示例应用。

如果你深入到这一步的话，这里有一个非阻塞数据流的示例，让你识别 websocket 中的作弊码。告诉我它的奖励... 不要忘记在 **reactor-net** 的 classpath 中添加你的 **Netty** 地址。

final Processor<Integer, Integer> keyboardStream = RingBufferProcessor.create();  
  
NetStreams.<String, String>httpServer(spec ->  
 spec  
 .codec(StandardCodecs.STRING\_CODEC)  
 .listen(3000)  
 )  
 .ws("/", channel -> {  
 System.out.println("Connected a websocket client: " + channel.remoteAddress());  
  
 return Streams  
 .wrap(keyboardStream)  
 .skipWhile(key -> KeyEvent.VK\_UP != key)  
 .buffer(10, 1)   
 .map(keys -> keys.size() == 10 &amp;&amp;  
 keys.get(0) == KeyEvent.VK\_UP &amp;&amp;  
 keys.get(1) == KeyEvent.VK\_UP &amp;&amp;  
 keys.get(2) == KeyEvent.VK\_DOWN &amp;&amp;  
 keys.get(3) == KeyEvent.VK\_DOWN &amp;&amp;  
 keys.get(4) == KeyEvent.VK\_LEFT &amp;&amp;  
 keys.get(5) == KeyEvent.VK\_RIGHT &amp;&amp;  
 keys.get(6) == KeyEvent.VK\_LEFT &amp;&amp;  
 keys.get(7) == KeyEvent.VK\_RIGHT &amp;&amp;  
 keys.get(8) == KeyEvent.VK\_B &amp;&amp;  
 keys.get(9) == KeyEvent.VK\_A  
 )  
 .map(isKonami -> isKonami ? "Konami!" : "Nah")  
 .nest()  
 .flatMap(konamis ->  
 channel.writeWith(konamis)  
 );  
 })  
 .start()  
 .await();  
  
keyboardStream.onNext(KeyEvent.VK\_RIGHT);  
keyboardStream.onNext(KeyEvent.VK\_UP);  
keyboardStream.onNext(KeyEvent.VK\_UP);  
keyboardStream.onNext(KeyEvent.VK\_DOWN);  
keyboardStream.onNext(KeyEvent.VK\_DOWN);  
keyboardStream.onNext(KeyEvent.VK\_LEFT);  
keyboardStream.onNext(KeyEvent.VK\_RIGHT);  
keyboardStream.onNext(KeyEvent.VK\_LEFT);  
keyboardStream.onNext(KeyEvent.VK\_RIGHT);  
keyboardStream.onNext(KeyEvent.VK\_B);  
keyboardStream.onNext(KeyEvent.VK\_A);  
keyboardStream.onNext(KeyEvent.VK\_C);  
keyboardStream.onComplete();  
  
System.out.println(konamis.await());

1. 注意 buffer(size, skip) 的使用，我们希望评估至少10按键之间的全部组合，10 乘 10。就是说我们要创建一个包含所有按键、且每个按键都有10个按键的列表。

---

1. (#\_footnoteref\_1) 除非你希望使用现阶段几乎独立的核心处理器。我们计划在下一步调整调度器和核心处理器，使它们相互协作。

2. (#\_footnoteref\_2) 有些人可能会挑战这种过度简化的愿景，但还是允许我们在这里更务实些吧 ：）

3. (#\_footnoteref\_3) 包括 Akka Streams(http://akka.io)，Ratpack(http://ratpack.io) 和 RxJava(https://github.com/ReactiveX/RxJava)