

Reactor 3 参考指南

Stephane Maldini, Simon Baslé

Version 3.3.5.RELEASE

Table of Contents

1. 于文	1
1.1. 最新版本 & 版 声明	1
1.2. 献本文	1
1.3. 取 助	1
1.4. 如何 始	1
2. 入	3
2.1. 介 Reactor	3
2.2. 前提	3
2.3. 理解 BOM	3
2.4. 入 Reactor	4
3. 式 程介	8
3.1. 阻塞更浪 源	8
3.2. 能 救?	9
3.3. 从命令式到 式 程	14
4. Reactor核心特性	17
4.1. Flux, 一个包含0-N个元素的 序列	17
4.2. Mono, 一个包含0-1 果的 序列	17
4.3. 建和 Flux或Mono	18
4.4. 以 程方式 建序列	25
4.5. 程和 度器	32
4.6. 理	35
4.7. 理器	50
5. Kotlin的支持	54
5.1. 要求	54
5.2. 展	54
5.3. 空安全	55
6.	56
6.1. 使用 StepVerifier 一个 景	56
6.2. 操	58
6.3. 用 StepVerifier 行 行后断言	60
6.4. 上下文	60
6.5. 用 TestPublisher 手 射	61
6.6. 用 PublisherProbe 行路径	62
7. Reactor	65
7.1. 典型的 式堆 追踪	65
7.2. 激活 模式 - 又名回	67
7.3. 在 模式下 取堆 跟踪	68
7.4. 可生 的全局	72

7.5. 序列	75
8. 暴露Reactor的指	78
8.1. 度器指	78
8.2. 生者指	78
9. 高特性和概念	81
9.1. 互用操作符用法	81
9.2. 与冷	84
9.3. 使用 ConnectableFlux 向多个者广播	87
9.4. 三批理	89
9.5. 用 ParallelFlux 并行化工作	92
9.6. 替的 Schedulers	93
9.7. 使用全局子	94
9.8. 式序列添加上下文	95
9.9. 理需要清理的象	103
9.10. 空安全	104
Appendix A: 我需要一个操作符?	106
A.1. 建一个新的序列	106
A.2. 已有的序列	107
A.3. 探序列	109
A.4. 序列	109
A.5. 理	110
A.6. 与的合作	111
A.7. 拆分 Flux	112
A.8. 回到同的世界	112
A.9. 广播 Flux 到多个 Subscribers	113
Appendix B: 常和最佳践, “我如何...?”	114
B.1. 如何包装一个同阻塞用?	114
B.2. 我在我的 Flux 上使用了一个操作符, 但似乎不用。什会?	114
B.3. 我的 Mono zipWith/zipWhen 从未被用	116
B.4. 如何使用 retryWhen 来模 retry(3)?	116
B.5. 我如何使用 retryWhen 行指数退避?	117
B.6. 使用 publishOn() 如何保程性?	118
B.7. 上下文日志 的好的方式是什么? (MDC)	119
Appendix C: Reactor-Extra	123
C.1. TupleUtils 和函数式接口	123
C.2. MathFlux 数学操作符	123
C.3. 重和重工具	124
C.4. 度器	124

Chapter 1. 于文

本章 要概述了Reactor参考文，不必依次 文，个章 都是独立的，尽管它常接其它的章。

1.1. 最新版本 & 版 声明

Reactor参考文 也提供了HTML版本的，最新可用的副本在 <https://projectreactor.io/docs/core/release/reference/index.html>

无 是 子版 是 版的文，只要 些副本文 包含了版 声明，且不 行任何的收，都可以供自己使用或者分享 他人。

1.2. 献本文

本参考文 是用 `Asciidoc` 写的，其源 位于 <https://github.com/reactor/reactor-core/tree/master/docs/asciidoc>。

如果 有任何改 或建， 迎提交PR。

我 建 check out源 到本地，便行gradle asciidoctor 任 行文 的 建以及 染效果。有些部分章 依 于其包含的文件，因此GitHub的 染并不 是完整的。

1.3. 取 助

使用Reactor， 可以通 以下几 方式 求 助：

- 在 [Gitter](#) 上与社区取得 系。
- 在[stackoverflow.com project-reactor 提](#)。
- 在Github issues上提交bug，我 密切 注 些： `reactor-core` (包括核心功能) 和 `reactor-addons` (涵 式 和 配器等)。



所有的Reactor 目都是 源的， 包括此文 ，如果 此文 存在 且想改 它， 参考。

1.4. 如何 始

- 如果 想直接 入 程，前往 [入](#)。
- 如果 接触 式 程， 可能 从 式 程介 始。
- 如果 比 熟悉Reactor的概念，只是在 合 的操作，却想不到相 的操作符， 看附 我需要 个操作符？。
- 了更深入地了解Reactor的核心功能， 至 [Reactor核心特性](#) 行了解：
 - 更多 于Reactor的 式 型在 `Flux`, 一个包含0-N个元素的 序列 和 `Mono`, 一个包含0-1 果的 序列 章 。

- 使用 `scheduler` 行 程上下文切 。
 - 理 在 理章 。
- 使用 元 ?在 `reactor-test` 目是可以做到的！ 看 。
 - 以 程方式 建序列 章 提供了更多高 的 建 式源的方式。
 - 高 特性和概念 章 涵 了其它高 主 。

Chapter 2. 入

本章 包含的信息有助于 使用Reactor，包含以下部分：

- 介 Reactor
- 前提
- 理解 BOM
- 入 Reactor

2.1. 介 Reactor

Reactor是一个完全非阻塞的JVM 式 程基，有着高效的需求管理（背 的形式）。它直接整合Java8的函数式API，尤其是 `CompletableFuture`, `Stream`, 有 `Duration` 提供了可 合的 化序列 API—`Flux`（于 [N] 个元素) and `Mono`（于 [0|1] 元素)—并广泛 式`Stream`。

Reactor 有支持非阻塞 程 通信的 `reactor-netty` 目， 用于微服 架，Reactor Netty HTTP (包括Websockets)，TCP和UDP提供了背 机制的 引。完全支持 式 解。

2.2. 前提

Reactor Core 行在 Java 8 及之上。

于 `org.reactivestreams:reactive-streams:1.0.3` 依。

Android的支持



- Reactor3并不 或正式支持Android（如果有很 的需求，考 使用RxJava2）。
- 然而它在Android SDK 26 (Android 0) 及之上 可以正常工作。
- 我 会尽最大的努力去 估有利于支持Android的 化。然而，我 不能 保 一点 ，必 根据具体的情况作出 个决定。

2.3. 理解 BOM

Reactor 3使用BOM (依 清) 模型 (从 `reactor-core 3.0.4` 始，以及 `Aluminium` 版本)。 尽管 些 件之 可能存在版本分 ，但 精 的 件列表，提供相 的版本，使其在一起能 行良好。

BOM本身是版本化的，它使用了一个代号和限定符的版本 方案。下面的列表展示了一些 例：

Aluminium-RELEASE

Californium-BUILD-SNAPSHOT

Aluminium-SR1

Bismuth-RELEASE

Californium-SR32

版本代号表示 上的MAJOR.MINOR数字，它 (大部分) 来于 元素周期表，按照字母 序 。

限定符（按照序）：

- **BUILD-SNAPSHOT**：和而建的。
- **M1..N**：里程碑或者人。
- **RELEASE**：代号系列中第一个GA（可用的）版本。
- **SR1..N**：代号系列中随后的GA版本—等同于修数字。（SR表示“服版本”）

2.4. 入 Reactor

正如 [前面提到的入理解BOM]，使用BOM和添加相的依在的工程中最使用Reactor的方式。注意，当添加一个依，必忽略版本，以便于从BOM中提取版本。

但是，如果想制的使用一个特定版本的件，可以像平一在添加依指定其版本。也可以完全放BOM，通件的版本指定其依。

2.4.1. Maven上的使用

Maven天然支持BOM的概念。首先需要通过添加下面的片段到的pom.xml来入BOM。

```
<dependencyManagement> ①
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>Bismuth-RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

① 注意 dependencyManagement，是除了常 dependencies 的部分。

如果的（dependencyManagement）在的pom中已存在，需要添加内容。

接下来，除了没有 `<version>` 以外，像往常一，添加的依到相的reactor工程中，如下所示。

```
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId> ①
        ②
    </dependency>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-test</artifactId> ③
        <scope>test</scope>
    </dependency>
</dependencies>
```

① 依 核心 。

② 没有版本 。

③ reactor-test 用作 reactive streams 行 元 。

2.4.2. Gradle上的使用

在Gradle5.0版本以前，没有 Maven BOM的核心支持，但是 可以使用Spring的 gradle依 管理 件。

首先，从Gradle 件入口 用 件，如下所示：

```
plugins {
    id "io.spring.dependency-management" version "1.0.7.RELEASE" ①
}
```

① 在写本文 ， 1.0.7.RELEASE是 件最新的版本。 然后更新它。

然后使用它 入BOM，如下所示：

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

最后，不添加版本号将依 添加到 的工程，如下所示：

```
dependencies {  
    implementation 'io.projectreactor:reactor-core' ①  
}
```

① 没有第三个： 版本分隔的部分，它取 于BOM。

从Gradle 5.0 始， 可以用原生的Gradle来支持BOM。

```
dependencies {  
    implementation platform('io.projectreactor:reactor-bom:Bismuth-RELEASE')  
    implementation 'io.projectreactor:reactor-core' ①  
}
```

① 没有第三个： 版本分隔的部分，它取 于BOM。

2.4.3. 路碑和快照

路碑和 者 的版本是通 Spring路碑 而不是中央 行 布。 使用如下片段，要将其添 加到 的 建配置文件中：

Example 1. Java中的路碑

```
<repositories>  
    <repository>  
        <id>spring-milestones</id>  
        <name>Spring Milestones Repository</name>  
        <url>https://repo.spring.io/milestone</url>  
    </repository>  
</repositories>
```

于Gradle， 使用下面的片段：

Example 2. Gradle中的路碑

```
repositories {  
    maven { url 'https://repo.spring.io/milestone' }  
    mavenCentral()  
}
```

同 的， 快照版本也是在一个 独的 用 中可用，如下面的例子所示：

*Example 3. Maven*中的快照版本

```
<repositories>
  <repository>
    <id>spring-snapshots</id>
    <name>Spring Snapshot Repository</name>
    <url>https://repo.spring.io/snapshot</url>
  </repository>
</repositories>
```

*Example 4. Gradle*中的快照版本

```
repositories {
  maven { url 'https://repo.spring.io/snapshot' }
  mavenCentral()
}
```

Chapter 3. 式 程介

Reactor是式程式的一，概括如下：

式程是一及数据流和化播的程式。意味着可以通程言松地表示静(如数)或(如事件射器)数据流。

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

作式程方向上的第一，Microsoft在.NET生中建了式(Rx)展。然后RxJava到了JVM上的式程。随着的推移，通Reactive Streams的努力，一套基于JVM式定接口与交互的准`Reactive Streams`出了。其接口已集成到了Java9中的Flow下。

式程常作察者模式的一展在面向象程言中体。可以将式流模式和熟悉的迭代器模式行比，它核心都基于Iterable-Iterator合。一个主要的区别是，迭代器基于拉的，而式流是基于推的。

迭代器是一命令式程模式，即使取完全取决于Iterable。事上，取决于者在序列中何用next()。在式流中，上面的合等于Publisher-Subscriber。但当有新的可用的数据到来，Publisher会者行通知，推是式的。当然，使用推送的操作表声明式的而不是命令式的：者注于表算而不是描述切的流控制。

除了推送数据外，以明的覆了理和完成方面。Publisher可以推送新的数据到其Subscriber(通用onNext)，也可以送(通用onError)或者完成(通用onComplete)信号。和完成信号都会止序列。可以如下：

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

方式非常活，用于零个，一个或者N个(包括无限序列，例如持的滴答声)。

但是首先需要思考的是，我什需要的式？

3.1. 阻塞更浪 源

代用程序有着巨大的并用，即使当代的硬件性能已提升了不少，但硬件的性能依然是一个因素。

通常有方式来提升用的性能：

- 使用更多的程和硬件源到并行化
- 在当前使用的源上求更高效的理

通常，Java者使用同方式程，做法在遇到性能瓶之前是可行的。当然此可能会引入更多的程来行相同的同代。但是像源利用率的展，很快会引入争和并的。

更糟的是，阻塞浪费。如果仔察，一旦程序及一些延操作（特别是I/O，例如数据求或者用），由于程（可能有很多程）等待数据而于空，致源的浪。

因此，并行化并不是。然了充分利用硬件源是有必要的，但是也来了性和容易造成源浪。

3.2. 能救？

先前提到的第二方式，求更高的效率，可以解决源浪。通写非阻塞的代，可以将行切到使用了相同底源的一个活任，然后在理完成后返回到当前任。

但是如何在JVM上写代？Java提供了程模型：

- **Callbacks**：方法没有一个返回，但是它外的了一个**callback**参数（lambda或者匿名），在当果可返回用。熟知的例子就是Swing的**EventListener**体系。
- **Futures**：方法立即返回一个**Future<T>**。理算得到个T，**Future**象取行了包装，个象可以一直直到返回。例如，**ExecutorService**使用**Future**象行**Callable<T>**任。

些技好？并不用于个用例，方式都有局限性。

回以合在一起，很容易致代以和（著名的地回）。

个例子：在界面上展示一个用的最先的五个喜好，如果都没有，在界面上行建。通用三个服（第一个提供喜好ID，第二个取喜好情，第三个提供情建），如下所示：

Example 5. 地回示例

```

userService.getFavorites(userId, new Callback<List<String>>() { ①
    public void onSuccess(List<String> list) { ②
        if (list.isEmpty()) { ③
            suggestionService.getSuggestions(new Callback<List<Favorite>>() {
                public void onSuccess(List<Favorite> list) { ④
                    UiUtils.submitOnUiThread(() -> { ⑤
                        list.stream()
                            .limit(5)
                            .forEach(uiList::show); ⑥
                    });
                }
            });
        } else {
            list.stream() ⑧
                .limit(5)
                .forEach(favId -> favoriteService.getDetails(favId, ⑨
                    new Callback<Favorite>() {
                        public void onSuccess(Favorite details) {
                            UiUtils.submitOnUiThread(() -> uiList.show(details));
                        }
                    }
                );
            }
        }
    }

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

- ① 我 提供了基于回 的服 :一个`Callback` 接口包含当 理成功和失 的 个方法。
- ② 第一个服 使用喜好列表ID 行回 。
- ③ 如果列表 空, 必 到 suggestionService。
- ④ suggestionService 将 List<Favorite> 到第二个回 。
- ⑤ 由于我 理的是UI, 我 需要 保我 的消 代 在UI 程。
- ⑥ 我 使用Java8 Stream 来限制建 的数量 5个, 并且将它 展示在UI的 形列表中。
- ⑦ 在 个 , 我 都以相同的方式 理 :将其 示在 出 口中。
- ⑧ 返回到喜好ID 。如果服 返回了完整的列表, 我 需要 到 favoriteService 来 取 的 Favorite 象。因 我 只需要五个, 所以我 首先将ID列表的流限制 5个。

⑨ 再一次回。一次我得到一个完整的 **Favorite** 象，我在UI程内将其推送到UI。

存在大量包含重且以追踪的代。在Reactor 相同的功能：

Example 6. 与回代等的Reactor代示例

```
userService.getFavorites(userId) ①
    .flatMap(favoriteService::getDetails) ②
    .switchIfEmpty(suggestionService.getSuggestions()) ③
    .take(5) ④
    .publishOn(UiUtils.uiThreadScheduler()) ⑤
    .subscribe(uiList::show, UiUtils::errorPopup); ⑥
```

① 我从一个喜好ID的流始。

② 我地将它 的 **Favorite** 象(**flatMap**)。在我有了一个 **Favorite** 的流。

③ 如果 **Favorite** 流空，我降到 **suggestionService**。

④ 我最多只注果流中的五个元素。

⑤ 最后我理UI程中的个元素。

⑥ 我通描述最如何理数据的形式来触流（在UI列表中示）以及出的操作（示出口）。

如果想要保在800ms内索到喜好ID，当耗，从存中取？在基于回的代中，是一个的任。在Reactor中，在中添加一个 **timeout** 操作符就得非常，如下所示：

Example 7. 超和回退的Reactor代示例

```
userService.getFavorites(userId)
    .timeout(Duration.ofMillis(800)) ①
    .onErrorResume(cacheService.cachedFavoritesFor(userId)) ②
    .flatMap(favoriteService::getDetails) ③
    .switchIfEmpty(suggestionService.getSuggestions())
    .take(5)
    .publishOn(UiUtils.uiThreadScheduler())
    .subscribe(uiList::show, UiUtils::errorPopup);
```

① 如果以上部分在800ms内没有射出元素，播一个。

② 如果生，降到 **cacheService**。

③ 的其余部分与前面的示例似。

Future象比回微好一点，但依然以行合，尽管Java8中 **CompletableFuture**其做了改。排多个 **Future**象在一起然是可以行的，但并不容易。外，**Future**有其它的：

- 用 **get()** 方法很容易致 **Future**象出一个阻塞的情况。

- 不支持惰性 算。
- 乏 多 和高 理的支持。

考 外一个示例： 取到一个ID列表，我 需要 取其名称或 信息并将其 合，且所有的操作都是 的。下面的例子使用 `CompletableFuture` 型列表 行此操作：

Example 8. CompletableFuture 合示例

```
CompletableFuture<List<String>> ids = ifhIds(); ①

CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> { ②
    Stream<CompletableFuture<String>> zip =
        l.stream().map(i -> { ③
            CompletableFuture<String> nameTask = ifhName(i); ④
            CompletableFuture<Integer> statTask = ifhStat(i); ⑤

                return nameTask.thenCombineAsync(statTask, (name, stat) -> "Name "
+ name + " has stats " + stat); ⑥
            });
    List<CompletableFuture<String>> combinationList =
zip.collect(Collectors.toList()); ⑦
    CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);

    CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray);
⑧
    return allDone.thenApply(v -> combinationList.stream()
        .map(CompletableFuture::join) ⑨
        .collect(Collectors.toList()));
});

List<String> results = result.join(); ⑩
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");
```

① 从一个 定需要 理的 id 列表 始。

② 一旦得到列表，我 需要 一 的 理。

③ 遍 列表中的 个元素。

④ 取 的名称。

⑤ 取 的任 。

⑥ 合并 果。

⑦ 在，我 有了表示所有 合任 的future列表。 了 行 些任 ， 我 需要 列表 数 。

⑧ 将数 `CompletableFuture.allOf`， 出到一个 Future 象，然后当所有的任 都完成后完成。

⑨ 棘手的是，`allOf` 返回 `CompletableFuture<Void>`，我 重新遍 future列表，使用 `join()` 收集其 果（ 里不会阻塞，因 `allOf` 已 保 些futures已 完成）。

⑩ 一旦触 了整个 ， 我 就可以等待其 行 果返回，且断言返回的 果。

由于Reactor提供了更多 箱即用的 合 算符，一个 程可以被 化如下：

Example 9. 与future代 等 的Reactor代 示例

```
Flux<String> ids = ifhrIds(); ①

Flux<String> combinations =
    ids.flatMap(id -> { ②
        Mono<String> nameTask = ifhrName(id); ③
        Mono<Integer> statTask = ifhrStat(id); ④

        return nameTask.zipWith(statTask, ⑤
            (name, stat) -> "Name " + name + " has stats " + stat);
    });

Mono<List<String>> result = combinations.collectList(); ⑥

List<String> results = result.block(); ⑦
assertThat(results).containsExactly( ⑧
    "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"
);
```

① 一次，我 从已提供的一个 序列 `ids` (`Flux<String>`) 始。

② 于序列中的 个元素，我 理（在 用的 `flatMap` 函数内部） 次。

③ 取 的名称。

④ 取 的 信息。

⑤ 合并 个 果。

⑥ 在 可用 聚合到 `List` 中。

⑦ 在生 中，我 会 一 通 `Flux` 的 合或者 。一般情况下，我 会返回 `result Mono`。由于我 在用作 ，我 阻塞等待 果 理完成，然后直接返回聚合的 列表。

⑧ 断言 果。

使用回 和 `Future` 象的 点是相似的，且是 `Publisher-Subscriber` 的 式 程所要解决的 。

3.3. 从命令式到 式 程

如Reactor之 的 式，旨在解决JVM上 些 “ ” 的 方法的 点，同 注一些其它的方面：

- 合性和易性
- 数据作 流操作，且有着 富的操作符

- 在 之前什 都不会 生
- 背 或消 者向生 者 送信号表示 布速率太快
- 与并 无 的高 抽象

3.3.1. 合性和易 性

"可 合性"是指能 排多个 任 , 我 使用先前任 的 果将 入提供 后 任 。 外, 我 可以 以fork-join的形式 行多个任 。此外, 我 能 用 任 作 散 件到更高 次的系 中。

排任 的能力是与代 的可 性和可 性 密 合。随着 理的 数和 性的 加, 写和 代 得越来越困 。正如我 看到的一 , 整个回 模型是非常 的, 但是其主要的 点之一是, 于 的 理, 需要从一个回 中 行一个回 , 其本身嵌套在 一个回 中, 依此 推。 混乱被称 " 地 回 "。正如 所料到的(或从 得知), 如此的代 是相当 以回 和推理的。

Reactor提供了 富的 合 , 其中在代 中反映了抽象 程, 并且所有内容通常保持在同一 (少 嵌套)。

3.3.2. 比流水

可以将 式 用数据 理当作在 装流水 上流 。Reactor既是流水 又是工作站。原料来源于(原 初始的 **Publisher**) 并最 作 一个 品, 准 推送到消 者 (**Subscriber**)。

原料可以通 各 和其它的中 , 或者将中 零件聚合成更大的流水 的一部分。如果在 某一点出 故障或者堵塞(也 品装箱所需), 当前被影 到的工作站可以向上游 出信号来限 制原料的流 。

3.3.3. 操作符

在Reactor中, 操作符是我 流水 比中的工作站。 个操作符添加行 到 **Publisher** 中, 并将上一 的 **Publisher** 包装到新的 例中。因此, 整个 被 接在一起, 数据源于第一个 **Publisher** 沿着 向下移 , 并通 个 接 行 。最 , **Subscriber** 完成 理。 住, 正如我 很快会看到的, 在 **Subscriber** **Publisher** 之前, 什 都不会 生。



理解操作符 建新的 例可以 助 避免一个常 的 , 会 致 在 中 使用的 算符未被 用。 参 FAQ中 **item**。

尽管在 式流 中根本没有指定操作符, Reactor作 秀 式 中之一的添加了 些操作, 提供了 富的操作符。且 及了很多方面, 从 的 和 到 的 排和 理。

3.3.4. **subscribe()** 之前什 都不会 生

在Reactor中, 当 写一个 **Publisher** , 情况下不会 始注入数据。相反, 可以 建 理(有助于重用和 合) 的抽象描述。

通 , 可以将 **Publisher** 与 **Subscriber** 行 定, 从而触 整个 中的流数据。 是在内部 的, 通 个 **request** 信号从 **Subscriber** 播到上游, 一直 回到 **Publisher**。

3.3.5. 背

向上游 播信号也用作 背，我在装流水 比中将其描述，当工作站的理速度比上游工作站慢，沿生 向上 送反 信号。

式流 所定 的 机制与 比非常接近：一个 者可以以 无界 模式工作，并源以其最快的速率推送所有的数据，也可以使用 `request` 机制向源 送信号，表明已准 好最多 理 n个元素。

中 操作符也可以在中途改 求。想一个以十 一 将元素 行分 的 `buffer` 操作符。如果 者求一个`buffer`，源可以生成十个元素。一些操作符 了 提前 取 的策略，能 避免 `request(1)` 往返，如果在 求之前生 元素的成本不太高，那 策略是有益的。

会将推模式 推拉混合，当元素随 可用，下游可以从上游 取n个元素。但是如果 些元素没准 好， 当它 被生 ，就会被上游推送到下游。

3.3.6. vs 冷

Rx系列的 区分了 大 式序列： 和 冷， 区 主要与 式流如何 者作出反 相：

- 一个 冷 的序列会 个 `Subscriber` 都重新 始，包括数据源。例如，如果源包装了一个HTTP 用，会 个 起一个新的HTTP 求。
- 一个 的序列 个 `Subscriber` 并非是从 始。更 切地， 到的 者会在 后接收到送的信号。但是注意，有些 的 式流可以 存或者重置全部或部分下 史。从一般的角度来看，一个 的序列即使没有 者 于 听（ 于`` 之前什 都不会 生`` 是个例外），也能下。

于Reactor上下文中 与冷的更多信息， 看 `reactor` 章。

Chapter 4. Reactor核心特性

Reactor 目的主要 件 `reactor-core`, 一个 注于 式流 并基于Java8的 式 。

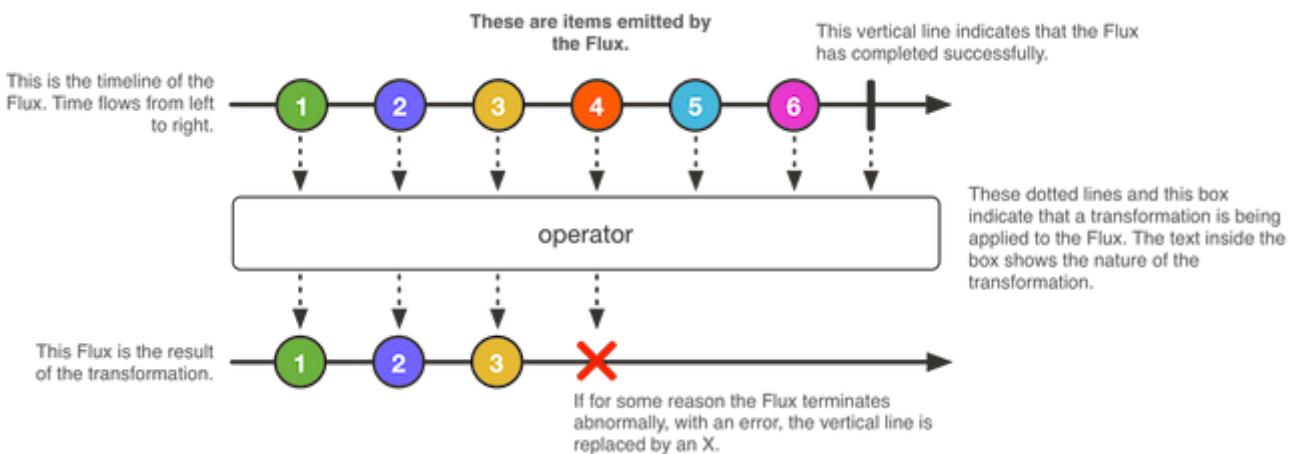
Reactor引入了可 合的 式 型， 些 型既 了 `Publisher` 又提供了 富的操作符：`Flux` 和 `Mono`。一个 `Flux` 象表示含有0..N个元素的 式序列。而一个 `Mono` 象表示 个 或 空(0..1) 的 果。

区 在 型中包含了一些 信息，表明 理的初略基数。例如，一个HTTP 求 一个 ，因 此 行 `count` 操作没有太大的意 。因此 于一次HTTP 用的 果表示 `Mono<HttpResponse>` 相 于 `Flux<HttpResponse>` 更有意 ，因 它 提供零 或者一 与上下文相 的的操作符。

操作符也能 到相 型，来改 要 理的最大基数。例如，`count` 操作符存在于 `Flux`，但它返回的是 `Mono<Long>`。

4.1. Flux, 一个包含0-N个元素的 序列

下 示了 `Flux` 如何 元素：

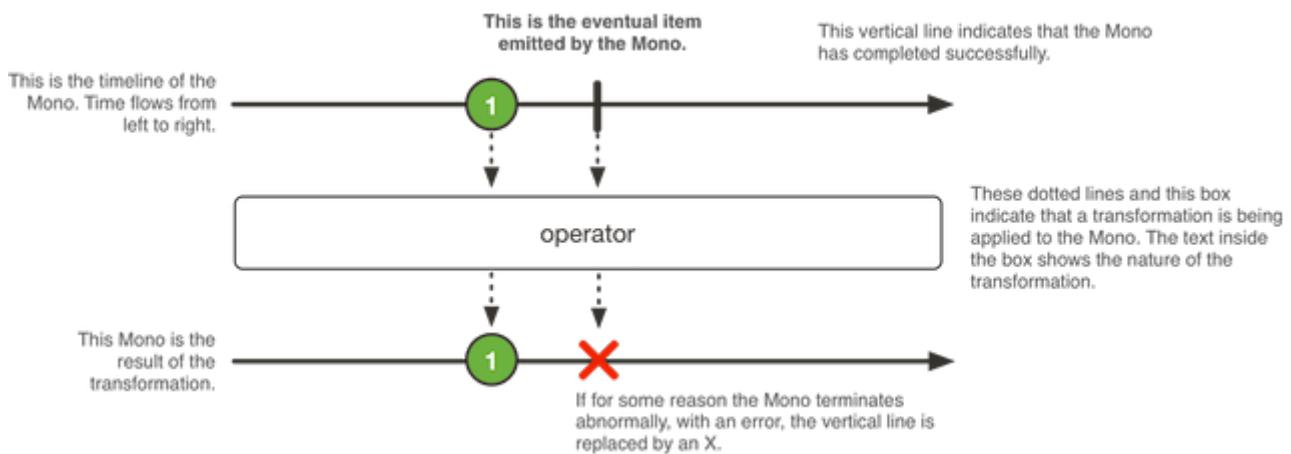


`Flux<T>` 是一个 准的 `Publisher<T>`，表示 出0到N个元素的 序列，可由完成或者 信号 行 性 止。在 式流 中， 三 型信号 用下游 者的 `onNext`，`onComplete`，和 `onError` 方法。

在如此之大可能信号的 的情况下，`Flux` 是通用的 式 型。 注意所有的事件，甚至是 止事件都是可 的：没有 `onNext` 事件但是有 `onComplete` 事件表示 一个 空 的有限序列，但去掉 `onComplete`，会有一个 无限 的空序列（除了 于取消的 外，并没有什 用）。同 ，无限序列并不一定 空的。例如，`Flux.interval(Duration)` 生成一个无 的且周期性的 出的 `Flux<Long>`。

4.2. Mono, 一个包含0-1 果的 序列

下 示了 `Mono` 如何 元素：



`Mono<T>` 是一个特定的 `Publisher<T>`, 最多可以 出一个元素, 可以被 `onComplete` 或 `onError` 信号性 止。

它 提供可用于 `Flux` 的子集操作符, 且一些操作符 (特 是那些将 `Mono` 和 一个 `Publisher` 合起来的操作符) 能 切 到 `Flux`。例如, `Mono#concatWith(Publisher)` 返回一个 `Flux`, 而 `Mono#then(Mono)` 返回 一个 `Mono`。

注意, 可以使用 `Mono` 来表示具有完成 (似于 `Runnable`) 概念的无 行 建。理。使用 `Mono<Void>` 来

4.3. 建和 Flux或Mono

使用各自 中 多的工厂方法之一是上手 `Flux` 和 `Mono` 的最 的方式。

例如, 要 建一个 `String` 序列, 可以枚 它 或者将它 放入到一个集合中, 并从中 建 `Flux`, 如下所示 :

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

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

其它工厂方法示例包括如下 :

```
Mono<String> noData = Mono.empty(); ①

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

Flux<Integer> numbersFromFiveToSeven = Flux.range(5, 3); ②
```

① 注意, 即使整个工厂方法没有任何 , 也需要 注泛型 型。

② 第一个参数是 的 始, 第二个参数是 生元素的个数。

在 `Flux` 和 `Mono` 使用 Java8 的 lambda 法。可以各各的 `.subscribe()` 形式，将 lambda 使用于不同的回合，如下面的方法名所示：

Example 10. 基于 Lambda 的 `Flux` 的形式

```
subscribe(); ①  
  
subscribe(Consumer<? super T> consumer); ②  
  
subscribe(Consumer<? super T> consumer,  
          Consumer<? super Throwable> errorConsumer); ③  
  
subscribe(Consumer<? super T> consumer,  
          Consumer<? super Throwable> errorConsumer,  
          Runnable completeConsumer); ④  
  
subscribe(Consumer<? super T> consumer,  
          Consumer<? super Throwable> errorConsumer,  
          Runnable completeConsumer,  
          Consumer<? super Subscription> subscriptionConsumer); ⑤
```

① 并触序列。

② 个生的做一些操作。

③ 理也会作出反。

④ 理和，且在序列成功完成行一些代。

⑤ 理和以及成功完成，但也要理 `subscribe` 用生的 `Subscription`。



些不同的形式会返回的引用，当不需要更多的数据，可以通过引用取消。取消后，源停止生数据并清除它所建的任何源。在 Reactor 中，使用通用 `Disposable` 接口来表示取消和清理行。

4.3.1. `subscribe` 方法示例

本包含了 `subscribe` 方法的五个名的最示例。以下代码示了一个没有参数的基本方法的例子：

```
Flux<Integer> ints = Flux.range(1, 3); ①  
ints.subscribe(); ②
```

① 置一个 `Flux`，在者接生成三个。

② 用最的方式。

前面的代码不会生任何可的出，但它起作用。`Flux`生三个。如果我提供一个 lambda，可以些可。`subscribe` 方法的下一个示例示了一使示的方法：

```
Flux<Integer> ints = Flux.range(1, 3); ①  
ints.subscribe(i -> System.out.println(i)); ②
```

① 置一个 Flux, 在者接生成三个。

②用能打印的或者行。

前面的代码生成以下输出：

```
1  
2  
3
```

了演示下一个方法名，我故意引入一个，如下例所示：

```
Flux<Integer> ints = Flux.range(1, 4) ①  
.map(i -> { ②  
    if (i <= 3) return i; ③  
    throw new RuntimeException("Got to 4"); ④  
});  
ints.subscribe(i -> System.out.println(i), ⑤  
    error -> System.err.println("Error: " + error));
```

① 置一个 Flux, 在者接生成四个。

②我需要一个 map, 我就可以一些行不同的理。

③于大多数，返回。

④于一个，制生。

⑤用包括理的或者行。

我在有 个lambda表达式：一个是我期望的内容，一个是。前面的代码生成以下输出：

```
1  
2  
3  
Error: java.lang.RuntimeException: Got to 4
```

subscribe 方法的下一个名包括理和完成事件理，如下示例所示：

```
Flux<Integer> ints = Flux.range(1, 4); ①
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> System.out.println("Done")); ②
```

① 置一个 `Flux`, 在 者 接 生成四个 。

② 用包含完成事件 理的 者 行 。

信号和完成信号都是 止事件, 并且彼此互斥 (永 不会同 得到 个信号)。 了完成消 , 我必 注意不要触 。

完成回 没有 入, 用一 空括号表示 : 它与 `Runnable` 接口中的 `run` 方法相匹配。上面的代 生以下出 :

```
1
2
3
4
Done
```

`subscribe` 方法的最后一个 名包括一个 `Consumer<Subscription>`。



形式要求 `Subscription` (其 行 `request(long)` 或 `cancel()` 做一些事情。否 `Flux` 会挂起。

以下示例 示了 `subscribe` 方法的最后一个 名 :

```
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(10)); ①
```

① 当我 , 我 会收到一个 `Subscription`。表示我 最多希望从源 (上会 出 4个元素并完成) 中接收到 10 个元素。

4.3.2. 用 `Disposable` 取消 `subscribe()`

所有 些基于lambda的 `subscribe()` 形式都有一个 `Disposable` 返回 型。在 情况下, `Disposable` 接口表示 都可以通 用其 `dispose()` 方法来 取消 。

于 `Flux` 或 `Mono` 来 , 取消是源 停止 生元素的信号。然而, 并不能保 立即 行 : 某些源可能

生元素的速度太快，以至于它甚至可以在收到取消指令之前就能完成。

`Disposables` 中提供了一些于 `Disposable` 的工具集。其中，`Disposables.swap()` 建了一个 `Disposable` 包装器，可以原子地取消和替一个具体的 `Disposable`。在 UI 景中会非常有用，例如，当用按，想取消求并将其替新的求。理包装本身会它。做会理掉当前的具体和所有未来的替。

一个有趣的工具方法是 `Disposables.composite(...)`。一个合可以收集多个 `Disposable` 一例如，与一个服用相的多个行中的求一并在后一次性理所有些求。一旦合的 `dispose()` 方法被用，任何添加一个 `Disposable` 的都会立即理它。

4.3.3. Lambda的替代方案：`BaseSubscriber`

外有一个更通用的 `subscribe` 方法，它接受一个完整的 `Subscriber`，而不是用lambda成一个。了助写我的 `Subscriber`，我提供了一个名 `BaseSubscriber` 的可展。



`BaseSubscriber`（或其子）的例是一用途，意味着，如果一个 `BaseSubscriber` 了第二个 `Publisher`，会取消第一个 `Publisher` 的。是因使用一个例次反了式流的，即 `Subscriber` 的 `onNext` 方法必不能被并行用。因此，只有在 `Publisher#subscribe(Subscriber)` 的用中直接声明匿名，匿名才是不的。

在我可以其中的一个。我称之为 `SampleSubscriber`。下面的例子示了如何将其附加到 `Flux`：

```
SampleSubscriber<Integer> ss = new SampleSubscriber<Integer>();
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> {System.out.println("Done");},
    s -> s.request(10));
ints.subscribe(ss);
```

下面的例子示了 `SampleSubscriber` 作 `BaseSubscriber` 的子：

```
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);
    }
}
```

`SampleSubscriber` 继承了 `BaseSubscriber`，在Reactor中，它作用于自定义的 `Subscribers` 的推抽象。一个子类提供了可以被重写并以此来整合作者行的子类。情况下，它会触发一个无界的请求，且行为方式与 `subscribe()` 完全相同。然而，当想要一个自定义的求求数量，扩展一个 `BaseSubscriber` 会更有用。

对于一个自定义的求求数量，最常见的是像我一样实现 `hookOnSubscribe(Subscription subscription)` 和 `hookOnNext(T value)`。在我自己的例子中，`hookOnSubscribe` 方法打印一条消息到控制台并发出第一个请求。然后 `hookOnNext` 方法打印一条消息并执行其它的请求，依次一个请求。

`SampleSubscriber` 生成以下输出：

```
Subscribed
1
2
3
4
```

`BaseSubscriber` 提供了一个 `requestUnbounded()` 方法来切换到无界模式（相当于 `request(Long.MAX_VALUE)`），以及一个 `cancel()` 方法。

它具有其它子类：`hookOnComplete`, `hookOnError`, `hookOnCancel`, 和 `hookFinally`（是在序列终止被调用，终止型作 SingalType 参数）。



几乎肯定要调用 `hookOnError`, `hookOnCancel`, 和 `hookOnComplete` 方法。可能还想调用 `hookFinally` 方法。`SampleSubscribe` 是一个有界的 `Subscriber` 的最小实现。

4.3.4. 于背和整求的方法

在Reactor中，于背，通向上游操作符发送 `request`, 将消息的力量回到源端。当前求的和有被称当前“需求”，或者“等待的求”。求的上限 `Long.MAX_VALUE`, 表示无限制的求（意思是“尽可能快的生成”——基本上是禁用了背压）。

第一个求来自于最前者，然而最直接的方式都会立即触发一个 `onError`。求的无界求：

- `subscribe()` 和大多数基于lambda的形式（具有 `Consumer<Subscription>` 的除外）
- `block()`, `blockFirst()` 和 `blockLast()`
- 在 `toIterable()` 或 `toStream()` 上迭代

自定义原始求最求的方法是使用重写了 `hookOnSubscribe` 方法的 `BaseSubscriber` 来实现 `subscribe`, 如下例所示：

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

前面的代码片段打印出以下内容：

```
request of 1
Cancelling after having received 1
```



在理一个求，必小心生足的需求以使序列前，否 Flux可能会被“住”。就是什 `BaseSubscriber` 在 `hookOnSubscribe` 无限的求。当重写个子，通常至少用一次 `request`。

改下游求的操作符

有一点要住的是，在表的求可以被上游中的个操作符重新整。一个例就是 `buffer(N)` 操作符：如果它接收到 `request(2)`，解需要个完整冲区。因此，由于冲区需要 `N` 个元素才能被已足求，因此 `buffer` 操作符将求重新整 $2 \times N$ 。

可能注意到有些操作符采用名 `prefetch` 的 `int` 入参数的形式。是一修改下游求的操作符。它通常用于理内部序列，从入的个元素（如 `flatMap`）中派生出 `Publisher`。

`Prefetch` 是些内部序列出的初始求行整的一方法。如果未指定，些操作符大多都以 `32` 的需求始。

些操作符通常了充化：一旦操作符看到75%的取求已完成，它将从上游重新求75%。是一式化，使些操作符能主即将到来的求。

最后，有个操作符可以直接整求：`limitRate` 和 `limitRequest`。

`limitRate(N)` 下游求行拆分，以便将其分批次向上游播。例如，`limitRate(10)` 出 `100` 的求会致最多 `10` 个求，分 `10` 个批次播到上游。注意，在形式下，`limitRate` 上了前面的充化。

操作符有一个可以整充数量（称 `lowTide`）的形式：`limitRate(highTide, lowTide)`。`lowTide 0` 将致格的批次 `highTide` 求，而不是由充策略一整批次。

一方面，`limitRequest(N)` 限制将下游求的最大求量。它将求累加到 `N`。如果一个 `request` 没有超的求 `N`，特定求将完全播到上游。在源出求量后，`limitRequest` 将序列已完成，向下游送 `onComplete` 信号，并取消源。

4.4. 以程方式建序列

在一中，我将通以程方式定其的事件（`onNext`, `onError`, 和 `onComplete`）来介 Flux 或 Mono 的建。所有些方法都有一个共同点，即它暴露了一个API来触我称之为 sink 的事件。上有一些sink的形式，后我将介。

4.4.1. 同 `generate`

Flux 的最的程建形式是通具有生成器功能的 `generate` 方法。

用于 `synchronous` 和一一射，意味着 sink是一个 `SynchronousSink`，并且次回用最多只能用其 `next()` 方法一次。然后可以外用 `error(Throwable)` 或 `complete()`，但是可的。

最有用的形式可能是一能保留一个状，可以在 sink的使用中参考个状来决定下一出什。然后生成器函数成了 `BiFunction<S, SynchronousSink<T>, S>`，其中 `<S>` 是状象的型。必初始状提供一个 `Supplier<S>`，的生成器函数在都会返回一个新的状。

例如， 可以使用 `int` 作 状：

Example 11. 基于状 `generate` 的示例

```
Flux<String> flux = Flux.generate(  
    () -> 0, ①  
    (state, sink) -> {  
        sink.next("3 x " + state + " = " + 3*state); ②  
        if (state == 10) sink.complete(); ③  
        return state + 1; ④  
    });
```

- ① 我 提供初始的状 0。
- ② 我 用状 来 要 出什 (3的乘法表中的一行)。
- ③ 我 可以用它来 什 时候停止。
- ④ 我 返回一个新的状，在下一个 用中使用 (除非 个序列在 个 用中 止)。

前面的代 生成3的乘法表，如下所示：

```
3 x 0 = 0  
3 x 1 = 3  
3 x 2 = 6  
3 x 3 = 9  
3 x 4 = 12  
3 x 5 = 15  
3 x 6 = 18  
3 x 7 = 21  
3 x 8 = 24  
3 x 9 = 27  
3 x 10 = 30
```

可以使用可 的 `<S>`。例如，上面的例子可以使用一个 独的 `AtomicLong` 作 状 来重写，在 一其 行修改：

Example 12. 可 状 形式

```
Flux<String> flux = Flux.generate(  
    AtomicLong::new, ①  
    (state, sink) -> {  
        long i = state.getAndIncrement(); ②  
        sink.next("3 x " + i + " = " + 3*i);  
        if (i == 10) sink.complete();  
        return state; ③  
    });
```

① 次，我 生成一个可 的 象作 状 。

② 我 在 里改 状 。

③ 我 返回相同 例作 新状 。



如果 的状 象需要清理一些 源， 使用 generate(Supplier<S>, BiFunction, Consumer<S>) 形式来清理最后的状 例。

下面的例子使用来包含一个 Consumer 的 generate 方法：

```
Flux<String> flux = Flux.generate(  
    AtomicLong::new,  
    (state, sink) -> { ①  
        long i = state.getAndIncrement(); ②  
        sink.next("3 x " + i + " = " + 3*i);  
        if (i == 10) sink.complete();  
        return state; ③  
    }, (state) -> System.out.println("state: " + state)); ④  
}
```

① 同 ，我 生成一个可 象作 状 。

② 我 在 里改 状 。

③ 我 返回相同 例作 新状 。

④ 我 把最后一个状 (11) 个 Consumer lambda的 出。

在包含数据 接或其他 源的状 需要在 程 束 理的情况下，Consumer lambda可以 接或以其它方式 理任何 在 程 束 完成的任 。

4.4.2. 和多 程：create

create是一 更高 的 程方式 建 Flux 的形式，它 合 多次 出，甚至来自多个 程。

它暴露了一个 FluxSink 及其 next, error 和 complete 方法。与 generate 相反，它没有基于状 的形式。但是，它可以在回 中触 多 程事件。



`create` 将 有的API和 式世界 接起来非常有用 - 例如基于 听器的 API。



`create` 不能使 的代 并行化 , 也不能使其 化, 即使它 可以 与 API一起使用。如果在 `create` lambda中阻塞了, 会使自己陷入死 和似副作用中。即使使用了 `subscribeOn`, 也需要警 , 即 阻塞的 `create` lambda (比如无限循 的 用 `sink.next(t)`) 会 住管道: 些 永 不会 行, 因 循 会耗尽 行它 的 程。使用 `subscribeOn(Scheduler, false)` 的形式: `requestOnSeparateThread = false` 将使用 `Scheduler` 程来 行 `create` , 并且 然可以通 在原始的 程中 行 `request` 来 数据流 。

假 使用了基于 听器的API。它按 理数据, 并有 个事件 : (1) 数据 已准 就 ; (2) 理已 完成 (止事件) , 如 `MyEventListener` 接口所示 :

```
interface MyEventListener<T> {  
    void onDataChunk(List<T> chunk);  
    void processComplete();  
}
```

可以使用 `create` 将其 接成 `Flux<T>` 中 :

```
Flux<String> bridge = Flux.create(sink -> {  
    myEventProcessor.register(④  
        new MyEventListener<String>() { ①  
  
            public void onDataChunk(List<String> chunk) {  
                for(String s : chunk) {  
                    sink.next(s); ②  
                }  
            }  
  
            public void processComplete() {  
                sink.complete(); ③  
            }  
        });  
    });
```

① 接到 `MyEventListener` API

② 中的 个元素都会成 `Flux` 中的一个元素。

③ `processComplete` 事件被 `onComplete`。

④ 所有 些都是在 `myEventProcessor` 行 完成的。

此外, 由于 `create` 可以 接 API并管理背 , 所以 可以通 声明 `OverflowStrategy` 来 化背 的行 :

- **IGNORE** 完全忽略下游背求。当下游列，可能会生 `IllegalStateException`。
- **ERROR** 当下游无法跟上，出 `IllegalStateException` 信号。
- **DROP** 如果下游没有准好接收信号，入的信号。
- **LATEST** 下游只能得来自上游的最新信号。
- **BUFFER** 在下游无法跟上，冲所有的信号（将无限制的冲，可能会致 `OutOfMemoryError`）。



`Mono` 也有一个 `create` 生成器。`Mono` 建的 `MonoSink` 不允多次出。它将在第一个信号之后所有信号。

4.4.3. 程：`push`

`push` 介于 `generate` 和 `create` 之，用于理来自个生者的事件。从某意上，它似于 `create`，因它也可以是的，并且可以使用所支持的任何溢出策略来管理背。但是，只能有一个生程可以用 `next`, `complete` 或者 `error`。

```
Flux<String> bridge = Flux.push(sink -> {
    myEventProcessor.register(
        new SingleThreadEventListener<String>() { ①

            public void onDataChunk(List<String> chunk) {
                for(String s : chunk) {
                    sink.next(s); ②
                }
            }

            public void processComplete() {
                sink.complete(); ③
            }

            public void processError(Throwable e) {
                sink.error(e); ④
            }
        );
    });
});
```

① 接到 `SingleThreadEventListener` API。

② 使用一个听器程中的 `next` 将事件推送到 `sink`。

③ 由同一听器程生成的 `complete` 事件。

④ `error` 事件也是由同一听器程生成的。

混合式推/拉模型

大多数式操作符，比如 `create`，都遵循混合的推/拉模型。我的意思是，尽管大多数理都是的（建使用推方法），但也有一个小的拉件：求。

从源来看，消费者从源拉取数据，直到第一次请求之前，它不会输出任何东西。只要有可用的数据，源会向消费者推送数据，但会在其请求数量的范围内。

注意，`push()` 和 `create()` 都允许多个一个 `onRequest` 消费者，以便于管理请求数量，并确保只有当有对应的请求时，才将数据推送到sink中。

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

            public void onMessage(List<String> messages) {
                for(String s : messages) {
                    sink.next(s); ③
                }
            }
        });
    sink.onRequest(n -> {
        List<String> messages = myMessageProcessor.getHistory(n); ①
        for(String s : message) {
            sink.next(s); ②
        }
    });
});
```

① 请求消息。

② 如果消息立即可用，将其推送到接收器。

③ 后到的其余消息也将被忽略。

在 `push()` 或 `create()` 之后清理

一个回调，`onDispose` 和 `onCancel`，在取消或终止时执行任何清理。`onDispose` 可用于 `Flux` 完成、或取消时执行清理。`onCancel` 可以用于在使用 `onDispose` 执行清理之前，执行任何特定于取消的操作。

```
Flux<String> bridge = Flux.create(sink -> {
    sink.onRequest(n -> channel.poll(n))
        .onCancel(() -> channel.cancel()) ①
        .onDispose(() -> channel.close()) ②
});
```

① `onCancel` 首先被调用，用于取消信号。

② 完成、或取消信号时用 `onDispose`。

4.4.4. 理

`handle`

方法有点不同：它是一个重载方法，意味着它被接到一个有的源（就像常

的操作符) 上。它存在于 `Mono` 和 `Flux` 中。

它 近于 `generate`, 从某 意 上 , 它使用 `SynchronousSink` 并只允 逐个的 出。但是, `handle` 可以用来从 一个源元素中生成一个任意 , 可能会跳 一些元素。 , 它可以作 `map` 和 `filter` 的 合。`handle`的方法 名如下 :

```
Flux<R> handle(BiConsumer<T, SynchronousSink<R>>);
```

我 考 一个例子。 式流 不允 序列中的 `null`。但是 想使用一个 先存在的方法作 `map`函数来 行 `map`, 而 方法有 返回`null` ?

例如, 下面的方法可以安全地 用于整数源 :

```
public String alphabet(int letterNumber) {  
    if (letterNumber < 1 || letterNumber > 26) {  
        return null;  
    }  
    int letterIndexAscii = 'A' + letterNumber - 1;  
    return "" + (char) letterIndexAscii;  
}
```

然后, 我 可以使用 `handle` 来 除任何空 :

Example 13. `handle` 用于 “映射和消除`null` ” 的 景

```
Flux<String> alphabet = Flux.just(-1, 30, 13, 9, 20)  
.handle((i, sink) -> {  
    String letter = alphabet(i); ①  
    if (letter != null) ②  
        sink.next(letter); ③  
});  
  
alphabet.subscribe(System.out::println);
```

① 映射到字母。

② 如果 “`map`函数” 返回`null`....

③ 通 不 用 `sink.next` 来 掉它。

将打印出 :

4.5. 程和 度器

Reactor，就像RxJava一样，可以被看作是可并行的。也就是说，并没有线程的并行模型。相反，它人掌握主线程。然而并不妨碍框架解决并行问题。

得 Flux 或 Mono 并不一定意味着它要在特定的线程上运行。相反，大多数操作符是在上一个操作符运行的 Thread 中工作。除非指定，否则最上面的操作符（源）本身运行在用了 subscribe() 的 Thread 上。下面的示例在一个新的线程上运行 Mono：

```
public static void main(String[] args) throws InterruptedException {
    final Mono<String> mono = Mono.just("hello "); ①

    Thread t = new Thread(() -> mono
        .map(msg -> msg + "thread ")
        .subscribe(v -> ②
            System.out.println(v + Thread.currentThread().getName()) ③
        )
    )
    t.start();
    t.join();

}
```

① Mono<String> 在 main 程序中声明。

② 但是，它是在 Thread-0 线程中运行的。

③ 因此，map 和 onNext 回调都是在 Thread-0 中运行。

上面的代码生成以下输出：

```
hello thread Thread-0
```

在Reactor中，线程模型和线程的位置由使用的 Scheduler 决定。Scheduler具有类似于 ExecutorService 的调度度，但有一个更强大的抽象使其可以做更多的事情，尤其是作为一个线程池和更广的调度器（虚拟，波动或立即调度等）。

The Schedulers 有可以运行上下文的静态方法：

- 无 行上下文 (`Schedulers.immediate()`) : 在 理 , 提交的 `Runnable` 将被直接 行, 有效地在当前的 `Thread` 上 行它 (可以 “空 象”或无操作的 `Scheduler`)。
- 一个 一可重用的 程 (`Schedulers.single()`)。注意, 此方法 所有 用者会重用相同的 程, 直到 度器 掉。如果 希望 次 用都有一个特定的 程, 使用 `Schedulers.newSingle()` 即可。
- 一个无界的 性 程池 (`Schedulers.elastic()`)。由于 `Schedulers.boundedElastic()` 的引入, 个 程池不再是首 的了, 因 它容易 藏背 而 致 程 多 (下文)。
- 一个有 界的 性 程池 (`Schedulers.boundedElastic()`)。就像它的前身 `elastic()` 一 , 它根据需要 建新的和 用空 的 程池。空 (60s) 的 程池也会被 。与 之前的 `elastic()` 不同的是, 建可支持的 程数有上限 (CPU核数 x 10)。最多可提交 10万个任 , 到上限后, 将在 程可用 重新 度任 (当延 度 , 延 在 程可用 始)。 于I/O阻塞任 , 是更好的 。`Schedulers.boundedElastic()` 是一 在自己 程上 行阻塞 理的便捷方式, 它不会占用其它的 源。 如何包装一个同 阻塞 用?, 但不会 系 来太多新 的 程 力。
- 一个固定worker池, 整 并行工作 (`Schedulers.parallel()`)。它 建和CPU核数相等的worker。

此外, 可以用 有的 `ExecutorService` 通 `Schedulers.fromExecutorService(ExecutorService)` 建一个 `Scheduler`。(也可以使用 `Executor` 行 建, 但不建 做。)

也可以通 使用 `newXXX` 方法 建各 度器 型的新 例。例如, `Schedulers.newParallel(yourScheduleName)` 建了一个新的并行 度器, 命名 `yourScheduleName`。



然 `boundedElastic` 是 了在无法避免的情况下 助 留的阻塞代 , 但 `single` 和 `parallel` 不是。因此, 使用Reactor阻塞API (在 的 度器和并行 度器内 `block()`, `blockFirst()`, `blockLast()` (以及迭代 `toIterable()` 或 `toStream()`)) 会 致 出 `IllegalStateException`。

通 建 `NonBlocking` 接口的 `Thread` 例, 自定 的 度器 也可以被 “非阻塞”。

一些操作符 使用 `Schedulers` 中特定的 度器 (通常会 提供一个不同的 度器)。例如, 用 `Flux.interval(Duration.ofMillis(300))` 工厂方法会 隔300ms 生一个 `Flux<Long>`。 情况下, 是由 `Schedulers.parallel()` 用的。下面 一行将 度器改 似于 `Schedulers.single()`的新 例。

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

Reactor提供了 方式来切 式 中的 行上下文 (或 `Scheduler`) : `publishOn` 和 `subscribeOn`。 者都取一个 `Scheduler`, 并 将 行上下文切 到 度器。但是 `publishOn` 在 中的位置是很重要的, 然而 `subscribeOn` 的位置却是无 要的。要理解 个区 , 首先要 住 之前什 都不会 生。

在Reactor中, 当 用 接操作符 , 可以根据需要在内部封装尽可能多的 `Flux` 和 `Mono` 。一旦 了, 一个 `Subscriber` 象的 就会 建出来, 向后 (沿着 向上) 到第一个生 者。 上是 藏掉的。 能看到的只是外 到 `Flux` (或 `Mono`)和 `Subscription`, 但是 些中 的操作符的 才是真正的工作。

有了些知，我可以仔看看 `publishOn` 和 `subscribeOn` 操作符：

4.5.1. `publishOn` 方法

和其它操作符一，`publishOn` 用在 的中 位置。它接收来自上游的信号，并在下游重放，同

在相的 `Scheduler` 中某个worker 行回。因此它影到后 操作符的 行（直到 中的一个 `publishOn`），具体如下：

- 将 行上下文改 由 `Scheduler` 的一个 `Thread`
- 根据 ，`onNext` 依次 用，所以 就占用了一个 程
- 除非它 在特定的 `Scheduler` 上工作，否 在 `publishOn` 之后的操作符将 在同一 程上 行

下面的示例使用了 `publishOn` 方法：

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4); ①

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i) ②
    .publishOn(s) ③
    .map(i -> "value " + i); ④

new Thread(() -> flux.subscribe(System.out::println)); ⑤
```

① 建一个包含四个 `Thread` 例的新的 `Scheduler`。

② 第一个 `map` 操作符在<5>的匿名 程中 行。

③ `publishOn` 将整个序列切 到从<1>中的 `Thread` 上。

④ 第二个 `map` 操作符在<1>的 `Thread` 上 行。

⑤ 个匿名 `Thread` 是 生 地方，打印是在最近的 行上下文中 生的，也就是 `publishOn` 中的那个。

4.5.2. `subscribeOn` 方法

当向下的 被 造，`subscribeOn` 用在 理上。因此，无 将 `subscribeOn` 放在 中的 个位置，始 会影 到源排放的上下文。然而， 不会影 到后 用 `publishOn` 的行 — 它 将 其之后的 行上下文切 。

- 改 整个 所 的 `Thread`
- 从 `Scheduler` 一个 程



只有 中的最早的 `subscribeOn` 用才会被 考 在内。

下面的例子使用了 `subscribeOn` 方法：

```

Scheduler s = Schedulers.newParallel("parallel-scheduler", 4); ①

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i) ②
    .subscribeOn(s) ③
    .map(i -> "value " + i); ④

new Thread(() -> flux.subscribe(System.out::println)); ⑤

```

- ① 建一个包含四个 `Thread` 例的新的 `Scheduler`。
- ② 第一个 `map` 操作符 行在四个 程中之一...
- ③ ...因 `subscribeOn` 从 <5> 始就会切 整个序列。
- ④ 第二个 `map` 也 行在相同的 程上。
- ⑤ 个匿名的 `Thread` 是最初 生 _ 的地方，但 `subscribeOn` 立即将其 移到 度器的四个 程之一。

4.6. 理



快速 看 于 理的操作符，参看相 操作符决策 。

在 式流中， 是 止事件。一旦 生，就会停止序列，并沿着操作符 向下 播到最后一 ，即 定 的 `Subscriber` 及其 `onError` 方法。

此 在 用程序 面 理。例如， 可以在UI中 示 通知或者在REST端点中 送一个有意 的 。因此， 者的 `onError` 方法 是被定 的。



如果没有定 ， `onError` 会 出一个 `UnsupportedOperationException` 常。 可以使用 `Exceptions.isErrorCallbackNotImplemented` 方法 一 和分 。

Reactor 提供了 理 中 理 的替代方法，即 操作符，下面的例子 示了如何做到 一点：

```

Flux.just(1, 2, 0)
    .map(i -> "100 / " + i + " = " + (100 / i)) // 将触 一个0的
    .onErrorReturn("Divided by zero :("); // 理例子

```



在学 理操作符之前， 必 住 式序列中的任何 都是一个 止事件。即使使用了 理操作符，它也不会 原来序列 行。相反，它将 `onError` 信号 一个新的序列（降 序列）的 始。 句 ，它会替 了 上游 止序列。

在我 可以逐一考 一 理的方式。当的 候，我 将与命令式 程的 **try** 模式并行使用。

4.6.1. 处理操作符

可能 用try-catch 捕 常的几 方法比 熟悉。最 得注意的是， 些方法包含以下几 :

- 捕 并返回一个静 。
- 捕 常并 行一个降 方法。
- 捕 常并 地 算一个降 的 。
- 捕 常， 封装成一个 **BusinessException**， 然后重新 外 出。
- 捕 常， 打印 的具体信息，并重新 外 出 常。
- 使用 **finally** 或Java 7以上支持的“try-with-resource” 法清理 源。

Reactor中，所有的 些方法都以 处理操作符的形式且具有相同的效果。 在深入 些操作符之前，我 首先要在 式 和try-catch 之 建立 的 系。

当 ， 在 的末端的 **onError** 回 似于一个 **catch** 。在 里，当 出一个 **Exception** ， 行会跳 到catch，如下面示例所示：

```
Flux<String> s = Flux.range(1, 10)
    .map(v -> doSomethingDangerous(v)) ①
    .map(v -> doSecondTransform(v)); ②
s.subscribe(value -> System.out.println("RECEIVED " + value), ③
            error -> System.err.println("CAUGHT " + error) ④
);
```

- ① 行了可能引 常的 。
- ② 如果一切 利， 行第二次 。
- ③ 个成功 的 都会打印出来。
- ④ 当 生 ， 止序列且 示 信息。

前面的例子在概念上与下面的try-catch 似：

```

try {
    for (int i = 1; i < 11; i++) {
        String v1 = doSomethingDangerous(i); ①
        String v2 = doSecondTransform(v1); ②
        System.out.println("RECEIVED " + v2);
    }
} catch (Throwable t) {
    System.err.println("CAUGHT " + t); ③
}

```

- ① 如果 里出 常
- ② ...跳 循 的其余部分...
- ③ ...直接 行到 里。

既然我 已 建立了 的 系，我 就可以看不同的 理情况及其等效的操作符。

静 降 的

`onErrorReturn` 等效于“捕 并返回一个静 ”。下面的例子 示了如何使用它：

```

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

```

下面的例子展示了在Reactor中相同的效果：

```

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

```

可以在 常上 用一个 `Predicate` 来决定是否恢 ，如下面的例子所示：

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn(e -> e.getMessage().equals("boom10"), "recovered10"); ①
```

① 当 常信息 "boom10" 返回

降 方法

如果 想要多个，并且有其他的（更安全的）方式 理数据， 可以使用 `onErrorResume`。相当于“捕 常并 行一个降 方法”。

例如，如果 名 上的 程正在从外部且不可 的服 中 取数据，但 也保留了一个相同数据的本地 存，而 些数据也 有点 期但是更可 ， 可以做以下操作：

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

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

下面的例子展示了在Reactor中相同的效果：

```
Flux.just("key1", "key2")
    .flatMap(k -> callExternalService(k) ①
        .onErrorResume(e -> getFromCache(k)) ②
    );

```

① 于 个 ， 用外部服 。

② 如果外部服 用失 ， 降 取 的 存中。注意，无 源的 是什 e 常，我 是 用相同的降 。

和 `onErrorReturn` 一， `onErrorResume` 有不同的形式 根据 常的 型或 `Predicate` 来 些 常需要降 。事 上，它需要一个 `Function`， 也 可以根据遇到的不同的 来 不同的降

序列来 行切 。下面的例子 示了如何做到 一点：

```
Flux.just("timeout1", "unknown", "key2")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(error -> { ①
        if (error instanceof TimeoutException) ②
            return getFromCache(k);
        else if (error instanceof UnknownKeyException) ③
            return registerNewEntry(k, "DEFAULT");
        else
            return Flux.error(error); ④
    })
);
```

① 函数可以 如何 。

② 如果源超 ， 本地 存。

③ 如果源中 未知， 建新的 象。

④ 其它的所有情况下，“重新 出 常”。

降 的

即使 没有其它（更安全的）的数据 理的方式， 可能也想从 收到的 常中 算出一个降 的 。 就相当于“捕 常并 地 算一个降 的 ”。

例如，如果 的返回 型 (`MyWrapper`) 有一个 用来保存 常的形式（参考 `Future.complete(T success)` 与 `Future.completeExceptionally(Throwable error)` ）， 可以 例化 保持 量并常。

一个命令式 程示例如下所示：

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

使用 `onErrorResume`， 可以像降 方法解决方案以相同的方式 行 式操作，略作修改，如下所示：

```
erroringFlux.onErrorResume(error -> Mono.just(①
    MyWrapper.fromError(error) ②
));
```

① 因 期望 `MyWrapper` 来表示 ， 所以 需要 `onErrorResume` 取一个 `Mono<MyWrapper>`，我 用 `Mono.just()` 来 。

② 我 需要 算出 常 。 里，我 通 使用相 的 `MyWrapper` 工厂方法 常 行包装来 。

捕 并重新 出 常

"捕 常，封装成一个 `BusinessException`，然后重新 外 出"，在命令式 程里面看起来就像下面 :

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

在 "降 方法" 示例中， `flatMap` 中的最后一行 了我 一个提示， 我 同 的 式操作，具体如下：

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

但是，有一 更直接的方法，可以使用 `onErrorMap` 到同 的效果：

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

面 日志或

如果 想 播，但 想在不修改序列的情况下 做出 (例如 日志)， 可以使用 doOnError 操作符。 相当于 “捕 常，打印 的具体信息，并重新 外 出 常” 的模式，如下面的例子所示：

```
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;
}
```

doOnError 操作符以及所有以 doOn 前 的操作符，有 被称 “ 面效 ”。它可以在不修改序列事件的情况下 到序列内部的事件。

就像前面的命令式 程例子一 ，下面的例子 然会 播 ，但至少可以 保我 到外部服 生了 故障。

```
LongAdder failureStat = new LongAdder();
Flux<String> flux =
Flux.just("unknown")
    .flatMap(k -> callExternalService(k) ①
        .doOnError(e -> {
            failureStat.increment();
            log("uh oh, falling back, service failed for key " + k); ②
        })
    ③
);
```

① 可能失 的外部服 用...

② ...被装 了日志和 的 面效果...

③ ...之后，它 然以 止，除非我 在 里使用 恢 操作符。

我 可以 想，我 有 数器 加来作 第二个 的 面效 。

使用Resources和Finally

最后一个与命令式 程 的是清理，通 使用 “使用 finally ” 或Java 7以上支持的 “try-with-resource” 法清理 源，如下所示：

Example 14. 命令式地使用finally

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

Example 15. 命令式地使用try-with-resource

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

者都有 的 式操作：doFinally 和 using。

doFinally 是 于 希望在序列 止（用 onComplete 或 onError） 或被取消 行的 面作用。它 了一个提示， 明是 型的 止方式触 面作用的。下面的例子 示了如何使用 doFinally：

式的finally: doFinally()

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

Flux<String> flux =
Flux.just("foo", "bar")
    .doOnSubscribe(s -> stats.startTimer())
    .doFinally(type -> { ①
        stats.stopTimerAndRecordTiming(); ②
        if (type == SignalType.CANCEL) ③
            statsCancel.increment();
    })
    .take(1); ④
```

① doFinally 消 止 型的 SignalType。

② 与 finally 代 似，我 是 。

③ 里我 也只在取消的情况下 行 量 。

④ take(1) 在 射一 元素后取消。

一方面，using 理了 Flux 来自于某 源的情况，且在 理 程中必 源

行操作。在下面的例子中，我 用 `Disposable` 替 “try-with-resource” 中的 `AutoCloseable` 接口：

Example 16. 可 的 源

```
AtomicBoolean isDisposed = new AtomicBoolean();
Disposable disposableInstance = new Disposable() {
    @Override
    public void dispose() {
        isDisposed.set(true); ④
    }

    @Override
    public String toString() {
        return "DISPOSABLE";
    }
};
```

在我 可以做相当于 “try-with-resource”的 式操作了，看起来像下面：

Example 17. 式的try-with-resource: using()

```
Flux<String> flux =
Flux.using(
    () -> disposableInstance, ①
    disposable -> Flux.just(disposable.toString()), ②
    Disposable::dispose ③
);
```

① 第一个lambda生成 源。 里，我 返回我 mock的 `Disposable`。

② 第二个lambda 理 源，返回一个 `Flux<T>`。

③ 当第二 中 `Flux` 止或被取消 ， 第三个lambda将被 行，以清理 源。

④ 并 行序列后，`isDisposed` 自 成 `true`。

演示 `onError` 的 止方面

了 明所有 些操作符都会在 生 致上游原始序列 止，我 可以用一个更直 的例子 `Flux.interval` 来 明。 `interval` 操作符 `x`个 位周期 加 `Long` 。下面的例子使用了 `interval` 操作符：

```
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); ①
```

① 注意，`interval` 是在 `timer Scheduler` 上 行的。如果我 想在 `main` 中 行 例子，我 需要在 里加一段 `sleep` 用， 用程序不会在没有 生任何 的情况下立即退出。

前面的例子 250ms 打印出一行，如下：

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

即使多了一秒的 行 ， `interval` 也没有多的周期。 个序列 被 止了。

重

于 理， 有 外一个有趣的操作符，在上一 所述的情况下， 可能会想到使用它。 名思， `retry` 重 生 的序列。

需要 住的是，它是通 重新 上游的 `Flux` 来工作的。 上是一个不同的序列，原始序列 然是 止的。 了 一点，我 依然用前面的例子，并添加 `retry(1)` 重 一次，而不是使用 `onErrorReturn`。下面的例子 示了如何做到 一点：

```

Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .elapsed() ①
    .subscribe(System.out::println, System.err::println); ②

Thread.sleep(2100); ③

```

① `elapsed` 将 个 与前一个 出后的持 起来。

② 我 想看看什 候出 `onError`。

③ 保我 有足 的 行4x2的 周期。

上面的例子 生以下 出：

```

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

```

① 从周期0，一个新的 `interval` 始。 外的250ms的持 从第四个周期 始，也就是 致常和后 重 的那个周期。

从前面的例子可以看到，`retry(1)` 重新 了一次原始的 `interval`，从0 始重新 。第二次，由于常依然 生，放 并向下游 播 。

`retry` (称 `retryWhen`) 有一个更高 的版本，使用伴随的 `Flux` 来告知是否 重 特定的故障。为了便于自定 `retry`的条件， 个伴随的 `Flux` 被操作符 建但是是由用 自己 装的。

伴随的 `Flux` 是一个 `Flux<RetrySignal>`，它被 一个 `Retry` 策略/函数，且作 `retryWhen` 的唯一的参数提供。作 用 ， 定 函数并使其返回新的 `Publisher<?>`。 `Retry` 是一个抽象 ，但如果 想用一个 的lambda (`Retry.from(Function)`) 来 伴随的 象，它提供了一个工厂方 法。

重 周期如下：

1. 次 生 (提供重 的可能性)，`RetrySignal` 都会被 送到伴随的 `Flux` 中，而 个 `Flux` 已 被 的函数装 了。 里的 `Flux` 可以看到目前 止所有的 。 `RetrySignal` 提供了 的 ，以及 的元数据。
2. 如果伴随的 `Flux` 生一个 ， 会 生重 。

- 如果伴随的 Flux 完成，被掉，重循停止，果序列也完成。
- 如果伴随的 Flux 生一个 (e)，重周期停止并生有 (e) 的序列。

前情况的区分很重要。只需完成伴随的就能有效地掉。考一下下面的方式，通使用 retryWhen 来模 retry(3)：

```
Flux<String> flux = Flux
    .<String>error(new IllegalArgumentException()) ①
    .doOnError(System.out::println) ②
    .retryWhen(Retry.from(companion -> ③
        companion.take(3))); ④
```

- ① 会不断生，用重。
- ② doOnError 可以我能在重之前和看到所有的失。
- ③ Retry 改自一个非常 的 Function lambda
- ④ 里，我前三个是可以重的 (take(3))，然后放。

上，前面的例子生一个空的 Flux，但是它成功地完成了。由于在同一个 Flux 上地 retry(3)会以最近的止，所以一个 retryWhen 例子与 retry(3) 不完全相同。

要想到同的行，需要一些外的技巧：

```
AtomicInteger errorCount = new AtomicInteger();
Flux<String> flux =
    Flux.<String>error(new IllegalArgumentException())
        .doOnError(e -> errorCount.incrementAndGet())
        .retryWhen(Retry.from(companion -> ①
            companion.map(rs -> { ②
                if (rs.totalRetries() < 3) return rs.totalRetries();
            ③
                else throw Exceptions.propagate(rs.failure()); ④
            })
        ));
});
```

- ① 我通改 Function lambda来自定 Retry，而不是提供一个具体的。
- ② 伴随的象出 RetrySignal，它了迄今止的重次数和最后一次失的次数。
- ③ 了允三次重，我考索引<3并返回一个来出（里我地返回索引）。
- ④ 了在中止序列，我在三次重之后出原始常。



人可以用 Retry 中暴露的建器来同的功能，也可以使用更流的重策略。例如：errorFlux.retryWhen(Retry.max(3));。



可以用 似的代 来 “ 等 和重 ” 模式，如FAQ中所示：

core提供的 `Retry` 助工具，`RetrySpec` 和 `RetryBackoffSpec`，都允 行高 定制，如：

- 可以触 重 的 常 置 `filter(Predicate)`
- 通 `modifyErrorFilter(Function)` 修改 一个先前 置的 器
- 触 重 触 器（即延 前后的回退）等副作用，只要重 有效（`doBeforeRetry()` 和 `doAfterRetry()` 是附加的）
- 在重 触 器周 触 一个 的 `Mono<Void>`，它允 在基本的延 的基 上添加 行 ，从而 一 延 触 器（`doBeforeRetryAsync` 和 `doAfterRetryAsync` 是附加的）
- 在 到最大 次数的情况下，通 `onRetryExhaustedThrow(BiFunction)` 自定 常。 情况下，使用了 `Exceptions.retryExhausted(...)`，可以通 `Exceptions.isRetryExhausted(Throwable)` 来区分。
- 激活 理瞬 （ 下文）

`Retry` 中的瞬 理使用 `RetrySignal#totalRetriesInARow()`： 了 是否重 和 算重 延 次 出 `onNext` ，使用的索引是一个替代索引且被重置 0。 做的后果是，如果重新 的数据源在再次失 之前 生了一些数据，那 之前的失 将不 入最大的重 次数。 在指数退避策略的情况下， 也意味着下一次的 将回到最小的 `Duration` 退避，而不是更 的 。 于生存比 久的源来 尤其有用，因 些源看到的是零星的 突 （或 瞬 ）， 次突 都 用自己的回退来重 。

```

AtomicInteger errorCount = new AtomicInteger(); ①
AtomicInteger transientHelper = new AtomicInteger();
Flux<Integer> transientFlux = Flux.<Integer>generate(sink -> {
    int i = transientHelper.getAndIncrement();
    if (i == 10) { ②
        sink.next(i);
        sink.complete();
    }
    else if (i % 3 == 0) { ③
        sink.next(i);
    }
    else {
        sink.error(new IllegalStateException("Transient error at " + i)); ④
    }
})
    .doOnError(e -> errorCount.incrementAndGet());

transientFlux.retryWhen(Retry.max(2).transientErrors(true)) ⑤
    .blockLast();
assertThat(errorCount).hasValue(6); ⑥

```

① 我 将 重 序列中的 数量。

② 我 generate 一个有突 的源。当 数器 到10 , 它将成功完成。

③ 如果 transientHelper 原子 量是 3 的倍数 , 我 就会 出 onNext, 从而 束当前的突 。

④ 在其他情况下, 我 会 出一个 onError。 就是3次中的 次, 所以2个 onError 突 中断了 1个 onNext。

⑤ 我 在 源上使用 retryWhen, 配置最多 2次重 , 但是以 transientErrors 模式。

⑥ 在 束 , 在 errorCount 中登 了 6 个 后, 序列到 onNext(10) 并完成。

如果没有 transientErrors(true), 在第二次突 , 将 到配置的最大 数 2, 并且在 出 onNext(3) 后, 序列将失 。

4.6.2. 理操作符或者函数中的 常

一般来 , 所有的操作符本身都可能包含有可能触 常或 用用 自定 的回 的代 , 些代 同 可能会失 , 所以它 都包含了某 形式的 理。

根据 , 未 的 常 是通 onError 行 播。例如, 在 map 函数中 出一个 RuntimeException 一个 onError, 如下代 所示 :

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

前面的代码打印出以下内容：

```
ERROR: java.lang.IllegalArgumentException: foo
```



可以通过使用`子`来整在`onError`之前的`Exception`。

然而，Reactor 定了一被是致命的常（例如`OutOfMemoryError`）。参考`Exceptions.throwIfFatal`方法。些意味着Reactor不能行，且将出而不是播。



在内部，也有某些情况下，由于并争，可能致`onError`或`onComplete`条件，未的常依然不能被播（最明的是在和求段）。当争生，不能播的会被“”掉。些情况在某程度上也可以通自定子行管理。参`除子`。

可能会：“需常的？”

例如，如果需要用一些声明了`throws`常的方法，依然需要在`try-catch`中理些常。但是，有几个：

1. 捕到常并从中恢。序列正常的行。
2. 捕常，将其封装成一个不的常，然后将其出（中断序列）。`Exceptions`工具可以助解决个（接下来我会到个）。
3. 如果需要返回一个`Flux`（例如，在`flatMap`中），那就用一个生的`Flux`来封装常，如下所示：`return Flux.error(checkedException)`。（个序列也会止。）

Reactor有一个`Exceptions`工具，可以用它来保只有当常被常才会被封装：

- 如果有必要，使用`Exceptions.propagate`方法来封装常。并且会首先用`throwIfFatal`且不会封装`RuntimeException`。
- 使用`Exceptions.unwrap`方法取原始的未包装的常（回到式特定常的次的根源）。

考下面的`map`的例子，它使用的`方法`可能致`IOException`常：

```
public String convert(int i) throws IOException {
    if (i > 3) {
        throw new IOException("boom " + i);
    }
    return "OK " + i;
}
```

假 在 `map` 使用 方法。 在必 式的捕 到 常，且 到`map`函数不能重新 外 出。所以 可以将其作 `RuntimeException` 常 播到`map`的 `onError` 方法中，如下所示：

```
Flux<String> converted = Flux
    .range(1, 10)
    .map(i -> {
        try { return convert(i); }
        catch (IOException e) { throw Exceptions.propagate(e); }
    });
});
```

以后，当 前面的 `Flux` 并 做出 （例如在用 界面），如果 想 IO 常做一些特殊的事情， 可以将其 原到原始 常。下面的例子 示了如何做到 一点：

```
converted.subscribe(
    v -> System.out.println("RECEIVED: " + v),
    e -> {
        if (Exceptions.unwrap(e) instanceof IOException) {
            System.out.println("Something bad happened with I/O");
        } else {
            System.out.println("Something bad happened");
        }
    }
);
```

4.7. 理器

`Processor`是一 特殊的 `Publisher`，也是 `Subscriber`。 意味着 可以 `subscribe` 一个 `Processor`（通常，它 了 `Flux`），但 也可以 用方法手 将数据注入到序列或 止它。

`Processor`有几 ， 一 都有一些特殊的 ， 但是在 始研究它 之前， 需要 自己以下 :

4.7.1. 我需要一个`Processor` ?

大多数 候， 尽量避免使用 `Processor`。它 很 正 使用，而且容易出 一些 端的情况。

- 如果 Processor 比 合 的情况，自己是否：
1. 操作符或者 合操作符是否 足要求？（看我需要 个操作符？）
 2. 能否用 “generator” 操作符代替？（通常，一些操作符用于 接非 式的API，提供了一个概念上 似于 Processor 的“sink”，也就是 它允 用数据手 填充或 止序列）。

如果在考察了上述的替代方案后，然 需要一个 Processor， 可用 理器概述 章 了解不同的 。

4.7.2. 使用 Sink 外 模式多 程安全生

与其直接使用Reactor的 Processors，不如通 一次性 用 sink() 来 得 Processor 的 Sink。

FluxProcessor 接收器可以安全地 多 程生 者，并可以被多 程并 的生成数据的 用使用。例如， 可以通 以下操作 UnicastProcessor 建一个 程安全的序列化接收器：

```
UnicastProcessor<Integer> processor = UnicastProcessor.create();
FluxSink<Integer> sink = processor.sink(overflowStrategy);
```

多个生 者 程可以通 行以下操作，在下面的序列化接收器上并 的生 数据：

```
sink.next(n);
```



尽管 FluxSink 用于 Processor 的多 程 手 入，但不可能将 者方法和接收器方法混合使用：必 将 FluxProcessor 到源 Publisher 或者通 它的 FluxSink 手 入。

从 next 溢出有 可能的方式， 具体取决于 Processor 及其配置：

- 无界 理器通 或 冲来 理溢出本身。
- 有界 理器在 IGNORE 策略上阻塞或“旋 ”，或 sink 指定的 overflowStrategy 行 。

4.7.3. 可用 理器概述

Reactor核心配 了几 Processor。并非所有的 理器都有相同的 ，但是它 大致分 三 。以下列表 要介 了 三 理器：

- 直接的 (DirectProcessor 和 UnicastProcessor)：些 理器只能通 用 直接操作（直接 用其 Sink 方法）来推送数据。
- 同 的 (EmitterProcessor 和 ReplayProcessor)：些 理器可以通 用 交互推送数据，也可以 上游的 Publisher 并同 消耗数据。



将事件 布到不同 程上的一 方法是将 `EmitterProcessor` 与 `publishOn(Scheduler)` 合使用。例如， 可以取代以前的 `TopicProcessor`，在3.3.0中，它使用了 `Unsafe` 操作，并已被移到 `reactor-extra`。

直接的 理器

直接的 Processor 是一个可以向零个或多个 Subscribers 送信号的 理器。最 的 例化，只需要一个 `DirectProcessor#create()` 静 工厂方法。 一方面，它具有不 理背 的局限性。因此，如果 通 `DirectProcessor` 推送了N个元素，但是如果有一个 者 求数量小于N，那 `DirectProcessor` 就会向 者 送 `IllegalStateException` 的信号。

一旦 Processor 止（通常是其接收器的 `error(Throwable)` 或者 `complete()` 方法被 用），它就会允 更多的 者 ，但会立即将 止信号 播 它 。

播 理器

播 Processor 可以通 使用内部 冲区来 理背 。不足之 是它最多只能有一个 Subscriber。

与直接 理器相比， `UnicastProcessor` 有更多的 ， 一点从几个 `create` 静 工厂的方法的存在可以反映出来。例如， 情况下，它是无界的：如果在其 Subscriber 没有 求数据的 候向它推送任何数量的数据，它将 冲所有的数据。

可以通 在 `create` 工厂方法中 内部 冲区提供一个自定 的 Queue 来改 一点。如果 列是有界的，那 当 冲区 了，并且没有收到来自下游的足 求 ， 理器可能会拒 数据的推送。

在 有界 的情况下， 也可以在 理器上建立一个回 ，在 个被拒 的元素上都会被 用，允 清理 些被拒 的元素。

射器 理器

一个 射器 Processor 可以 射 多个 者，同 其 个 者提供背 。它 可以 到 Publisher 并同 其信号。

最 始，在它没有 者的 候，它 然可以接收一些数据推送，最大 可配置的 `bufferSize`。此后，如果没有 Subscriber 并消 数据，那 就会 用 `onNext` ，直到 理器被耗尽 止（ 只能同 生）。

因此，第一个 Subscriber 在 ， 最多接收到 `bufferSize` 个元素。但是，此后， 理器停止向其它 者重播 些信号。 些后 的 者只接受到在 后通 理器推送的信号。内部 冲区 用于背 。

情况下，如果它的所有 者都被取消了（基本上意味着它 已 全部取消 ），它将清除其内部 冲区并停止接收新的 者。 可以通 使用 `create` 静 工厂方法中的 `autoCancel` 参数来 此 行 整。

重播 理器

重播 Processor 存了直接通 其 `sink()` 直接推送或来自上游 Publisher 的元素，并将其重播 后面的 者。

可以用多 配置来 建它：

- 存一个元素 (`cacheLast`)。
- 存一个有限的 史 (`create(int)`) 或 无界的 史 (`create()`)。
- 存基于 的重播 口 (`createTimeout(Duration)`)。
- 存 史 大小和 口的 合 (`createSizeOrTimeout(int, Duration)`)。

Chapter 5. Kotlin的支持

Kotlin 是一 JVM (和其它平台) 的静 型 言，可以 写 而 雅的代 ，同 与 有的 Java 展 提供了很好的互操作性。

本 介 了Reactor Kotlin的支持。

5.1. 要求

Reactor支持Kotlin 1.1+，需要 `kotlin-stdlib` (或 `kotlin-stdlib-jre7` 或 `kotlin-stdlib-jre8` 其中之一)。

5.2. 展

从 `Dysprosium-M1` (即 `reactor-core 3.3.0.M1`) 始，Kotlin 展被移到一个 的 `reactor-kotlin-extensions` 模 ， 模 用新的以 `reactor.kotlin` 始的包名代替之前的 `reactor`。

因此，不推 使用 `reactor-core` 模 中的Kotlin 展。新的依 `groupId` 和 `artifactId`是：

`io.projectreactor.kotlin:reactor-kotlin-extensions`

得益于 大的 Java互操作性和 Kotlin 展，Reactor的Kotlin API使用常 的Java API，并通一些Kotlin特有的API 行 ， 些API可以在Reactor中 箱即用。

 住必 要 入Kotlin 展后才能使用。例如， 意味着只有在 入 `import reactor.kotlin.core.publisher.toFlux` 的情况下，`Throwable.toFlux` Kotlin 展才能使用。也就是 ， 似于静 入，在大多数情况下，IDE 会自 建 入。

例如， `Kotlin 化 型参数` `JVM 泛型 型擦除`提供了一个解决方案，且Reactor提供了一些 展来利用个特性。

下表比 了Java中的Reactor和Kotlin中的Reactor 展。

Java	Kotlin 展
<code>Mono.just("foo")</code>	<code>"foo".toMono()</code>
<code>Flux.fromIterable(list)</code>	<code>list.toFlux()</code>
<code>Mono.error(new RuntimeException())</code>	<code>RuntimeException().toMono()</code>
<code>Flux.error(new RuntimeException())</code>	<code>RuntimeException().toFlux()</code>
<code>flux.ofType(Foo.class)</code>	<code>flux.ofType<Foo>() 或 flux.ofType(Foo::class)</code>
<code>StepVerifier.create(flux).verifyComplete()</code>	<code>flux.test().verifyComplete()</code>

[Reactor KDoc API](#)列出并列出了所有可用的Kotlin 展。

5.3. 空安全

Kotlin的特性之一是空安全，它在干利落地理了 `null`，而不是在行到著名的 `NullPointerException` 常。通过可空性的声明和“有或者无”的表可以使⽤更加安全，而不需要花代⾏如 `Optional` 的封装。（Kotlin允使⽤具有空的函数造。看于[Kotlin空安全的全面指南](#)。）

尽管Java不允在其型系中表示null安全，但通在 `reactor.util.annotation` 包中声明的友好的注，Reactor整个Reactor API提供空安全。情况下，Kotlin中使用的Java API中的型会被平台型，于些型，空可以放。Kotlin支持JSR 305注和Reactor可空性注。Kotlin者提供了整个Reactor API的空安全，在理 `null` 相的。

可以通添加 `-Xjsr305` 器来配置JSR 305的，并使⽤以下：
`-Xjsr305={strict|warn|ignore}`。

于Kotlin 1.1.50+的版本，行与 `-Xjsr305=warn` 相同。`strict` 表示需要考Reactor API的完全空安全，但被是性的，因Reactor API的可空性声明即使是在小的行版之也会生演，也可能会在未来加更多的。



尚不支持泛型型参数和量参数，数元素的可空性，但是会在即将布的版本中出。有最新的信息，看此。

Chapter 6.

无 是写了一个 的Reactor操作符 是自己的操作符，自 化 都是一个不 的 。

Reactor自 了一些 用于 的元素， 到自己的artifact：reactor-test。在 reactor-core 中， 能 在Github 到 目。

要在 中使用它， 必 将其添加作 依 。下面的例子 示了如何在Maven中 添加 reactor-test 作 依 ：

Example 18. Maven中的reactor-test, 在 <dependencies>

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
    ①
</dependency>
```

①如果 使用了 BOM， 不需要指定 <version>。

下面的例子 示了如何在Gradle中添加 reactor-test 作 依 ：

Example 19. Gradle中的reactor-test, 修改 dependencies

```
dependencies {
    testCompile 'io.projectreactor:reactor-test'
}
```

reactor-test 的三个主要用途如下：

- 用 StepVerifier 逐 一个序列是否遵循 定的 景。
- 生 数据，以便用 TestPublisher 下游操作符（包括 自己的操作符）的行 。
- 在可以 多个可 的 Publisher 的序列(例如，一个使用 switchIfEmpty 的 ， 行 Publisher 的探 ，以 保它已被使用 (即已))中。

6.1. 使用 StepVerifier 一个 景

Reactor序列最常 的情况是在 代 中定 了一个 Flux 或者 Mono (例如，可能是由一个方法返回的)，想要 它在 的行 。

情况可以很好地 化 定 一个 “ 景”，可以根据事件，一 定 的期望。 可能会 到并回答 如下面的 ：

- 下一个 期的事件是什 ？

- 希望 Flux 射出一个特定的 ?
- 或者在接下来的300ms内什 都不做？

可以通 StepVerifier API来表 一切。

例如， 可以在 的代 中使用下面的工具方法来装 一个 Flux :

```
public <T> Flux<T> appendBoomError(Flux<T> source) {
    return source.concatWith(Mono.error(new IllegalArgumentException("boom")));
}
```

了 它， 要 以下 景：

我希望 个 Flux 首先 生 thing1， 然后 生 thing2， 接着 生 有 消息 的 boom。 并 些期望。

在 StepVerifier API中， 可以 化 以下 :

```
@Test
public void testAppendBoomError() {
    Flux<String> source = Flux.just("thing1", "thing2"); ①

    StepVerifier.create( ②
        appendBoomError(source)) ③
        .expectNext("thing1") ④
        .expectNext("thing2")
        .expectErrorMessage("boom") ⑤
        .verify(); ⑥
}
```

① 因 我 的方法需要一个源 Flux， 所以定 一个 的源来作 。

② 建一个 StepVerifier 建器， 用于封装和 Flux。

③ 要 的 Flux (用我 工具方法的 果)。

④ 我 期望在 生的第一个信号是 onNext， 是 thing1。

⑤ 我 期望的最后一个信号是以 onError 止序列。 个 常 包含 boom 信息。

⑥ 重要的是通 用 verify() 触 。

个API是一个 建器。 首先 建 StepVerifier 并 要 的序列。 提供了一 方法 ， 可以 :

- 表 下一个信号 生的期望。如果收到任何其他信号（或者信号的内容与 期不符）， 整个 都会 以一个有意 的 AssertionError 失 。例如， 可以使用 expectNext(T...) 和 expectNextCount(long)。

- 消下一个信号。当想要略序列的一部分，或者想信号的内容用一个自定的 `assertion`（例如，为了是否存在 `onNext` 事件并断言生的列表元素个数大小 5），例如，可以使用 `consumeNextWith(Consumer<T>)`。
- 行如停或行任意代等其他操作。例如，如果想要操一个特定的状态或上下文。此，可以使用 `thenAwait(Duration)` 和 `then(Runnable)`。

于止事件，相的期望方法（`expectComplete()` 和 `expectError()` 及其其它的形式）会切到一个无法再表期望的API中。在最后一，能做的就是再 `StepVerifier` 上行一些外的配置，然后触，通常用 `verify()` 或其其它形式之一。

此，`StepVerifier` 了被的 `Flux` 或 `Mono`，并触序列的始，将一个新信号与景中的下一个行比。只要些符合，就成功。一旦有一个差，将会出 `AssertionError`。



住 `verify()`，它触了。了提供助，API包含了一些快捷方法，将止期望与用 `verify()` 合起来：`verifyComplete()`, `verifyError()`, `verifyErrorMessage(String)` 等。

注意，如果其中一个基于lambda的期望出一个 `AssertionError`，会按原告，失。于自定断言是很有用。



情况下，`verify()` 方法和派生的快捷方法（`verifyThenAssertThat`, `verifyComplete()` 等）没有超。它可以无限制地阻塞。可以使用 `StepVerifier.setDefaultTimeout(Duration)` 些方法全局置一个超，或者用 `verify(Duration)` 指定一个超。

6.1.1. 更好地失

`StepVerifier` 提供了个，以更好地定是个期望致失：

- `as(String)`：用在大多数 `expect*` 方法之后，用于出先前期望的描述。如果期望失，其信息会包含描述。止期望和 `verify` 不能使用方式行描述。
- `StepVerifierOptions.create().scenarioName(String)`：通使用 `StepVerifierOptions` 来建的 `StepVerifier`，可以使用 `scenarioName` 方法整个景命名，个名字也可用于在断言信息中。

注意，在情况下，只能保在使用 `StepVerifier` 它自己的方法生的 `AssertionError` 信息中使用描述和名称（例如，手出一个常或通 `assertNext` 中的断言，不会将描述或名称添加到消息中）。

6.2. 操

可以使用基于的操作符的 `StepVerifier` 来避免相的行。可以通过 `StepVerifier.withVirtualTime` 建器来。

它看起来像下面个例子：

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

个虚的功能在Reactor的 Schedulers 工厂中 入一个自定 的 Scheduler。由于 些 操作符通常使用 的 Schedulers.parallel() 度器，所以用 VirtualTimeScheduler 代替它就行了。但是，一个重要的前提是，个操作符必 在虚 度器被激活后 例化。

了 加 情况正 生的几率，StepVerifier 不接受 的 Flux 作 入。withVirtualTime 需要一个 Supplier，在完成 度器的 置后，它会 慢地引 建被 的Flux的 例。



格外小心，保在 加 下 Supplier<Publisher<T>> 能被使用。否 ，虚 无法被保 。特 是避免在 代 中 Flux 的 早 例化和并 Supplier 返回 量。相反， 是在lambda中 例化 Flux。

理 的期望方法有 ，不管有没有虚 ，都是有效的。

- `thenAwait(Duration)`：停 的 算（允 一些信号 生或延 耗尽）。
- `expectNoEvent(Duration)`：也可以 序列在 定的持 内 生元素，但是如果在 段 内有 其它 信号 生， 失 。

在 典模式下， 方法都会 停 程的 定 ，而在虚 模式下， 会提前虚 。



expectNoEvent 也将 subscription 一个事件。如果 把它作 第一 使用，通常会失 ，因 会 到 信号。用 expectSubscription().expectNoEvent(duration) 代替它。

了快速 算我 上面的 `Mono.delay` 的行 ，可以通 以下方式完成代 的 写：

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
    .expectSubscription() ①
    .expectNoEvent(Duration.ofDays(1)) ②
    .expectNext(0L) ③
    .verifyComplete(); ④
```

① 前面的 tip。

② 期待一整天都不会有什 事情 生。

③ 然后期待 射数据是 0。

④ 然后期待完成（并触 ）。

我 可以使用上面的 `thenAwait(Duration.ofDays(1))`，但 `expectNoEvent` 能 保什 都不会 早 生。

注意，`verify()` 返回一个 Duration。 是整个 的 持 。

虚不是。所有的 **Schedulers** 都是被替相同的 **VirtualTimeScheduler**。在某些情况下，可以定程，因在期望表前，虚并未始，从而致在期望等待的数据只能提前生。在大多数情况下，需要将虚提前，才能出序列。无限序列的虚也会受到限制，可能会占用序列和行所在的程。



6.3. 用 StepVerifier 行 行后断言

在描述了的景中的最期望后，可以切到一个充的断言API，而不是触 **verify()**。此，需要使用 **verifyThenAssertThat()**。

verifyThenAssertThat() 返回一个 **StepVerifier.Assertions** 象，一旦整个景成功地行了，可以使用它来断言一些状元素（因它同会用 **verify()**）。典型（即高）的用法是捕被某些操作符的元素并断言它（参 [子章](#)）。

6.4. 上下文

于 [上下文](#) 的更多信息，看 [式序列添加上下文](#)。

在 **Context** 的播程中，**StepVerifier** 附一些期望：

- **expectAccessibleContext**：返回一个 **ContextExpectations** 象，可以使用一个象来置在播 **Context** 的期望。保用 **then()** 能返回到序列期望集。
- **expectNoAccessibleContext**：置了一个期望，使其在被的操作符上不能播任何 **Context**。最有可能生在当的不是式的 **Publisher** 或没有任何可以播 **Context**（例如，生成器源）的操作符

此外，可以通使用 **StepVerifierOptions** 来建器，将特定于的初始 **Context** 到 **StepVerifier**。

下面的片段展示了些特性：

```

StepVerifier.create(Mono.just(1).map(i -> i + 10),
    StepVerifierOptions.create().withInitialContext(Context.of("thing1", "thing2")))
①
    .expectAccessibleContext() ②
    .contains("foo", "bar") ③
    .then() ④
    .expectNext(11)
    .verifyComplete(); ⑤

```

① 通 使用 `StepVerifierOptions` 建 `StepVerifier` 并 一个初始化的 `Context`。

② 始 置 于 `Context` 播的期望。此一 就可以 保 `Context` 的 播。

③ 特定 `Context` 期望的一个例子。它必 包含 "thing1" 的 "thing2"。

④ 我 使用 `then()` 切 回 数据 置正常 期望。

⑤ 我 不要忘 整个期望集合 行 `verify()`。

6.5. 用 `TestPublisher` 手 射

于更高的 用例来 , 完全掌握数据源, 能 触 精心 的信号, 使之与 要 的特定情况 密 配合会更有用。

一 情况是当 已 了自己的操作符, 想要 其在 式流 的行 , 特 是其源不能 很 好表 。

于 情况, `reactor-test` 提供了 `TestPublisher` 。 是一个能 以 程方式触 各 信号的 `Publisher<T>` :

- `next(T)` 和 `next(T, T…)` 触 1到n个 `onNext` 信号。
- `emit(T…)` 触 1到n个 `onNext` 信号并 行 `complete()`。
- `complete()` 以 `onComplete` 信号 止。
- `error(Throwable)` 以 `onError` 信号 止。

可以通 `create` 工厂方法 得一个表 良好的 `TestPublisher`。外, 也可以通 使用 `createNonCompliant` 工厂方法 建一个表 不好的 `TestPublisher`。后者从 `TestPublisher.Violation` 枚 中取一个或多个 。些 定 了生 者可以忽略 中的 些部分。些枚 包括 :

- `REQUEST_OVERFLOW`: 允 在 求不足的情况下 行 `next` 用, 且不会触 `IllegalStateException`。
- `ALLOW_NULL`: 允 `null` 行 `next` 用而不会触 `NullPointerException` 常。
- `CLEANUP_ON_TERMINATE`: 允 多次 送 止信号。包括 `complete()`、`error()` 和 `emit()`。
- `DEFER_CANCELLATION`: 允 `TestPublisher` 忽略取消信号并 送信号, 就好像取消信号 掉了与所 信号的比 一 。

最后, `TestPublisher` 保持着 后的内部状 , 可以通 它的各 `assert*` 方法 行断言。

可以使用 `flux()` 和 `mono()`，将其 `Flux` 或 `Mono`。

6.6. 用 PublisherProbe 行路径

在造的操作符，可能会遇到有几个可能的行途，由不同的子序列具体化的情况。

大多数时候，一些子序列会生一个特定的 `onNext` 信号，能通看最果来断言其已行。

例如，考下面的方法，它从源建一个操作符，如果源空，使用 `switchIfEmpty` 来回退到一个特定替代的源：

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

可以使用了 `switchIfEmpty` 的一个分支，如下所示：

```
@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();
}
```

但是，想想看一个例子，方法生一个 `Mono<Void>`。它等待源完成，行一个外的任并完成。如果源空，必行似于 `Runnable` 的降任。下面的例子示了情况：

```

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()) ①
        .switchIfEmpty(doWhenEmpty); ②
}

```

① `then()` 忽略命令 行 果。它只 心它是否完成了。

② 如何区分 个都是空序列的情况？

了 的 `processOrFallback` 方法 行了 `doWhenEmpty` 分支， 需要写一些 板。即 需要一个 `Mono<Void>` :

- 捕 已 的事 。
- 在整个 程 束后断言 事 。

在3.1版本之前， 需要 个 想要断言的状 手 一个 `AtomicBoolean`， 并将相 的 `doOn*` 回 附加到 需要 估的生 者上。当需要 常使用 模式 ， 可能会有很多繁 的模版。幸 的是， 3.1.0 引入了 `PublisherProbe` 的替代方案。下面的例子展示了如何使用它：

```

@Test
public void testCommandEmptyPathIsUsed() {
    PublisherProbe<Void> probe = PublisherProbe.empty(); ①

    StepVerifier.create(processOrFallback(Mono.empty(), probe.mono())) ②
        .verifyComplete();

    probe.assertWasSubscribed(); ③
    probe.assertWasRequested(); ④
    probe.assertWasNotCancelled(); ⑤
}

```

① 建一个 空序列的探 。

② 通 用 `probe.mono()` 探 代替 `Mono<Void>`。

③ 序列完成后，探 可断言它已被使用。 能 它是否已被 ...

④ ...以及 求的数据...

⑤ ...以及是否被取消。

可以通 用 `.flux()` 替 `.mono()`， 及 探 代替 `Flux<T>`。 于 想要探 行途 并也需要探

射数据的情况，可以使用 `PublisherProbe.of(Publisher)` 来封装任何 `Publisher<T>`。

Chapter 7. Reactor

从命令式同 程 式切 到 式 式 程有 会 人望而生畏。学 曲 中最 的 之一，就 是在出 如何分析和 。

在命令式 程中， 通常是相当直接的。 可以 堆 跟踪， 看 的根源。完全是代 出 了故障 ？ 故障是否 生在其它 的代 中？如果是 ， 代 的 一部分是 用了 ， 可能是 入了不正 的参数，从而 致故障？

7.1. 典型的 式堆 追踪

当 向 代 ， 事情会 得更加 。

考 一下下面的堆 信息：

```
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:445)
    at reactor.core.publisher.FluxFlatMap$FlatMapMain.onNext(FluxFlatMap.java:379)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseabl
e.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$MapFuseableSubscriber.request(FluxMapFuseab
le.java:162)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.onSubscribe(FluxFlatMap.java:332)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFu
seable.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)
```

里面有很多事情。我 得到一个 `IndexOutOfBoundsException` 常，告 我 源 射了超 一个以上的元素。

从下一行提到的 `MonoSingle`，我 可能很快就能推断出 源 一个Flux或者Mono。因此，似乎是从一个 `single` 操作符 的 。

参照 `Mono#single` 操作符的Javadoc，我 可以看到 `single` 有一个 定：源必 精 地 射一个元素。看来我 有一个源 射了超 一个以上的元素，因此 反了 一 定。

我 可以更深入地 掘并 定那个源？下面 几行， 我 的 助不大。它 通 多次 用 `subscribe` 和 `request`， 我 了解了似乎是一个 式 的内部。

通 略 些行，我 至少可以 始形成一个出 的 的：它似乎 及到 `MonoSingle` 和 `FluxFlatMap`，`FluxRange`（ 个在堆 跟踪中都能 得几行，但 体上 三个 都 及到了）。所以也 是一个 `range().flatMap().single()` ？

但是如果我在用中大量使用模式？然不能明什，的搜索single并不能。然后，最后一行的是我自己的一些代。最，我真相越来越近的。

不，等一下。当我跳到源文件，我只看到一个先存在到Flux被，如下所示：

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

所有些都是在生的，但是Flux本身并没有在那里声明。更糟的是，当我到声明量的地方，我会看到下面的内容：

```
public Mono<String> toDebug; // 忽略公共属性
```

量没有在它被声明的地方被例化。我必假一个最坏的情况，即我可能有几个不同的代路徑在应用程序中置它。我然不定是一个引起的。



有点似于Reactor的行，而不是。

我想要更容易的是操作符添加到的位置，即Flux声明的地方。我通常将其称Flux的“装配”。

7.2. 激活模式 - 又名回



本描述了最但也是最慢的方式来用能力，因它捕一个操作符上的堆。了解更的方式，看checkpoint()替代方案，以及于更高性能的全局，看可生的全局。

尽管于微有点的来，堆跟踪能是能表一些信息，但是我可以看出，在更高的案例中，它本身的效果并不理想。

幸的是，Reactor自有了用于的装配施。

可以通在用（或至少在所疑的Flux或者Mono例化前）自定Hooks.onOperator子来，如下：

```
Hooks.onOperatorDebug();
```

通在里包装操作符的造和捕堆跟踪始Flux（和Mono）操作符方法（装配到的）的用行。因是在声明操作符完成的，所以子在之前被激活，所以最安全的方式是直接在用始就激活它。

之后，如果生常，失的操作符能引用捕的信息并将其附加到堆跟踪中。我将捕的装配信息称回。

在下一中，我将看到堆跟踪有什么不同，以及如何解一些新的信息。

7.3. 在模式下取堆跟踪

当我再次使用最原始的例子，但在激活 `operatorStacktrace` 功能的时候，堆跟踪如下：

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
    at
reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
    at
reactor.core.publisher.FluxOnAssembly$OnAssemblySubscriber.onNext(FluxOnAssembly.java:375) ①
...
②
...
    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.debuggingActivated(GuideTests.java:1000)
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: ③
Assembly trace from producer [reactor.core.publisher.MonoSingle] : ④
    reactor.core.publisher.Flux.single(Flux.java:6676)
    reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949)
    reactor.guide.GuideTests.populateDebug(GuideTests.java:962)
    org.junit.rules.TestWatcher$1.evaluate(TestWatcher.java:55)
    org.junit.rules.RunRules.evaluate(RunRules.java:20)
Error has been observed by the following operator(s): ⑤
    |_ Flux.single
reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949) ⑥
```

① 是新的常信息：我可以看到捕堆的包装操作符。

② 除此之外，第一部分的堆依然几乎相同，示了一些操作符（所以我里去掉了一些代码片段）内部。

③ 就是回始出的地方。

④ 首先，我得到一些装配操作符的信息。

⑤ 当通操作符播，我可以从到尾（端到端）追到。

⑥ 个看到的操作符都会提到使用的用和行。

捕的堆跟踪会作的 `OnAssemblyException` 常附加到原始中。它分部分，但第一部分是最有趣的。它示了触常的操作符的造路径。在里，它示了是 `scatterAndGather` 方法中建的 `single` 致了，它本身是通 JUnit 行的 `populateDebug` 方法用的。

在我已掌握了足的信息来到罪魁首，我 `scatterAndGather` 方法行有意的研究：

```
private Mono<String> scatterAndGather(Flux<String> urls) {  
    return urls.flatMap(url -> doRequest(url))  
        .single(); ①  
}
```

① 果然， 里是 `single` 方法。

在我 可以看到 致 的根本原因是 `flatMap` 几个url 行了几个HTTP 用，但是 个 用是用 `single` 串 起来的，限制性太 了。 短的使用 `git blame`，并与 行代 的作者 行了 短的后，我 他打算使用限制性 小的 `take(1)` 来代替。

我的 已 解决了。

在看下在堆 跟踪中的 一行：

```
Error has been observed by the following operator(s):
```

在 个特殊的例子中， 堆 跟踪的第二部分并不一定有意思，因 上 生在 中的最后一个操作符（最接近 的那个）。考 一个例子可能会更清晰：

```
FakeRepository.findAllUserByName(Flux.just("pedro", "simon", "stephane"))  
    .transform(FakeUtils1.applyFilters)  
    .transform(FakeUtils2.enrichUser)  
    .blockLast();
```

在想象一下，在 `findAllUserByName` 中，有一个 `map` 失 了。在 里，我 将看到下面的的回：

```
Error has been observed by the following operator(s):
|_ Flux.map
reactor.guide.FakeRepository.findAllUserByName(FakeRepository.java:27)
|_ Flux.map
reactor.guide.FakeRepository.findAllUserByName(FakeRepository.java:28)
|_ Flux.filter
reactor.guide.FakeUtils1.lambda$static$1(FakeUtils1.java:29)
|_ Flux.transform
reactor.guide.GuideDebuggingExtraTests.debuggingActivatedWithDeepTraceback(GuideDe
buggingExtraTests.java:40)
|_ Flux.elapsed
reactor.guide.FakeUtils2.lambda$static$0(FakeUtils2.java:30)
|_ Flux.transform
reactor.guide.GuideDebuggingExtraTests.debuggingActivatedWithDeepTraceback(GuideDe
buggingExtraTests.java:41)
```

的是操作符 中被通知 的那部分：

1. 常源于第一个 `map`。
2. 它被第二个 `map` 看到了（ 上 一个方法都 于 `findAllUserByName` 方法）。
3. 然后通 一个 `filter` 和 `transform` 看到它， 表示 的那部分是由可重用的 函数 成（ 里是 `applyFilters` 工具方法）的。
4. 最后，通 一个 `elapsed` 和 `transform` 看到它。`elapsed` 是由第二个 的 函数所使用。



当回 作 短的 常被附加到原始 中， 可能在某 程度上会干 一 使用此机
制的 一 常： 合 常。 常可以直接通 `Exceptions.multiple(Throwable...)`
来 建，或者通 一些可能 接多个 源（如 `Flux#flatMapDelayError`）的操作符来
建。它 可以通 `Exceptions.unwrapMultiple(Throwable)` 展 到 `List` 中，在
情况下，回 会被 合的一个 件，并成 返回的 `List`
的一部分。相反，如果不需要的 ，可以通 `Exceptions.isTraceback(Throwable)`
来 回，并使用 `Exceptions.unwrapMultipleExcludingTracebacks(Throwable)`
将其排除在展 之外。

我 在 里以 的形式 理，而 建堆 追踪的代 是非常高的。 就是 什 个 功能只 以可
把控的方式激活，并只能当作最后的 法。

7.3.1. `checkpoint()` 替代方案

模式是全局的，它影 到 用程序中的 一个操作符，并将其 装到 `Flux` 或 `Mono` 中。 做的好
是允 事后 ！无 是什 ，我 都可以 取更多的信息来 。

正如我 前面看到的， 全局的能力是以 性性能（由于填充的堆 跟踪的数量） 代 的。如果我 知
道可能是某个操作符有 ，那 个代 可以降低。但是，我 通常不知道 个操作符出 了 ，除非
我 明 地看到 ，看到自己 失了 装信息，然后修改代 激活 装追踪，希望再次 察到相同的
。

在那 景下，我 必 切 到 模式，并做好准 以便更好地 察到第二次出 的 ， 一次捕 到 所有 外的信息。

如 能 出 在 用中 装的，且 其可用性至 重要的 式 。那 可以使用 `checkpoint()` 操作符将 技 合。

可以将此操作符 接到方法 中。`checkpoint` 操作符的工作原理与像 子版本一 ，但 用于特定 的 接。

里 有 `checkpoint(String)` 的 一 形式，能 添加唯一的 String 符到 合的回 。就省略了堆 追踪，并依 描述来 装配的位置。`checkpoint(String)` 于普通的 `checkpoint` 代 要低。

`checkpoint(String)` 在其 出中（ 搜索 会非常方便）包含了“light”，如下面例子所示：

```
...
Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly site of producer [reactor.core.publisher.ParallelSource] is identified by
light checkpoint [light checkpoint identifier].
```

最后但并非最不重要的，如果 想要添加更通用的描述到 点，但 然依 堆 跟踪机制来 装配的位置， 可以通 使用 `checkpoint("description", true)` 来 制 行 行 。我 在又回到了回 的初始信息，并添加了 `description`，如下例所示：

```
Assembly trace from producer [reactor.core.publisher.ParallelSource], described as
[descriptionCorrelation1234] : ①
```

```
    reactor.core.publisher.ParallelFlux.checkpoint(ParallelFlux.java:215)
```

```
reactor.core.publisher.FluxOnAssemblyTest.parallelFluxCheckpointDescriptionAndForc
eStack(FluxOnAssemblyTest.java:225)
```

```
Error has been observed by the following operator(s):
```

```
    |_ ParallelFlux.checkpoint
```

```
reactor.core.publisher.FluxOnAssemblyTest.parallelFluxCheckpointDescriptionAndForc
eStack(FluxOnAssemblyTest.java:225)
```

① `descriptionCorrelation1234` 是 `checkpont` 中提供的描述。

描述可以是一个静 的 符或用 可 的描述，也可以是更广泛的相 ID（例如，在HTTP 求的情况下，来自于一个 求 的）。



当全局 和本地 `checkpoint()` 都 用 ， 点的快照堆 作 短的 出附加到 察操作符 之后，并遵循相同的声明性 序。

7.4. 可生 的全局

Reactor工程自 了一个独立的Java代理，可以 的代 并添加 信息，而不需要花 捕 个操作符 用的堆 追踪的代 。其行 似于[激活 模式 - 又名回](#)，但是没有 行 的性能 。

要在 的 用程序使用它，必 将其添加 依 型。

下面的例子 示了如何在Maven中添加 `reactor-tools` 依 型：

Example 21. Maven中的`reactor-tools`, 在 `<dependencies>`

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-tools</artifactId>
    ①
</dependency>
```

① 如果 使用了[BOM](#)， 不需要指定 `<version>`。

下面的例子 示了如何在Gradle中添加 `reactor-tools` 依 型：

Example 22. Gradle中的`reactor-tools`, 更改 `dependencies`

```
dependencies {
    compile 'io.projectreactor:reactor-tools'
}
```

当然需要 式地初始化它：

```
ReactorDebugAgent.init();
```



由于 工具会在加 的 候会 的 ， 所以最好把它放在`main(String[])`方法中的所有其它功能之前：

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

如果 不能及早地（例如在 中） 行初始化， 也可以 有的 行重新 理：

```
ReactorDebugAgent.init();
ReactorDebugAgent.processExistingClasses();
```



注意，由于需要迭代所有加的并用，重新理需要花几秒的。在一些用位置没有使用它。

7.4.1. 局限性

`ReactorDebugAgent` 作一个Java代理，并使用 `ByteBuddy` 行自我附加。自我附加可能不用某些JVM，参考`ByteBuddy`的文取更多信息。

7.4.2. 作 Java代理 行`ReactorDebugAgent`

如果的境不支持`ByteBuddy`的自我附加，可以将 `reactor-tools` 作 Java代理行：

```
java -javaagent reactor-tools.jar -jar app.jar
```

7.4.3. 在建 行`ReactorDebugAgent`

也可以在建行 `reactor-tools`，要做到一点，需要将其作 `ByteBuddy` 的建工具的件来使用。



将只用于的目的，路径下的并不会被。

Example 23. reactor-tools 和 ByteBuddy 的 Maven 件

```
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-tools</artifactId>
        ①
        <classifier>original</classifier> ②
        <scope>runtime</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>net.bytebuddy</groupId>
            <artifactId>byte-buddy-maven-plugin</artifactId>
            <configuration>
                <transformations>
                    <transformation>
<plugin>reactor.tools.agent.ReactorDebugByteBuddyPlugin</plugin>
                    </transformation>
                </transformations>
            </configuration>
        </plugin>
    </plugins>
</build>
```

① 如果 使用了 BOM， 不需要指定 `<version>`。

② 里的 `classifier` 很重要。

Example 24. reactor-tools 和 ByteBuddy 的 Gradle 件

```
plugins {
    id 'net.bytebuddy.byte-buddy-gradle-plugin' version '1.10.9'
}

configurations {
    byteBuddyPlugin
}

dependencies {
    byteBuddyPlugin(
        group: 'io.projectreactor',
        name: 'reactor-tools',
        ①
        classifier: 'original', ②
    )
}

byteBuddy {
    transformation {
        plugin = "reactor.tools.agent.ReactorDebugByteBuddyPlugin"
        classPath = configurations.byteBuddyPlugin
    }
}
```

① 如果 使用了 BOM， 不需要指定 <version>。

② 里的 classifier 很重要。

7.5. 序列

除了堆 追踪 和分析之外，在工具包中有 外一个 大的工具是在 序列中追踪和 事件的能力。

`log()` 操作符可以做到 一点。 接在序列中，能 它上游的 个 `Flux` 或 `Mono` 事件（包括 `onNext`，`onError` 和 `onComplete` 以及 ，取消和 求）。

于 行日志的 明

`log` 操作符使用 `Loggers` 工具，它通 `SLF4J` 提取常用的日志 架，如`Log4J`和`Logback`，如果`SLF4J`不可用，到控制台。

控制台后 方案使用 `System.err` 用于 `WARN` 和 `ERROR` 日志，其它的都是 `System.out`。

如果 更喜 `JDK`的 `java.util.logging` 的后 方案，比如在`3.0.x`中，可以通 将 `reactor.logging.fallback` 系 属性 置 `JDK` 来得到。

在所有的情况下，当在生 境中 日志，注意配置底 日志 架尽量使用 非阻塞的方式—例如，`Logback`中的 `AsyncAppender` 或 `Log4j 2`中的 `AsyncLogger`。

例如，假 我 已 配置且激活了`Logback`，并配置了一条像 `range(1,10).take(3)` 的 。通 在 `take` 之前放置一个 `log`，我 可以深入了解其工作原理，以及它向上游 播什 的事件，如下例所示：

```
Flux<Integer> flux = Flux.range(1, 10)
    .log()
    .take(3);
flux.subscribe();
```

将打印出以下内容（通 日志 的控制台附加器）：

```
10:45:20.200 [main] INFO reactor.Flux.Range.1 - | onSubscribe([Synchronous
Fuseable] FluxRange.RangeSubscription) ①
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | request(unbounded) ②
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(1) ③
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(2)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | onNext(3)
10:45:20.205 [main] INFO reactor.Flux.Range.1 - | cancel() ④
```

在里，除了器自己的格式（，程，消息），`log()`操作符可以用其自己的格式
出一些内容：

- ①如果使用的操作符在中多次出，`reactor.Flux.Range.1`是日志的自分。它可以区分个操作符的事件被（在情况下，是range）。可以通过`log(String)`方法名使用自定覆符，在几个分的字符后，的事件被打印出来。里，我得到一次`onSubscribe`用，一次`request`用，三次`onNext`和一次`cancel`用。于第一行，在`onSubscribe`中，我得到了Subscriber的，通常是于操作符的具体化。在方括号之，我得到外的信息，包括操作符是否可以通同或合行自动化。
- ②在第二行，我可以看到，一个从下游向上播的无界的求。
- ③然后range送三个。
- ④在最后一行，我看到`cancel()`。

最后一行（4）是最有意思的。我可以看到里的`take`的作用。在看到足多的元素射后，将序列剪短。而言之，一旦射了用求的数量后，`take()`就会致源`cancel()`。

Chapter 8. 暴露Reactor的指

Reactor 目是一个旨在提高性能和更好地利用 源的 。但是要真正了解一个系 的性能，最好是能其各 件。

也是 什 Reactor提供了一个内置的 [Micrometer](#) 集成的原因。



如果Micrometer不在 路径上，指 将是不可操作的。

8.1. 度器指

Reactor中的 个 操作都是通 程和 度器中描述的 度器抽象来完成的。 就是 什 控 的度器是很重要的，注意 指 始出 可疑的情况并作出相 反 。

要 用 度器指 ， 需要使用以下方法：

```
Schedulers.enableMetrics();
```



在 度器 建 行 。建 尽早 用 方法。



如果 使用的是Spring Boot，那 最好将 用放在 `SpringApplication.run(Application.class, args)` 用之前。

一旦 度器指 被 用，并且只要它在 路径上，Reactor将使用Micrometer的支持来 背后的大多数度器的 行器。

于暴露的指 ， 参考 [Micrometer的文](#) ，如：

- executor_active_threads
- executor_completed_tasks_total
- executor_pool_size_threads
- executor_queued_tasks
- executor_secounds_{count, max, sum}

由于一个 度器可能有多个 行器， 个 行器指 都有一个 `reactor_scheduler_id` 。



Grafana + Prometheus用 可以使用 建 表板，其中包含 程，已完成任 ，任 列和其它有用的指 。

8.2. 生 者指

有 ， 在 式管道的某个 段能 指 是非常有用的。

一 方法是将 手 推送到 的指 后端。 一 是使用Reactor内置的 Flux/Mono 的指 集成，并解析它。

考 下面的管道：

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

了 用 个 Flux (从 listenToEvents() 返回的) 的指 ，我 需要 其命名并 用指 收集：

```
listenToEvents()
    .name("events") ①
    .metrics() ②
    .doOnNext(event -> log.info("Received {}", event))
    .delayUntil(this::processEvent)
    .retry()
    .subscribe();
```

① 在 一 段， 个指 都将会被 定 “事件”。

② Flux#metrics 操作符 用指 告并使用管道中的最后一个的名称。

只需要加上 一个操作符，就会暴露出一大堆有用的指 ！

指 名	型	描述
reactor.subscribed	数	了多少 式序列
reactor.malformed.source	数	从 常的源 (即onComplete 之后的onNext) 接收到的事件
reactor.requested	分 概括	所有 者 命名Flux的 求量，直到至少有一个无界数量的求止
reactor.onNext.delay	器	量onNext信号之 的延
reactor.flow.duration	器	从 到序列 止或取消之 的持。添加状 以指定什事件 致 器 束 (onComplete、onError、cancel)。

想知道 的事件 理由于某些 而重 了多少次？ reactor.subscribed，因 retry() 操作符在 生 会重新 生 者源。

“ 秒的事件数”指 感 趣？ 量 reactor.onNext.delay 的 数的速率。

想在 听器 出 得到告警？ `status=error` 的 `reactor.flow.duration` 是 的朋友。

8.2.1. 常用

个指 都有以下共同的：

名称	描述	例
<code>type</code>	生 者 型	"Mono"
<code>flow</code>	当前流的名称，由 <code>.name()</code> 操作符 置	"events"

8.2.2. 自定

允 用 添加自定 到其 式：

```
listenToEvents()
    .tag("source", "kafka") ①
    .name("events")
    .metrics() ②
    .doOnNext(event -> log.info("Received {}", event))
    .delayUntil(this::processEvent)
    .retry()
    .subscribe();
```

① 置一个自定 “source” “kafka”。

② 除了上述常 的 外，所有 告的指 都会有 `source=kafka` 。

Chapter 9. 高 特性和概念

本章涵 了Reactor的高 特性和概念，包括以下内容：

- 互用操作符用法
- 与冷
- 使用 `ConnectableFlux` 向多个 者广播
- 三 批 理
- 用 `ParallelFlux` 并行化工作
- 替 的 `Schedulers`
- 使用全局 子
- 式序列添加上下文
- 空安全
- 理需要清理的 象

9.1. 互用操作符用法

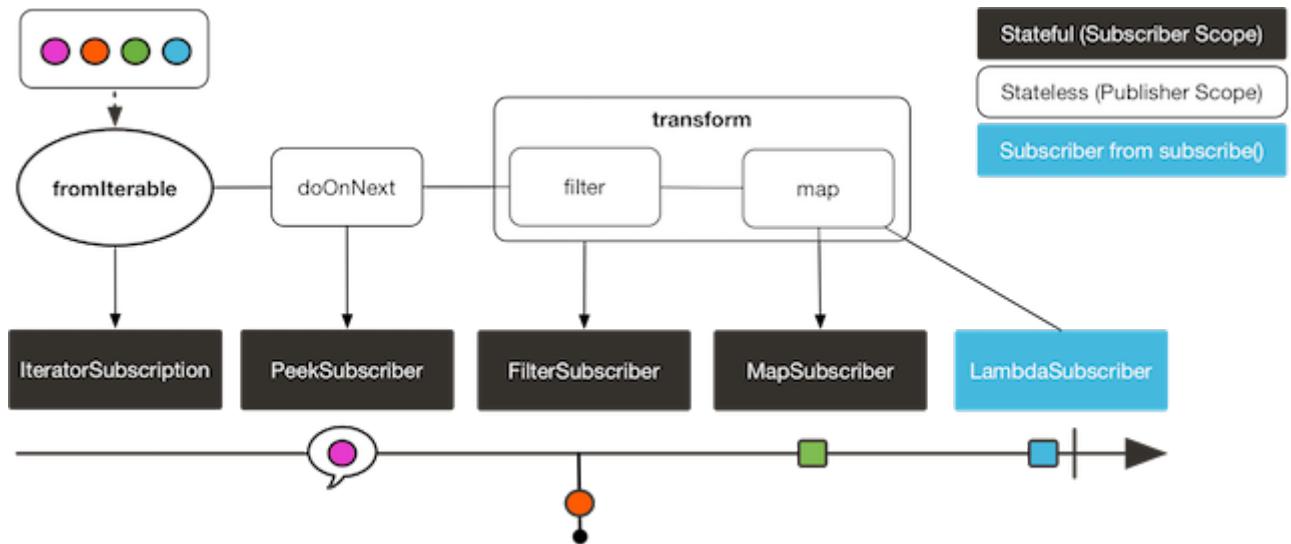
从 的代 角度来看，代 用通常是一件好事。Reactor提供了一些可以 助 重用和互用代 的方式，特 是 于 可能想在代 中 常 用的操作符或操作符的 合。如果 想将操作符 作 配方， 可以 建一个操作符“食 ”的配方。

9.1.1. 使用 `transform` 操作符

`transform` 操作符可以 将操作符 的一部分封装成一个函数。 个函数在 装 被 用到原始操作上，使用封装的操作符来 行 。 做会将相同的操作 用于序列的所有 者，基本上相当于直接操作符。下面的代 示了一个例子：

```
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));
```

下 示了 `transform` 操作符如何封装流：



前面的例子 生以下 出：

```

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

```

9.1.2. 使用 `transformDeferred` 操作符

`transformDeferred` 操作符 似于 `transform`, 也 可以 将操作符封装在一个函数中。主要区
在 于, 此 函 数 是 基 于 一 个 者 用 于 原始序 列。意 味 着 一 个 函 数 上 可 以 产
者 (通 某 状) 生 不 同 的 操 作 符 。下 面 的 代 码 示 了 一 个 例 子 :

```

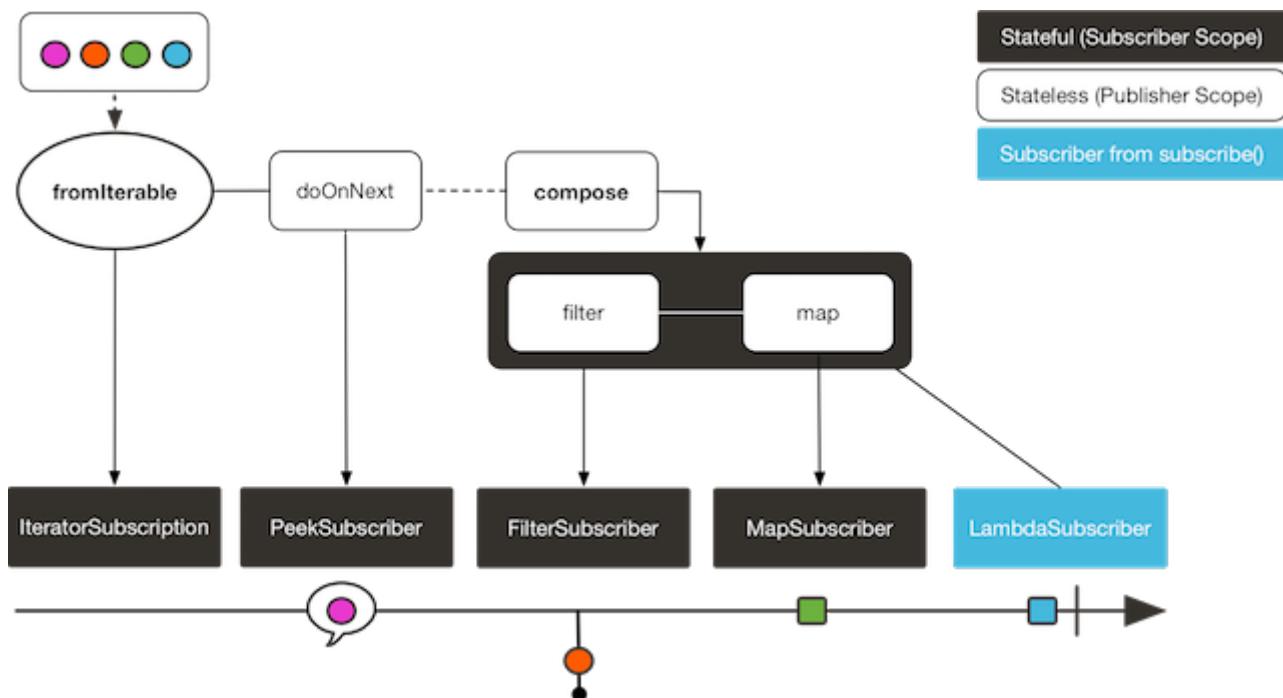
AtomicInteger ai = new AtomicInteger();
Function<Flux<String>, Flux<String>> filterAndMap = f -> {
    if (ai.incrementAndGet() == 1) {
        return f.filter(color -> !color.equals("orange"))
            .map(String::toUpperCase);
    }
    return f.filter(color -> !color.equals("purple"))
       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

```

下图展示了 `transformDeferred` 操作符如何处理一个操作者：



上面的例子生成以下输出：

```
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
```

9.2. 与冷

到目前 止，我 已 所有的 `Flux` (和 `Mono`) 都是一 的：它 都表示一个 的数据序列，在 之前没有任何事情 生。

但 上，生 者有 大派系： 与冷。

前面的描述 用于冷的 生 者。它 会 个 重新生成数据。如果没有 建 ， 数据永 不会被生成。

考 一个HTTP 求： 个新的 生 者触 一次HTTP 用，但是如果没有人 果感 趣，就不 行 用。

一方面， 生 者不依 于任何数量的 生 者。它 可能会立即 始 布数据，并在 当有一个新的 `Subscriber` 出 (在 情况下，当它 后， 生 者只能看到 出的新元素) 做。 于 的生 者来 ， 在 前 会 生 一些事情。

Reactor中 数不多的 操作符的一个例子就是 `just`：在 装 直接 取 ， 然后向 它的任何人重新 出。再次 比于HTTP的 用，如果 取的数据是一次HTTP 用的 果，那 只有一次 用，即初始化 `just` 。

要将 `just` 成一个冷的生 者， 可以使用 `defer`。在我 的示例中，它将HTTP 求推 到 的 时候 (并会 致 于 个新的 生 者有 独的 用)。



Reactor中的 大多数 生 者 展了 `Processor`。

考 外 个例子。下面的代 示的是第一个例子：

```

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));

```

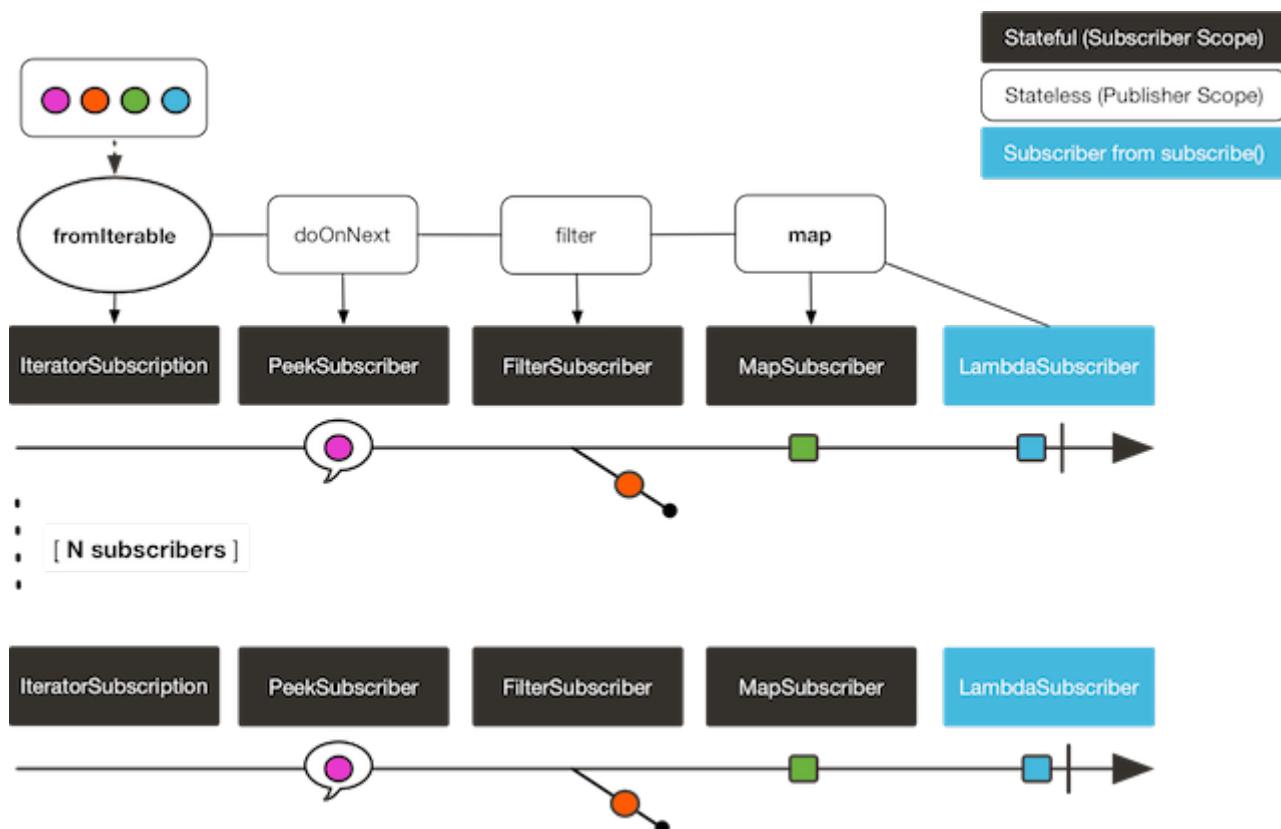
第一个例子 生以下 出：

```

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

```

下 示了重播行：



个 者都能捕 所有的四 色，因 个 者都会 致操作符在 Flux 上定 的 程 行。

将第一个例子和第二个例子 行比 ，如下代 所示：

```
DirectProcessor<String> hotSource = DirectProcessor.create();

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

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

hotSource.onNext("blue");
hotSource.onNext("green");

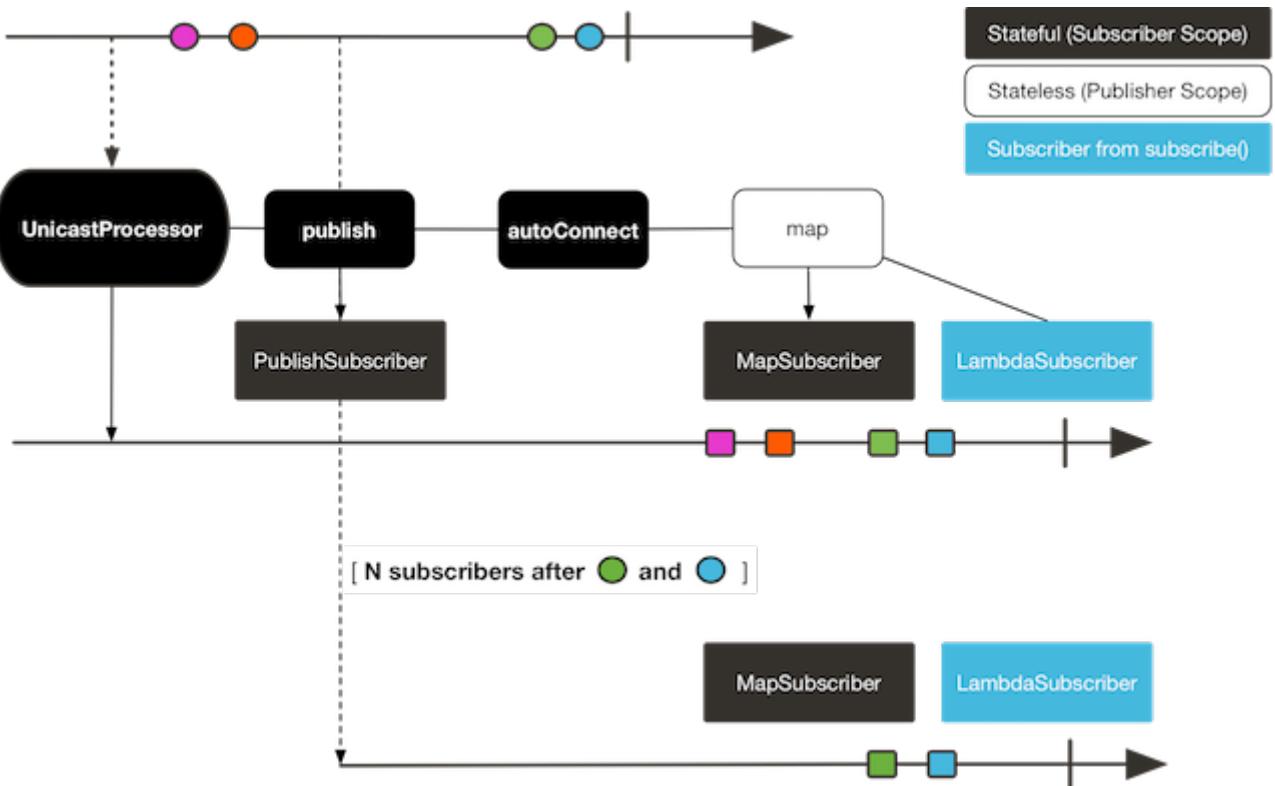
hotFlux.subscribe(d -> System.out.println("Subscriber 2 to Hot Source: "+d));

hotSource.onNext("orange");
hotSource.onNext("purple");
hotSource.onComplete();
```

第二个例子 生以下 出：

```
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
```

下 示了 是如何广播的：



者1捕 了所有四 色。在前面 色 生后 建 者2, 只捕 了后面 色。 差 致 了 ORANGE 和 PURPLE 的 出加倍。Flux上操作符所描述的 程, 无 何 被添加, 都会 行。

9.3. 使用 ConnectableFlux 向多个 者广播

有 , 可能不想延 , 只是推 某些 理到 者的 时候, 而 上是想 他 中的几个聚合, 然后 触 和数据生成。

就是 `ConnectableFlux` 的作用。Flux API 中包含了 个主要的模式, 可以返回一个 `ConnectableFlux` : `publish` 和 `replay`。

- `publish` 地 各个 者的需求, 在背 方面, 通 些 求 源。最 得注意的是, 如果任何 者有一个挂起的 求 `0`, `publish` 将 停向源的 求。
- `replay` 冲第一次 始的数据, 直到 到可配置的限制 (在 和 冲区大小上)。它将重新 出数据 后 者。

`ConnectableFlux` 提供了 外的方法来管理下游 与原始源的 。 些 外方法包括 :

- 一旦 Flux 到足 多的 , 可以手 用 `connect()`。 将触 上游源的 。
- 一旦 到 `n` 个 , `autoConnect(n)` 可以自 做相同的事情。
- `refCount(n)` 不 自 跟踪到来的 , 而且 可以 些 何 被取消。如果跟踪的 者不足, 源将 ”`disconnected`“, 如果 后有 外的 者出 , 将 致 源 生新的 。
- `refCount(int, Duration)` 加了一个 “ 限期 ”。一旦跟踪的 者数量太低, 它会在断 源之前等待 `Duration` 持 , 有可能 足 多的新的 者 入并再次超 接 。

看下面的例子 :

```
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();
```

前面的代码生成以下输出：

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

下面的代码使用 `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 -> {}, () -> {});
```

前面的代码生成下面的输出：

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

9.4. 三 批 理

当 有很多的元素并且想把它 分批的 候，在Reactor中， 大致有三个解决方案：分 ， 口化和 冲 。 三者概念上接近，因 它 将一个 `Flux<T>` 重新分配到一个集合中。分 和 口化会 建一个 `Flux<Flux<T>>`，而 冲 将聚合到一个 `Collection<T>`。

9.4.1. 用 `Flux<GroupedFlux<T>>` 分

分 是将源的 `Flux<T>` 分成多个批次的行 ， 一个批次匹配一个 。

相 的操作符是 `groupBy`。

个 都表示 一个 `GroupedFlux<T>`， 可以通 用其 `key()` 方法来得到 。

的内容的 性是没有必要的。一旦一个源的元素 生成一个新的 ， 的 就会被打 ， 并且与 匹配的元素就会出 在 中（几个 可以同 打 ）。

意味着 ：

1. 是不相交的（一个源元素只属于一个 ）。
2. 可以包含原始序列中不同位置的元素。
3. 永不 空。

下面的例子按 是偶数 是奇数 行分 ：

```

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)) //如果是空的，就示出来
            .map(String::valueOf) //映射字符
            .startWith(g.key()))
)
.expectNext("odd", "1", "3", "5", "11", "13")
.expectNext("even", "2", "4", "6", "12")
.verifyComplete();

```



分最合具有中等到低数的情况。必制性地使用（例如，通过 `flatMap`），以便 `groupBy` 从上游取数据并更多的提供数据。有，一个束成倍加并致挂起，例如当基数高且 `flatMap` 消的并性太低。

9.4.2. Flux<Flux<T>> 口化

口化是将源 `Flux<T>` 根据大小，定界的或界定的 `Publisher` 的准，将源 `Flux<T>` 拆分口的操作。

相的操作符是 `window`, `windowTimeout`, `windowUntil`, `windowWhile` 和 `windowWhen`。

与 `groupBy` 不同的是，后者是根据入的随机重，口（大多数候）是按序打的。

不，有些形式依然是可以重的。例如，在 `window(int maxSize, int skip)` 中，`maxSize` 参数是口后的元素数，而 `skip` 参数是当新的口后源中元素的数量。如果 `maxSize > skip`，会在前一个口前打一个新的口，然后个口重。

下面的例子示的是重的口：

```

StepVerifier.create(
    Flux.range(1, 10)
        .window(5, 3) //重口
        .concatMap(g -> g.defaultIfEmpty(-1)) //空口示-1
)
.expectNext(1, 2, 3, 4, 5)
.expectNext(4, 5, 6, 7, 8)
.expectNext(7, 8, 9, 10)
.expectNext(10)
.verifyComplete();

```



使用相反的配置 (`maxSize < skip`)，某些源的元素被，不属于任何口。

在通 `windowUntil` 和 `windowWhile` 行基于的口化的情况下，后源的元素与

不匹配也可能会致空口，如下例所示：

```
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) // 分触 奇数 1 3 5
    .expectNext(2, 4, 6) // 11 触
    .expectNext(12) // 13 触
    // 但是没有出空的完成口（将包含外的匹配元素）
    .verifyComplete();
```

9.4.3. 用 `Flux<List<T>>` 冲

冲似于口化，但有以下的不同：与生口（个都是一个`Flux<T>`）相反，它生冲区（即`Collection<T>`--情况下`List<T>`）。

用于冲的操作符与口化操作符相同：`buffer`, `bufferTimeout`, `bufferUntil`, `bufferWhile`, 和`bufferWhen`。

当相的口化操作符打一个口，冲操作符建一个新的集合并始向其中添加元素。当口，冲操作符出集合。

冲也可以致源元素或具有重的冲区，如下例所示：

```
StepVerifier.create(
    Flux.range(1, 10)
        .buffer(5, 3) // 重冲区
)
    .expectNext(Arrays.asList(1, 2, 3, 4, 5))
    .expectNext(Arrays.asList(4, 5, 6, 7, 8))
    .expectNext(Arrays.asList(7, 8, 9, 10))
    .expectNext(Collections.singletonList(10))
    .verifyComplete();
```

与在口化中不同，`bufferUntil`和`bufferWhile`不会出空的冲区，如下例所示：

```
StepVerifier.create(  
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)  
        .bufferWhile(i -> i % 2 == 0)  
)  
.expectNext(Arrays.asList(2, 4, 6)) // 11 触  
.expectNext(Collections.singletonList(12)) // 13 触  
.verifyComplete();
```

9.5. 用 `ParallelFlux` 并行化工作

如今，随着多核架 得普遍，能 松 并行化工作很重要。Reactor提供了一 特殊的 `ParallelFlux`，它暴露了 并行化而 化的操作符，从而 助我 了 一点。

要 得一个 `ParallelFlux`， 可以在任何 `Flux` 上使用 `parallel()` 操作符。
方法本身并不会使工作并行化。而是将 分 “道”（ 情况下， 道的数量与 CPU的核数相同）。

了告 生成的 `ParallelFlux` 在 里 行 个 道（并且，通 展，并行化地 行 道）， 必 使用 `runOn(Scheduler)`。注意有一个推 的 用的 `Scheduler` 用于并行化工作：`Schedulers.parallel()`。

比 一下下面 个例子：

```
Flux.range(1, 10)  
.parallel(2) ①  
.subscribe(i -> System.out.println(Thread.currentThread().getName() + " -> " +  
i));
```

① 我 制使用多个 道而不是依 于CPU的核数。

```
Flux.range(1, 10)  
.parallel(2)  
.runOn(Schedulers.parallel())  
.subscribe(i -> System.out.println(Thread.currentThread().getName() + " -> " +  
i));
```

第一个例子 生以下 出：

```
main -> 1
main -> 2
main -> 3
main -> 4
main -> 5
main -> 6
main -> 7
main -> 8
main -> 9
main -> 10
```

第二个例子正 地并行化在 个 程上，如下面的 出所示：

```
parallel-1 -> 1
parallel-2 -> 2
parallel-1 -> 3
parallel-2 -> 4
parallel-1 -> 5
parallel-2 -> 6
parallel-1 -> 7
parallel-1 -> 9
parallel-2 -> 8
parallel-2 -> 10
```

如果 一旦并行 理 的序列， 想要恢 “正常” 的 Flux，并按 序的方式 用其余的操作符 ， 可以使用 ParallelFlux 上的 sequential() 方法。

注意，如果 用一个 Subscriber subscribe ParallelFlux， 会 式的 用 sequential() ，但当使用基于lambda形式的 subscribe 不能。

要注意的是， subscribe(Subscriber<T>) 合并了所有的 道，而 subscribe(Consumer<T>) 行所有的 道。如果 subscribe() 方法具有lambda， 个lambda 行的次数与 道 行的次数相同。

可以通 groups() 方法来 各个 道或 “groups” 作 一个 Flux<GroupedFlux<T>>，并通 composeGroup() 方法 其 用其他的操作符。

9.6. 替 的 Schedulers

正如我 在 程和 度器一 中所描述的那 ，Reactor核心自 了几个 Scheduler 。然 是可以通 new* 工厂方法 建新的 例，但 个 Scheduler 格都有一个 的 例 例，可直接通 工厂方法（例如 Schedulers.boundedElastic() 与 Schedulers.newBoundedElastic(…) ）。

些 的 例是操作符使用的，如果没有明 指定一个 Scheduler ，需要一个 Scheduler

例。例如，`Flux#delayElements(Duration)` 使用 `Schedulers.parallel()` 例。

但是，在某些情况下，可能需要以交叉的方式使用其他东西来更改一些例，而不必保用的个操作符都有指定的 `Scheduler` 作参数。一个例子就是通过包装的调度器来量个调度任花的，以行的目的。句，可能想要改的 `Schedulers`。

可以通 `Schedulers.Factory` 来更改的调度器。情况下，`Factory` 通似的方法建所有准的 `Scheduler`。可以用的自定覆些方法。

此外，工厂暴露了一自定方法：`decorateExecutorService`。它在 `ScheduledExecutorService` (即使是非例，例如通用 `Schedulers.newParallel()` 建的) 所支持的个式核心 `Scheduler` 建程中用。

允整要使用的 `ScheduledExecutorService`：的是暴露 `Supplier`，并根据所配置的 `Scheduler` 的型，可以完全 `supplier` 并返回自己的例，或可以通 `get()` 得到例并将其包装。



一旦建了足需要的 `Factory`，必通用 `Schedulers.setFactory(Factory)` 来其行置。

最后，在 `Schedulers` 中有最后一个可定制的子：`onHandleError`。当提交到 `Scheduler` 的 `Runnable` 任出 `Exception` (注意，如果行任的 `Thread` 置了 `UncaughtExceptionHandler` 理器，理器和子都会被用) 用。

9.7. 使用全局子

Reactor 有一可配置的回，Reactor 操作符在各情况下都会用它。它都被置在 Hooks 中，分三：

- 除子
- 内部子
- 装子

9.7.1. 除子

当源操作符不符合式流，除子将被用。些超出了正常的行路径 (即，它不能通 `onError` 播)。

通常，尽管之前已用了 `onCompleted`，`Publisher` 也会在操作符上用 `onNext`。在情况下，`onNext` 的将被除。于无的 `onError` 信号也是如此。

相的子 `onNextDropped` 和 `onErrorDropped`，允些除提供一个全局的 `Consumer`。例如，如果需要的 (因不会到式的其他部分)，可以使用它来除和清理与某个相的源。

置次子是附加的：用提供的个消者。可以使用 `Hooks.resetOn*Dropped()` 方法将子完全重置。

9.7.2. 内部子

在行 `onNext`, `onError`, 和 `onComplete` 方法出意外的 `Exception`, 操作符将用 `onOperatorError` 子。

与前一不同, 然是在正常的行路径内。一个典型的例子就是有 `map` 函数的 `map` 操作符出 `Exception` (例如, 除以零)。在一点上, 然可以通平常的 `onError` 方式, 正是操作符需要做的。

首先, 它通 `onOperatorError` `Exception`。个子可以 (以及相的致的) 并更改 `Exception`。当然, 可以在一旁做一些事情, 比如日志并返回原始的 `Exception`。

注意, 可以多次置 `onOperatorError` 子。可以特定的 `BiFunction` 提供一个 `String` 符, 后不同的用将些函数接起来, 些函数都会被行。一方面, 重使用同一个次可以替之前置的函数。

因此, 可以完全重置 (通使用 `Hooks.resetOnOperatorError()`) 子的行或只指定的 `key` 行部分重置 (通使用 `Hooks.resetOnOperatorError(String)`)。

9.7.3. 装子

些子和操作符的生命周期密相。当一个操作符装 (即例化) 被用。`onEachOperator` 通返回不同的 `Publisher` 允改装在中的个操作符。`onLastOperator` 也是似的, 除了在的最后一个操作符即 `subscribe` 用之前被用。

如果想要用横切 `Subscriber` 来装所有的操作符, 可以研究一下 `Operators#lift*` 方法, 以助各型的 `Reactor` 的 `Publishers` (`Flux`, `Mono`, `ParallelFlux`, `GroupedFlux` 和 `ConnectableFlux`), 以及它的 `Fuseable` 版本。

像 `onOperatorError` 一, 些子是累的, 可以用一个来。它也可以被部分或全部重置。

9.7.4. 子

`Hooks` 工具提供了个子。可以通过相的方法来性的替行, 而不是自己定子:

- `onNextDroppedFail()`: `onNextDropped` 用于出一个 `Exceptions.failWithCancel()` 常。它在在除的。要回到以前的出行, 使用 `onNextDroppedFail()`。
- `onOperatorDebug()`: 此方法会激活模式。它与 `onOperatorError` 子密相, 因此用 `resetOnOperatorError()` 也能重置它。因它在内部使用了一个特定的, 也可以通使用 `resetOnOperatorDebug()` 独重置它。

9.8. 式序列添加上下文

从命令式程点到式程思遇到的重大技挑之一在于如何程化。

与可能的命令式程相反, 在式程中, 可以使用 `Thread` 理几个大致同 (上, 在非阻塞的) 行的序列。行也可以很容易且常从一个程跳到一个程。

定于使用依于程模型使得更“定”的特性的者相当困, 例如 `ThreadLocal`。因

它可以 把数据与 程 起来，但在 式上下文中使用它就 得很棘手。因此，依 于 `ThreadLocal` 的 与Reactor一起使用 ，至少 来了新的挑 。最糟 的是，它 不能工作或者甚至失 。使用Logback的MDC来存 并 日志相 性ID就是 情况的一个典型例子。

使用 `ThreadLocal` 的通常解决方法是通 使用（例如）`Tuple2<T, C>` 按 序将上下文的数据 `C` 沿 数据 `T` 移 。 看起来不好，并且将正交 系（上下文数据）泄露到方法和 `Flux` 名中。

从 3.1.0 版本 始，Reactor自 了 似于 `ThreadLocal` 的一个高 功能，但可以 用于 `Flux` 或 `Mono` 而不是 `Thread`。 个特性称 `Context`。

了 明它是什 子的，下面的例子同 从 `Context` 写入和 取：

```
String key = "message";
Mono<String> r = Mono.just("Hello")
    .flatMap( s -> Mono.subscriberContext()
        .map( ctx -> s + " " + ctx.get(key)))
    .subscriberContext(ctx -> ctx.put(key, "World"));

StepVerifier.create(r)
    .expectNext("Hello World")
    .verifyComplete();
```

在下面的章 中，我 将介 `Context` 以及如何使用它，以便 最 能 理解前面的例子。



是一个更 合 人 的高 功能。它需要充分理解 `Subscription` 的生命周期，并且用于 。

9.8.1. Context API

`Context` 是一个 似于 `Map` 的接口。它存 ，并允 根据 取 存 的 。更具体地：

- 和 都属于 `Object` 型，因此一个 `Context` 例可以包含来源于不同 和源的任意数量且有巨大差 的 。
- `Context` 是不可更改的。
- 使用 `put(Object key, Object value)` 存 一个 ，返回一个新的 `Context` 例。 可以通 使用 `putAll(Context)` 将 个上下文合并到一个新的上下文中。
- 可以通 `hasKey(Object key)` 是否存在。
- 使用 `getOrDefault(Object key, T defaultValue)` 来 取 （ ）或如果 `Context` 例没有 返回 。
- 使用 `getOrEmpty(Object key)` 得一个 `Optional<T>` (`Context` 例 存 ）。
- 使用 `delete(Object key)` 来 除与某个 相 的 ，返回一个新的 `Context`。

当建一个 `Context`，可以通 使用静 的 `Context.of` 方法 建最多5个的 `Context` 例。它 取2, 4, 6, 8或10个 `Object` 例， `Object` 例都是要添加到 `Context` 的。

外， 也可以通 使用 `Context.empty()` 建一个空的 `Context`。

9.8.2. 将 `Context` 定到 `Flux` 并 写

了使 `Context` 有用，它必 与一个特定的序列 定，并且可以被 中的 个操作符。注意，操作符必 是Reactor的原生操作符，因 `Context` 是Reactor所特有的。

上，`Context` 与 中的 个 `Subscriber` 所 定。它使用 `Subscription` 播机制使其在 个操作符上都可用，从最后的 `subscribe` 始向上移 。

了填充只能在 完成的 `Context`， 需要使用 `subscriberContext` 操作符。

`subscriberContext(Context)` 合并 提供的 `Context` 和来自下游（ 住，`Context` 是从 的底部向上播的）的 `Context`。 是通 用 `putAll` 完成的，从而 生一个新的上游 `Context`。



也可以使用更高 的 `subscriberContext(Function<Context, Context>)`。它从下游接收 `Context` 的状 ，它允 根据需要 加或 除 ，并返回新的 `Context` 来使用。 甚至可以决定返回一个完全不同的 例，尽管 上不建 （ 做可能会影 到依 于 `Context` 的第三方 ） 做。

9.8.3. 取 `Context`

一旦 填充了一个 `Context`， 就可以 索数据。在大多数情况下，将信息放到 `Context` 的 是在最 用 ，而利用 些信息是第三方 中，因 些 通常在客 端代 的上游。

从上下文中 取数据的工具是 `Mono.subscriberContext()` 静 方法。

9.8.4. 的 `Context` 示例

本 中的示例是 了更好地理解一些使用 `Context` 的注意事 。

首先，我 回 一下我 引言中的 示例，如下示例所示：

```

String key = "message";
Mono<String> r = Mono.just("Hello")
    .flatMap( s -> Mono.subscriberContext() ②
        .map( ctx -> s + " " + ctx.get(key))) ③
    .subscriberContext(ctx -> ctx.put(key, "World")); ①

StepVerifier.create(r)
    .expectNext("Hello World") ④
    .verifyComplete();

```

- ① 操作符 以 `subscriberContext(Function)` 用 束， 用以 "message" ， 将 "World" 放到 `Context` 中。
- ② 我 源元素 行 `flatMap`， 用 `Mono.subscriberContext()` 具体化 `Context`。
- ③ 然后， 我 用 `map` 来提取与 "message" 相 的数据，并将其与原来的 行 接。
- ④ 由此 生 `Mono<String>` 出 "Hello World"。



上面的数字与 的行 序没有 系。它代表的是 行 序。即使 `subscriberContext` 是 的最后一部分，它 然是最先被 行（由于它 性 以及 信号从下至上流的事 ）的那个。



在 的操作符 中， 写入 `Context` 和 取 `Context` 的相 位置是很重要的。`Context` 是不可 的， 其内容只能被它上面的操作符看到， 如下面的例子：

```

String key = "message";
Mono<String> r = Mono.just("Hello")
    .subscriberContext(ctx -> ctx.put(key, "World")) ①
    .flatMap( s -> Mono.subscriberContext()
        .map( ctx -> s + " " +
ctx.getOrDefault(key, "Stranger"))); ②

StepVerifier.create(r)
    .expectNext("Hello Stranger") ③
    .verifyComplete();

```

- ① 在 中写入 `Context` 的位置太在上面了。
- ② 因此，在 `flatMap` 中， 里没有我 的 。而是使用了一个 。
- ③ 由此 生的 `Mono<String>` 出 "Hello Stranger"。

下面的例子 演示了 `Context` 的不可 特性， 以及 `Mono.subscriberContext()` 如何始 返回由 `subscriberContext` 用 置的 `Context`：

```

String key = "message";

Mono<String> r = Mono.subscriberContext() ①
    .map( ctx -> ctx.put(key, "Hello")) ②
    .flatMap( ctx -> Mono.subscriberContext()) ③
    .map( ctx -> ctx.getOrDefault(key,"Default")); ④

StepVerifier.create(r)
    .expectNext("Default") ⑤
    .verifyComplete();

```

- ① 我 将 Context 具体化
- ② 在 map 中，我 将其
- ③ 我 在 flatMap 中重新 了 Context
- ④ 我 在 Context 中 取
- ⑤ 的 没有 "Hello"

同，在多次 将同一个 写入 Context 的情况下，写入的相 序也很重要。取 Context 的操作符会看到最接近它 置的 ，如下例所示：

```

String key = "message";
Mono<String> r = Mono.just("Hello")
    .flatMap( s -> Mono.subscriberContext()
        .map( ctx -> s + " " + ctx.get(key)))
    .subscriberContext(ctx -> ctx.put(key, "Reactor")) ①
    .subscriberContext(ctx -> ctx.put(key, "World")); ②

StepVerifier.create(r)
    .expectNext("Hello Reactor") ③
    .verifyComplete();

```

- ① 写入 "message"。
- ② 一次 写入 "message"。
- ③ map 只看到了最接近它（在它下面） 置的："Reactor"。

在前面的例子中，Context 在 期 被填充了 "World"。然后 信号向上移 ，一个写操作生了。就 生了第二个不可 的 Context，其 "Reactor"。之后，数据 始流 。flatMap 看到他最近的 Context，也就是我的第二个 Context，其 "Reactor"。

可能会想知道，Context 是否会随着数据信号一起 播。如果是那 情况下的 ，再在 个写操作之放置 一个 flatMap，那 就会使用最上面的 Context。但事 并未如此，下面的例子就 明了 一点：

```

String key = "message";
Mono<String> r = Mono.just("Hello")
    .flatMap( s -> Mono.subscriberContext()
        .map( ctx -> s + " " + ctx.get(key))) ③
    .subscriberContext(ctx -> ctx.put(key, "Reactor")) ②
    .flatMap( s -> Mono.subscriberContext()
        .map( ctx -> s + " " + ctx.get(key))) ④
    .subscriberContext(ctx -> ctx.put(key, "World")); ①

StepVerifier.create(r)
    .expectNext("Hello Reactor World") ⑤
    .verifyComplete();

```

- ① 第一次写操作。
- ② 第二次写操作。
- ③ 第一个 flatMap 看到第二次写入的 。
- ④ 第二个 flatMap 将第一次的 果与第一次写入的 接起来。
- ⑤ Mono 出 "Hello Reactor World"。

原因是 Context 与 Subscriber 相 ， 个操作符通 下游的 Subscriber 来 求 Context。

最后一个有趣的 播情况是将 Context 也被写到 flatMap 中，如下例所示：

```

String key = "message";
Mono<String> r =
    Mono.just("Hello")
        .flatMap( s -> Mono.subscriberContext()
            .map( ctx -> s + " " + ctx.get(key)))
        )
        .flatMap( s -> Mono.subscriberContext()
            .map( ctx -> s + " " + ctx.get(key))
            .subscriberContext(ctx -> ctx.put(key, "Reactor"))
        ①
        )
        .subscriberContext(ctx -> ctx.put(key, "World")); ②

StepVerifier.create(r)
    .expectNext("Hello World Reactor")
    .verifyComplete();

```

- ① 个 subscriberContext 不会影 到 flatMap 之外的任何 西。
- ② 个 subscriberContext 会影 到主序列的 Context。

在前面的例子中，最 出的 "Hello World Reactor" 而不是 "Hello Reactor World"，因 写

"Reactor" 的 `subscriberContext` 是作 第二个 `flatMap` 的内部序列的一部分。因此，它不可 或通 主序列 播，且第一个 `flatMap` 也看不到它。 播和不可 性将 建中 内部序列的操作符（例如 `flatMap`）中的 `Context` 隔 。

9.8.5. 完整的例子

在我 可以考 一个更真 的例子，一个 从 `Context` 中 取信息：一个将 `Mono<String>` 作 `PUT` 的数据源，但同 也会 一个特定的上下文 ，以将相 的ID添加到 求 中的 式HTTP客 端。

从用 的角度来看， 用如下：

```
doPut("www.example.com", Mono.just("Walter"))
```

了 播一个相 的ID，它将 用如下：

```
doPut("www.example.com", Mono.just("Walter"))
    .subscriberContext(Context.of(HTTP_CORRELATION_ID, "2-j3r9afaf92j-afkaf"))
```

正如前面的代 片段所示，用 代 使用 `subscriberContext` 填充具有 `HTTP_CORRELATION_ID` 的 `Context`。操作符的上游是由HTTP客 端 返回的 `Mono<Tuple2<Integer, String>>` (HTTP 的 表示)。所以它有效地将信息从用 代 架代 。

下面的例子 示了从 架角度的模 代 ， 取上下文， 到相 ID并“ 造 求”。

```

static final String HTTP_CORRELATION_ID = "reactive.http.library.correlationId";

Mono<Tuple2<Integer, String>> doPut(String url, Mono<String> data) {
    Mono<Tuple2<String, Optional<Object>>> dataAndContext =
        data.zipWith(Mono.subscriberContext() ①
                    .map(c -> c.getOrDefault(HTTP_CORRELATION_ID))); ②

    return dataAndContext
        .<String>handle((dac, sink) -> {
            if (dac.getT2().isPresent()) { ③
                sink.next("PUT <" + dac.getT1() + "> sent to " + url + " with
header X-Correlation-ID = " + dac.getT2().get());
            }
            else {
                sink.next("PUT <" + dac.getT1() + "> sent to " + url);
            }
            sink.complete();
        })
        .map(msg -> Tuples.of(200, msg));
}

```

① 通过 `Mono.subscriberContext()` 具体化 `Context`。

② 提取相 `ID` 的 `Optional`。

③ 如果 存在于上下文中， 使用相 的 `ID` 作 。

架代 段用 `Mono.subscriberContext()` 数据 `Mono`。 架提供了 `Tuple2<String, Context>`， 并且上下文中包含了来自下游（因 它位于直接 的路径）的 `HTTP_CORRELATION_ID` 条目。

然后， 架代 使用 `map` 提取 的 `Optional<String>`，如果 条目存在，它将 的相 `ID` 作 `X-Correlation-ID`。 最后一部分由 `handle` 模 。

整个 架代 使用的相 `ID` 的整个 可以写成如下所示：

```

@Test
public void contextForLibraryReactivePut() {
    Mono<String> put = doPut("www.example.com", Mono.just("Walter"))
        .subscriberContext(Context.of(HTTP_CORRELATION_ID, "2-j3r9afaf92j-
afkaf"))
        .filter(t -> t.getT1() < 300)
        .map(Tuple2::getT2);

    StepVerifier.create(put)
        .expectNext("PUT <Walter> sent to www.example.com with header X-
Correlation-ID = 2-j3r9afaf92j-afkaf")
        .verifyComplete();
}

```

9.9. 理需要清理的象

在非常特殊的情况下，的用程序可能会理那些一旦不再使用就需要某形式清理的型。是一个高的景—例如，当有引用数象或理堆外象。Netty的ByteBuf就是一个很好的例子。

了保此象行正的清理，需要基于 Flux-by-Flux 以及在几个全局子（参考[使用全局子](#)）中其行明：

- `doOnDiscard Flux/Mono` 操作符
- `onOperatorError` 子
- `onNextDropped` 子
- 操作符特定的理器

是必要的，因一个子都考到了特定的清理子集，用可能希望（例如）除了在 `onOperatorError` 中的清理之外，需要特定的理。

注意，有些操作符不太合理需要清理的象。例如，`bufferWhen` 可以引入重的冲区，意味着我之前使用的已的“本地子”可能会将第一个冲区被，并清理其中的一个元素，而个元素在第二个冲区中然有效。

! 了便于清理，所有些子必是等的。在某些情况下，它可能会被多次用于同一象。与行 `instanceof` 的 `doOnDiscard` 操作符不同，全局的子理的例可以是任何 `Object`。区分些例需要清理和不需要清理，取决于用的。

9.9.1. `doOnDiscard` 操作符或者本地子

子用于清理那些永不会被用代暴露的象。它旨在用于在正常情况下行的流（而不是推送很多元素且被 `onNextDropped` 覆的源）的清理子。

它是局部的，即它是通操作符激活的，并且用于定的 Flux 或者 Mono。

很明的情况包括从上游元素的操作符。一些元素永远不会到下一个操作符（或最是正常行途的一部分。因此，它被到doOnDiscard子。可能使用doOnDiscard子的例子包括以下情况：

- `filter`：不符合器的被“”。
- `skip`：跳的将被。
- `buffer(maxSize, skip)` 与 `maxSize < skip`：“的冲区”—冲区之的元素被。

但doOnDiscard并不局限于操作符，而且被用于在内部数据行排以到背目的操作符。更具体地，在大多数情况下，在取消程中很重要。从源中先提取数据，然后按需布到者的操作符可能在被取消未出数据。的操作符使用doOnDiscard子在取消清理它内部的背列。



doOnDiscard(Class, Consumer)的次用都是与其它操作符一起的，以使其只能被其上游的操作符看到并使用。

9.9.2. onOperatorError 子

onOperatorError子旨在以横向的方式修改（似于AOP的捕和重新出常）。

当在理onNext信号期生，将要出的元素被onOperatorError。

如果型的元素需要清理，需要在onOperatorError子中它，可能是在重写代之上。

9.9.3. onNextDropped 子

于格式不正的Publishers，在某些情况下，操作符可能在期没有元素的情况下（通常是在收到onError或onComplete信号之后）接收到一个元素。在情况下，不期望的元素是“除的”—即onNextDropped子。如果有必要清理的型，必在onNextDropped子中到些型，并在那里清理代。

9.9.4. 操作符特定的理器

一些理冲区或将收集作其操作的一部分的操作符，有着特定的理器，以理所收集的数据不向下游播的情况。如果使用的此操作符的型需要清理，需要在些理器中行清理。

例如，`distinct`有—个回，在操作符止（或取消）用回函数，以便清除用于判断元素是否不同的集合。情况下，集合是一个`HashSet`，清理的回函数`HashSet::clear`。但是，如果理的是引用数的象，可能想把它更改一个更的理器，能在用`clear()`之前`release`集合中的个元素。

9.10. 空安全

尽管Java不允在其型系中表示null安全，但Reactor在提供了注解来声明API的可空性，似与Spring5提供的注解。

Reactor使用些注解，但是它也可以用于任何基于Reactor的Java目中来声明可空的API。方法体内使用的型的可空性不在此功能之内。

些注解是用 [JSR 305](#) 行元注解(一 被IntelliJ IDEA之 的工具支持的潜在JSR), Java 人 提供与空安全相 的有用的警告, 以避免 行 出 [NullPointerException](#)。JSR 305 元注解允 IDE厂商以通用的方式提供空安全支持, 而不必 Reactor注解提供硬 支持。



在Kotlin 1.1.5+ 中, 不需要也不建 在 的 目 路径下依 JSR 305。

它 也被Kotlin使用, Kotlin原生支持 [空安全](#)。 看 [—](#) 了解更多 信息。

`reactor.util.annotation` 包中提供了以下注解 :

- `@NotNull` : 表示特定的参数, 返回 或字段不能 `null`。 (在使用 `@NotNullApi` 的参数和返回上不需要它) 。
- `@Nullable` : 表示参数, 返回 或字段可以 `null`。
- `@NotNullApi` : 表示参数和返回 非空的包 注解。



尚不支持泛型 型参数, 量参数和数 元素的可空性。 看 [issue #878](#) 取最新信息。

Appendix A: 我需要一个操作符？



在本中，如果一个操作符是 Flux 或 Mono，它会相地加上前。普通的操作符是没有前的。当一个特定的用例被操作符合覆，它以方法用的形式呈，在括号中加上前点和参数，如下所示：`.methodCall(parameter)`。

我想理：

- 建一个新的序列...
- 已有的序列
- 序列
- 探序列
- 理
- 与的合作
- 拆分 Flux
- 回到同的世界
- 广播 Flux 到多个 Subscribers

A.1. 建一个新的序列...

- 出 T，且我已有了一个：just
 - ...来自 `Optional<T> : Mono#justOrEmpty(Optional<T>)`
 - ...从可能`空` T : `Mono#justOrEmpty(T)`
- 出一个由方法返回的 T：也是 just
 - ...延取：使用 `Mono#fromSupplier` 或在 `defer` 中包装 just
- 出几个 T，可以明地列：`Flux#just(T…)`
- 迭代：
 - 数：`Flux#fromArray`
 - 集合或可迭代：`Flux#fromIterable`
 - 整数：`Flux#range`
 - 一个提供了一个流：`Flux#fromStream(Supplier<Stream>)`
- 从各源出的，例如：
 - 一个 `Supplier<T> : Mono#fromSupplier`
 - 任：`Mono#fromCallable, Mono#fromRunnable`
 - 一个 `CompletableFuture<T> : Mono#fromFuture`
- 完成：`empty`
- 立即出：`error`
 - ...延建 `Throwable : error(Supplier<Throwable>)`

- 不做任何事情：`never`
- 在 决定：`defer`
- 依 于一次性 源：`using`
- 以 程的方式生成事件（可以使用状 ）：
 - 同 逐个生成：`Flux#generate`
 - （也可以同 ）， 可以一次 射多个信号：`Flux#create` (`Mono#create`)
没有
射多个信号的特性)

A.2. 已有的序列

- 我想 已有的数据：
 - 按一 一（如字符串的 度）：`map`
 - ...通 映射：`cast`
 - ... 了 一个源 的索引：`Flux#index`
 - 在1 n的基 上（如字符串到其字符）：`flatMap` + 使用一个工厂方法
 - 在1 n的基 上， 一个源元素和/或状 提供 程行 ：`handle`
 - 一个源条目（如,url到http 求） 行一个 任 ：`flatMap` + 一个 的 `Publisher` 返回方法
 - ...忽略一些数据：在flatMap lambda中有条件地返回一个 `Mono.empty()`
 - ...保留原始序列的 序：`Flux#flatMapSequential` (会立即触 理，并重新排序 果)
 - ... 任 可以从一个 `Mono` 源返回多个 ：`Mono#flatMapMany`
- 我想将 元素添加到 有序列中：
 - 在 始：`Flux#startWith(T…)`
 - 在末尾：`Flux#concatWith(T…)`
- 我想聚合一个 `Flux`：(假 前 `Flux#`)
 - 聚合到List：`collectList`, `collectSortedList`
 - 聚合 Map：`collectMap`, `collectMultiMap`
 - 聚合到任意容器中：`collect`
 - 聚合 序列的大小：`count`
 - 通 在 个元素之 用函数（例如， 算 和）：`reduce`
 - ...但 出 个中 ：`scan`
 - 从 聚合 布 ：
 - 用于所有 (和)：`all`
 - 用于至少一个 (或)：`any`
 - 是否存在任何 ：`hasElements`
 - 是否存在特定 ：`hasElement`
- 我想 合生 者...

- 按序：`Flux#concat` 或 `.concatWith(other)`
 - ...延任何，直到剩余的生者全部出后：`Flux#concatDelayError`
 - ...上后布者：`Flux#mergeSequential`
- 按出的序（合后的出和它来一）：`Flux#merge / .mergeWith(other)`
 - ...不同型（合并）：`Flux#zip / Flux#zipWith`
- 按配合：
 - 将n个Mono合成一个`Tuple2 : Mono#zipWith`
 - 全部完成后从n个Mono中合并：`Mono#zip`
- 通配合它的止而合并：
 - 将1个Mono和任何源合并：`Mono<Void> : Mono#and`
 - 全部完成从n个源合并：`Mono#when`
 - 合并任意容器型：
 - 次所有的都出：`Flux#zip`（直到最小基数）
 - 次有新的：`Flux#combineLatest`
- 只考先出的序列：`Flux#first, Mono#first, mono.or(otherMono).or(thirdMono), flux.or(otherFlux).or(thirdFlux)`
- 由源序列中的元素触：`switchMap`（一个源元素都映射到生者）
- 由生者序列中的下一个生者始触：`switchOnNext`
- 我想重一个已有序列：`repeat`
 - ...隔一段：`Flux.interval(duration).flatMap(tick → myExistingPublisher)`
- 我有一个空序列，但...
 - 我要一个代替：`defaultIfEmpty`
 - 我想要一个序列：`switchIfEmpty`
- 我有一个序列，但我不感兴趣：`ignoreElements`
 - ...我要想把完成的内容表示：`Mono : then`
 - ...我想等待一个任束后再去完成：`thenEmpty`
 - ...我想在尾切到一个`Mono : Mono#then(mono)`
 - ...我想在末尾出一个：`Mono#thenReturn(T)`
 - ...我想在最后切到`Flux : thenMany`
- 我有一个想要延完成的`Mono`...
 - ...直到的派生的生者完成止：`Mono#delayUntil(Function)`
- 我想将元素地展成一个序列，并出合...
 - ...首先展的度：`expand(Function)`
 - ...首先展的深度：`expandDeep(Function)`

A.3. 探序

- 在不修改最序的情况下，我希望：
 - 得到通知/行其它行（有称“副作用”）：
 - 出：`doOnNext`
 - 完成：`Flux#doOnComplete`, `Mono#doOnSuccess`（包括果，如果有的）
 - 止：`doOnError`
 - 取消：`doOnCancel`
 - 序列的“始”：`doFirst`
 - 与 `Publisher#subscribe(Subscriber)` 定的
 - 后：`doOnSubscribe`
 - 如 `subscribe` 后的 `Subscription`
 - 与 `Subscriber#onSubscribe(Subscription)` 定的
 - 求：`doOnRequest`
 - 完成或：：`doOnTerminate` (`Mono`包括果，如果有的)
 - 但在播到下游之后：`doAfterTerminate`
 - 任何型的信号，表示信号：`Flux#doOnEach`
 - 任何止条件（完成，，取消）：`doFinally`
 - 内部生了什：`log`
- 我想知道所有的事件：
 - 个事件都表示 `Signal` 象：
 - 在序列外的回中：`doOnEach`
 - 而不是原始的 `onNext` 出：`materialize`
 - ...然后返回到 `onNexts`：`dematerialize`
 - 作日志中的一行：`log`

A.4. 序列

- 我想一个序列：
 - 基于任意条件：`filter`
 - ...算：`filterWhen`
 - 限制出象的型：`ofType`
 - 完全忽略：`ignoreElements`
 - 忽略重的：
 - 在整个序列（集合）中：`Flux#distinct`
 - 在后出的条目（重数据除）之：`Flux#distinctUntilChanged`

- 我只想只保留序列的一个子集：
 - 取N个元素
 - 在序列的 : `Flux#take(long)`
 - ...按 周期 : `Flux#take(Duration)`
 - ...只有第一个元素作 一个 `Mono` : `Flux#next()`
 - ...使用 `request(N)` 而不是取消 : `Flux#limitRequest(long)`
 - 在序列的末尾 : `Flux#takeLast`
 - 直到 足条件 (包括) 止 : `Flux#takeUntil` (基于) , `Flux#takeUntilOther` (基于生者的 外的生 者)
 - 足条件 (不包括) : `Flux#takeWhile`
- 最多取一个元素 :
 - 在某个位置 : `Flux#elementAt`
 - 在末尾 : `.takeLast(1)`
 - ...如果 空 出 : `Flux#last()`
 - ...如果 空, 出 : `Flux#last(T)`
- 通 跳 元素 :
 - 在序列的 : `Flux#skip(long)`
 - 按 周期 : `Flux#skip(Duration)`
 - 在序列的末尾 : `Flux#skipLast`
 - 直到 足条件 (包括) 止 : `Flux#skipUntil` (基于) , `Flux#skipUntilOther` (基于生者的 外的生 者)
 - 足条件 (不包括) : `Flux#skipWhile`
- 通 采 元素 :
 - 按 周期: `Flux#sample(Duration)`
 - 将第一个元素保留在采 口中, 而不是最后一个 : `sampleFirst`
 - by a publisher-based window: `Flux#sample(Publisher)`
 - 基于生 者 口 : `Flux#sample(Publisher)`
 - 基于生 者的 “超 ” : `Flux#sampleTimeout` (个元素触 一个生 者, 如果 生 者与下一个不重 出)
- 我期望最多1个元素 (如果多于一个, 会出) ...
 - 如果序列 空, 我想要一个 : `Flux#single()`
 - 如果序列 空, 我想要一个 : `Flux#single(T)`
 - 我也接收空序列 : `Flux#singleOrDefault`

A.5. 理

- 我想 建一个 序列 : `error...`
 - ...替 成功完成的 `Flux` : `.concat(Flux.error(e))`

- ...替 成功 出的 Mono : .then(Mono.error(e))
- ...如果onNext之 的 隔太 : timeout
- ...延 : error(Supplier<Throwable>)
- 我想等效于try/catch :
 - 出 : error
 - 捕 常 :
 - 回退到 : onErrorReturn
 - 回退到 一个 Flux 或 Mono : onErrorResume
 - 包装并重新 出 : .onErrorMap(t → new RuntimeException(t))
 - finally : doFinally
 - Java 7 始的使用方式 : using 工厂方法
- 我想从 中恢 ...
 - 通 回退 :
 - 回退到某个 : onErrorReturn
 - 回退到 Publisher 或 Mono, 取决于 的不同 : Flux#onErrorResume 和 Mono#onErrorResume
 - 策略 (最大 次数) : retry, retry(long)
 - ...由伴随的Flux触 : retryWhen
 - ...使用 准的回退策略 (的指数退避) : retryWhen(Retry.backoff(…)) (中的其他工厂方法)
- 我想 理背 " " (从上游 求最大 求量, 并在下游没有 生足 多 求 用 策略) ...
 - 通 出一个特殊的 IllegalStateException : Flux#onBackpressureError
 - 通 掉多余的 : Flux#onBackpressureDrop
 - ...除了最后一次看到的 : Flux#onBackpressureLatest
 - 通 冲多余的 (有界或无界) : Flux#onBackpressureBuffer
 - ...当有界 冲区也溢出 用 策略 : 使用 BufferOverflowStrategy 的 Flux#onBackpressureBuffer

A.6. 与 的合作

- 我想将 出和 量的 (Tuple2<Long, T>) 相 ...
 - 自 始 : elapsed
 - 自始至今 (算机) : timestamp
- 如果在 次 出之 有太多的延 , 我希望序列被中断 : timeout
- 我想得到以固定 隔的 周期 : Flux#interval
- 我想在初始延 后 出一个 0 : 静 的 Mono.delay
- 我想引入延 :

- 在 `↑onNext` 信号之后 : `Mono#delayElement`, `Flux#delayElements`
- 在 前 生 : `delaySubscription`

A.7. 拆分 Flux

- 我想按一个 界条件把 `Flux<T>` 拆分成 `Flux<Flux<T>>` :
 - 大小 : `window(int)`
 - ...重 或下降的 口 : `window(int, int)`
 - : `window(Duration)`
 - ...重 或下降的 口 : `window(Duration, Duration)`
 - 大小或 (到 数或者超 后 口) : `windowTimeout(int, Duration)`
 - 按元素上的 : `windowUntil`
 - ...在下一个 口中 (`cutBefore` 形式) 出触 界的元素 : `.windowUntil(predicate, true)`
 - ...当元素匹配 保持 口打 : `windowWhile` (不匹配的元素不被 出)
 - 由控件生 者中的 `onNext` 表示的任意 界 : `window(Publisher)`, `windowWhen`
- 我想把 `Flux<T>` 和 界内的 冲元素拆分在一起...
 - 分成 `List`:
 - 通 大小 界 : `buffer(int)`
 - 冲区重 或下降 : `buffer(int, int)`
 - 通 持 界 : `buffer(Duration)`
 - ... 冲区重 或下降 : `buffer(Duration, Duration)`
 - 通 大小或 界 : `bufferTimeout(int, Duration)`
 - 通 任意条件 界 : `bufferUntil(Predicate)`
 - ...将触 界的元素放到下一个 冲区中 : `.bufferUntil(predicate, true)`
 - ...在 匹配 行 冲, 并 除触 界的元素 : `bufferWhile(Predicate)`
 - 由控件生 者中的 `onNext` 表示的任意 界 : `buffer(Publisher)`, `bufferWhen`
 - 拆分成任意“集合”型 `C` : 使用 似 `buffer(int, Supplier<C>)` 的形式
- 我想分割一个 `Flux<T>`, 以便有相同特征的元素可以在同一个子flux中 : `groupBy(Function<T,K>)` TIP: 注意, 将返回一个 `Flux<GroupedFlux<K, T>>`, 一个内部的 `GroupedFlux` 共享相同的 `K`, 并可通过 `key()` 。
 - 由控件生 者中的 `onNext` 表示的任意 界 : `buffer(Publisher)`, `bufferWhen`

A.8. 回到同 的世界

注意：如果从在 “ 非阻塞” (`parallel()` 和 `single()`) 的 `Scheduler` 用, 除了 `Mono#toFuture` 之外的所有 些方法都会 出一个 `UnsupportedOperatorException` 常。

- 我有一个 `Flux<T>`, 我希望 :
 - 阻塞到我得到一个一个元素 止 : `Flux#blockFirst`

- ...超 阻塞 : `Flux#blockFirst(Duration)`
- 阻塞到我得到最后一个元素 止 (如果 空, 返回null) : `Flux#blockLast`
 - 超 阻塞 : `Flux#blockLast(Duration)`
- 同 切 到 `Iterable<T>` : `Flux#toIterable`
- 同 切 到Java 8 `Stream<T>` : `Flux#toStream`
- 我有一个 `Mono<T>`, 我希望 :
 - 阻塞到我 取到 止 : `Mono#block`
 - ...超 阻塞 : `Mono#block(Duration)`
 - `CompletableFuture<T>` : `Mono#toFuture`

A.9. 广播 Flux 到多个 Subscribers

- 我想将多个 `Subscriber` 接到一个 `Flux` 上 :
 - 并决定何 用 `connect()` 触 源 : `publish()` (返回一个 `ConnectableFlux`)
 - 并立即触 源 (后面的 者看到随后的数据) : `share()`
 - 当注 了足 的 者后永久 接源 : `.publish().autoConnect(n)`
 - 当 者高于/低于 自 接和取消源 : `.publish().refCount(n)`
 - ...但是 个新的 者一个机会, 它 在取消之前有机会 来 : `.publish().refCountGrace(n, Duration)`
- 我想 存来自 `Publisher` 的数据并将其重新 出到以后的 者 :
 - 最多 n 个元素 : `cache(int)`
 - 存在 `Duration` (生命周期) 内的看到的最新元素 : `cache(Duration)`
 - ...但保留不超 n 个元素 : `cache(int, Duration)`
 - 但没有立即触 源 : `Flux#replay` (返回一个 `ConnectableFlux`)

Appendix B: 常 和最佳 践, “我如何...?”

本 涵 以下内容 :

- 如何包装一个同 阻塞 用 ?
- 我在我的 Flux 上使用了一个操作符, 但似乎不 用。什 会 ?
- 我的 Mono zipWith/zipWhen 从未被 用
- 如何使用 retryWhen 来模 retry(3)?
- 我如何使用 retryWhen 行指数退避 ?
- 使用 publishOn() 如何 保 程 性 ?
- 上下文日志 的好的方式是什 ? (MDC)

B.1. 如何包装一个同 阻塞 用 ?

通常情况下, 信息源是同 且阻塞的。了在 的Reactor 用中 理 些源, 使用以下方式 :

```
Mono blockingWrapper = Mono.fromCallable(() -> { ①
    return /* make a remote synchronous call */ ②
});
blockingWrapper = blockingWrapper.subscribeOn(Schedulers.boundedElastic()); ③
```

① 通 使用 fromCallable 建新的 Mono。

② 返回 阻塞 源。

③ 保 个 都 生在来自 Schedulers.boundedElastic() 建的一个 用的 程worker上。

使用 Mono, 因 源只返回一个 。 使用 Schedulers.boundedElastic, 因 它会 建一个 的 程来等待阻塞 源而不影 其它的非阻塞 理, 同 保 可 建 程数量的限制, 并且在峰 期 阻塞的任 可以排 和延 。

注意, subscribeOn 并不会 为 Mono。它指定了当 用 生 要使用什 的 Scheduler。

B.2. 我在我的 Flux 上使用了一个操作符, 但似乎不 用。 什 会 ?

保 的 .subscribe() 的 量已 受到 用 量的操作符的影 。

Reactor操作符就是修 符。它 返回不同的 例来包装源序列并添加了行 。 就是 什 使用操作符的 首 方式是 接 用。

比 以下 个例子 :

Example 25. 没有 接(不正)

```
Flux<String> flux = Flux.just("something", "chain");
flux.map(secret -> secret.replaceAll(".", "*")); ①
flux.subscribe(next -> System.out.println("Received: " + next));
```

① 就出在 里。 果没有附加到 `flux` 量。

Example 26. 没有 接(正)

```
Flux<String> flux = Flux.just("something", "chain");
flux = flux.map(secret -> secret.replaceAll(".", "*"));
flux.subscribe(next -> System.out.println("Received: " + next));
```

下面的例子就更好了 (因 比) :

Example 27. 使用 接(最佳)

```
Flux<String> secrets = Flux
    .just("something", "chain")
    .map(secret -> secret.replaceAll(".", "*"))
    .subscribe(next -> System.out.println("Received: " + next));
```

第一个版本 出如下：

```
Received: something
Received: chain
```

其它 个版本 出的 期 ， 如下所示：

```
Received: ****
Received: ***
```

B.3. 我的 Mono zipWith/zipWhen 从未被用

看下面的例子：

```
myMethod.process("a") // this method returns Mono<Void>
    .zipWith(myMethod.process("b"), combinator) //this is never called
    .subscribe();
```

如果源 Mono 空 或 Mono<Void> (无 出于何 目的, Mono<Void> 都是空的) , 某些 合永 不会被用。

于 如 zip 静 方法或 zipWith、zipWhen 操作符之 的任何 器来 都是典型的情况, 它 (根据定) 需要从 个源中 取一个元素来生成其 出。

因此, 在 zip 源上使用数据抑制操作符是有 的。数据抑制操作符的例子包括 then(), thenEmpty(Publisher<Void>), ignoreElements() 和 ignoreElement(), 以及 when(Publisher...)。

同 , 使用 Function<T,?> 来 整它 行 的操作符, 例如 flatMap, 至少需要 出一个元素, 以便于 Function 有机会 用。将 些 用于空 (或 <Void>) 序列, 永 不会 生一个元素。

可以使用 .defaultIfEmpty(T) 和 .switchIfEmpty(Publisher<T>) 将空的 T 序列分 替 或回退的 Publisher<T>, 有助于避免某些情况。注意, 并不 用于 Flux<Void>/Mono<Void> 源, 因 只能切 到 一个 Publisher<Void>, 它 然保 是空的。下面的例子使用了 defaultIfEmpty :

Example 28. 在 zipWhen 之前使用 defaultIfEmpty

```
myMethod.emptySequenceForKey("a") // this method returns empty Mono<String>
    .defaultIfEmpty("") // this converts empty sequence to just the empty
    String
    .zipWhen(aString -> myMethod.process("b")) //this is called with the empty
    String
    .subscribe();
```

B.4. 如何使用 retryWhen 来模 retry(3)?

retryWhen 操作符可能相当 。希望下面的代 片段可以通 模 一个 的 retry(3) 来 助 理解它是如何工作的 :

```

AtomicInteger errorCount = new AtomicInteger();
Flux<String> flux =
    Flux.<String>error(new IllegalArgumentException())
        .doOnError(e -> errorCount.incrementAndGet())
        .retryWhen(Retry.from(companion -> ①
            companion.map(rs -> { ②
                if (rs.totalRetries() < 3) return rs.totalRetries();
            }
        )));

```

① 我通常改用 Function lambda 来自定 Retry，而不是提供一个具体的。

② 伴随的象限会输出 RetrySignal，它记录了迄今为止的重试次数和最后一次失败的次数。

③ 了允许三次重试，我考虑索引 < 3 并返回一个来指出（这里我直接地返回索引）。

④ 了在终止序列中，我在三次重试之后输出原始常量。

B.5. 我如何使用 retryWhen 行指数退避？

指数退避会生成重新尝试，每次尝试之间的延时会越来越长，这样就不会使源系统崩溃，也不会有全面崩溃的理由。原因是，如果源系统已经处于一个不确定的状态，不太可能立即从中恢复过来。因此，盲目的立即重试可能会生成一个错误，增加不确定性因素。

自从 3.3.4.RELEASE 版本之后，Reactor 自带了一个用于重试的构建器，可与 Flux#retryWhen 一起使用：Retry.backoff。

下面的例子演示了一个重试构建器的使用，在重试前后的消息。它可以重试并增加每次尝试之间的延时（代码：延时 = 重试次数 * 100 毫秒）：

```

AtomicInteger errorCount = new AtomicInteger();
Flux<String> flux =
Flux.error(new IllegalStateException("boom"))
    .doOnError(e -> { ①
        errorCount.incrementAndGet();
        System.out.println(e + " at " + LocalTime.now());
    })
    .retryWhen(Retry
        .backoff(3, Duration.ofMillis(100)).jitter(0d) ②
        .doAfterRetry(rs -> System.out.println("retried at " +
LocalTime.now())) ③
        .onRetryExhaustedThrow((spec, rs) -> rs.failure()) ④
    );

```

- ① 我 将 源 出的 ， 并 其 行 。
- ② 我 配置了一个指数退避重 ， 最多 3次， 没有 。
- ③ 我 了重 生的 。
- ④ 情况下， 会 出一个 `Exceptions.retryExhausted` 常， 最后一个 `failure()` 作 失原因。 里我 将其自定 直接以 `onError` 的形式 出原因。

后， 它将失 并在打印出以下内容后 止：

```

java.lang.IllegalArgumentException at 18:02:29.338
retried at 18:02:29.459 ①
java.lang.IllegalArgumentException at 18:02:29.460
retried at 18:02:29.663 ②
java.lang.IllegalArgumentException at 18:02:29.663
retried at 18:02:29.964 ③
java.lang.IllegalArgumentException at 18:02:29.964

```

- ① 100 秒后第一次重
- ② 200 秒后第二次重
- ③ 300 秒后第三次重

B.6. 使用 `publishOn()` 如何 保 程 性？

如 度器 中所描述的， `publishOn()` 可以用来切 行上下文。`publishOn()` 操作符会影 到 程上下文，在它下面的 中的其它操作符都会在 个上下文中 行， 直到出 一个新的 `publishOn`。因此， `publishOn` 的位置很重要。

看下面的例子：

```
EmitterProcessor<Integer> processor = EmitterProcessor.create();
processor.publishOn(scheduler1)
    .map(i -> transform(i))
    .publishOn(scheduler2)
    .doOnNext(i -> processNext(i))
    .subscribe();
```

`map()` 中的 `transform` 函数在 `scheduler1` 的工作 程上 行，`doOnNext()` 中的 `processNext` 方法 行在 `scheduler2` 的工作 程上。 个 都有自己的工作 程，因此，推送到相 者的所有元素都是在在同一个 `Thread` 上 布。

可以使用 程 度器来 保在 上不同 段或不同 者的 程 性。

B.7. 上下文日志 的好的方式是什 ？ (MDC)

大多数日志 架都允 行上下文日志 ，允 用 存 反映在日志模式中的 量，通常是通 叫做MDC (“映射 断上下文”) 的 `Map` 的方式来。 是Java中 `ThreadLocal` 最常 的用法之一，因此，此模式假 所 的代 与 `Thread`是一 一的 系。

在Java 8之前， 可能是一个安全的假 ，但随着Java 言中函数式 程元素的出 ，情况 生了一些 化...

我 以一个命令式API的示例 例， API使用了模版方法模式，然后切 一个更具有函数式 格的。使 用模版方法模式， 承 了作用。 在，在更 函数式的方法中， 高 函数来定 算法的“ ”。 在的 西更多的是声明性的而不是命令式的， 使得 可以自由地决定 个 在 里 行。例如，知 道了 些 的底 算法可以并行化， 就可以使用 `ExecutorService` 并行地 行某些 。

函数式API的一个具体例子是Java 8中引入的 `Stream` 及其 `parallel()` 格。在并行 `Stream` 中使用MDC 行日志 并不是免 的午餐：需要 保MDC在 个 中都能被捕 并重新 用。

函数式 格支持 的 化，因 个 都是 程不可知的和引用透明的，但是它可以打破MDC 一 `Thread` 的假 。 保所有 段都可以 任何 型的上下文信息的最 用的方法是通 合 上下文。在Reactor 程中，我 遇到了同一 的 ，我 希望避免 明了的方法。 就是引入 `Context` 的原因：只要使用 `Flux` 和 `Mono` 作 返回 ，它就会通 行 播， 段(操作符)探 到其下游 段的 `Context`。因此，Reactor中没有使用 `ThreadLocal`，而是提供了 个 似于map的 象，它 定到一个 `Subscription` 而不是 `Thread`。

既然我 已 定了MDC “只是在工作”，并不是在声明性API做出的最佳假 ，那 我 如何在 式流中的事件 (`onNext`, `onError` 和 `onComplete`) 行与事件相 的上下文日志 句？

当想要以直接和明 的方式 与 些信号的 系 ，FAQ的 个条目提供了一个可能的中 解决方案。 保事先 式序列添加上下文 章 ，尤其是如何在操作符 的底部 行写操作，以使其 上方的操作符能看到。

要从 `Context` 取上下文信息到MDC中，最 的方式是使用一些 板代 将日志 句封装在一个 `doOnEach` 操作符中。 板代 取决于 的日志 架/抽象和要放入MDC中的信息，因此它必 位于 的代 中。

下面是 一个 个MDC 量的 助函数的例子，并着重于使用Java9 的 **Optional API** **onNext** 事件：

```
public static <T> Consumer<Signal<T>> logOnNext(Consumer<T> logStatement) {
    return signal -> {
        if (!signal.isOnNext()) return;
        Optional<String> toPutInMdc =
            signal.getContext().getOrDefault("CONTEXT_KEY"); ②

        toPutInMdc.ifPresentOrElse(tpim -> {
            try (MDCCloseable cMdc = MDCCloseable("MDC_KEY", tpim)) { ③
                logStatement.accept(signal.get()); ④
            }
        },
        () -> logStatement.accept(signal.get())); ⑤
    };
}
```

① `doOnEach` 信号包括 `onComplete` 和 `onError`。在 个例子中，我 只 `onNext` 感 趣

② 我 将从Reactor的 `Context` (看 `Context API` 章) 提取一个有趣的 。

③ 在 个例子中，我 使用SLF4J 2中的 `MDCCloseable`，允 使用try-with-resource 法在 行日志 句后自 清理MDC。

④ 用方以 `Consumer<T>` (`onNext` 的消 者) 的形式提供正 的日志 句

⑤ 如果在 `Context` 中没有 置 期的 ， 使用 一 方式， MDC中不放置任何 西

使用 个 板代 可以 保我 是MDC的良好公民：我 在 行日志 句之前就 置了一个 ， 并在 行完之后立即将其 除。在后 的日志 句中，不会有 染MDC的 。

当然， 只是一个建 。 可能 从 `Context` 取多个 或在出 `onError` 情况 一些事情更感 趣。 可能想要 些情况 建其它 助方法，或者 建一个使用 外的lambda来覆 更多 域的方法。

在任何情况下：前面的 助方法的使用都可能 似于下面的 式web控制器：

```

@GetMapping("/byPrice")
public Flux<Restaurant> byPrice(@RequestParam Double maxPrice,
@RequestHeader(required = false, name = "X-UserId") String userId) {
    String apiId = userId == null ? "" : userId; ①

    return restaurantService.byPrice(maxPrice)
        .doOnEach(logOnNext(r -> LOG.debug("found restaurant {} for ${}", ②
            r.getName(), r.getPricePerPerson())))
        .subscriberContext(Context.of("CONTEXT_KEY", apiId)); ③
}

```

① 我 需要从 求 中 取上下文信息，将其放到 Context 中

② 在 里，我 使用 doOnEach 将我 的 助方法 用到 Flux 中。 住：操作符可以看到在它 下面定 的 Context 。

③ 我 使用 的 CONTEXT_KEY 将 求 中的 写入到 Context。

在 配置中，restaurantService 可以在一个共享 程上 出数据，但日志 能 个 求引用正 的 X-UserId。

了完整起，我 可以看到 日志 的 助方法看起来像：

```

public static Consumer<Signal<?>> logOnError(Consumer<Throwable>
errorLogStatement) {
    return signal -> {
        if (!signal.isOnError()) return;
        Optional<String> toPutInMdc =
signal.getContext().getOrDefault("CONTEXT_KEY");

        toPutInMdc.ifPresentOrElse(tpim -> {
            try (MDC.MDCCloseable cMdc = MDC.putCloseable("MDC_KEY", tpim)) {
                errorLogStatement.accept(signal.getThrowable());
            }
        },
        () -> errorLogStatement.accept(signal.getThrowable()));
    };
}

```

除了我 Signal 是否是一个 onError 以及我 向日志 句lambda提供 个 (Throwable) 之外，没有什 化。

在控制器中 用 个 助方法和我 之前做的非常相似：

```
@GetMapping("/byPrice")
public Flux<Restaurant> byPrice(@RequestParam Double maxPrice,
@RequestParam(required = false, name = "X-UserId") String userId) {
    String apiId = userId == null ? "" : userId;

    return restaurantService.byPrice(maxPrice)
        .doOnEach(logOnNext(v -> LOG.info("found restaurant {}", v)))
        .doOnEach(logOnError(e -> LOG.error("error when searching
restaurants", e)) ①
        .subscriberContext(Context.of("CONTEXT_KEY", apiId));
}
```

① 如果 `restaurantService` 出错，它将在哪里用MDC上下文下来

Appendix C: Reactor-Extra

于 reactor-core 具有高 需求的用 ， reactor-extra 模 包含了 外的操作符和工具集。

由于 是一个 独的 件， 需要 式地将其添加到 建中。下面的例子 示了在Gradle中如何操作：

```
dependencies {
    compile 'io.projectreactor:reactor-core'
    compile 'io.projectreactor.addons:reactor-extra' ①
}
```

①除了core模 外， 需要添加extra模 。 于使用BOM 不需要指定版本的原因，其它 信息 以及在Maven中的用法， 看入 [Reactor](#)。

C.1. TupleUtils 和函数式接口

reactor.function 包含了 充Java 8 Function, Predicate 和 Consumer 的函数式接口， 用于三到八个 。

TupleUtils 提供了静 方法。 些方法充当 些函数式接口的lambda与 的 Tuple 上的 似接口之 的 梁。

可以 松地 理任何 Tuple 的独立部分，如下例所示：

```
.map(tuple -> {
    String firstName = tuple.getT1();
    String lastName = tuple.getT2();
    String address = tuple.getT3();

    return new Customer(firstName, lastName, address);
});
```

可以将前面的例子改写如下：

```
.map(TupleUtils.function(Customer::new)); ①
```

① (因 Customer 造器符合 Consumer3 函数式接口 名)

C.2. MathFlux 数学操作符

reactor.math 包含了 Flux 特定版本的 MathFlux，提供了包括 max, min, sumInt, averageDouble 等数学操作符。

C.3. 重 和重 工具

`reactor.retry` 包含了 助 写 `Flux#repeatWhen` 和 `Flux#retryWhen` 函数的工具集。入口点分 是 `Repeat` 和 `Retry` 接口中的工厂方法。

可以将 一个接口都用作 可 的 建器，并且它 可以正 的 在 的操作符中要使用的 `Function` 名。

从3.2.0 始， 些工具集提供的最高 的重 策略之一也是 `reactor-core` 主要模 的一部分。`Flux#retryBackoff` 操作符可作 指数退避来使用。

从3.3.4 始， `Retry` 建器直接在core中提供，并且有了更多的可能的定制，基于一个封装了 之外的 外状 的 `RetrySignal`。

C.4. 度器

Reactor-extra 有几个 的 度器：

- `ForkJoinPoolScheduler` (在 `reactor.scheduler.forkjoin` 包中)：使用Java的 `ForkJoinPool` 行任。
- `SwingScheduler` (在 `reactor.swing` 包中)：在Swing UI事件循 程 `EDT` 中 行任 。
- `SwtScheduler` (在 `reactor.swing` 包中)：在SWT UI事件循 程中 行任 。