Reactor

> Project Reactor is a fully non-blocking foundation with back-pressure support included. It's the foundation of the reactive stack in the Spring ecosystem and is featured in projects such as Spring WebFlux, Spring Data, and Spring Cloud Gateway.

1、关于

- Reactor 是 完全非阻塞 并提供高效管理操作
- 直接与JavaAPI交互，如 function api，CompletableFuture，Stream，Duration
- 定义 [0|1|N] 序列
  - 提供两种响应式数据；Flux[N]，Mono[0|1]
- 非阻塞IO
  - 完美适配微服务架构。为HTTP、TCP、UDP、提供"背压"网络引擎
- 高效的消息传递
  - Reactor 的 操作与调度功能能适配高吞吐率系统
  - 每秒1亿的消息速率，却能占用更少的内存和线程资源
- 万物皆可响应式
  - 无需手写 Reactive Streams ， Reactor提供快捷工具和丰富的 函数式响应式流 API，比如
    - SpringBoot与Webflux
    - CloudFoundry Java Client
    - RSocket与R2DBC
- Reactor核心技术栈
  - Reactor Core： 基于Java8响应式流核心功能
  - Reactor Test：测试工具
  - Reactor Extra： Flux额外扩展
  - Reactor Netty：使用Netty的HTTP, TCP, UDP Clients/Servers
  - Reactor Adapter： 适配其他响应式
  - Reactor Kafka： 响应式Kafka操作
  - Reactor RabbitMQ：响应式MQ操作
  - Reactor Pool：响应式池
  - BlockHound：java代理，从非阻塞线程中探查阻塞式调用
  - Reactor Core .NET
  - Reactor Core JS

2、入门

1、Reactor介绍

> Reactor is a fully non-blocking reactive programming foundation for the JVM, with efficient demand management (in the form of managing "backpressure"). It integrates directly with the Java 8 functional APIs, notably `CompletableFuture`, `Stream`, and `Duration`. It offers composable asynchronous sequence APIs — `Flux` (for [N] elements) and `Mono` (for [0|1] elements) — and extensively implements the Reactive Streams specification.

> Reactor also supports non-blocking inter-process communication with the `reactor-netty` project. Suited for Microservices Architecture, Reactor Netty offers backpressure-ready network engines for HTTP (including Websockets), TCP, and UDP. Reactive encoding and decoding are fully supported.

2、环境

Reactor Core runs on `Java 8` and above.

3、获取Reactor

1、maven

> maven依赖导入

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>2020.0.20</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

> 引入模块

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

2、gradle

```
plugins {
    id "io.spring.dependency-management" version "1.0.7.RELEASE"
}
```

```
dependencyManagement {
     imports {
          mavenBom "io.projectreactor:reactor-bom:2020.0.20"
     }
}
```

> since gradle5.0

```
dependencies {
     implementation platform('io.projectreactor:reactor-bom:2020.0.20')
     implementation 'io.projectreactor:reactor-core'
}
```

3、响应式编程

Reactor 是响应式编程规范的一种实现，响应式编程定义如下：

> Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. This means that it becomes possible to express static (e.g. arrays) or dynamic (e.g. event emitters) data streams with ease via the employed programming language(s).
>
> 响应式编程式一种异步的编程模型，关注数据流以及变化传播。这意味着可以通过编程语言轻松地表达静态（例如数组）或动态（例如事件发射器）数据流。

— https://en.wikipedia.org/wiki/Reactive_programming

作为响应式编程第一步，Microsoft 在 .NET 生态创建了 Reactive Extensions (Rx) 库 。然后，RxJava在JVM上实现了响应式编程。随着时间推移，Java出现了响应式流标准，该规范定义了JVM上反应性库的一组接口和交互规则。它的接口已集成到Java9的Flow类下。

响应式编程经常会被用来和面向对象语言的观察者模式类比。我们也可以和迭代器模式类比，比如遍历Map的k,v。区别在于，迭代器式拉模式，响应式是推模式。

iterator是一种急性子编程模式，开发者自己选择何时访问next()元素。对应响应式编程中`Publisher-Subscriber`。但是Publisher通知Subscriber最新可用的值，以推送的方式给Subscriber，开发者使用响应式编程，只需要关心数据的处理计算逻辑，而不关心流程控制迭代逻辑。

除了Pushing Values，还可以手动定义 error-handling、completion。Publisher调用 onNext 把新值Push给 Subscriber，也可以调用 onError 、onComplete发送错误/完成信号，来中断给Subcriber的流。

```
onNext x 0..N [onError | onComplete]
```

这种方法非常灵活（操作0，1，N个数）。

1、阻塞是浪费

现在应用能处理大量并发用户，但是软硬件性能依然是核心话题。

常见两种方式提升应用性能：

- 1、并行：使用更多线程，更多硬件资源

- 2、寻求更高效率：每种资源使用找到最高效的方式

通常，Java开发人员使用阻塞代码编写程序。在性能瓶颈出现之前，这种方式都很ok。接下来，就会考虑用更多的线程，运行阻塞式代码。但是，这种资源利用率的扩展可以快速引入争议和并发问题。

糟糕的是，阻塞是种资源浪费。如果您仔细观察，一旦程序涉及一些延迟（尤其是I/O，例如数据库请求或网络调用），资源就会浪费，因为现在的线程（可能很多线程）现在闲置，等待数据。

因此，并行不是银弹。需要发挥硬件完全能力，但资源浪费因素也很复杂。

2、异步-拯救?

前面提到的第二种方法是寻求提高效率，可以作为解决资源浪费问题的解决方案。通过编写异步，非阻滞代码，您可以让执行切换到另一个使用相同基础资源的活动任务，然后在异步处理完成后返回当前过程。

但是，如何在JVM上产生异步代码？ Java提供了两种异步编程模型：

- **Callbacks**：异步方法没有返回值，但结果会传递给回调函数。比如 Swing的EventListener
- **Futures**：异步立即返回一个 `Future<T>`，异步处理完后返回T，Future提供了T的访问直到T完成。如：ExecutorService 使用 Future 对象 运行 Callable 任务

这些技术够好吗？并非每种用例，两种方法都有局限性。

1、回调的局限性

> 回调很难共同组成，很快就会导致难以阅读和维护的代码（称为“回调地狱”）。
>
> 例如：showing the top five favorites from a user on the UI or suggestions if she does not have a favorite. This goes through three services (one gives favorite IDs, the second fetches favorite details, and the third offers suggestions with details), as follows:

```
userService.getFavorites(userId, new Callback<List<String>>() {
  public void onSuccess(List<String> list) {
    if (list.isEmpty()) {
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) {
          UiUtils.submitOnUiThread(() -> {
            list.stream()
                .limit(5)
                .forEach(uiList::show);
            });
        }

        public void onError(Throwable error) {
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream()
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId,
```

```
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

以上代码在响应式编程中可表示为如下

```
userService.getFavorites(userId)
           .flatMap(favoriteService::getDetails)
           .switchIfEmpty(suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe(uiList::show, UiUtils::errorPopup);
```

如果想保证favorite IDs在800ms内查询完成，如果太长时间，就从缓存中获取。可以如下编写

```
userService.getFavorites(userId)
           .timeout(Duration.ofMillis(800))
           .onErrorResume(cacheService.cachedFavoritesFor(userId))
           .flatMap(favoriteService::getDetails)
           .switchIfEmpty(suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe(uiList::show, UiUtils::errorPopup);
```

2、Future局限性

`Future`的对象比回调要好一些，但是尽管有Java 8改进的`CompletableFuture`，但它们在组合方面仍然表现不佳。将`Future`多个对象编排是可行的，但并不容易。另外，`Future`还有其他问题：

- 通过调用get（）方法，很容易与Future对象一起出现另一个阻塞情况。
- 不支持懒计算
- 缺乏对多值和高级错误处理的支持。

> 场景： We get a list of IDs from which we want to fetch a name and a statistic and combine these pair-wise, all of it asynchronously. The following example does so with a list of type `CompletableFuture`:

```
CompletableFuture<List<String>> ids = ifhIds();
```

```
CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> {
    Stream<CompletableFuture<String>> zip =
            l.stream().map(i -> {
                CompletableFuture<String> nameTask = ifhName(i);
                CompletableFuture<Integer> statTask = ifhStat(i);

                return nameTask.thenCombineAsync(statTask, (name, stat) -> "Name
" + name + " has stats " + stat);
            });
    List<CompletableFuture<String>> combinationList =
zip.collect(Collectors.toList());
    CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);

    CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray);
    return allDone.thenApply(v -> combinationList.stream()
            .map(CompletableFuture::join)
            .collect(Collectors.toList()));
});

List<String> results = result.join();
assertThat(results).contains(
        "Name NameJoe has stats 103",
        "Name NameBart has stats 104",
        "Name NameHenry has stats 105",
        "Name NameNicole has stats 106",
        "Name NameABSLAJNFOAJNFOANFANSF has stats 121");
```

> Reactor拥有更丰富的合并操作。可简化为如下

```
Flux<String> ids = ifhrIds();

Flux<String> combinations =
        ids.flatMap(id -> {
            Mono<String> nameTask = ifhrName(id);
            Mono<Integer> statTask = ifhrStat(id);

            return nameTask.zipWith(statTask,
                    (name, stat) -> "Name " + name + " has stats " + stat);
        });

Mono<List<String>> result = combinations.collectList();

List<String> results = result.block();
assertThat(results).containsExactly(
        "Name NameJoe has stats 103",
        "Name NameBart has stats 104",
        "Name NameHenry has stats 105",
        "Name NameNicole has stats 106",
        "Name NameABSLAJNFOAJNFOANFANSF has stats 121"
);
```

3、从命令式到响应式编程

Reactive 库（例如Reactor）旨在解决JVM上“经典”异步方法的这些缺点，同时还要关注其他一些方面：

- 组合性和可读性（Composability and Readability）
- 使用丰富的 操作符，操作管理一个数据流
- 不 **subscribe** ，就不会有事情发生
- 背压 或 消费者通过给生产者信号来控制数据的生产速率不会太高
- 高阶高度抽象 是 未知并发

1、组合性和可读性

- Reactor提供丰富任务组合式API
- 编写的代码具有可读性

2、流水线

> 响应式数据处理，类似装配线一样。
>
> 数据流过装配线，会被加工成相应的样子。Reactor既是传送带、又是工作站
>
> 原始数据从 `Publisher` 流入装配线使用各种 `operator` 加工后推给 `Subscriber`
>
> 原材料经过各种中间步骤加工。如果某个点有故障或者堵塞，可以向上游发送信号以限制原材料流动

3、操作符（Operators）

Reactor中 Operators 就是那个装配线上的工作站。每个 Operators 把自己行为添加到 `Publisher` 并且包装前面的步骤生成一个新的 `Publisher` 实例。整个链路通过这样被链接，当原始数据从第一个 `Publisher` 出来后，会从整个链路流下来，被每个环节进行加工。最终，一个 `Subscriber` 完成整个处理。

> 记住：直到 `Subcriber` 去 `Publisher` 订阅数据，否则什么都不会发生；
>
> 理解 Operators 是创建一个新实例，可以帮助我们避免出现一些常见错误，比如：编码期间需要链式组装整个流水线

虽然反应流规范根本没有指定Operators ，但reactor等这些库提供了丰富的操作符。覆盖很多方面，从简单的转换和过滤到复杂的编排和错误处理。

4、什么都不会发生，直到 `subscribe()`

在Reactor中，当您编写发布者链时，默认情况下，数据不会开始泵入它。相反，您可以对异步过程进行抽象描述（可以帮助可重复使用和组成）。

通过 **subscribe** 的行为，您将 **Publisher** 与 **Subscriber** 联系起来，这触发了整个链中的数据流。这是通过从 **Subscriber** 的单个 **request** 信号在上游传播的单个请求信号来实现的，一直返回到源出版商。

5、Backpressure（背压）

上游信号传播用来实现背压。当装配线某个工作站很慢时，通过给上游发送信号，减慢整个流水线速度。

`Reactive Streams`定义的规范真实机制类比如下：

- 一个`Subscriber`可以在 unbounded 模式下工作，让源把所有数据以最快的速率推送过来
- 或者可以使用`request()`机制给数据源发送信号，代表自己准备好处理 n 个数据了

中间操作符还可以修改中途的`request`，想象 一个`buffer`操作符把10个元素组织成一组。如果`subscriber`请求一个`buffer`，他就会接到10个元素。一些操作符也实现了 **prefetching**（预取）策略， 避免每次`request(1)`的整个往返过程浪费。（注：request(1)给上游先发一个信号，然后上游产生一个数据，然后再交给下游，这是一个往返过程）

这种Push模型一般是`push-pull-hybrid（推拉混合）`，下游准备好后就可以从上游`pull（拉）`n个元素。

6、Hot vs Cold

RX家族的响应式库区分了两个常见的响应式序列：冷热。这种区别主要是响应式流对Subcriber的响应方式有关：

- 一个**冷序**开始于每个Subscriber，包括数据源。例如，如果源包装HTTP调用，则为每种订阅制作新的HTTP请求。
- 每个`Subcriber`的热序列不会从头开始。相反，后来的 Subcriber 在订阅后会发出信号。但是请注意，一些热反应流可以完全或部分地缓存或重播排放历史。从一般的角度来看，当没有订阅者正在侦听时，热序列甚至可以发出（“订阅之前什么都没有发生”规则的例外）。

4、Reactor Core 特性

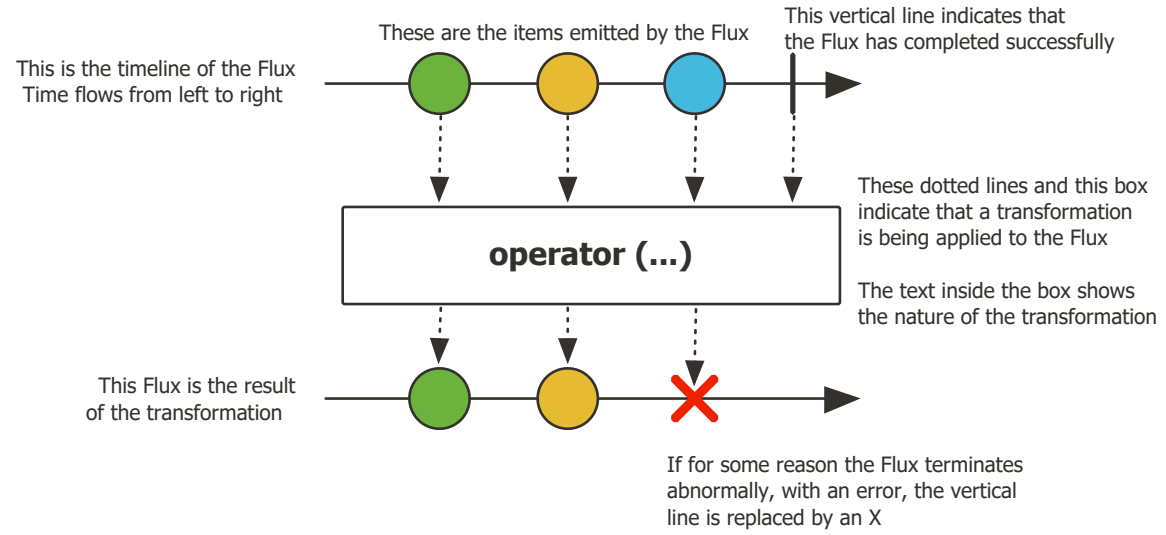
Reactor项目主要模块是`reactor-core`，这是一个响应式库，侧重于反应流的规范，并针对Java 8。

Reactor 引入了可复合的响应式类型，可以实现`Publisher`，但也提供了丰富的操作符：Flux和 Mono。Flux对象表示0..n项的响应式序列，而Mono对象表示单值或空的结果（0..1）。

这种区别将语义信息带入了类型中，表明异步处理的粗糙基数。例如，HTTP请求仅产生一个响应，因此进行计数操作没有太多意识。因此，将这种HTTP调用的结果表达为`Mono<httpresponse>`，比将其表示为`Flux <httpresponse>`更有意义。

操作符也会改变类型，比如`count`会把`Flux`变成`Mono<Long>`

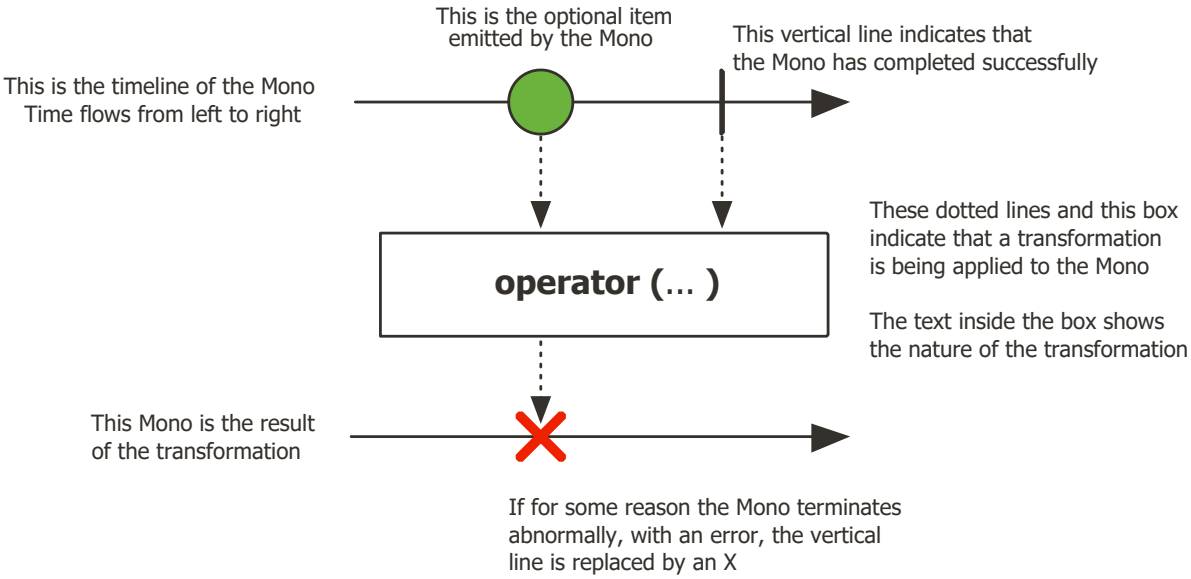
1、Flux，异步0-N元素序列



`Flux<T>`是一个标准的`Publisher<T>`，他代表0-N元素的异步序列，可以由完成或错误信号中断这个序列。在响应流规范中，这三种形式会由下游`Subscriber`进行调用，如`onNext`、`onComplete`、`onError`

由于信号可能会随时发生，`Flux`是一个常见的响应类型。注意：所有事件、甚至终止事件都是可选的：没有`onNext`事件，但是`onComplete`事件代表一个空的有限序列，但是移除`onComplete`事件并且我们有一个无限空序列。相似的，无限序列不一定是空的。比如，`Flux.interval(Duration)`基于时钟周期产生一个无限的`Flux<Long>`序列

2、Mono，异步0-1元素序列



`Mono<T>`是一个特别的`Publisher<T>`，使用`onNext`信号发射最多一个数据，使用`onComplete`（成功的Mono，有或没值都行）成功中断，使用`onError`只发送一个错误信号（失败的Mono）

大多数`Mono`的实现期望`Subcriber`能在调用`onNext`之后立即调用`onComplete`。`Mono.never()`例外：他不会发送任何信号，除了用于测试外，其他场景都没啥用。另一方面，明确静止`onNext`和`onError`组合

`Mono`提供`Flux`的一些自操作，某些操作符（尤其是组合`Mono`和其他`Publisher`）和可以切换到`Flux`。比如：`Mono#concatWith(Publisher)`返回Flux，`Mono#then(Mono)`返回`Mono`

注意，您可以使用Mono来表示仅具有完成概念（类似于可运行的）的无价异步过程。要创建一个，您可以使用一个空的`Mono<void>`。

3、创建Flux、Mono并Subcribe的一些方式

最简单方式使用`Flux`、`Mono`是使用他们类提供的一些工厂方法。

例如，创建一个String序列，可以枚举或者创建一个集合，然后用Flux创出序列。

```
Flux<String> seq1 = Flux.just("foo", "bar", "foobar");

List<String> iterable = Arrays.asList("foo", "bar", "foobar");
Flux<String> seq2 = Flux.fromIterable(iterable);
```

其他方式

```
Mono<String> noData = Mono.empty();

Mono<String> data = Mono.just("foo");

Flux<Integer> numbersFromFiveToSeven = Flux.range(5, 3);
```

可以使用Java8提供的lambda订阅这些Flux和Mono序列。用`.subscribe()`：

> 基于lambda方式：

```
subscribe();

subscribe(Consumer<? super T> consumer);

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

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

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

1、`subscribe`示例

- 无参方式
  - ```
    Flux<Integer> ints = Flux.range(1, 3);
    ints.subscribe(); //有消费线程，只是看不见
    ```
- 消费
  - ```
    Flux<Integer> ints = Flux.range(1, 3);
    ints.subscribe(i -> System.out.println(i)); //消费并处理
    ```
- 错误信号

    - ```
      Flux<Integer> ints = Flux.range(1, 4)
            .map(i -> {
              if (i <= 3) return i;
              throw new RuntimeException("Got to 4");
            });
      ints.subscribe(i -> System.out.println(i),
            error -> System.err.println("Error: " + error));
      ```

- 完成事件
    - ```
      Flux<Integer> ints = Flux.range(1, 4);
      ints.subscribe(i -> System.out.println(i),
          error -> System.err.println("Error " + error),
          () -> System.out.println("Done"));
      ```

- 消费行为
    - ```
      Flux<Integer> ints = Flux.range(1, 4);
      ints.subscribe(i -> System.out.println(i),
          error -> System.err.println("Error " + error),
          () -> System.out.println("Done"),
          sub -> sub.request(1));
      ```

    - 定制消费行为request(1)，则只会要上游1个

2、使用 `Disposable`，取消subcribe()

每个`subscribe()`返回`Disposable`对象。Disposable 接口可以调用 dispose() 来取消订阅。

对于`Flux`或`Mono`，取消是一个信号来告诉源停止生产数据。但是，这个不能保证是立即的：一些源可能会在接到信号，或者取消命令之前产生很多数据。

一些围绕Disposable的工具，在Disposables类中。**Disposables.swap()** 创建 **Disposable** 的包装器可以允许我们自动取消或者替换具体的**Disposable**。这个非常有用，比如：在UI场景，我们想取消请求，并且替换成我们点击按钮的新请求。

其他有趣的工具是`Disposables.composite(…)`,这个可以让我们收集一些`Disposable`，例如：多个正在执行的请求关联一个服务调用。当我们组合的dispose()方法被调用，任何尝试添加其他`Disposable`立即处理他。

3、lambda的替代方式：`BaseSubscriber`

```
SampleSubscriber<Integer> ss = new SampleSubscriber<Integer>();
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(ss);
```

```
package io.projectreactor.samples;

import org.reactivestreams.Subscription;

import reactor.core.publisher.BaseSubscriber;

public class SampleSubscriber<T> extends BaseSubscriber<T> {

    public void hookOnSubscribe(Subscription subscription) {
        System.out.println("Subscribed");
```
```

```
        request(1);
    }

    public void hookOnNext(T value) {
        System.out.println(value);
        request(1);
    }
}
```

BaseSubscriber 也提供 **requestUnbounded()** 方法去切换为unbounded模式（等同于request(Long.MAX_VALUE)），以及cancel方法

他还有其他钩子：`hookOnComplete`、`hookOnError`, `hookOnCancel`, and `hookFinally`(finally总是在流中断的时候被调用，并传入SignalType参数)

> 大量场景需要实现 `hookOnError`, `hookOnCancel`, and `hookOnComplete` 方法，可能需要实现 `hookFinally`方法。SampleSubscriber 是处理请求的最小实现参考。

4、背压和重塑请求的方式

Reactor中怎么实现背压。consumer的压力通过发送request传播回上游数据源。当前请求的总和有时候用当前`demand`或者`pending request`代表。demand的上限是Long.MAX_VALUE，代表一个Unbounded请求（意思"尽最大速度生产"-基本上是禁用背压）

第一个请求从最后一个subscriber订阅的时候来，往往，最直接的订阅方式是立即触发一个unbounded请求 of Long.MAX_VALUE，使用如下方式：

- `subscribe()`
- `block()`, `blockFirst()` and `blockLast()`
- 使用 `toIterable()` or `toStream()`进行迭代

最简单方式定义原始请求是使用`BaseSubscriber`的`hookOnSubscribe`方法。

```
Flux.range(1, 10)
    .doOnRequest(r -> System.out.println("request of " + r))
    .subscribe(new BaseSubscriber<Integer>() {

      @Override
      public void hookOnSubscribe(Subscription subscription) {
        request(1);
      }

      @Override
      public void hookOnNext(Integer integer) {
        System.out.println("Cancelling after having received " + integer);
        cancel();
      }
    });
```

1、操作符&下游需求改变

要记住的一件事是，在上游链中的每个操作可以重塑在订阅级别上表达的需求。教科书案例是`buffer（n）`操作符：如果它收到`request（2）`，则将其解释为对两个完整缓冲区的需求。结果，由于缓冲区需要将n个元素视为已满，因此缓冲区操作员将请求重塑为2 xN。

您可能还注意到，某些操作员具有称为prefetch的int输入参数的变体。这是修改下游请求的另一类操作员。这些通常是处理内部序列的运算符，从每个传入元素（如Flatmap）中得出发布者。

Prefetch可以指定请求数据量，默认32

操作符通常会有 **补充优化**：75%的预数据填满后，会从上游再请求75%。这种优化使操作符能主动预支上游请求

几个操作符能直接调整请求：`limitRate`和`limitRequest`

- **limitRate(N)**
  - 把下游请求数量分割成更多批次。如：100个请求，使用limitRate(10)，最终会10批，每批10个请求上游。
  - limitRate实现了 **补充优化**
  - `limitRate(highTide, lowTide)`允许变速率。
- **limitRequest(N)**

4、编程式创建序列

创建Flux或Mono，关联onNext、onError、onComplete事件。这些方法触发事件我们称为sink（渠）

1、同步 `generate`

- 基本使用

```
Flux<String> flux = Flux.generate(
    () -> 0,
    (state, sink) -> {
      sink.next("3 x " + state + " = " + 3*state);
      if (state == 10) sink.complete();
      return state + 1;
    });
```

- 可变状态

```
Flux<String> flux = Flux.generate(
    AtomicLong::new,
    (state, sink) -> {
      long i = state.getAndIncrement();
      sink.next("3 x " + i + " = " + 3*i);
      if (i == 10) sink.complete();
      return state;
    });
```

2、异步多线程`create`

`create`是一个更高级的创建Flux的方法，可以每轮创建很多元素，甚至使用多线程方式。

暴露一个`FluxSink`，可以使用`next`、`error`和`complete`方法。

- 使用`create`桥接一个`Flux<T>`
  - ```
    interface MyEventListener<T> {
        void onDataChunk(List<T> chunk);
        void processComplete();
    }


    Flux<String> bridge = Flux.create(sink -> {
        myEventProcessor.register(
          new MyEventListener<String>() {

            public void onDataChunk(List<String> chunk) {
              for(String s : chunk) {
                sink.next(s);
              }
            }

            public void processComplete() {
                sink.complete();
            }
        });
    });
    ```
  - **OverflowStrategy**: 溢出策略
    - `IGNORE`：完全忽略 下游背压requests. 下游队列慢会抛出 IllegalStateException
    - `ERROR`: 下游不能再接受元素，发送 IllegalStateException 信号
    - `DROP`: 删除过来的信号，如果下游还没有准备好接受
    - `LATEST`: 下游仅获取最新信号
    - `BUFFER`: 默认，buffer缓存所有信号，如果下游不能消费。（如果是一个无界Buffer可能导致 OOM）

3、异步单线程：push

```
Flux<String> bridge = Flux.push(sink -> {
    myEventProcessor.register(
      new SingleThreadEventListener<String>() {

        public void onDataChunk(List<String> chunk) {
          for(String s : chunk) {
            sink.next(s);
          }
        }

        public void processComplete() {
            sink.complete();
        }

        public void processError(Throwable e) {
            sink.error(e);
```
```

```
        }
    });
});
```

1、push/pull 混合模型

```
Flux<String> bridge = Flux.create(sink -> {
    myMessageProcessor.register(
      new MyMessageListener<String>() {

        public void onMessage(List<String> messages) {
          for(String s : messages) {
            sink.next(s);
          }
        }
    });
    sink.onRequest(n -> {
        List<String> messages = myMessageProcessor.getHistory(n);
        for(String s : messages) {
           sink.next(s);
        }
    });
});
```

2、push()、create()后清理

```
Flux<String> bridge = Flux.create(sink -> {
    sink.onRequest(n -> channel.poll(n))
        .onCancel(() -> channel.cancel())
        .onDispose(() -> channel.close())
    });
```

4、Handle

```
Flux<String> alphabet = Flux.just(-1, 30, 13, 9, 20)
    .handle((i, sink) -> {
        String letter = alphabet(i);
        if (letter != null)
            sink.next(letter);
    });

alphabet.subscribe(System.out::println);
```

5、线程与调度

```
public static void main(String[] args) throws InterruptedException {
  final Mono<String> mono = Mono.just("hello ");

  Thread t = new Thread(() -> mono
      .map(msg -> msg + "thread ")
      .subscribe(v ->
          System.out.println(v + Thread.currentThread().getName())
      )
  )
  t.start();
  t.join();

}
```

- **Mono** 在 main线程创建
- 但是在Thread-0中消费
- 意味着，map，onNext都会在Thread-0中运行

Reactor框架中，`Scheduler`决定怎么执行任务，类似于`ExecutorService`进行调度。

`Schedulers`的一些静态方法访问执行上下文

- 没有Execution上下文：`Schedulers.immediate()`： 处理期间，提交的 Runnable 被当前线程直接执行；
- 一个，可复用线程(`Schedulers.single()`)：所有调用者共用同一个线程，直到 Scheduler 结束。每个调用用一个线程可使用`Scheduler.newSingle()`
- 无界弹性线程池(`Schedulers.elastic()`)： 这种模式下，**Schedulers.boundedElastic()**不再是首选，因为它具有隐藏背压问题并导致太多线程的趋势（见下文）；
- 有界弹性线程池(**Schedulers.boundedElastic()**): 向之前的**elastic()**，他创建新的 worker 池并复用空闲线程。Worker 池长时间空闲（60s）就会被清理。不像 **elastic()**，他有后台线程的最大容量（默认 CPU核心 10倍）。100000 任务提交，队列容量满，就会在线程可用的时候出发重新调度（当是一个延迟调度，线程可用的时候，才开始计算延迟）。I/O阻塞式工作就适用这个。Scheduler.boundedelastic（）是提供自己的线程的方便方式，以免将其他资源绑定在一起。（参考：如何包装同步的，阻塞调用？）但是不会用新线程向系统施加太大压力。
- 固定worker线程池：`Schedulers.parallel()`。创建出和CPU核心一样多的worker

可以使用**Schedulers.fromExecutorService(ExecutorService)**基于前面的`ExecutorService`创建出`Scheduler`。

可以使用newXXX创建一个Scheduler示例。比如：`Schedulers.newParallel(yourScheduleName)`创建一个名字yourScheduleName为并行Scheduler。

> 可以使用 boundedElastic 来改善以前的阻塞式代码，single 和 parallel 。使用Reactor的API（block（），blockfirst（），blocklast（）等待结果，会抛出IllegalStateException

```
Flux.interval(Duration.ofMillis(300))//创建一个无限序列

//默认是使用  Schedulers.parallel()


Flux.interval(Duration.ofMillis(300), Schedulers.newSingle("test"))
```

Reactor 提供两种方式切换执行上下文，`publishOn` and `subscribeOn`。 nothing happens until you subscribe.

Reactor中操作链路的时候，可以wrap `Flux` 和 `Mono` 实现。一旦 subscribe，一个Subscriber对象创建，

1、`publishOn`

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4);

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i)
    .publishOn(s)
    .map(i -> "value " + i);

new Thread(() -> flux.subscribe(System.out::println));
```

2、`subscribeOn`

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4);

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i)
    .subscribeOn(s)
    .map(i -> "value " + i);

new Thread(() -> flux.subscribe(System.out::println));
```

6、错误处理

响应式流中，errors是一个中断事件。error发送后，会停止序列并且传播给下游 Subscriber 的 onError 方法

如果 onError 么有定义，就会抛出 **UnsupportedOperationException** 。 非常建议定义onError，这样在错误的时候可以响应UI，Rest等场景

```
Flux.just(1, 2, 0)
    .map(i -> "100 / " + i + " = " + (100 / i)) //this triggers an error with 0
    .onErrorReturn("Divided by zero :("); // error handling example
```

> 公知：任何响应式流中的错误都是一个中断事件。

1、错误处理操作符

常见异常处理机制：

- Catch and return a static default value.
- Catch and execute an alternative path with a fallback method.
- Catch and dynamically compute a fallback value.

- Catch, wrap to a `BusinessException`, and re-throw.
- Catch, log an error-specific message, and re-throw.
- Use the `finally` block to clean up resources or a Java 7 “try-with-resource” construct.

```
Flux<String> s = Flux.range(1, 10)
    .map(v -> doSomethingDangerous(v))
    .map(v -> doSecondTransform(v));
s.subscribe(value -> System.out.println("RECEIVED " + value),
            error -> System.err.println("CAUGHT " + error)
);
```

以上类似下面

```
try {
    for (int i = 1; i < 11; i++) {
        String v1 = doSomethingDangerous(i);
        String v2 = doSecondTransform(v1);
        System.out.println("RECEIVED " + v2);
    }
} catch (Throwable t) {
    System.err.println("CAUGHT " + t);
}
```

1、Static Fallback Value

```
try {
  return doSomethingDangerous(10);
}
catch (Throwable error) {
  return "RECOVERED";
}
```

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn("RECOVERED");
```

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn(e -> e.getMessage().equals("boom10"), "recovered10");
```

2、Fallback Method

```
String v1;
try {
  v1 = callExternalService("key1");
}
catch (Throwable error) {
  v1 = getFromCache("key1");
}

String v2;
try {
```

```
  v2 = callExternalService("key2");
}
catch (Throwable error) {
  v2 = getFromCache("key2");
}
```

```
Flux.just("key1", "key2")
    .flatMap(k -> callExternalService(k)
        .onErrorResume(e -> getFromCache(k))
    );
```

`onErrorReturn`, `onErrorResume`

```
Flux.just("timeout1", "unknown", "key2")
    .flatMap(k -> callExternalService(k)
        .onErrorResume(error -> {
            if (error instanceof TimeoutException)
                return getFromCache(k);
            else if (error instanceof UnknownKeyException)
                return registerNewEntry(k, "DEFAULT");
            else
                return Flux.error(error);
        })
    );
```

3、Dynamic Fallback Value

```
try {
  Value v = erroringMethod();
  return MyWrapper.fromValue(v);
}
catch (Throwable error) {
  return MyWrapper.fromError(error);
}
```

```
erroringFlux.onErrorResume(error -> Mono.just(
        MyWrapper.fromError(error)
));
```

4、Catch and Rethrow

```
try {
  return callExternalService(k);
}
catch (Throwable error) {
  throw new BusinessException("oops, SLA exceeded", error);
}
```

```
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(original -> Flux.error(
            new BusinessException("oops, SLA exceeded", original))
    );
```

5、Log or React on the Side

```
try {
  return callExternalService(k);
}
catch (RuntimeException error) {
  //make a record of the error
  log("uh oh, falling back, service failed for key " + k);
  throw error;
}
```

```
LongAdder failureStat = new LongAdder();
Flux<String> flux =
Flux.just("unknown")
    .flatMap(k -> callExternalService(k)
        .doOnError(e -> {
            failureStat.increment();
            log("uh oh, falling back, service failed for key " + k);
        })

    );
```

6、Using Resources and the Finally Block

```
Stats stats = new Stats();
stats.startTimer();
try {
  doSomethingDangerous();
}
finally {
  stats.stopTimerAndRecordTiming();
}
```

```
try (SomeAutoCloseable disposableInstance = new SomeAutoCloseable()) {
  return disposableInstance.toString();
}
```

doFinally()

```
Stats stats = new Stats();
LongAdder statsCancel = new LongAdder();

Flux<String> flux =
Flux.just("foo", "bar")
    .doOnSubscribe(s -> stats.startTimer())
    .doFinally(type -> {
        stats.stopTimerAndRecordTiming();
        if (type == SignalType.CANCEL)
          statsCancel.increment();
    })
    .take(1);
```

7、Terminal Aspect of `onError`

```
Flux<String> flux =
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .onErrorReturn("Uh oh");

flux.subscribe(System.out::println);
Thread.sleep(2100);
```

```
tick 0
tick 1
tick 2
Uh oh
```

8、Retrying

```
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .elapsed()
    .subscribe(System.out::println, System.err::println);

Thread.sleep(2100);
```

```
259,tick 0
249,tick 1
251,tick 2
506,tick 0
248,tick 1
253,tick 2
java.lang.RuntimeException: boom
```

Retry 循环逻辑:

- 每次发生错误的时候，一个 RetrySignal 就发给 我们的Flux。Flux将会有整个重试的上下文。RetrySignal 提供错误信息上下文数据
- **Flux** 发送一个值，retry开始
- `Flux` 完成，错误将被吃掉，retry循环停止，并且结果序列完成
- `Flux` 产生一个error，重试循环会使用错误停止结果序列

```
Flux<String> flux = Flux
    .<String>error(new IllegalArgumentException())
    .doOnError(System.out::println)
    .retryWhen(Retry.from(companion ->
        companion.take(3)));
```

```
AtomicInteger errorCount = new AtomicInteger();
Flux<String> flux =
        Flux.<String>error(new IllegalArgumentException())
                .doOnError(e -> errorCount.incrementAndGet())
                .retryWhen(Retry.from(companion ->
                        companion.map(rs -> {
                            if (rs.totalRetries() < 3) return rs.totalRetries();
                            else throw Exceptions.propagate(rs.failure());
                        })
                ));
```

2、使用操作符或函数处理异常

```
Flux.just("foo")
    .map(s -> { throw new IllegalArgumentException(s); })
    .subscribe(v -> System.out.println("GOT VALUE"),
               e -> System.out.println("ERROR: " + e));
```

异常处理几种方式：

- Catch the exception and recover from it. The sequence continues normally.
- Catch the exception, wrap it into an *unchecked* exception, and then throw it (interrupting the sequence). The `Exceptions` utility class can help you with that (we get to that next).
- If you need to return a `Flux` (for example, you are in a `flatMap`), wrap the exception in an error-producing `Flux`, as follows: `return Flux.error(checkedException)`

7、Processors（处理器） and Sinks（水槽）

Processors 是特殊的 Publisher，也是一个 **Subscriber**。它们最初是作为中间步骤的可能表示，然后可以在反应流实现之间共享。但是，在Reactor中，这样的步骤是由Publisher的operators 代表的。

常见错误是，遇到Processor时，从Subscriber接口直接调用`onNext`，`onComplete` 和`onError`方法。

这些手动调用应该小心，特别是Reactive Streams规范中定义的一些外部同步调用。实际上，Processors 可能会略有用，除非遇到基于反应流的API，该API需要通过 Subscriber 而不是暴露 Publisher。

Sinks 通常是更好的选择。Sink 是一个允许安全手动触发信号的类。可以关联Subscriber也可以独立

从`3.4.0`开始，Sink 成为一等公民，并且正在完全淘汰 Processors ：

- `FluxProcessor`和`MonoProcessor`会被从3.5.0中一处

- sinks 不是操作符产生的，是通过工程方法构造的 Sinks 类
- 以后推荐 用 Sinks 替换 processor

1、多线程安全生产，使用Sinks.one和Sinks.many

```
Sinks.Many<Integer> replaySink = Sinks.many().replay().all();
```

```
//thread1
replaySink.emitNext(1, FAIL_FAST);

//thread2, later
replaySink.emitNext(2, FAIL_FAST);

//thread3, concurrently with thread 2
EmitResult result = replaySink.tryEmitNext(3); //would return
FAIL_NON_SERIALIZED
```

```
Flux<Integer> fluxView = replaySink.asFlux();
fluxView
    .takeWhile(i -> i < 10)
    .log()
    .blockLast();
```

Sinks:

- **many().multicast()**: 仅新数据推送给subscriber的时候发送数据。
- `many().unicast()`:与上述相同，随着数据在第一个订户Buffer被缓冲之前推动的流动。
- **many().replay()**: 订阅者来时重放之前旧数据，并开始推送后面新数据
- **one()**: sink只推一个数据给订阅者
- **empty()**: 推一个中断信号。（error或complete）

2、 可用的Sinks

Reactor 3 Reference Guide (projectreactor.io)

> **Sinks.many().unicast().onBackpressureBuffer(args?)**
>
> **Sinks.many().multicast().onBackpressureBuffer(args?)**
>
> **Sinks.many().multicast().directAllOrNothing()**
>
> **Sinks.many().multicast().directAllOrNothing()**

> **Sinks.many().multicast().directBestEffort()**
>
> **Sinks.many().replay()**
>
> **Sinks.unsafe().many()**
>
> **Sinks.one()**
>
> **Sinks.empty()**

5、测试

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

1、`StepVerifier`

```
@Test
public void testAppendBoomError() {
  Flux<String> source = Flux.just("thing1", "thing2");

  StepVerifier.create(
    appendBoomError(source))
    .expectNext("thing1")
    .expectNext("thing2")
    .expectErrorMessage("boom")
    .verify();
}
```

2、操作时间

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
```

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
    .expectSubscription()
    .expectNoEvent(Duration.ofDays(1))
    .expectNext(0L)
    .verifyComplete();
```

3、`StepVerifier` 后置断言

After having described the final expectation of your scenario, you can switch to a complementary assertion API instead of triggering `verify()`. To do so, use `verifyThenAssertThat()` instead.

`verifyThenAssertThat()` returns a `StepVerifier.Assertions` object, which you can use to assert a few elements of state once the whole scenario has played out successfully (because it also calls `verify()`). Typical (albeit advanced) usage is to capture elements that have been dropped by some operator and assert them (see the section on Hooks).

4、测试`Context`

```
StepVerifier.create(Mono.just(1).map(i -> i + 10),

StepVerifierOptions.create().withInitialContext(Context.of("thing1", "thing2")))
                    .expectAccessibleContext()
                    .contains("thing1", "thing2")
                    .then()
                    .expectNext(11)
                    .verifyComplete();
```

5、`TestPublisher`手动发送

For both cases, `reactor-test` offers the `TestPublisher` class. This is a `Publisher<T>` that lets you programmatically trigger various signals:

- `next(T)` and `next(T, T…)` triggers 1-n `onNext` signals.
- `emit(T…)` triggers 1-n `onNext` signals and does `complete()`.
- `complete()` terminates with an `onComplete` signal.
- `error(Throwable)` terminates with an `onError` signal.

6、`PublisherProbe`检查执行路径

```
public Flux<String> processOrFallback(Mono<String> source, Publisher<String>
fallback) {
    return source
            .flatMapMany(phrase -> Flux.fromArray(phrase.split("\\s+")))
            .switchIfEmpty(fallback);
}
```

```
@Test
public void testSplitPathIsUsed() {
    StepVerifier.create(processOrFallback(Mono.just("just a  phrase with
tabs!"),
            Mono.just("EMPTY_PHRASE")))
                .expectNext("just", "a", "phrase", "with", "tabs!")
                .verifyComplete();
}

@Test
public void testEmptyPathIsUsed() {
    StepVerifier.create(processOrFallback(Mono.empty(),
Mono.just("EMPTY_PHRASE")))
                .expectNext("EMPTY_PHRASE")
                .verifyComplete();
}
```

```
private Mono<String> executeCommand(String command) {
    return Mono.just(command + " DONE");
}

public Mono<Void> processOrFallback(Mono<String> commandSource, Mono<Void>
doWhenEmpty) {
    return commandSource
            .flatMap(command -> executeCommand(command).then())
            .switchIfEmpty(doWhenEmpty);
}
```

```
@Test
public void testCommandEmptyPathIsUsed() {
    PublisherProbe<Void> probe = PublisherProbe.empty();

    StepVerifier.create(processOrFallback(Mono.empty(), probe.mono()))
                .verifyComplete();

    probe.assertWasSubscribed();
    probe.assertWasRequested();
    probe.assertWasNotCancelled();
}
```

6、Debugging Reactor

1、一个Reactor堆栈

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
    at
reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.tryEmitScalar(FluxFlatMap.java:44
5)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.onNext(FluxFlatMap.java:379)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFusea
ble.java:121)
    at
reactor.core.publisher.FluxRange$RangeSubscription.slowPath(FluxRange.java:154)
    at
reactor.core.publisher.FluxRange$RangeSubscription.request(FluxRange.java:109)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuse
able.java:162)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.onSubscribe(FluxFlatMap.java:332)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMap
Fuseable.java:90)
    at reactor.core.publisher.FluxRange.subscribe(FluxRange.java:68)
    at reactor.core.publisher.FluxMapFuseable.subscribe(FluxMapFuseable.java:63)
    at reactor.core.publisher.FluxFlatMap.subscribe(FluxFlatMap.java:97)
    at reactor.core.publisher.MonoSingle.subscribe(MonoSingle.java:58)
```

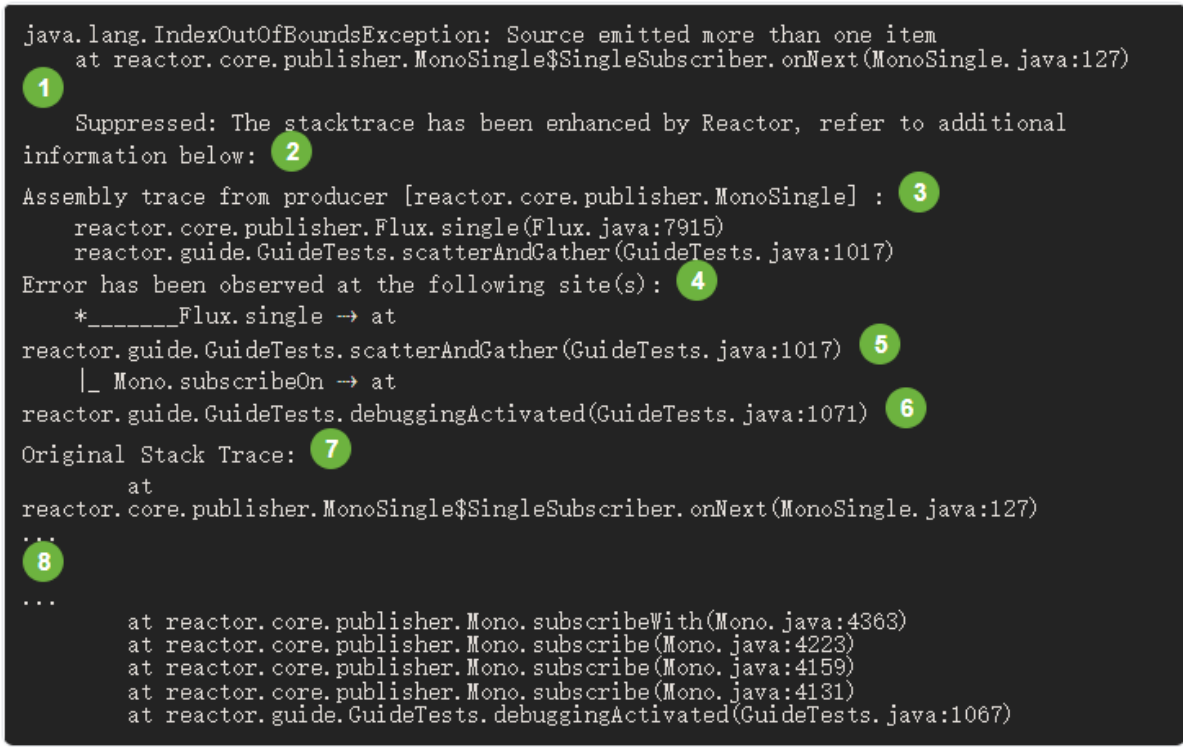
```
    at reactor.core.publisher.Mono.subscribe(Mono.java:3096)
    at reactor.core.publisher.Mono.subscribeWith(Mono.java:3204)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3090)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3057)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3029)
    at reactor.guide.GuideTests.debuggingCommonStacktrace(GuideTests.java:995)
```

```
toDebug
    .subscribeOn(Schedulers.immediate())
    .subscribe(System.out::println, Throwable::printStackTrace);
```

2、激活Debug模式-aka tracebacks

Hooks.onOperatorDebug();

3、Stack Trace in Debug Mode



1	**The original stack trace is truncated to a single frame.**
2	This is new: We see the wrapper operator that captures the stack. This is where the traceback starts to appear.
3	First, we get some details about where the operator was assembled.
4	Second, we get a notion of operator chain(s) through which the error propagated, from first to last (error site to subscribe site).
5	Each operator that saw the error is mentioned along with the user class and line where it was used. Here we have a "root".
6	Here we have a simple part of the chain.
7	The rest of the stack trace is moved at the end...
8	...showing a bit of the operator's internals (so we removed a bit of the snippet here).

4、生产环境全局debug

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

```
public static void main(String[] args) {
    ReactorDebugAgent.init(); //
    SpringApplication.run(Application.class, args);
}
```

7、暴露Reactor指标metrics

1、Scheduler metrics

> Schedulers.enableMetrics();
>
> - executor_active_threads
> - executor_completed_tasks_total
> - executor_pool_size_threads
> - executor_queued_tasks
> - executor_secounds_{count, max, sum}

2、Publisher metrics

```
listenToEvents()
    .doOnNext(event -> log.info("Received {}", event))
    .delayUntil(this::processEvent)
    .retry()
    .subscribe();
```

```
listenToEvents()
    .name("events")
    .metrics()
    .doOnNext(event -> log.info("Received {}", event))
    .delayUntil(this::processEvent)
    .retry()
    .subscribe();
```

3、tags

```
listenToEvents()
    .name("events")
    .tag("source", "kafka")
    .metrics()
    .doOnNext(event -> log.info("Received {}", event))
    .delayUntil(this::processEvent)
    .retry()
    .subscribe();
```

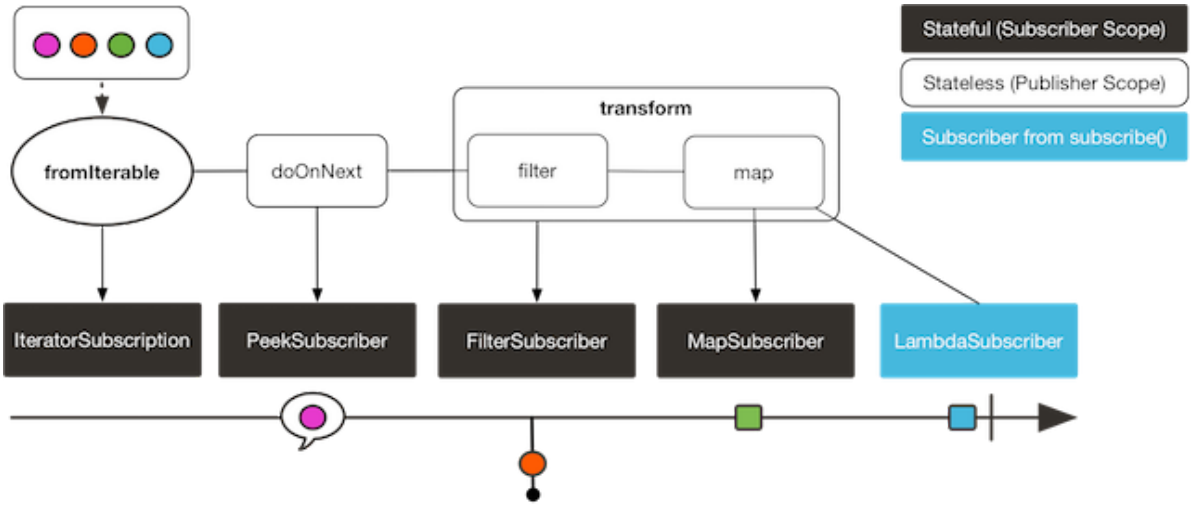
8、高级特性&概念

- Mutualizing Operator Usage： 互动操作符用法
- Hot Versus Cold： 冷热
- Broadcasting to Multiple Subscribers with `ConnectableFlux`： 多播
- Three Sorts of Batching： 批量树排序
- Parallelizing Work with `ParallelFlux`： 并行任务
- Replacing Default `Schedulers`：自定义Schedulers
- Using Global Hooks: 全局钩子
- Adding a Context to a Reactive Sequence：给响应式序列添加上下文
- Null Safety： null安全
- Dealing with Objects that Need Cleanup： 处理需要清理的对象

1、Mutualizing Operator Usage

1、`transform`

```
Function<Flux<String>, Flux<String>> filterAndMap =
f -> f.filter(color -> !color.equals("orange"))
      .map(String::toUpperCase);

Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))
    .doOnNext(System.out::println)
    .transform(filterAndMap)
    .subscribe(d -> System.out.println("Subscriber to Transformed MapAndFilter:
"+d));
```

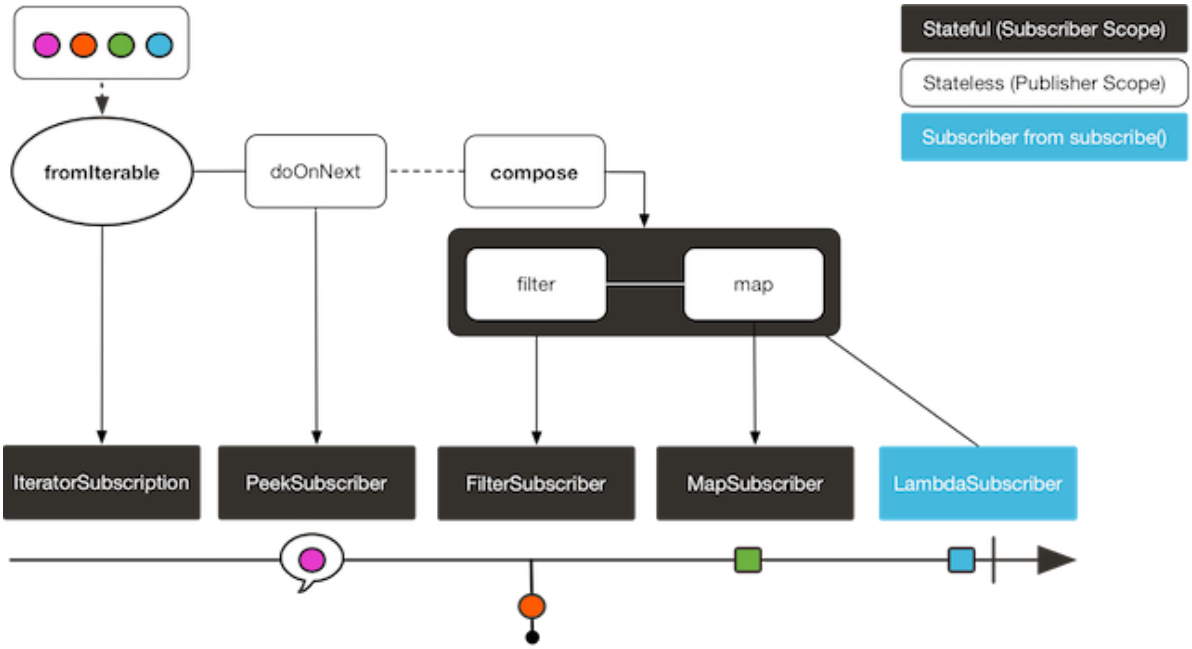


结果:

```
blue
Subscriber to Transformed MapAndFilter: BLUE
green
Subscriber to Transformed MapAndFilter: GREEN
orange
purple
Subscriber to Transformed MapAndFilter: PURPLE
```

2、`transformDeferred`

```
AtomicInteger ai = new AtomicInteger();
Function<Flux<String>, Flux<String>> filterAndMap = f -> {
    if (ai.incrementAndGet() == 1) {
return f.filter(color -> !color.equals("orange"))
        .map(String::toUpperCase);
    }
    return f.filter(color -> !color.equals("purple"))
            .map(String::toUpperCase);
};

Flux<String> composedFlux =
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))
    .doOnNext(System.out::println)
    .transformDeferred(filterAndMap);

composedFlux.subscribe(d -> System.out.println("Subscriber 1 to Composed
MapAndFilter :"+d));
composedFlux.subscribe(d -> System.out.println("Subscriber 2 to Composed
MapAndFilter: "+d));
```



> 结果:

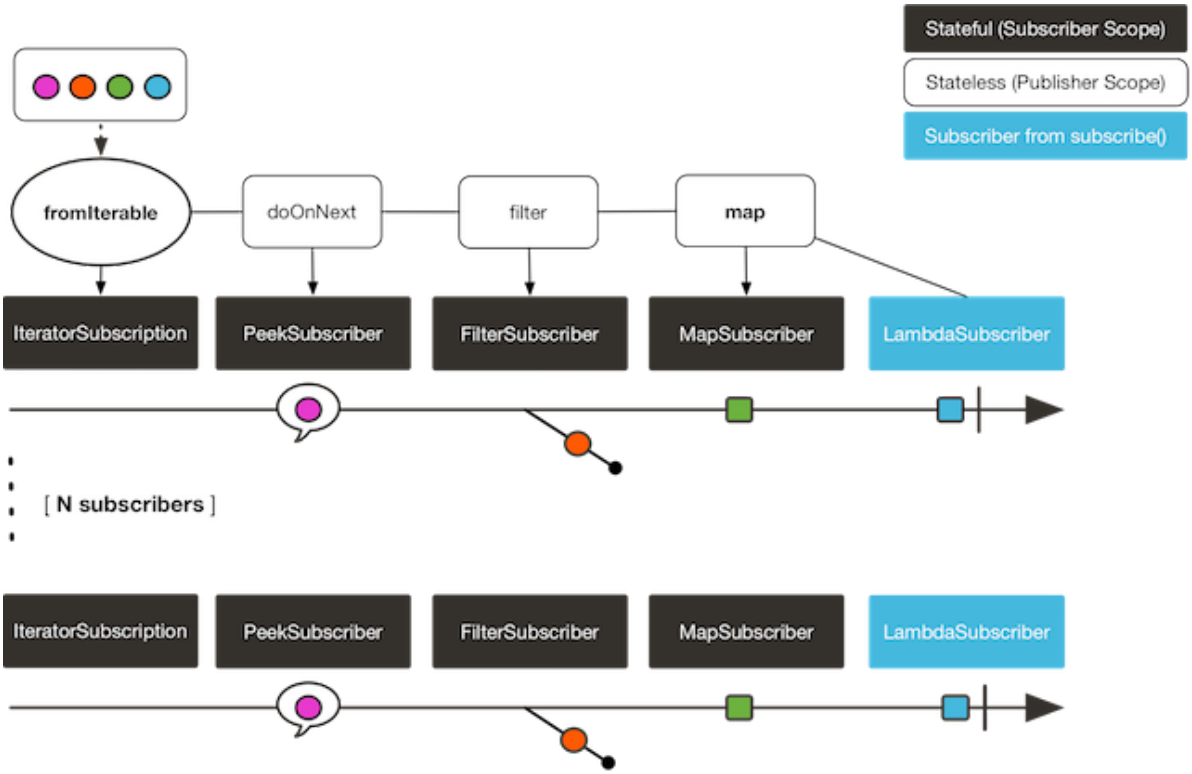
```
blue
Subscriber 1 to Composed MapAndFilter :BLUE
green
Subscriber 1 to Composed MapAndFilter :GREEN
orange
purple
Subscriber 1 to Composed MapAndFilter :PURPLE
blue
Subscriber 2 to Composed MapAndFilter: BLUE
green
Subscriber 2 to Composed MapAndFilter: GREEN
orange
Subscriber 2 to Composed MapAndFilter: ORANGE
purple
```

2. Hot Versus Cold

```
Flux<String> source = Flux.fromIterable(Arrays.asList("blue", "green", "orange",
"purple"))
                          .map(String::toUpperCase);

source.subscribe(d -> System.out.println("Subscriber 1: "+d));
source.subscribe(d -> System.out.println("Subscriber 2: "+d));
```

> 结果

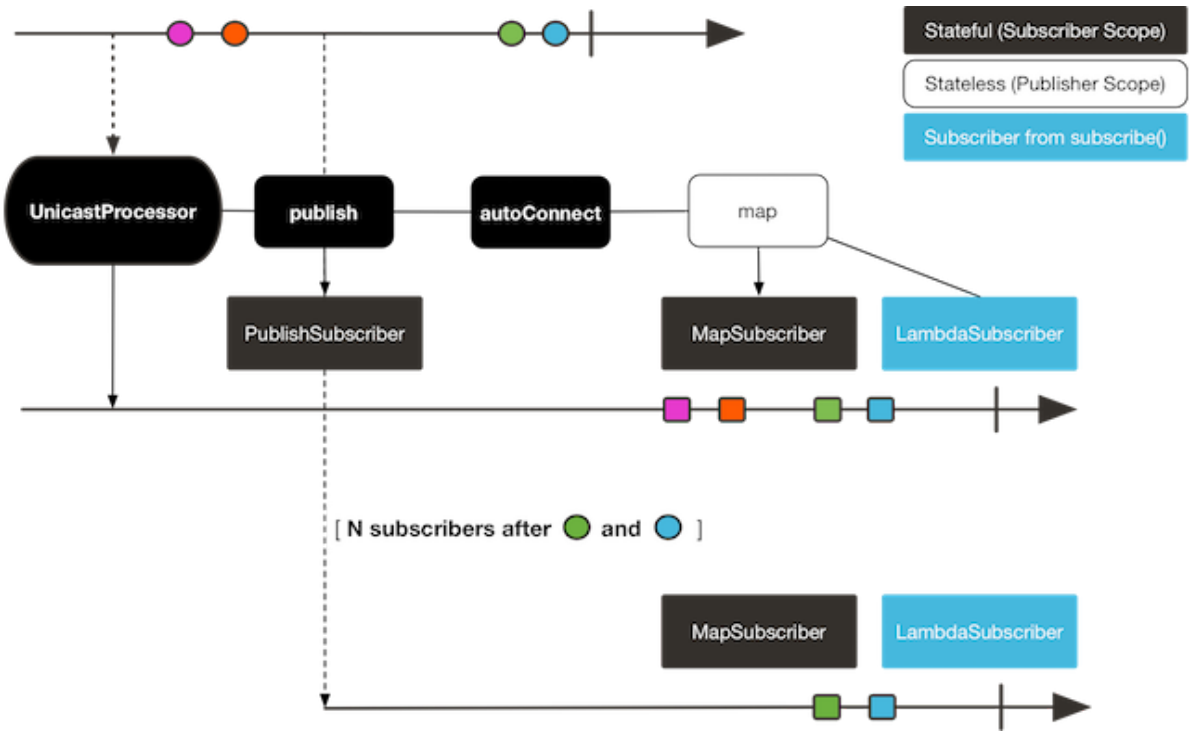
```
Subscriber 1: BLUE
Subscriber 1: GREEN
Subscriber 1: ORANGE
Subscriber 1: PURPLE
Subscriber 2: BLUE
Subscriber 2: GREEN
Subscriber 2: ORANGE
Subscriber 2: PURPLE
```




```
Sinks.Many<String> hotSource =
Sinks.unsafe().many().multicast().directBestEffort();

Flux<String> hotFlux = hotSource.asFlux().map(String::toUpperCase);

hotFlux.subscribe(d -> System.out.println("Subscriber 1 to Hot Source: "+d));

hotSource.emitNext("blue", FAIL_FAST);
hotSource.tryEmitNext("green").orThrow();

hotFlux.subscribe(d -> System.out.println("Subscriber 2 to Hot Source: "+d));

hotSource.emitNext("orange", FAIL_FAST);
hotSource.emitNext("purple", FAIL_FAST);
hotSource.emitComplete(FAIL_FAST);
```

> 结果

```
Subscriber 1 to Hot Source: BLUE
Subscriber 1 to Hot Source: GREEN
Subscriber 1 to Hot Source: ORANGE
Subscriber 2 to Hot Source: ORANGE
Subscriber 1 to Hot Source: PURPLE
Subscriber 2 to Hot Source: PURPLE
```



3、Broadcasting to Multiple Subscribers with `ConnectableFlux`

ConnectableFlux:

- **publish**
- **replay**

ConnectableFlux

- **connect()**
- **autoConnect(n)**
- **refCount(n)**
- **refCount(int, Duration)**

```
Flux<Integer> source = Flux.range(1, 3)
                           .doOnSubscribe(s -> System.out.println("subscribed to
source"));

ConnectableFlux<Integer> co = source.publish();

co.subscribe(System.out::println, e -> {}, () -> {});
co.subscribe(System.out::println, e -> {}, () -> {});

System.out.println("done subscribing");
Thread.sleep(500);
System.out.println("will now connect");

co.connect();
```

> 结果:

```
done subscribing
will now connect
subscribed to source
1
1
2
2
3
3
```

autoConnect

```
Flux<Integer> source = Flux.range(1, 3)
                           .doOnSubscribe(s -> System.out.println("subscribed to
source"));

Flux<Integer> autoCo = source.publish().autoConnect(2);

autoCo.subscribe(System.out::println, e -> {}, () -> {});
System.out.println("subscribed first");
Thread.sleep(500);
System.out.println("subscribing second");
autoCo.subscribe(System.out::println, e -> {}, () -> {});
```

结果：

```
subscribed first
subscribing second
subscribed to source
1
1
2
2
3
3
```

4、Three Sorts of Batching

1、`Flux<GroupedFlux<T>>`

```
StepVerifier.create(
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)
        .groupBy(i -> i % 2 == 0 ? "even" : "odd")
        .concatMap(g -> g.defaultIfEmpty(-1) //if empty groups, show them
                .map(String::valueOf) //map to string
                .startWith(g.key())) //start with the group's key
    )
    .expectNext("odd", "1", "3", "5", "11", "13")
    .expectNext("even", "2", "4", "6", "12")
    .verifyComplete();
```

2、Windowing with `Flux<Flux<T>>`

```
StepVerifier.create(
    Flux.range(1, 10)
        .window(5, 3) //overlapping windows
        .concatMap(g -> g.defaultIfEmpty(-1)) //show empty windows as -1
    )
        .expectNext(1, 2, 3, 4, 5)
        .expectNext(4, 5, 6, 7, 8)
        .expectNext(7, 8, 9, 10)
        .expectNext(10)
        .verifyComplete();
```

```
StepVerifier.create(
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)
        .windowWhile(i -> i % 2 == 0)
        .concatMap(g -> g.defaultIfEmpty(-1))
    )
        .expectNext(-1, -1, -1) //respectively triggered by odd 1 3 5
        .expectNext(2, 4, 6) // triggered by 11
        .expectNext(12) // triggered by 13
        // however, no empty completion window is emitted (would contain extra
matching elements)
        .verifyComplete();
```

附录：

操作符

弹珠图marble diagrams

最佳实战

Reactor-Extra