# Операторы Flux / Mono в Reactor

**1. Введение в Project Reactor**

Реактивное программирование поддерживается Spring Framework, начиная с версии 5. Эта поддержка построена на основе Project Reactor.

Project Reactor (или просто Reactor) - это библиотека Reactive для создания неблокирующих приложений на JVM, основанная на спецификации Reactive Streams. Reactor - это основа реактивного стека в экосистеме Spring, и он разрабатывается в тесном сотрудничестве со Spring. WebFlux, веб-фреймворк с реактивным стеком Spring, использует Reactor в качестве базовой зависимости.

**1.1 Модули Reactor**

Проект Reactor состоит из набора модулей, перечисленных в документации Reactor. Модули встраиваемы и совместимы. Основным артефактом является Reactor Core, который содержит реактивные типы Flux и Mono, которые реализуют интерфейс Publisher Reactive Stream (подробности см. в [**первом сообщении этой серии**](https://habr.com/ru/post/565004/)) и набор операторов, которые могут применяться к ним.

Некоторые другие модули:

* Reactor Test - предоставляет некоторые утилиты для тестирования реактивных потоков
* Reactor Extra - предоставляет некоторые дополнительные операторы Flux
* Reactor Netty - неблокирующие клиенты и серверы TCP, HTTP и UDP с поддержкой обратного давления - на основе инфраструктуры Netty
* Reactor Adapter - адаптер для других реактивных библиотек, таких как RxJava2 и Akka Streams
* Reactor Kafka - реактивный API для Kafka, который позволяет публиковать и получать сообщения в Kafka.

**1.2 Настройка проекта**

Прежде чем мы продолжим, если вы хотите настроить проект и запустить некоторые из приведенных ниже примеров кода, сгенерируйте новое приложение Spring Boot с помощью **Spring Initializr**. В качестве зависимости выберите Spring Reactive Web. После импорта проекта в вашу среду IDE взгляните на файл POM, и вы увидите, что добавлена ​​зависимость spring-boot-starter-webflux, которая также внесет зависимость ядра-реактора. Также в качестве зависимости добавлен тест-реактор. Теперь вы готовы к запуску следующих примеров кода.

...

<**dependencies**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-webflux</**artifactId**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-test</**artifactId**>

<**scope**>test</**scope**>

<**exclusions**>

<**exclusion**>

<**groupId**>org.junit.vintage</**groupId**>

<**artifactId**>junit-vintage-engine</**artifactId**>

</**exclusion**>

</**exclusions**>

</**dependency**>

<**dependency**>

<**groupId**>io.projectreactor</**groupId**>

<**artifactId**>reactor-test</**artifactId**>

<**scope**>test</**scope**>

</**dependency**>

</**dependencies**>

...

**2. Возможности Reactor Core**

Reactor Core определяет реактивные типы Flux и Mono.

**2.1 FLUX и MONO**

Flux - это Publisher, который может испускать от 0 до N элементов, а Mono может испускать от 0 до 1 элемента. Оба они завершаются либо сигналом завершения, либо ошибкой, и они вызывают методы onNext, onComplete и onError нижестоящего подписчика. Помимо реализации функций, описанных в спецификации Reactive Streams, Flux и Mono предоставляют набор операторов для поддержки преобразований, фильтрации и обработки ошибок.

В качестве первого упражнения перейдите к классу тестирования, созданному в вашем новом проекте, добавьте следующий пример и запустите его:

@Test

**void** **simpleFluxExample**() {

Flux<String> fluxColors = Flux.just("red", "green", "blue");

fluxColors.subscribe(System.out::println);

}

Метод just создает поток, который испускает предоставленные элементы, а затем завершается. Ничего не передается, пока кто-нибудь на это не подпишется. Чтобы подписаться на него, мы вызываем метод subscribe и в этом случае просто распечатываем отправленные элементы. Создание Mono также может быть выполнено с помощью метода just, с той лишь разницей, что разрешен только один параметр.

**2.2 Объединение операторов**

Взгляните на [Flux API,](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html) и вы увидите, что почти все методы возвращают Flux или Mono, что означает, что операторы могут быть связаны. Каждый оператор добавляет поведение к Publisher (Flux или Mono) и переносит Publisher предыдущего шага в новый экземпляр. Данные поступают от первого издателя и перемещаются по цепочке, трансформируясь каждым оператором. В конце концов, подписчик завершает процесс. Обратите внимание, что ничего не происходит, пока подписчик не подпишется на издателя.

Существует оператор log(), который обеспечивает регистрацию всех сигналов Reactive Streams, происходящих за кулисами. Просто измените последнюю строку приведенного выше примера на

*fluxColors.log().subscribe(System.out::println);*

и перезапустите тест. Теперь вы увидите, что к выходным данным добавляется следующее:

2020-09-12 16:16:39.779 INFO 6252 --- [ main] reactor.Flux.Array.1 : | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1 : | request(unbounded)

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1 : | onNext(red)

red

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1 : | onNext(green)

green

2020-09-12 16:16:39.781 INFO 6252 --- [ main] reactor.Flux.Array.1 : | onNext(blue)

blue

2020-09-12 16:16:39.782 INFO 6252 --- [ main] reactor.Flux.Array.1 : | onComplete()

Теперь, чтобы увидеть, что произойдет, если вы исключите вызов subscribe(), снова измените последнюю строку кода на следующую и повторно запустите тест:

fluxColors.log();

Как вы увидите из выходных данных журнала, сейчас никакие элементы не отправляются - поскольку нет подписчика, инициирующего процесс.

**2.3 Поиск подходящего оператора**

Reactor предоставляет длинный список операторов, и в качестве помощи в поиске подходящего оператора для конкретного варианта использования есть специальное [приложение](https://projectreactor.io/docs/core/release/reference/index.html#which-operator) в справочной документации Reactor. Он разделен на различные категории, как показано в таблице ниже.

|  |  |
| --- | --- |
| **КАТЕГОРИЯ ОПЕРАТОРА** | **ПРИМЕРЫ** |
| Создание новой последовательности | just, fromArray, fromIterable, fromStream |
| Преобразование существующей последовательности | map, flatMap, startWith, concatWith |
| Заглядывать в последовательность | doOnNext, doOnComplete, doOnError, doOnCancel |
| Фильтрация последовательности | filter, ignoreElements, distinct, elementAt, takeLast |
| Обработка ошибок | onErrorReturn, onErrorResume, retry |
| Работаем со временем | elapsed, interval, timestamp, timeout |
| Расщепление потока | buffer, groupBy, window |
| Возвращаясь к синхронному миру | block, blockFirst, blockLast, toIterable, toStream |
| Многоадресная рассылка потока нескольким подписчикам | publish, cache, replay |

Теперь не стесняйтесь создать несколько небольших примеров, в которых используются некоторые из этих операторов, и посмотреть, что произойдет, когда вы их запустите. Например, с помощью оператора map (который преобразует элементы, создаваемые путем применения синхронной функции к каждому элементу):

@Test

**void** **mapExample**() {

Flux<String> fluxColors = Flux.just("red", "green", "blue");

fluxColors.map(color -> color.charAt(0)).subscribe(System.out::println);

}

Или оператор zip, который объединяет несколько источников вместе (ожидая, пока все источники испускают один элемент, и объединяет их в кортеж):

@Test

**void** **zipExample**() {

Flux<String> fluxFruits = Flux.just("apple", "pear", "plum");

Flux<String> fluxColors = Flux.just("red", "green", "blue");

Flux<Integer> fluxAmounts = Flux.just(10, 20, 30);

Flux.zip(fluxFruits, fluxColors, fluxAmounts).subscribe(System.out::println);

}

**3. Обработка ошибок**

Как описано в предыдущем сообщении в блоге, в Reactive Streams ошибки - это терминальные события. При возникновении ошибки вся последовательность останавливается, и ошибка передается методу onError подписчика, который всегда должен быть определен. Если не определено, onError вызовет исключение UnsupportedOperationException.

Как вы видите, запустив следующий пример, третье значение никогда не генерируется, поскольку второе значение приводит к ошибке:

@Test

**public** **void** **onErrorExample**() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i));

fluxCalc.subscribe(value -> System.out.println("Next: " + value),

error -> System.err.println("Error: " + error));

}

Результат будет выглядеть так:

Next: 10 / -1 = -10

Error: java.lang.ArithmeticException: / by zero

Также можно обрабатывать ошибки в середине реактивной цепочки, используя операторы обработки ошибок:

Метод onErrorReturn будет выдавать резервное значение, когда наблюдается ошибка указанного типа. Это можно сравнить с перехватом исключения и возвратом статического запасного значения в императивном программировании. См. Пример ниже:

@Test

**public** **void** **onErrorReturnExample**() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i))

.onErrorReturn(ArithmeticException.**class**, "**Division** **by** 0 **not** **allowed**");

fluxCalc.subscribe(value -> System.out.println("Next: " + value),

error -> System.err.println("Error: " + error));

}

и результат:

Next: 10 / -1 = -10

Next: Division by 0 not allowed

Как видите, использование оператора обработки ошибок таким образом все еще не позволяет продолжить исходную реактивную последовательность (третье значение здесь также не генерируется), а скорее заменяет ее. Если недостаточно просто вернуть какое-то значение по умолчанию, вы можете использовать этот onErrorResume метод, чтобы подписаться на резервного издателя при возникновении ошибки. Это можно сравнить с перехватом исключения и вызовом резервного метода в императивном программировании. Если, например, вызов внешней службы завершается неудачно, реализация onErrorResume может быть связана с извлечением данных из локального кеша.

**4. Тестирование**

Модуль Reactor Test предоставляет служебные программы, которые могут помочь в тестировании поведения вашего Flux или Mono. В этом помогает API StepVerifier. Вы создаете StepVerifier и передаете его издателю для тестирования. StepVerifier подписывается на Publisher при вызове метода verify, а затем сравнивает выданные значения с вашими определенными ожиданиями.

См. следующий пример:

@Test

**public** **void** **stepVerifierTest**() {

Flux<String> fluxCalc = Flux.just(-1, 0, 1)

.map(i -> "10 / " + i + " = " + (10 / i));

StepVerifier.create(fluxCalc)

.expectNextCount(1)

.expectError(ArithmeticException.**class**)

.**verify**();

}

Для объекта создается StepVerifier, fluxCalc и определяются два ожидания: сначала ожидается, что будет выдана одна String, а затем должна быть выдана ошибка с типом ArithmeticException. С помощью вызова verify StepVerifier начинает подписываться на Flux, и инициируется поток.

StepVerifier также имеет другие функции, такие как включение утверждений после выполнения и поддержка виртуального времени, чтобы избежать длительного времени выполнения тестов, связанных с операторами, основанными на времени.

Модуль Reactor Test также предоставляет другой API, TestPublisher который представляет собой Publisher, которым вы можете напрямую управлять, инициируя события onNext, onComplete и onError для целей тестирования.

**5. Модель параллелизма**

Как вы, возможно, уже заметили из вывода журнала simpleFluxExample, до сих пор наш издатель выполнялся в основном потоке так же, как подписчик. Это связано с тем, что Reactor не применяет модель параллелизма. Вместо этого выполнение большинства операторов будет продолжено в том же потоке, оставляя выбор за разработчиком. Модель выполнения определяется тем Scheduler, что используется.

Есть два способа переключения контекста выполнения в реактивной цепочке: publishOn и subscribeOn. Отличается следующее:

publishOn(Scheduler scheduler) влияет на выполнение всех последующих операторов (если не указано иное)

subscribeOn(Scheduler scheduler) изменяет поток, из которого подписывается вся цепочка операторов, на основе самого раннего вызова subscribeOn в цепочке. Это не влияет на поведение последующих вызовов publishOn

Класс Schedulers содержит статические методы, чтобы обеспечить контекст выполнения, например:

parallel() - Фиксированный пул воркеров, настроенный для параллельной работы, создавая столько воркеров, сколько ядер ЦП.

single() - Одиночная многоразовая нить. Этот метод повторно использует один и тот же поток для всех вызывающих, пока Планировщик не будет удален. Если вместо этого вам нужен выделенный поток для каждого вызова, вы можете использовать Schedulers.newSingle () для каждого вызова.

boundedElastic() - Динамически создает ограниченное количество рабочих. Он имеет ограничение на количество поддерживающих потоков, которые он может создать, и может ставить задачи в очередь для перепланирования, когда поток становится доступным. Это хороший выбор для обертывания синхронных, блокирующих вызовов.

immediate() - немедленно запускается в исполняемом потоке, не переключая контекст выполнения

fromExecutorService(ExecutorService) - может использоваться для создания Планировщика из любого существующего ExecutorService

Выполните следующий пример и посмотрите на поведение:

@Test

**public** **void** **publishSubscribeExample**() {

Scheduler schedulerA = Schedulers.newParallel("Scheduler A");

Scheduler schedulerB = Schedulers.newParallel("Scheduler B");

Scheduler schedulerC = Schedulers.newParallel("Scheduler C");

Flux.just(1)

.map(i -> {

System.out.println("First map: " + Thread.currentThread().getName());

**return** i;

})

.subscribeOn(schedulerA)

.map(i -> {

System.out.println("Second map: " + Thread.currentThread().getName());

**return** i;

})

.publishOn(schedulerB)

.map(i -> {

System.out.println("Third map: " + Thread.currentThread().getName());

**return** i;

})

.subscribeOn(schedulerC)

.map(i -> {

System.out.println("Fourth map: " + Thread.currentThread().getName());

**return** i;

})

.publishOn(schedulerA)

.map(i -> {

System.out.println("Fifth map: " + Thread.currentThread().getName());

**return** i;

})

.blockLast();

}

Взглянув на вывод (показан ниже), вы можете увидеть, что первая и вторая операции map выполняются в потоке из планировщика A, поскольку первый subscribeOn в цепочке переключается на этот планировщик, и это влияет на всю цепочку. Перед третьей операцией map выполняется publishOn, переключающий контекст выполнения на Scheduler B, в результате чего третья и четвертая операции map выполняются в этом контексте (поскольку вторая subscribeOn не будет иметь никакого эффекта). И, наконец, есть новый метод publishOn, который переключает обратно на Планировщик A перед последней операцией map.

First map: Scheduler A-4

Second map: Scheduler A-4

Third map: Scheduler B-3

Fourth map: Scheduler B-3

Fifth map: Scheduler A-1

**6. BACKPRESSURE (противодавление)**

Как вы могли вспомнить из первой части этой серии блогов, противодавление - это способность потребителя сигнализировать производителю, с какой скоростью выброса он может справиться, чтобы он не перегружался.

В приведенном ниже примере показано, как подписчик может контролировать скорость передачи, вызывая request(n) метод в Subscription.

@Test

**public** **void** **backpressureExample**() {

Flux.range(1,5)

.subscribe(**new** Subscriber<Integer>() {

**private** Subscription s;

**int** counter;

@Override

**public** **void** **onSubscribe**(Subscription s) {

System.out.println("onSubscribe");

**this**.s = s;

System.out.println("Requesting 2 emissions");

s.request(2);

}

@Override

**public** **void** **onNext**(Integer i) {

System.out.println("onNext " + i);

counter++;

**if** (counter % 2 == 0) {

System.out.println("Requesting 2 emissions");

s.request(2);

}

}

@Override

**public** **void** **onError**(Throwable t) {

System.err.println("onError");

}

@Override

**public** **void** **onComplete**() {

System.out.println("onComplete");

}

});

}

Запустите его, и вы увидите, что по запросу одновременно генерируются два значения:

onSubscribe

Requesting 2 emissions

onNext 1

onNext 2

Requesting 2 emissions

onNext 3

onNext 4

Requesting 2 emissions

onNext 5

onComplete

В Subscription также есть cancelметод, позволяющий запросить Издателя остановить эмиссию и очистить ресурсы.

**7. Холодные и горячие Publisher**

Доступны два типа Publisher - cold и hot (холодные и горячие). Пока что мы сосредоточились на холодных Publisher. Как мы заявляли ранее, ничего не происходит, пока мы не подпишемся - но на самом деле это верно только для холодных издателей.

Холодный Publisher генерирует новые данные для каждой подписки. Если подписки нет, данные никогда не генерируются. Напротив, hot издатель не зависит от подписчиков. Он может начать публикацию данных без подписчиков. Если подписчик подписывается после того, как издатель начал передавать значения, он получит только значения, выпущенные после его подписки.

Publisher в Reactor по умолчанию не работают. Один из способов создания горячего Publisher - это вызвать publish() метод в Flux. Это вернет ConnectableFlux<T>, у которого есть метод connect() для запуска передачи значений. Подписчики должны затем подписаться на этот ConnectableFlux вместо исходного Flux.

Давайте посмотрим на простой холодный и горячий Publisher, чтобы увидеть различное поведение. В приведенном ниже примере coldPublisherExample оператор interval используется для создания потока, который генерирует значения long, начинающиеся с 0.

@Test

**public** **void** **coldPublisherExample**() **throws** InterruptedException {

Flux<Long> intervalFlux = Flux.interval(Duration.ofSeconds(1));

Thread.sleep(2000);

intervalFlux.subscribe(i -> System.out.println(String.format("Subscriber A, value: %d", i)));

Thread.sleep(2000);

intervalFlux.subscribe(i -> System.out.println(String.format("Subscriber B, value: %d", i)));

Thread.sleep(3000);

}

При запуске будет получен следующий результат:

Subscriber A, value: 0

Subscriber A, value: 1

Subscriber A, value: 2

Subscriber B, value: 0

Subscriber A, value: 3

Subscriber B, value: 1

Subscriber A, value: 4

Subscriber B, value: 2

Теперь вы можете задаться вопросом, почему что-то происходит, когда основной поток спит, но это потому, что оператор интервала по умолчанию выполняется в планировщике Schedulers.parallel(). Как видите, оба подписчика получат значения, начинающиеся с 0.

Теперь давайте посмотрим, что происходит, когда мы используем ConnectableFlux:

@Test

**public** **void** **hotPublisherExample**() **throws** InterruptedException {

Flux<Long> intervalFlux = Flux.interval(Duration.ofSeconds(1));

ConnectableFlux<Long> intervalCF = intervalFlux.publish();

intervalCF.connect();

Thread.sleep(2000);

intervalCF.subscribe(i -> System.out.println(String.format("Subscriber A, value: %d", i)));

Thread.sleep(2000);

intervalCF.subscribe(i -> System.out.println(String.format("Subscriber B, value: %d", i)));

Thread.sleep(3000);

}

На этот раз мы получаем следующий результат:

Subscriber A, value: 2

Subscriber A, value: 3

Subscriber A, value: 4

Subscriber B, value: 4

Subscriber A, value: 5

Subscriber B, value: 5

Subscriber A, value: 6

Subscriber B, value: 6

Как мы видим, на этот раз ни один из подписчиков не получает исходные значения 0 и 1. Они получают значения, которые отправляются после подписки. Вместо того, чтобы вручную запускать публикацию, с помощью этого autoConnect(n)метода также можно настроить ConnectableFlux так, чтобы он запускался после n подписок.

**8. Прочие возможности**

**8.1 Завершение синхронного, блокирующего вызова**

Когда необходимо использовать источник информации, который является синхронным и блокирующим, в Reactor рекомендуется использовать следующий шаблон:

Mono blockingWrapper = Mono.fromCallable(() -> {

**return** /\* make a remote synchronous call \*/

});

blockingWrapper = blockingWrapper.subscribeOn(Schedulers.boundedElastic());

Метод fromCallable создает Mono, который производит его значение с помощью прилагаемого Callable. Используя Schedulers.boundedElastic(), мы гарантируем, что каждая подписка выполняется на выделенном однопоточном работнике, не влияя на другую неблокирующую обработку.

**8.2 Контекст**

Иногда возникает необходимость передать некоторые дополнительные, обычно технические данные, через реактивный конвейер. Сравните это с привязкой некоторого состояния к потоку с помощью ThreadLocal в императивном мире.

Reactor имеет функцию, которая в некоторой степени сравнима с ThreadLocal, но может применяться к Flux или Mono вместо Thread, называемая a Context. Это интерфейс, похожий на Map, где вы можете хранить пары ключ-значение и получать значение по его ключу. Контекст прозрачно распространяется по всему реактивному конвейеру и может быть легко доступен в любой момент, вызвав метод Mono.subscriberContext().

Контекст может быть заполнен во время подписки путем добавления вызова метода subscriberContext(Function) или subscriberContext(Context) метода в конце вашего реактивного конвейера, как показано в методе тестирования ниже..

**8.3 SINKS**

Rector также предлагает возможность создавать Flux или Mono, программно определяя события onNext, onError и onComplete. Для этого предоставляется так называемый API-интерфейс приемника, запускающий события. Существуют несколько различных вариантов раковин, чтобы узнать больше об этом, читайте далее в справочной документации: [**Программное создание последовательности**](https://projectreactor.io/docs/core/release/reference/#producing)

**8.4 Отладка**

Отладка реактивного кода может стать проблемой из-за его функционального декларативного стиля, в котором фактическое объявление (или «assembly ») и обработка сигнала («execution») не происходят одновременно. Обычная трассировка стека Java, генерируемая приложением Reactor, не будет включать никаких ссылок на ассемблерный код, что затрудняет определение фактической основной причины распространенной ошибки.

Чтобы получить более значимую трассировку стека, которая включает информацию о сборке (также называемую трассировкой), вы можете добавить вызов Hooks.onOperatorDebug() в свое приложение. Однако это нельзя использовать в производственной среде, потому что это связано с перемещением тяжелого стека и может отрицательно повлиять на производительность.

Для использования в производственной среде Project Reactor предоставляет отдельный Java-агент, который инструментирует ваш код и добавляет отладочную информацию, не требуя больших ресурсов для захвата трассировки стека при каждом вызове оператора. Чтобы использовать его, вам нужно добавить reactor-tools артефакт в свои зависимости и инициализировать его при запуске приложения Spring Boot:

**public** **static** **void** **main**(String[] args) {

ReactorDebugAgent.init();

SpringApplication.run(Application.**class**, **args**);

}

**8.5 Метрики**

Reactor предоставляет встроенную поддержку для включения и отображения показателей как для планировщиков (Schedulers), так и для издателей (Publishers). Дополнительные сведения см. в разделе «[Метрики](https://projectreactor.io/docs/core/release/reference/#metrics)» Справочного руководства.

**9. Подводя итог…**

В этом сообщении в блоге представлен обзор Project Reactor, в основном сосредоточенный на функциях Reactor Core. [**Следующий блог**](https://habr.com/ru/post/565056/) в этой серии будет о WebFlux - реактивном фреймворке Spring, который использует Reactor в качестве реактивной библиотеки!

1. Введение в Spring WebFlux

Исходный веб-фреймворк для Spring - Spring Web MVC - был построен для Servlet API и контейнеров Servlet.

WebFlux был представлен как часть Spring Framework 5.0. В отличие от Spring MVC, он не требует Servlet API. Он полностью асинхронный и неблокирующий, реализует спецификацию Reactive Streams через проект Reactor.

WebFlux требует Reactor в качестве основной зависимости, но он также может взаимодействовать с другими реактивными библиотеками через Reactive Streams.

Spring WebFlux поддерживает две разные модели программирования: на основе аннотаций и функциональную.

**1.1.1 Аннотированные контроллеры**

Если вы работали со Spring MVC, модель на основе аннотаций будет выглядеть довольно знакомой, поскольку в ней используются те же аннотации из веб-модуля Spring, что и в Spring MVC. Основное отличие состоит в том, что теперь методы возвращают реактивные типы Mono и Flux. Пример RestController с использованием модели на основе аннотаций:

@RestController

@RequestMapping("/students")

**public** **class** **StudentController** {

@Autowired

**private** StudentService studentService;

**public** **StudentController**() {

}

@GetMapping("/{id}")

**public** Mono<ResponseEntity<Student>> getStudent(@PathVariable **long** id) {

**return** studentService.findStudentById(id)

.map(ResponseEntity::ok)

.defaultIfEmpty(ResponseEntity.notFound().build());

}

@GetMapping

**public** Flux<Student> **listStudents**(@RequestParam(name = "name", required = **false**) String name) {

**return** studentService.findStudentsByName(name);

}

@PostMapping

**public** Mono<Student> **addNewStudent**(@RequestBody Student student) {

**return** studentService.addNewStudent(student);

}

@PutMapping("/{id}")

**public** Mono<ResponseEntity<Student>> updateStudent(@PathVariable **long** id, @RequestBody Student student) {

**return** studentService.updateStudent(id, student)

.map(ResponseEntity::ok)

.defaultIfEmpty(ResponseEntity.notFound().build());

}

@DeleteMapping("/{id}")

**public** Mono<ResponseEntity<Void>> deleteStudent(@PathVariable **long** id) {

**return** studentService.findStudentById(id)

.flatMap(s ->

studentService.deleteStudent(s)

.then(Mono.just(**new** ResponseEntity<Void>(HttpStatus.OK)))

)

.defaultIfEmpty(**new** ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

}

Некоторые пояснения к функциям, использованным в примере:

map функция используется для преобразования элемента, испускаемого Mono, применяя функцию синхронной к нему.

flatMap функция используется для преобразования элемент, испускаемый Mono асинхронно, возвращая значение, излучаемого другим Mono.

defaultIfEmpty функция обеспечивает значение по умолчанию, если Mono завершается без каких - либо данных.

**1.1.2 Функциональные конечные точки**

Модель функционального программирования основана на лямбда-выражении и оставляет за приложением полную обработку запроса.

Он основан на концепциях HandlerFunctions и RouterFunctions.

HandlerFunctions используются для генерации ответа на данный запрос:

@FunctionalInterface

**public** **interface** **HandlerFunction**<**T** **extends** **ServerResponse**> {

Mono<T> **handle**(ServerRequest request);

}

RouterFunction используется для маршрутизации запросов к HandlerFunctions:

@FunctionalInterface

**public** **interface** **RouterFunction**<**T** **extends** **ServerResponse**> {

Mono<HandlerFunction<T>> route(ServerRequest request);

...

}

Продолжая с тем же примером ученика, мы получим что-то вроде следующего, используя функциональный стиль.

A StudentRouter:

@Configuration

**public** **class** **StudentRouter** {

@Bean

**public** RouterFunction<ServerResponse> **route**(StudentHandler studentHandler){

**return** RouterFunctions

.route(

GET("/students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)), studentHandler::getStudent)

.andRoute(

GET("/students")

.and(accept(APPLICATION\_JSON)), studentHandler::listStudents)

.andRoute(

POST("/students")

.and(accept(APPLICATION\_JSON)),studentHandler::addNewStudent)

.andRoute(

PUT("students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)), studentHandler::updateStudent)

.andRoute(

DELETE("/students/{id:[0-9]+}")

.and(accept(APPLICATION\_JSON)), studentHandler::deleteStudent);

}

}

И StudentHandler:

@Component

**public** **class** **StudentHandler** {

**private** StudentService studentService;

**public** **StudentHandler**(StudentService studentService) {

**this**.studentService = studentService;

}

**public** Mono<ServerResponse> **getStudent**(ServerRequest serverRequest) {

Mono<Student> studentMono = studentService.findStudentById(

Long.parseLong(serverRequest.pathVariable("id")));

**return** studentMono.flatMap(student -> ServerResponse.ok()

.body(fromValue(student)))

.switchIfEmpty(ServerResponse.notFound().build());

}

**public** Mono<ServerResponse> **listStudents**(ServerRequest serverRequest) {

String name = serverRequest.queryParam("name").orElse(**null**);

**return** ServerResponse.ok()

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.findStudentsByName(name), Student.**class**);

}

**public** Mono<ServerResponse> **addNewStudent**(ServerRequest serverRequest) {

Mono<Student> studentMono = serverRequest.bodyToMono(Student.**class**);

**return** studentMono.flatMap(student ->

ServerResponse.status(HttpStatus.OK)

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.addNewStudent(student), Student.**class**));

}

**public** Mono<ServerResponse> **updateStudent**(ServerRequest serverRequest) {

**final** **long** studentId = Long.parseLong(serverRequest.pathVariable("id"));

Mono<Student> studentMono = serverRequest.bodyToMono(Student.**class**);

**return** studentMono.flatMap(student ->

ServerResponse.status(HttpStatus.OK)

.contentType(MediaType.APPLICATION\_JSON)

.body(studentService.updateStudent(studentId, student), Student.**class**));

}

**public** Mono<ServerResponse> **deleteStudent**(ServerRequest serverRequest) {

**final** **long** studentId = Long.parseLong(serverRequest.pathVariable("id"));

**return** studentService

.findStudentById(studentId)

.flatMap(s -> ServerResponse.noContent().build(studentService.deleteStudent(s)))

.switchIfEmpty(ServerResponse.notFound().build());

}

}

Некоторые пояснения к функциям, использованным в примере:

switchIfEmpty функция имеет ту же цель, defaultIfEmpty, но вместо того, чтобы обеспечить значение по умолчанию, она используется для обеспечения альтернативного Mono.

Сравнивая две модели, мы видим, что:

Для использования функционального варианта требуется еще немного кода для таких вещей, как получение входных параметров и синтаксический анализ до ожидаемого типа.

Не полагаясь на аннотации, но написание явного кода предлагает некоторую большую гибкость и может быть лучшим выбором, если нам, например, нужно реализовать более сложную маршрутизацию.

1.2 Поддержка сервера

WebFlux работает в средах выполнения, отличных от сервлетов, таких как Netty и Undertow (неблокирующий режим), а также в средах выполнения сервлетов 3.1+, таких как Tomcat и Jetty.

По умолчанию стартер Spring Boot WebFlux использует Netty, но его легко переключить, изменив зависимости Maven или Gradle.

Например, чтобы переключиться на Tomcat, просто исключите spring-boot-starter-netty из зависимости spring-boot-starter-webflux и добавьте spring-boot-starter-tomcat:

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-webflux</**artifactId**>

<**exclusions**>

<**exclusion**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-netty</**artifactId**>

</**exclusion**>

</**exclusions**>

</**dependency**>

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-tomcat</**artifactId**>

</**dependency**>

1.3 Конфигурация

Spring Boot обеспечивает автоматическую настройку Spring WebFlux, которая хорошо работает в общих случаях. Если вам нужен полный контроль над конфигурацией WebFlux, можно использовать аннотацию @EnableWebFlux (эта аннотация также потребуется в простом приложении Spring для импорта конфигурации Spring WebFlux).

Если вы хотите сохранить конфигурацию Spring Boot WebFlux и просто добавить дополнительную конфигурацию WebFlux, вы можете добавить свой собственный класс @Configuration типа WebFluxConfigurer (но без @EnableWebFlux).

2. Защита ваших конечных точек

Чтобы получить поддержку Spring Security WebFlux, сначала добавьте в свой проект зависимость spring-boot-starter-security. Теперь вы можете включить его, добавив @EnableWebFluxSecurity аннотацию в свой класс Configuration (доступно с Spring Security 5.0).

В следующем упрощенном примере будет добавлена ​​поддержка двух пользователей, один с ролью USER, а другой с ролью ADMIN, принудительно применить базовую аутентификацию HTTP и потребовать роль ADMIN для любого доступа к пути /student/admin:

@EnableWebFluxSecurity

**public** **class** **SecurityConfig** {

@Bean

**public** MapReactiveUserDetailsService **userDetailsService**() {

UserDetails user = User

.withUsername("user")

.password(passwordEncoder().encode("userpwd"))

.roles("USER")

.build();

UserDetails admin = User

.withUsername("admin")

.password(passwordEncoder().encode("adminpwd"))

.roles("ADMIN")

.build();

**return** **new** MapReactiveUserDetailsService(user, admin);

}

@Bean

**public** SecurityWebFilterChain **securityWebFilterChain**(ServerHttpSecurity http) {

**return** http.authorizeExchange()

.pathMatchers("/students/admin")

.hasAuthority("ROLE\_ADMIN")

.anyExchange()

.authenticated()

.and().httpBasic()

.and().build();

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** **new** BCryptPasswordEncoder();

}

}

Также можно защитить метод, а не путь, сначала добавив аннотацию @EnableReactiveMethodSecurity к вашей конфигурации:

@EnableWebFluxSecurity

@EnableReactiveMethodSecurity

**public** **class** **SecurityConfig** {

...

}

А затем добавляем @PreAuthorize аннотацию к защищаемым методам. Например, мы можем захотеть, чтобы наши методы POST, PUT и DELETE были доступны только для роли ADMIN. Затем к этим методам можно применить аннотацию PreAuthorize, например:

@DeleteMapping("/{id}")

@PreAuthorize("hasRole('ADMIN')")

**public** Mono<ResponseEntity<Void>> deleteStudent(@PathVariable **long** id) {

...

}

Spring Security предлагает дополнительную поддержку, связанную с приложениями WebFlux, например защиту CSRF, интеграцию OAuth2 и реактивную аутентификацию X.509. Для получения дополнительной информации прочтите следующий раздел в документации Spring Security: [Реактивные приложения](https://docs.spring.io/spring-security/site/docs/current/reference/html5/#reactive-applications)

3. Веб-клиент

Spring WebFlux также включает реактивный, полностью неблокирующий веб-клиент. У него есть функциональный, свободный API, основанный на Reactor.

Давайте рассмотрим (еще раз) упрощенный пример того, как WebClient можно использовать для запроса нашего StudentController:

**public** **class** **StudentWebClient** {

WebClient client = WebClient.create("http://localhost:8080");

**public** Mono<Student> **get**(**long** id) {

**return** client

.get()

.uri("/students/" + id)

.headers(headers -> headers.setBasicAuth("user", "userpwd"))

.retrieve()

.bodyToMono(Student.**class**);

}

**public** Flux<Student> **getAll**() {

**return** client.get()

.uri("/students")

.headers(headers -> headers.setBasicAuth("user", "userpwd"))

.retrieve()

.bodyToFlux(Student.**class**);

}

**public** Flux<Student> **findByName**(String name) {

**return** client.get()

.uri(uriBuilder -> uriBuilder.path("/students")

.queryParam("name", name)

.build())

.headers(headers -> headers.setBasicAuth("user", "userpwd"))

.retrieve()

.bodyToFlux(Student.**class**);

}

**public** Mono<Student> **create**(Student s) {

**return** client.post()

.uri("/students")

.headers(headers -> headers.setBasicAuth("admin", "adminpwd"))

.body(Mono.just(s), Student.**class**)

.**retrieve**()

.**bodyToMono**(**Student**.**class**);

}

**public** Mono<Student> **update**(Student student) {

**return** client

.put()

.uri("/students/" + student.getId())

.headers(headers -> headers.setBasicAuth("admin", "adminpwd"))

.body(Mono.just(student), Student.**class**)

.**retrieve**()

.**bodyToMono**(**Student**.**class**);

}

**public** Mono<Void> **delete**(**long** id) {

**return** client

.delete()

.uri("/students/" + id)

.headers(headers -> headers.setBasicAuth("admin", "adminpwd"))

.retrieve()

.bodyToMono(Void.**class**);

}

}

4. Тестирование

Для тестирования вашего реактивного веб-приложения WebFlux предлагает WebTestClient, который поставляется с API, аналогичным WebClient.

Давайте посмотрим, как мы можем протестировать наш StudentController с помощью WebTestClient:

@ExtendWith(SpringExtension.**class**)

@**SpringBootTest**(**webEnvironment** = SpringBootTest.WebEnvironment.RANDOM\_PORT)

**class** **StudentControllerTest** {

@Autowired

WebTestClient webClient;

@Test

@WithMockUser(roles = "USER")

**void** **test\_getStudents**() {

webClient.get().uri("/students")

.header(HttpHeaders.ACCEPT, "application/json")

.exchange()

.expectStatus().isOk()

.expectHeader().contentType(MediaType.APPLICATION\_JSON)

.expectBodyList(Student.**class**);

}

@Test

@WithMockUser(roles = "ADMIN")

**void** **testAddNewStudent**() {

Student newStudent = **new** Student();

newStudent.setName("some name");

newStudent.setAddress("an address");

webClient.post().uri("/students")

.contentType(MediaType.APPLICATION\_JSON)

.accept(MediaType.APPLICATION\_JSON)

.body(Mono.just(newStudent), Student.**class**)

.**exchange**()

.**expectStatus**().**isOk**()

.**expectHeader**().**contentType**(**MediaType**.**APPLICATION\_JSON**)

.**expectBody**()

.**jsonPath**("$.**id**").**isNotEmpty**()

.**jsonPath**("$.**name**").**isEqualTo**(**newStudent**.**getName**())

.**jsonPath**("$.**address**").**isEqualTo**(**newStudent**.**getAddress**());

}

...

}

5. WEBSOCKETS и RSOCKET

5.1 Веб-сокеты

В Spring 5 WebSockets также получает дополнительные реактивные возможности. Чтобы создать сервер WebSocket, вы можете создать реализацию WebSocketHandler интерфейса, которая содержит следующий метод:

Mono<Void> **handle**(WebSocketSession session)

Этот метод вызывается при установке нового соединения WebSocket и позволяет обрабатывать сеанс. Он принимает в WebSocketSession качестве входных данных и возвращает Mono <Void>, чтобы сигнализировать о завершении обработки сеанса приложением.

WebSocketSession имеет методы, определенные для обработки входящих и исходящих потоков:

Flux<WebSocketMessage> **receive**()

Mono<Void> **send**(Publisher<WebSocketMessage> messages)

Spring WebFlux также предоставляет WebSocketClient реализации для Reactor Netty, Tomcat, Jetty, Undertow и стандартной Java.

Для получения дополнительной информации прочтите следующую главу в документации Spring's Web on Reactive Stack: [WebSockets](https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-websocket)

5.2 RSOCKET

RSocket - это протокол, моделирующий семантику реактивных потоков по сети. Это двоичный протокол для использования в транспортных потоках байтовых потоков, таких как TCP, WebSockets и Aeron. В качестве введения в эту тему я рекомендую следующий пост в блоге, который написал мой коллега Pär: [An introduction to RSocket](https://callistaenterprise.se/blogg/teknik/2020/06/05/rsocket-part-1/)

А для получения дополнительной информации о поддержке Spring Framework протокола [RSocket](https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#rsocket)

6. Подводя итог…

Это сообщение в блоге продемонстрировало, как WebFlux можно использовать для создания реактивного веб-приложения.

# Работа с backpressure в Reactor

**Backpressure**(back pressure) – это то, с чем рано или поздно сталкивается почти любой разработчик, а для некоторых это становится частой и серьезной проблемой.

В мире программирования термин “backpressure” является аналогией, позаимствованной из физики. В двух словах, backpressure – это сопротивление или некоторая сила, действующая в направлении, противоположном желаемому направлению движению частиц в трубе. В контексте программирования эту фразу можно переиначить: …это сопротивление желаемому потоку данных приложения.

Цель любого приложения – это получение входных данных и приведение их к некоторому желаемому виду. Когда возникает backpressure, между входом и выходом возникает некое сопротивление. В большинстве случаев, узкое место возникает из-за ограничений в вычислительных мощностях – данные не успевают обрабатываться с всё нарастающей скоростью их поступления. Но есть и другие формы backpressure, например, когда приложение ожидает реакции пользователя на некоторое событие, которая, по понятным причинам, может запаздывать.

Иногда под backpressure понимается не само явление как таковое, а конкретный механизм его обработки.

Пример чтения и записи файлов

Запись файла процесс более медленный, чем чтение. Если комбинация ОС, жесткого диска и библиотек обеспечивает эффективную скорость чтения в 150 мб/с, а записи – в 100 мб/с, то при чтении файла с последующей его записью обратно на диск, вы должны записывать в буфер “лишние” 50 мб каждую секунду.

**Решение очевидно:** читать ровно столько, сколько сможете записать. Практически все библиотеки имеют соответствующие абстракции для работы с этими случаями.

Взаимодействие между серверами

Микросервисная архитектура, где каждый микросервис может разворачиваться на отдельном сервере, пользуется все большей популярностью. Backpressure в этом случае возникает, когда один сервер отправляет запросы другому слишком быстро и второй сервер не успевает их обрабатывать.

Предположим, что **Сервер A** отправляет **Серверу B** 100 запросов в секунду, но **Сервер B** может обрабатывать только 75 из них. Имеем дефицит в 25 запросов в секунду. В любом случае, **Сервер B** должен каким-то образом работать с backpressure. Один из вариантов – это буферизация избыточных запросов, но если они будут сыпаться с прежней скоростью, однажды свободная память сервера закончится. Можно игнорировать избыточные запросы, но во многих случаях это запрещается требованиями к системе.

Идеальный случай, когда **Сервер B** может сам контролировать поток запросов от **Сервера А**, но и это не всегда возможно. Например, если **Сервер А** генерирует запросы от имени пользователя. Мы же не можем сказать пользователям “притормозите!” (хотя иногда стОит!).

Стратегии backpressure

Так как же работать с backpressure? Если опустить вариант увеличения производительности серверного железа, остаётся всего три опции:

Контролировать источник (увеличивать частоту или снижать – решает приёмник)

Буферизация (временно хранить запросы, которые не были обработаны “на лету”)

Игнорировать (пропускать обработку части запросов)

**Контроль источника** – безусловно лучший вариант. Если говорить о случаях реального применения, то единственный возможный оверхэд – реализация самого механизма контроля.

К сожалению этот вариант не всегда реализуем. Например случай с валом пользовательских запросов самый очевидный, ведь не так-то просто договориться с пользователями!

**Буферизация**обычно выбирается в качестве следующего пути решения проблемы. Но надо держать в уме, что неограниченная буферизация опасна, так как приводит к утечкам памяти с последующим падением сервера. Зачастую лучше начать игнорировать запросы, чем продолжать складывать их в буфер, отнимая остатки памяти у сервера.

**Игнорирование**входящих запросов это последняя стратегия, которая часто сочетается с буферизацией. Обычно “сбрасывают” какую-то часть запросов ежесекундно, чтобы увеличить время существования буфера.

Укрощаем backpressure

Существует ряд паттернов, которые чаще всего используются при работе с backpressure вне зависимости от используемого языка программирования.

Pull

С использованием pull-стримов потребитель контролирует источник. Обычно это вариант работы 1 к 1, то есть запрос – ответ, но есть паттерны и для request(n) (например **Flowables**в **RxJava**).

Push

В push-стримах источник направляет сообщение в сторону потребителя если последний доступен и готов обработать запрос. Обычно такие стримы используются при обработке действий пользователя. Существует множество библиотек, самая популярная из которых – **RxJava**.

Немного сетевой теории

Для понимания механизма работы backpressure в фреймворке **WebFlux** нужно вспомнить, какой транспортный протокол используется при взаимодействии по умолчанию. Обычно серверы общаются между собой через TCP-соединения. TCP скорее взаимодействует с байтами, чем с элементами логики приложения. Обычно, когда говорят о backpressure, подразумевают контроль количества принятых или отправленных по сети логических объектов и, хотя TCP имеет собственные средства для контроля потоков, они всё же работают для байтов, а не для объектов.

TCP Flow control

**Flow control** (далее – контроль потока) означает, что TCP гарантирует, что отправитель не подавляет получателя , отправляя пакеты быстрее, чем тот может их обработать. Это и есть backpressure в контексте транспортного уровня модели OSI. Идея в том, что получатель даёт отправителю обратную связь с данными о своём текущем статусе.

Что происходит, когда мы отправляем данные по сети? Отправитель записывает данные в сокет, транспортный уровень (в нашем случае TCP) упаковывает их в сегмент и передаёт их в сетевой уровень (IP), который каким-то путём доставляет пакет получателю.

На принимающей стороне сетевой уровень доставляет эти данные до уровня TCP, что делает их доступными для получателя. TCP хранит данные для отправки в общем буфере отправки, а данные для получения – в общем буфере получения. Когда приложение готово, оно читает данные из буфера получения.

Контроль потока гарантирует, что мы не отправляем новых пакетов, если буфер получения уже полон, так как получатель все равно не сможет обработать эти запросы и будет вынужден сбросить (проигнорировать) их. Для управления объемом данных для отправки через TCP, получатель объявляет своё окно (receive window), то есть объем свободного места в буфере получения.

При получении нового пакета TCP отправляет ack сообщение отправителю, подтверждающее, что пакет получен корректно. Вместе с ack сообщением передаётся и размер текущего окна получения, поэтому отправитель знает, что он может или не может отправлять новые данные.

Вернёмся к WebFlux

В текущей реализации WebFlux backpressure регулируется транспортным протоколом, но не отражает реального состояния получателя.

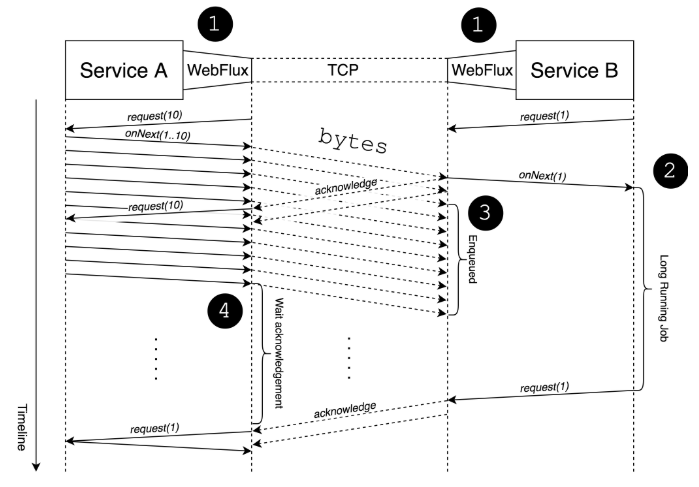
  
<https://stackoverflow.com/questions/52244808/backpressure-mechanism-in-spring-web-flux>

Диаграмма выше отражает взаимодействие между двумя микросервисами, где левый отправляет потоки данных, а правый – их принимает.

WebFlux берёт на себя работу по преобразованию объектов в байты и обратно для дальнейшей передачи через TCP.

Начало длительной обработки элемента, которая по окончании запрашивает следующий элемент.

В этой точке WebFlux удерживает байты, приходящие по сети, без отправки ack, так как бизнес-логика принимающего микросервиса занята обработкой предыдущего запроса.

Из-за природы контроля потока TCP, левый микросервис может по-прежнему отправлять данные в сеть.

Как видно из диаграммы, demand получателя отличается от demand отправителя (demand измеряем в логических объектах). Это значит, что их demand изолирован и работает только в случае взаимодействия WebFlux с бизнес-уровнем приложения. То есть контроль backpressure в WebFlux не так справедлив, как бы мы хотели.

А если backpressure, но для HTTP-запросов?

Информация о backpressure не отправляется по протоколу HTTP (он просто-напросто не поддерживает такой функционал). Поэтому всё снова упирается в TCP flow control и буферы отправки и получения. Demand’ы отправки/получения в **Reactive Streams** также опираются на TCP flow control.

В Spring MVC подразумевается, что приложения могут блокировать текущий поток, например для выполнения удалённых вызовов в другие системы. Поэтому контейнеры сервлетов используют относительно большой пул потоков для сглаживания последствий блокировок.

WebFlux и другие неблокирующие серверы предполагают запуск неблокирующих приложений, поэтому используют небольшой пул потоков фиксированного размера (так называемые event loop workers).

Модель потоков Spring WebFlux

На WebFlux сервере можно увидеть следующие потоки:

На оригинальном сервере Spring WebFlux существует один поток для работы сервера и несколько потоков для обработки запросов (обычно их количество равно числу ядер в процессоре).

Реактивный WebClient работает в режиме event loop.

Reactor обеспечивает абстракции пула потоков, которые называются scheduler’ами. Для переключения обработки на другой пул используется метод publishOn(). Планировщики (scheduler’ы) названы в соответствии со стратегией работы, например parallel для CPU-ориентированной параллельной обработки с ограниченным количеством потоков, elastic для I/O – с большим количеством потоков.

Под капотом WebFlux трудится **Reactor-Netty** – обёртка над **Netty**с поддержкой backpressure. **Reactor-Netty** создаёт Runtime.getRuntime().availableProcessors() \* 2 потоков для использования во внутреннем EventLoopThreadPool. Если входящий запрос завершается блокирующим удалённым вызовом, то его нужно соответствующим образом обернуть, чтобы текущий поток мог вернуться в **event-queue** пул для обработки новых запросов.

Для тяжёлых вычислительных операций имеет смысл использовать выделенный пул потоков. Это гарантирует, что **EventLoop**Netty будет только принимать и отправлять данные по сети, не тратя ресурсы на другие операции:

@PostMapping

public Mono<CpuIntensiveResult> cpuIntensiveProcessingHandler(

Mono<CpuIntensiveInput> monoInput

) {

**return** monoInput

.publishOn(Schedulers.fromExecutorService(myOwnDedicatedExecutor))

.map(i -> doCpuIntensiveInImperativeStyle(i));

}

Если интересен ручной контроль количества запросов, можно посмотреть в сторону метода [limiRate(int n)](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html#limitRate-int-)