从 Reactive 到 WebFlux

理解 Reactive

关于 Reactive 的一些讲法

- Reactive 是异步非阻塞编程
- Reactive 能够提升程序性能
- Reactive 解决传统编程模型遇到的困境

Reactive 框架

- Java 9 Flow API
- RxJava
- Reactor

传统编程模型中的某些困境

Reactor 认为阻塞可能是浪费的

> ### 3.1. Blocking Can Be Wasteful
>
> Modern applications can reach huge numbers of concurrent users, and, even though the capabilities of modern hardware have continued to improve, performance of modern software is still a key concern.
>
> There are broadly two ways one can improve a program's performance:
>
> 1. **parallelize**: use more threads and more hardware resources.
> 2. **seek more efficiency** in how current resources are used.
>
> Usually, Java developers write programs using blocking code. This practice is fine until there is a performance bottleneck, at which point the time comes to introduce additional threads, running similar blocking code. But this scaling in resource utilization can quickly introduce contention and concurrency problems.

> Worse still, blocking wastes resources.
>
> So the parallelization approach is not a silver bullet.

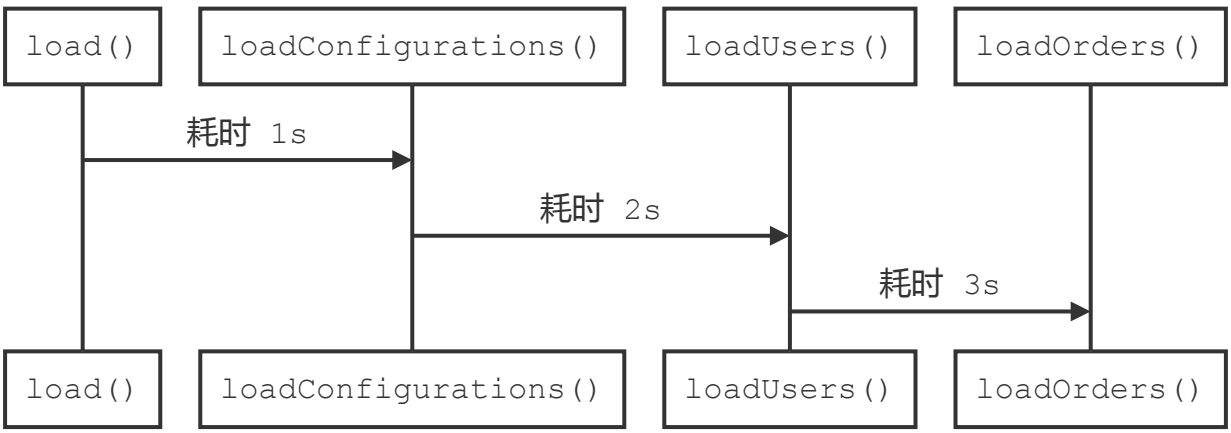
观点归纳

- 阻塞导致性能瓶颈和浪费资源
- 增加线程可能会引起资源竞争和并发问题
- 并行的方式不是银弹（不能解决所有问题）

理解阻塞的弊端

阻塞场景 - 数据顺序加载

加载流程如下图所示：



- Java 实现

```
public class DataLoader {

    public final void load() {
        long startTime = System.currentTimeMillis(); // 开始时间
        doLoad(); // 具体执行
        long costTime = System.currentTimeMillis() - startTime; // 消耗时间
        System.out.println("load() 总耗时：" + costTime + " 毫秒");
    }

    protected void doLoad() { // 串行计算
        loadConfigurations();    //  耗时 1s
        loadUsers();                  //  耗时 2s
        loadOrders();                // 耗时 3s
    } // 总耗时 1s + 2s  + 3s  = 6s
```

```
    protected final void loadConfigurations() {
        loadMock("loadConfigurations()", 1);
    }

    protected final void loadUsers() {
        loadMock("loadUsers()", 2);
    }

    protected final void loadOrders() {
        loadMock("loadOrders()", 3);
    }

    private void loadMock(String source, int seconds) {
        try {
            long startTime = System.currentTimeMillis();
            long milliseconds = TimeUnit.SECONDS.toMillis(seconds);
            Thread.sleep(milliseconds);
            long costTime = System.currentTimeMillis() - startTime;
            System.out.printf("[线程 : %s] %s 耗时 :  %d 毫秒\n",
                    Thread.currentThread().getName(), source, costTime);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        new DataLoader().load();
    }

}
```

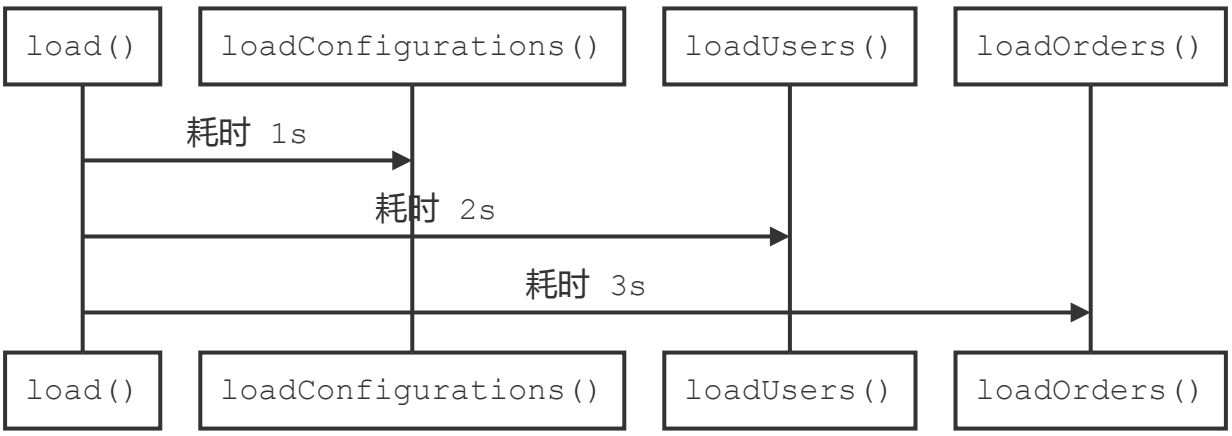
结论

> 由于加载过程串行执行的关系，导致消耗实现线性累加。Blocking 模式即串行执行 。

理解并行的复杂

并行场景 - 并行数据加载

- 图示



- Java 代码

```
public class ParallelDataLoader extends DataLoader {

    protected void doLoad() {  // 并行计算
        ExecutorService executorService = Executors.newFixedThreadPool(3); // 创建线程池
        CompletionService completionService = new
ExecutorCompletionService(executorService);
        completionService.submit(super::loadConfigurations, null);      //  耗时 >= 1s
        completionService.submit(super::loadUsers, null);               //  耗时 >= 2s
        completionService.submit(super::loadOrders, null);              //  耗时 >= 3s

        int count = 0;
        while (count < 3) { // 等待三个任务完成
            if (completionService.poll() != null) {
                count++;
            }
        }
        executorService.shutdown();
    }  // 总耗时 max(1s, 2s, 3s)  >= 3s

    public static void main(String[] args) {
        new ParallelDataLoader().load();
    }

}
```

结论

> 明显地，程序改造为并行加载后，性能和资源利用率得到提升，消耗时间取最大者。

延伸思考

1. 如果阻塞导致性能瓶颈和资源浪费的话，Reactive 也能解决这个问题？
2. 为什么不直接使用 `Future#get()` 方法强制所有任务执行完毕，然后再统计总耗时？
3. 由于以上三个方法之间没有数据依赖关系，所以执行方式由串行调整为并行后，能够达到性能提升的效果。如果方法之间存在依赖关系时，那么提升效果是否还会如此明显，并且如何确保它们的执行顺序？

Reactor 认为异步不一定能够救赎

> ## 3.2. Asynchronicity to the Rescue?
>
> The second approach (mentioned earlier), seeking more efficiency, can be a solution to the resource wasting problem. By writing *asynchronous*, *non-blocking* code, you let the execution switch to another active task **using the same underlying resources** and later come back to the current process when the asynchronous processing has finished.
>
> Java offers two models of asynchronous programming:
>
> - **Callbacks**: Asynchronous methods do not have a return value but take an extra `callback` parameter (a lambda or anonymous class) that gets called when the result is available. A well known example is Swing's `EventListener` hierarchy.
> - **Futures**: Asynchronous methods return a `Future<T>` **immediately**. The asynchronous process computes a `T` value, but the `Future` object wraps access to it. The value is not immediately available, and the object can be polled until the value is available. For instance, `ExecutorService` running `Callable<T>` tasks use `Future` objects.
>
> Are these techniques good enough? Not for every use case, and both approaches have limitations.
>
> Callbacks are hard to compose together, quickly leading to code that is difficult to read and maintain (known as "Callback Hell").
>
> Futures are a bit better than callbacks, but they still do not do well at composition, despite the improvements brought in Java 8 by `CompletableFuture`.

观点归纳

- Callbacks 是解决非阻塞的方案，然而他们之间很难组合，并且快速地将代码引导至 "Callback Hell" 的不归路
- Futures 相对于 Callbacks 好一点，不过还是无法组合，不过 `CompletableFuture` 能够提升这方面的不足

理解 "Callback Hell"

- Java GUI 示例

```
public class JavaGUI {

    public static void main(String[] args) {
        JFrame jFrame = new JFrame("GUI 示例");
        jFrame.setBounds(500, 300, 400, 300);
        LayoutManager layoutManager = new BorderLayout(400, 300);
        jFrame.setLayout(layoutManager);
        jFrame.addMouseListener(new MouseAdapter() { // callback 1
            @Override
            public void mouseClicked(MouseEvent e) {
                System.out.printf("[线程 : %s] 鼠标点击，坐标(X : %d, Y : %d)\n",
                        currentThreadName(), e.getX(), e.getY());
            }
        });
        jFrame.addWindowListener(new WindowAdapter() {  // callback 2
            @Override
            public void windowClosing(WindowEvent e) {
                System.out.printf("[线程 : %s] 清除 jFrame... \n", currentThreadName());
                jFrame.dispose(); // 清除 jFrame
            }

            @Override
            public void windowClosed(WindowEvent e) {
                System.out.printf("[线程 : %s] 退出程序... \n", currentThreadName());
                System.exit(0); // 退出程序
            }
        });
        System.out.println("当前线程：" + currentThreadName());
        jFrame.setVisible(true);
    }

    private static String currentThreadName() { // 当前线程名称
        return Thread.currentThread().getName();
    }
}
```

结论

> Java GUI 以及事件/监听模式基本采用匿名内置类实现，即回调实现。从本例可以得出，鼠标的点击确实没有被其他线程给阻塞。不过当监听的维度增多时，Callback 实现也随之增多。同时，事件/监听者模式的并发模型可为同步或异步。

> 回顾
>
> - Spring 事件/监听器（同步/异步）：
>   - 事件：`ApplicationEvent`
>   - 事件监听器：`ApplicationListener`
>   - 事件广播器：`ApplicationEventMulticaster`
>   - 事件发布器：`ApplicationEventPublisher`
> - Servlet 事件/监听器
>   - 同步
>   - 事件：`ServletContextEvent`
>   - 事件监听器：`ServletContextListener`
>   - 异步
>   - 事件：`AsyncEvent`
>   - 事件监听器：`AsyncListener`

理解 `Future` 阻塞问题

如果 `DataLoader` 的 `loadOrders()` 方法依赖于 `loadUsers()` 的结果，而 `loadUsers()` 又依赖于 `loadConfigurations()` ，调整实现：

```
public class FutureBlockingDataLoader extends DataLoader {

    protected void doLoad() {
        ExecutorService executorService = Executors.newFixedThreadPool(3); // 创建线程池
        runCompletely(executorService.submit(super::loadConfigurations));
        runCompletely(executorService.submit(super::loadUsers));
        runCompletely(executorService.submit(super::loadOrders));
        executorService.shutdown();
    }

    private void runCompletely(Future<?> future) {
        try {
            future.get();
        } catch (Exception e) {
        }
    }

    public static void main(String[] args) {
        new FutureBlockingDataLoader().load();
    }
```

```
}
```

结论

> `Future#get()` 方法不得不等待任务执行完成，换言之，如果多个任务提交后，返回的多个 Future 逐一调用 `get()` 方法时，将会依次 blocking，任务的执行从并行变为串行。这也是之前 ""延伸思考"问答 2 的答案：
>
> > 2.为什么不直接使用 `Future#get()` 方法强制所有任务执行完毕，然后再统计总耗时？

理解 `Future` 链式问题

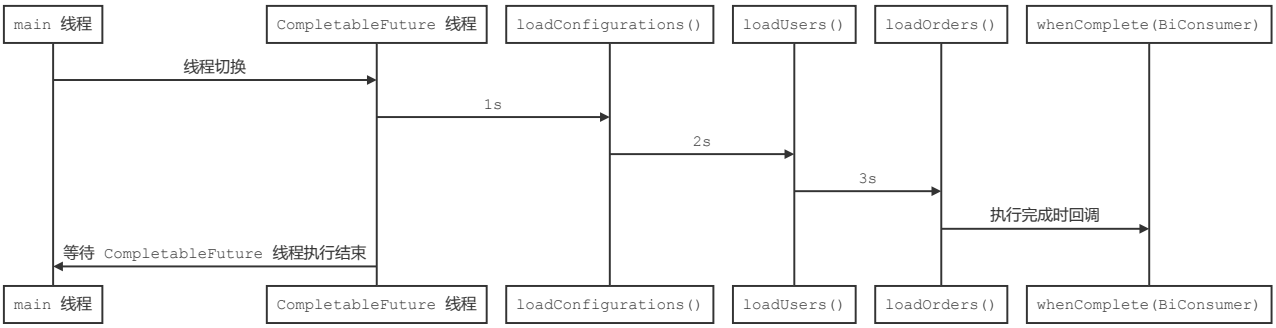
由于 `Future` 无法实现异步执行结果链式处理，尽管 `FutureBlockingDataLoader` 能够解决方法数据依赖以及顺序执行的问题，不过它将并行执行带回了阻塞（串行）执行。所以，它不是一个理想实现。不过 `CompletableFuture` 可以帮助提升 `Future` 的限制：

```
public class ChainDataLoader extends DataLoader {

    protected void doLoad() {
        CompletableFuture
                .runAsync(super::loadConfigurations)
                .thenRun(super::loadUsers)
                .thenRun(super::loadOrders)
                .whenComplete((result, throwable) -> { // 完成时回调
                    System.out.println("加载完成");
                })
                .join(); // 等待完成
    }

    public static void main(String[] args) {
        new ChainDataLoader().load();
    }
}
```

- 图解



结论

1. 如果阻塞导致性能瓶颈和资源浪费的话，Reactive 也能解决这个问题？
2. `CompletableFuture` 属于异步操作，如果强制等待结束的话，又回到了阻塞编程的方式，那么 Reactive 也会面临同样的问题吗？
3. `CompletableFuture` 让我们理解到非阻塞不一定提升性能，那么 Reactive 也会这样吗？

Reactive Streams JVM 认为异步系统和资源消费需要特殊处理

> Handling streams of data—especially “live” data whose volume is not predetermined—requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be carefully controlled such that a fast data source does not overwhelm the stream destination. Asynchrony is needed in order to enable the parallel use of computing resources, on collaborating network hosts or multiple CPU cores within a single machine.

观点归纳：

- 流式数据容量难以预判
- 异步编程复杂
- 数据源和消费端之间资源消费难以平衡

Reactive 是要解决以上所有问题吗？

思考

- Reactive 到底是什么？
- Reactive 的使用场景在哪里？
- Reactive 存在怎样限制/不足？

Reactive Programming 定义

The Reactive Manifesto

> Reactive Systems are: Responsive, Resilient, Elastic and Message Driven.
>
> > https://www.reactivemanifesto.org/

关键字：

- 响应的（Responsive）
- 适应性强的（Resilient）
- 弹性的（Elastic）
- 消息驱动的（Message Driven）

侧重点：

- 面向 Reactive 系统
- Reactive 系统原则

维基百科

> Reactive programming is a declarative programming paradigm concerned with **data streams** and the **propagation of change**. With this paradigm it is possible to express static (e.g. arrays) or dynamic (e.g. event emitters) data streams with ease, and also communicate that an inferred dependency within the associated execution model exists, which facilitates the automatic propagation of the changed data flow.
>
> > https://en.wikipedia.org/wiki/Reactive_programming

关键字：

- 数据流（data streams ）
- 传播变化（ propagation of change）

侧重点：

- 数据结构
  - 数组（arrays）
  - 事件发射器（event emitters）
- 数据变化

技术连接：

- 数据流：Java 8 `Stream`
- 传播变化：Java `Observable` / `Observer`
- 事件：Java `EventObject` / `EventListener`

Spring Framework

> The term "reactive" refers to programming models that are built around **reacting to change** — network component reacting to I/O events, UI controller reacting to mouse events, etc. In that sense **non-blocking** is reactive because instead of being blocked we are now in the mode of reacting to notifications as operations complete or data becomes available.
>
> > https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-why-reactive

关键字：

- 变化响应（reacting to change ）
- 非阻塞（non-blocking ）

侧重点：

- 响应通知
  - 操作完成（operations complete ）
  - 数据可用（data becomes available ）

技术连接：

- 非阻塞：Servlet 3.1 `ReadListener` / `WriteListener`
- 响应通知：Servlet 3.0 `AsyncListener`

ReactiveX

> ReactiveX extends the observer pattern to support sequences of data and/or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.
>
> > http://reactivex.io/intro.html

关键字：

- 观察者模式（Observer pattern ）
- 数据/事件序列（Sequences of data and/or events）
- 序列操作符（Opeators）
- 屏蔽并发细节（abstracting away...）

侧重点：

- 设计模式
- 数据结构
- 数据操作
- 并发模型

技术连接：

- 观察者模式：Java `Observable` / `Observer`
- 数据/事件序列：Java 8 `Stream`
- 数据操作：Java 8 `Stream`
- 屏蔽并发细节（abstracting away...）：`Exectuor` 、`Future`、`Runnable`

Reactor

> The reactive programming paradigm is often presented in object-oriented languages as an extension of the Observer design pattern. One can also compare the main reactive streams pattern with the familiar Iterator design pattern, as there is a duality to the Iterable-Iterator pair in all of these libraries. One major difference is that, while an Iterator is pull-based, reactive streams are push-based.
>
> > http://projectreactor.io/docs/core/release/reference/#intro-reactive

关键字：

- 观察者模式（Observer pattern ）
- 响应流模式（Reactive streams pattern ）
- 迭代器模式（Iterator pattern）
- 拉模式（pull-based）
- 推模式（push-based）

侧重点：

- 设计模式
- 数据获取方式

技术连接：

- 观察者模式：Java `Observable` / `Observer`
- 响应流模式：Java 8 `Stream`
- 迭代器模式：Java `Iterator`

@andrestaltz

> Reactive programming is programming with **asynchronous data streams**.
>
> In a way, **this isn't anything new**. Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive is that **idea on steroids**. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc.
>
> > "What is Reactive Programming?"

关键字：

- 异步（asynchronous ）
- 数据流（data streams ）
- 并非新鲜事物（not anything new ）
- 过于理想化（idea on steroids ）

侧重点：

- 并发模型
- 数据结构
- 技术本质

技术连接：

- 异步：Java `Future`
- 数据流：Java 8 `Stream`

Reactive Programming 特性

编程模型（Programming Models ）

- 响应式编程
- 函数式编程

> 参考资源：WebFlux 编程模型

对立模型 - Imperative programming

> Imperative programming is a programming paradigm that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates.
>
> > https://en.wikipedia.org/wiki/Imperative_programming

小结

Reactive Programming：同步或异步非阻塞执行，数据传播被动通知

Imperative programming：同步阻塞执行，数据主动获取

设计模式（Design Patterns）

- 扩展模式：观察者（Observer）
  - 推模式（push-based）
- 混合模式：反应堆（Reactor）、Proactor
- 对立模式：迭代器（Iterator）
  - 拉模式（pull-based）

模式对比

An Observable(RxJava) is the asynchronous/push "dual" to the synchronous/pull Iterable

event	Iterable (pull)	Observable (push)
data	`T next()`	`onNext(T)`
discover error	throws `Exception`	`onError(Exception)`
complete	`!hasNext()`	`onCompleted()`

小结

Reactive Programming 作为观察者模式（Observer）的延伸，在处理流式数据的过程中，并非使用传统的命令编程方式（ Imperative programming ）同步拉取数据，如迭代器模式（Iterator），而是采用同步或异步非阻塞地推拉相结合的方式，响应数据传播时的变化。

数据结构（Data Structure）

- 流式（Streams）
- 序列（Sequences ）
- 事件（Events）

小结

> A stream is a sequence of **ongoing events ordered in time**.
>
> > "What is Reactive Programming?"

并发模式（Concurrency Model）

- 非阻塞（Non-Blocking）
  - 同步（Synchronous ）
  - 异步（Asynchronous ）

小结

屏蔽并发编程细节，如线程、同步、线程安全以及并发数据结构。

Reactive Programming 使用场景

Reactive Streams JVM

> The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary.
>
> > https://github.com/reactive-streams/reactive-streams-jvm

主要目的：

- 管理流式数据交换（ govern the exchange of stream data ）
- 异步边界（ asynchronous boundary ）

Spring Framework

> Reactive and non-blocking generally do not make applications run faster. They can, in some cases, for example if using the `WebClient` to execute remote calls in parallel. On the whole it requires more work to do things the non-blocking way and that can increase slightly the required processing time.
>
> The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load because they scale in a more predictable way.

主要目的：

- 通常并非让应用运行更快速（ generally do not make applications run faster ）
- 利用较少的资源提升伸缩性（ scale with a small, fixed number of threads and less memory ）

ReactiveX

> The ReactiveX Observable model allows you to treat streams of asynchronous events with the same sort of simple, composable operations that you use for collections of data items like arrays. It frees you from tangled webs of callbacks, and thereby makes your code more readable and less prone to bugs.

主要目的：

- 更好可读性（ more readable ）
- 减少 bugs（ less prone to bugs ）

核心技术：

- 异步（ asynchronous ）
- 同顺序（ same sort ）
- 组合操作（ composable operations ）

Java 原生技术限制：

- Stream 存在组合限制

Reactor

> Composability and readability
>
> Data as a flow manipulated with a rich vocabulary of operators
>
> Nothing happens until you subscribe
>
> Backpressure or the ability for the consumer to signal the producer that the rate of emission is too high
>
> High level but high value abstraction that is concurrency-agnostic

主要目的：

- 结构性和可读性（Composability and readability）
- 高层次并发抽象（High level abstraction）

核心技术：

- 丰富的数据操作符（ rich vocabulary of operators）
- 背压（Backpressure）
- 订阅式数据消费（Nothing happens until you subscribe）

Java 原生技术限制：

- Stream 有限操作符
- Stream 不支持背压
- Stream 不支持订阅

总结 Reactive Programming

Reactive Programming 作为观察者模式（Observer）的延伸，不同于传统的命令编程方式（ Imperative programming）同步拉取数据的方式，如迭代器模式（Iterator）。而是采用数据发布者同步或异步地推送到数据流（Data Streams）的方案。当该数据流（Data Steams）订阅者监听到传播变化时，立即作出响应动作。在实现层面上，Reactive Programming 可结合函数式编程简化面向对象语言语法的臃肿性，屏蔽并发实现的复杂细节，提供数据流的有序操作，从而达到提升代码的可读性，以及减少 Bugs 出现的目的。同时，Reactive Programming 结合背压（Backpressure）的技术解决发布端生成数据的速率高于订阅端消费的问题。

Reactive Streams 规范

Reactive Streams is a standard and specification for Stream-oriented libraries for the JVM that

- process a potentially unbounded number of elements
- in sequence,
- asynchronously passing elements between components,
- with mandatory non-blocking backpressure.

API 组件

Publisher

数据发布者，数据上游

接口

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

Subscriber

数据订阅者，数据上游

接口

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

信号事件

- onSubscribe：当下游订阅时
- onNext：当下游接收数据时
- onComplete：当数据流（Data Streams）执行完成时
- onError： 当数据流（Data Streams）执行错误时

Subscription

订阅信号控制

接口

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

信号操作

- request：请求上游元素的数量
- cancel：请求停止发送数据并且清除资源

Processor

消息发布者和订阅者综合体

接口

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

背压（Backpressure）

维基百科

> The term is also used analogously in the field of information technology to describe the build-up of data behind an I/O switch if the buffers are full and incapable of receiving any more data; the transmitting device halts the sending of data packets until the buffers have been emptied and are once more capable of storing information. It also refers to an algorithm for routing data according to congestion gradients (see backpressure routing).

关键字：

- I/O 切换（I/O switch ）
- 缓冲填满（the buffers are full ）
- 数据无法接受（incapable of receiving any more data）
- 传输设备（transmitting device ）
- 停止发送数据包（halts the sending of data packets ）

Reactive Streams JVM

> Backpressure is an integral part of this model in order to allow the queues which mediate between threads to be bounded.
>
> Since back-pressure is mandatory the use of unbounded buffers can be avoided. In general, the only time when a queue might grow without bounds is when the publisher side maintains a higher rate than the subscriber for an extended period of time, but this scenario is handled by backpressure instead.
>
> > https://github.com/reactive-streams/reactive-streams-jvm

关键字：

- 线程和边界间调停（mediate between threads to be bounded）
- 发布者维持速率高于订阅者（publisher side maintains a higher rate than the subscriber）
- 背压处理（handled by backpressure）

Reactor

> Propagating signals upstream is also used to implement **backpressure**, which we described in the assembly line analogy as a feedback signal sent up the line when a workstation processes more slowly than an upstream workstation.
>
> The real mechanism defined by the Reactive Streams specification is pretty close to the analogy: a subscriber can work in *unbounded* mode and let the source push all the data at its fastest achievable rate or it can use the `request` mechanism to signal the source that it is ready to process at most `n` elements.

关键字：

- Propagating signals upstream（传播上游信号）
- 无边界模式（*unbounded* mode）
- 处理最大元素数量（process at most `n` elements）

总结背压

假设下游Subscriber工作在无边界大小的数据流水线时，当上游Publisher提供数据的速率快于下游Subscriber的消费数据速率时，下游Subscriber将通过传播信号（request）到上游Publisher，请求限制数据的数量（ Demand ）或通知上游停止数据生产。

Reactor 框架运用

核心 API

`Mono`

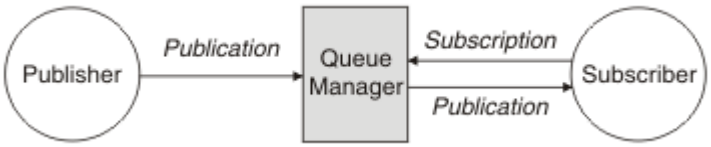
定义：0-1 的非阻塞结果

实现：Reactive Streams JVM API `Publisher`

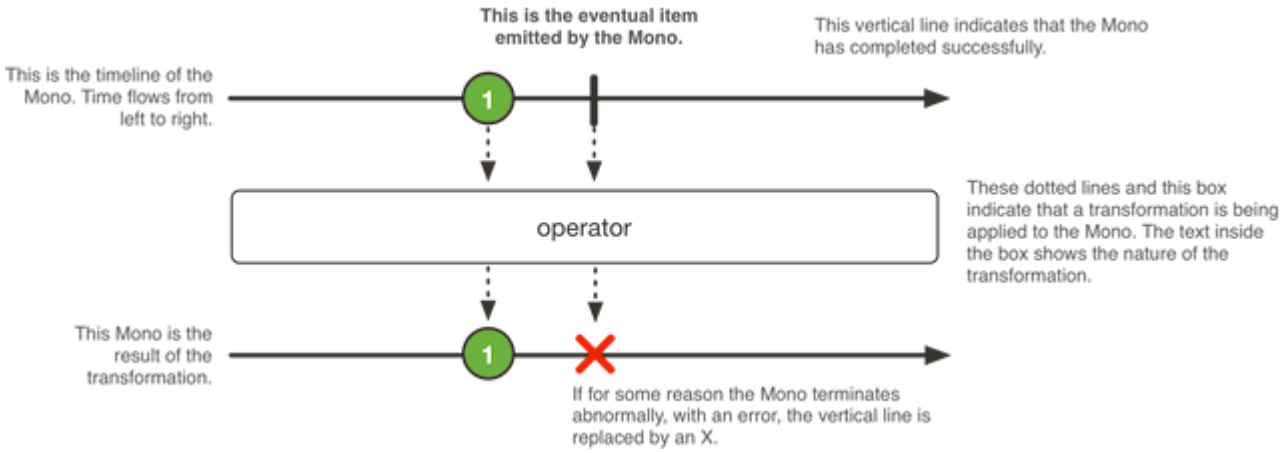
类比：非阻塞 `Optional`

类似模式

点对点模式



图解



`Flux`

定义：0-N 的非阻塞序列

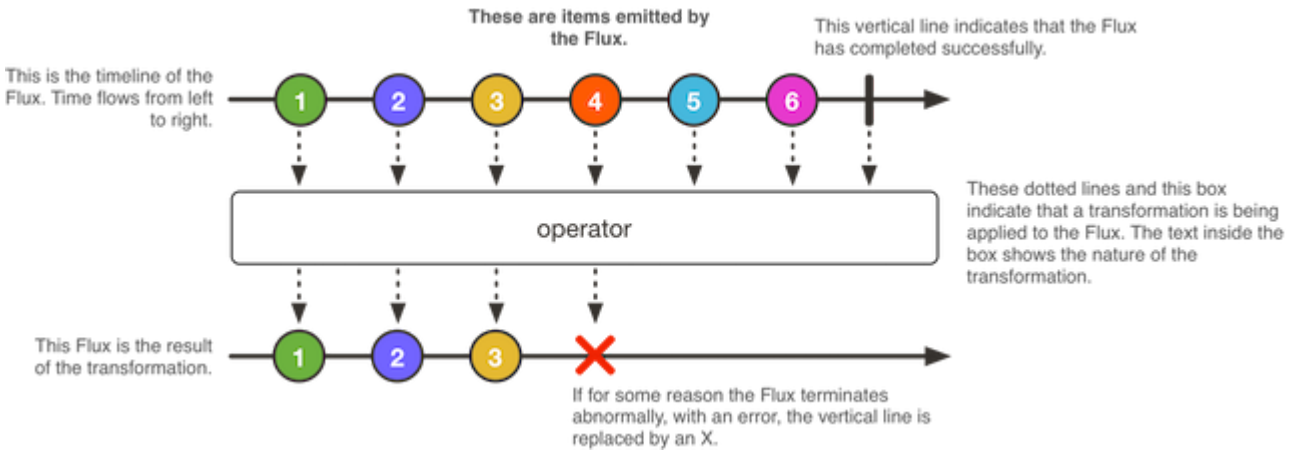
实现：Reactive Streams JVM API `Publisher`

类比：非阻塞 `Stream`

类似模式

发布者/订阅者模式

图解



`Scheduler`

定义：Reactor 调度线程池

- 当前线程：`Schedulers.immediate()`
  - 等价关系：Thread.currentThread()
- 单复用线程：`Schedulers.single()`
  - 内部名称："single"
  - 线程名称："single"
  - 线程数量：单个
  - 线程idel时间：Long Live
  - 底层实现：ScheduledThreadPoolExecutor (core 1)
- 弹性线程池：`Schedulers.elastic()`
  - 内部名称："elastic"
  - 线程名称："elastic-evictor-{num}"
  - 线程数量：无限制（unbounded）
  - 线程idel时间：60 秒
  - 底层实现：ScheduledThreadPoolExecutor
- 并行线程池：`Schedulers.parallel()`
  - 内部名称："parallel"
  - 线程名称："parallel-{num}"
  - 线程数量：处理器数量
  - 线程idel时间：60 秒
  - 底层实现：ScheduledThreadPoolExecutor

实战

Maven 依赖

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
</dependency>
```

同步处理

`generate` 方法

`generate` 方法生成同步并且一对一提交 `Flux` ，sink 实现 `SynchronousSink`

```
Flux<String> flux = Flux.generate(
                () -> 0, // 数据源
                (value, sink) -> { // value 为当前执行的值，sink 是 单信号 Subscriber 的抽象
                    sink.next("value : " + value);
                    if (value == 10) sink.complete();
                    return value + 1;
                });
flux.subscribe(Utils::println);
```

`create` 方法

`create` 方法生成

`handle` 方法

```
Flux.range(0, 10).handle((item, sink) -> {
  if (item % 2 == 0) {
    sink.next("Even : " + item);
  }
}).subscribe(Utils::println);
```