ChatGPT

Проект Reactor – основные концепции Reactive Streams и Reactor Core

Введение в реактивное программирование и Reactive Streams

Реактивное программирование – это парадигма асинхронного программирования, в которой основной единицей работы являются **потоки данных** и распространение изменений [1]. В реактивном подходе поток данных представляет последовательность событий (элементов), и потребители реагируют на поступление этих событий, вместо того чтобы запрашивать их активно.

В Java основой стандарта реактивного программирования является спецификация **Reactive Streams** – набор интерфейсов и правил взаимодействия для библиотек реактивного программирования на JVM [2]. Эти интерфейсы были включены в Java 9 в виде `java.util.concurrent.Flow` API. Основные интерфейсы Reactive Streams: - **Publisher** (издатель) - **Subscriber** (подписчик) - **Subscription** (подписка) - **Processor** (процессор)

Reactive Streams можно рассматривать как развитие шаблона **Observer** (Наблюдатель), но с важным отличием: вместо привычной модели *"pull"* (как в `Iterator`), реактивные потоки используют модель *"push"* – то есть **издатель** сам отправляет данные подписчику по мере их появления [3]. Такой подход позволяет обрабатывать асинхронные последовательности событий, включая передачу ошибок и сигналов завершения, единообразным образом.

В реактивной модели общение ведётся с помощью специальных сигналов: - `onNext` – сигнал с новым элементом данных (может поступить 0 или более раз). - `onComplete` – сигнал об успешном завершении потока (больше данных не будет). - `onError` – сигнал об ошибке, завершает поток с ошибкой.

После передачи `onError` или `onComplete` поток считается **завершённым**, дальнейшие `onNext` недопустимы [4]. Последовательность событий можно описать схемой:

```
onNext (x) * N -> (onComplete | onError) [завершающий сигнал]
```

Ниже рассматриваются основные компоненты Reactive Streams и их роль.

Основные интерфейсы Reactive Streams

Publisher (Издатель)

`Publisher<T>` – производитель (источник) реактивного потока данных типа `T`. Издатель генерирует элементы и передаёт их подписчику через вызовы `onNext`. Кроме данных, он может послать сигнал об **окончании последовательности** (`onComplete`) или об **ошибке** (`onError`)

[5] [6]. Важная особенность: издатель не начинает испускать данные, пока не появится подписчик и не будет запрошено определённое количество элементов (см. Backpressure ниже).

Каждый `Publisher` по спецификации Reactive Streams обязан корректно обрабатывать **backpressure** – то есть не подавать элементов быстрее, чем запрошено подписчиком. Подробнее про механизм backpressure – в разделе ниже.

> **Пример:** В Project Reactor классы `Flux` и `Mono` являются реализациями интерфейса `Publisher` [5] [6]. `Flux` генерирует 0..N элементов, `Mono` – 0..1 элемент (подробности о них – далее).

Subscriber (Подписчик)

`Subscriber<T>` – потребитель данных из потока. Подписчик подписывается на Publisher и получает: - Вызов `onSubscribe(Subscription s)` – при успешной подписке (содержит объект `Subscription` для управления потоком). - Вызовы `onNext(T item)` – при поступлении нового элемента `item` от издателя. - Вызов `onComplete()` – если издатель нормально завершил поток. - Вызов `onError(Throwable e)` – если произошла ошибка при обработке или генерации данных.

Методы `onComplete` и `onError` являются *терминальными*: после них никаких сигналов больше не придёт.

Подписчик управляет скоростью потребления через `Subscription` – он может запрашивать определённое количество элементов методом `request(n)` у связанной подписки (см. ниже про Subscription). Таким образом, подписчик сигнализирует издателю, сколько элементов он готов обработать (это и есть механизм **backpressure**).

> **Важно:** Если подписчик не определил обработку ошибок (не переопределён `onError`), то по умолчанию Reactor выбросит неперехваченную ошибку (`OnErrorNotImplemented`) [7]. Поэтому **на практике всегда следует предоставлять обработчик ошибки** либо через подписку, либо через оператор error handling (см. раздел об обработке ошибок).

Subscription (Подписка)

`Subscription` представляет собой связь между издателем и подписчиком. Именно через этот объект осуществляется **управление потоком данных**: - `request(long n)` – запрос n элементов у издателя. Подписчик вызывает этот метод, чтобы сообщить, сколько элементов он готов принять. Издатель не должен отправлять больше, чем запрошено. - `cancel()` – отмена подписки (отписка). После отмены издатель должен прекратить посылать сигналы этому подписчику.

Этот механизм лежит в основе **обратного давления (backpressure)**. В начале, как только подписчик подключается (`onSubscribe`), он обычно запрашивает определённое число элементов. Издатель, получив `request(n)`, может отправить *до* n элементов. Затем подписчик может снова вызвать `request(...)` чтобы получить ещё. Если подписчик не успевает обрабатывать данные, он может запросить меньше элементов или вовсе не запрашивать, тем самым «притормаживая» издателя.

В Reactor и многих реализациях, **если запросить** `Long.MAX_VALUE` **элементов**, это считается запросом *без ограничений*, т.е. **неограниченный поток** (издатель будет слать данные на полной скорости) [8]. Такой неограниченный запрос фактически отключает backpressure.

> **По умолчанию** большинство способов подписаться в Reactor сразу делают неограниченный (`Long.MAX_VALUE`) запрос данных [9]. Поэтому, если нужен тонкий контроль, подписчик должен явно управлять запросами (например, использовать `BaseSubscriber`, см. далее).

Processor (Процессор)

`Processor<T,R>` – это *особый тип*, сочетающий поведение и `Publisher<T>` (издателя), и `Subscriber<R>` (подписчика) [10]. Проще говоря, процессор может подписаться на один поток и одновременно предоставлять выходной поток для других подписчиков. Он используется как промежуточное звено – получает данные, возможно их преобразует, и пересылает дальше.

Однако, на практике в Reactor **нечасто требуется реализовывать собственный Processor**. Большинство задач решаются встроенными *операторами* Reactor (которые сами по себе внутренне являются процессорами, управляя подпиской и публикацией). Ранее в Reactor были реализованы различные классы Processor (например, `EmitterProcessor`, `ReplayProcessor` и др.), но **начиная с версии 3.5.0 они объявлены устаревшими и заменяются на API под названием** Sinks**** [11] [12].

Sinks – это специализированные объекты в Reactor для безопасного ручного генератора сигналов. С помощью Sinks можно императивно отправлять события `next`, `error`, `complete` в поток, при этом Reactor обеспечивает соблюдение правил Reactive Streams (например, защищает от одновременной эмиссии из разных потоков) [13] [10]. Sinks предоставляют разные политики буферизации и multicast/unicast поведения (подробнее о Sinks – в соответствующем разделе ниже).

> **Совет:** На собеседовании могут спросить про **Processor** – достаточно объяснить, что это **комбинация издателя и подписчика** (наследует оба интерфейса). В современных версиях Reactor вместо прямого использования Processor рекомендуется API Sinks для ручной отправки событий, так как Sinks автоматически обрабатывают вопросы потокобезопасности и backpressure [13] [10].

Flux и Mono – основные типы Reactor

Reactor Core предоставляет два основных реактивных типа: - **Flux** – поток из *0 или более* элементов (0..N). - **Mono** – поток максимум из *одного* элемента (0..1).

Оба типа реализуют интерфейс `Publisher` из Reactive Streams. Рассмотрим их подробнее.

Flux – поток 0..N элементов

`Flux<T>` представляет **асинхронную последовательность** элементов типа T, которая может содержать любое количество элементов (включая бесконечный поток) и завершается сигналом завершения или ошибкой [5].

Особенности Flux: - Может не выдать ни одного элемента (только сразу завершиться) – например, `Flux.empty()` эмитит только `onComplete` без `onNext`. - Может выдать конечное число элементов и затем завершиться (`onComplete`), либо выдать бесконечный поток элементов (в этом случае `onComplete` не вызывается, поток не завершается сам по себе). - Может завершиться ошибкой в любой момент, тогда последовательность прерывается сигналом `onError`.

Таким образом, Flux – наиболее общий тип. **Примеры Flux**: последовательность чисел, события из очереди сообщений, поток изменений файловой системы и т.д. Даже бесконечные источники (например, `Flux.interval(Duration)` – бесконечный поток с периодическими тиками таймера) представлены типом Flux [14].

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

Пример выше создает Flux из трёх строковых элементов. После эмиссии этих трёх значений поток автоматически завершится `onComplete`.

> **Важно:** Flux *ленивый* – он не начнёт испускать элементы без подписчика. Подписка запускает выполнение потока (см. раздел о подписке ниже). До вызова `subscribe()` Flux лишь описывает *потенциальный* поток вычислений, но ничего не происходит [15].

Mono – результат 0..1 элемент

`Mono<T>` – это специализированный Publisher, который может выдать **не более одного** элемента типа T. Возможны три варианта поведения Mono: 1. Эмитит один элемент (`onNext`) и затем `onComplete`. 2. Сразу `onComplete` без элемента (то есть пустой результат). 3. `onError` – завершение с ошибкой (не выдав ни одного элемента).

Mono удобно использовать для асинхронных операций, которые возвращают единичный результат (или не возвращают ничего). Например, **HTTP-запрос** на получение объекта может быть представлен как `Mono<Response>`, **запись в базу** – как `Mono<Void>` (возвращает только сигнал завершения).

Особенности Mono: - Если Mono успешен и возвращает значение, то за единичным `onNext` сразу последует `onComplete` [6]. - Mono никогда не выдаёт более одного `onNext`. Комбинация из `onNext` **и** `onError` считается нарушением (недопустимо выдать значение и затем ошибку) [16]. То есть Mono либо успешен (может иметь значение или быть пустым), либо завершён ошибкой. - Mono предоставляет ограниченный набор операторов по сравнению с Flux. Многие операторы, применённые к Mono, возвращают снова Mono; но операторы комбинирования (например, объединение двух Mono) обычно дают результат типа Flux [17]. Например, `Mono.concatWith(Publisher)` возвращает Flux, а вот `Mono.then(Mono)` – снова Mono, потому что последовательное выполнение двух Mono всё равно даёт не больше одного результата.

Примеры Mono:

```
Mono<String> noData = Mono.empty();            // пустой Mono<String>, сразу завершится
Mono<Integer> data = Mono.just(10);            // успешный Mono с одним числом 10
Mono<Double> error = Mono.error(new RuntimeException("oops")); // завершится ошибкой
```

Mono можно использовать для обозначения операций без полезного значения, но с сигналом о завершении – в этом случае используют тип `Mono<Void>`. Например, `Mono<Void>` может сигнализировать, что некая фоновая задача завершилась (успешно или с ошибкой).

> **Совет:** Интервьюеры часто спрашивают **разницу Flux vs Mono**. Здесь главное: **Flux** – для нескольких элементов, **Mono** – максимум для одного. Ещё можно упомянуть, что у Flux есть специальные операторы для агрегации потоков (группировка, буферизация и т.п.), тогда как Mono – это по сути аналог `CompletableFuture` или *Promise* на один результат [6]. Mono часто используется для представления ответов в реактивных Web API (HTTP запрос -> Mono<Response>), а Flux – для потоков данных, сообщений и пр.

Создание Flux/Mono и подписка на них

Reactor предоставляет множество *фабричных методов* для удобного создания Flux и Mono из разных источников данных. Рассмотрим некоторые простые способы.

Фабричные методы создания

- **Из известных значений**: `Flux.just(T... values)` создает Flux, испускающий заданные значения подряд и завершающийся. Аналогично, `Mono.just(T value)` – Mono с одним значением.
- **Из коллекций или массивов**: `Flux.fromIterable(Iterable)` или `Flux.fromStream(Stream)` может создать поток из элементов коллекции или стрима.
- **Диапазоны чисел**: `Flux.range(int start, int count)` генерирует `count` последовательных чисел начиная с `start` [18].
- **Пустые последовательности**: `Flux.empty()` и `Mono.empty()` – немедленно завершающиеся без элементов.
- **Ошибка**: `Flux.error(Throwable)` / `Mono.error(Throwable)` – сразу завершаются с ошибкой.
- **Отложенное создание**: `Mono.fromCallable(...)` или `Mono.defer(...)` – позволяет выполнить функцию или лямбду при подписке и получить её результат как Mono (если функция бросает исключение, Mono завершится ошибкой).

Пример использования фабричных методов:

```
Flux<String> seq1 = Flux.just("foo", "bar", "baz");
List<String> list = Arrays.asList("foo", "bar", "baz");
Flux<String> seq2 = Flux.fromIterable(list);

Mono<String> mono1 = Mono.just("hello");
```

```
Mono<String> mono2 = Mono.empty();

Flux<Integer> numbers = Flux.range(5, 3);  // выдаст 5,6,7 и завершится
```

В примере `seq1` и `seq2` – эквивалентные Flux из трёх строк. `mono1` содержит строку, а `mono2` – пустой (сразу complete). `numbers` выдаст последовательность [5,6,7].

Подписка на поток (`subscribe`)

Создав Flux или Mono, необходимо **подписаться** на него, чтобы начать получение данных. Метод `subscribe()` у Flux/Mono запускает поток и регистрирует обработчики для приходящих сигналов.

В классе `Flux` определено несколько перегрузок `subscribe(...)` для разных случаев [19]:

```
flux.subscribe();                                  // (1) начать выполнение без
обработчиков (кроме ошибок по умолчанию)
flux.subscribe(value -> { ... });                  // (2) обработка каждого
значения
flux.subscribe(value -> { ... },
               error -> { ... });                  // (3) + обработка ошибки
flux.subscribe(value -> { ... },
               error -> { ... },
               () -> { ... });                    // (4) + обработка успешного
завершения
flux.subscribe(value -> { ... },
               error -> { ... },
               () -> { ... },
               subscription -> { ... });          // (5) + доступ к Subscription
```

1. **Без параметров** – просто запустить выполнение. Часто используется в примерах, где важен лишь сам факт запуска, либо в тестах. Если произойдёт ошибка, а обработчик не указан, она будет брошена как `OnErrorNotImplemented`.
2. **С** `Consumer<T>` – обработчик для каждого элемента (`onNext`). Используется, когда нам нужно что-то сделать с каждым элементом (например, вывести на экран).
3. **С обработчиком ошибки** – позволяет перехватить и обработать ошибку (`onError`), например, залогировать или вывести сообщение.
4. **С обработчиком завершения** – код, который выполнится когда поток успешно закончился (`onComplete`).
5. **С доступом к** `Subscription` – в эту версию передаётся четвертым аргументом потребитель, который получает объект `Subscription`. Это даёт возможность вручную контролировать запрос элементов (вызовом `subscription.request(n)`) или отменять подписку (`subscription.cancel()`).

Методы `subscribe` возвращают объект `Disposable`, с помощью которого тоже можно отменить подписку вызовом `.dispose()` – это удобно сохранить, если нужно отменять поток по каким-то условиям в дальнейшем [20]. В Reactor `Disposable` – это интерфейс маркер, указывающий, что ресурс можно освободить; для подписки его вызов эквивалентен `cancel()` на Subscription.

Пример 1: простая подписка без обработчиков:

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

Этот код запустит выполнение Flux, который излучит 1, 2, 3 и завершится. Однако мы не указали никаких действий при получении данных, поэтому результат никак не отображается.

Пример 2: подписка с обработкой значений:

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

Вывод в консоль:

```
Получено: 1
Получено: 2
Получено: 3
```

Каждый элемент обработан лямбдой, которая печатает его.

Пример 3: обработка ошибки при подписке:

```
Flux<Integer> ints = Flux.range(1, 5)
    .map(i -> {
        if (i <= 3) return i;
        else throw new RuntimeException("Got to " + i);
    });

ints.subscribe(
    i -> System.out.println("NEXT: " + i),
    err -> System.err.println("Ошибка: " + err.getMessage())
);
```

В примере мы намеренно генерируем ошибку, когда число превышает 3. Подписчик печатает элементы до ошибки и затем выводит сообщение об ошибке:

```
NEXT: 1
NEXT: 2
NEXT: 3
Ошибка: Got to 4
```

Обратите внимание: числа 4 и 5 не напечатались, потому что при попытке обработать 4 выброшено исключение, которое привело к `onError` и завершению потока. Элемента 5 Reactor уже не обрабатывал.

Пример 4: ручное управление Subscription:
Иногда требуется самому регулировать, сколько элементов запрашивать. Можно использовать перегрузку subscribe с четвертым параметром или реализовать собственный Subscriber. Проще всего – воспользоваться классом `BaseSubscriber<T>` из Reactor. Он позволяет переопределить методы-хуки для управления подпиской. Например:

```
Flux<Integer> source = Flux.range(1, 10).doOnRequest(r ->
    System.out.println("Запрошено " + r + " элементов")
);

source.subscribe(new BaseSubscriber<Integer>() {
    @Override
    protected void hookOnSubscribe(Subscription subscription) {
        System.out.println("Подписка произошла, запрашиваем 1 элемент");
        request(1);  // запросим по одному элементу
    }
    @Override
    protected void hookOnNext(Integer value) {
        System.out.println("Получен: " + value + ", запрашиваем следующий");
        request(1);  // запрашиваем следующий элемент после получения
текущего
    }
});
```

Вывод:

```
Запрошено 1 элементов
Получен: 1, запрашиваем следующий
Запрошено 1 элементов
Получен: 2, запрашиваем следующий
Запрошено 1 элементов
...
Получен: 10, запрашиваем следующий
Запрошено 1 элементов
```

В этом примере мы **последовательно** запрашиваем по одному элементу. Метод `doOnRequest` на источнике просто логирует запросы: видно, что каждый раз запрашивается 1. Такой подход полезен, если нужно строже контролировать скорость потребления.

> **Важно:** Если самостоятельно манипулировать запросами, нужно не забывать **запросить хотя бы что-то**, иначе поток может не сдвинуться с места и «застрять» без элементов [21]. В `BaseSubscriber` по умолчанию (если не переопределять `hookOnSubscribe`) при подписке сразу запрашивается `Long.MAX_VALUE` элементов, чтобы избежать застопоривания [22]. В приведённом примере мы переопределили поведение, поэтому сами вызываем `request(1)` при подписке.

Обратное давление (Backpressure)

Backpressure (*обратное давление*, или контроль нагрузки) – ключевой механизм Reactive Streams, позволяющий **сбалансировать скорость производителя и потребителя** данных. Идея в том, что подписчик запрашивает ровно столько элементов, сколько способен обработать, тем самым **предотвращая переполнение буферов и перегрузку** памяти или CPU [23].

Как упоминалось, `Subscription.request(n)` – сигнал backpressure. Рассмотрим подробнее, как это работает в Reactor:

- **Первичный запрос**. После подписки подписчик обычно делает первый `request`. Если используется простая форма `subscribe()` или с лямбдами (без явного указания Subscription), Reactor **автоматически запрашивает Long.MAX_VALUE** элементов [9]. Это эквивалентно «шли данные без ограничений» – то есть по сути отключает backpressure. Такой подход удобен для **конечных подписчиков**, которые хотят получить все данные целиком, или для случаев, когда известен малый объём данных. Но в случаях большого или бесконечного потока нужно быть осторожным: unbounded запрос может привести к переполнению, если источник очень «производительный».

- **Явное управление спросом.** Если мы хотим регулировать потребление, можно:

- Использовать `subscribe(..., Consumer<Subscription>)` перегрузку и внутри лямбды `subscriptionConsumer` вызвать `request(n)` с нужным числом.
- Либо, как показано выше, использовать `BaseSubscriber` и вызывать `request` в `hookOnSubscribe` и других хуках.

- **Протокол propagation.** Когда подписчик вызывает `request(n)`, этот сигнал поднимается вверх по цепочке операторов к исходному Publisher. Каждый оператор может трансформировать запрос (например, буферизующий оператор может умножить запрос) [24], но в итоге источник получает информацию, сколько элементов нужно выпустить. Если источник не может сгенерировать столько (например, поток конечен и элементов меньше), он просто выдаст, что есть, и завершится.

- **Изменение запроса операторами.** Некоторые операторы Reactor **изменяют стратегию запроса**:

- Например, `buffer(n)` – группирует элементы в буферы по n штук. Если вниз по цепочке запросили 2 элемента, `buffer(5)` превратит это во внутренний запрос 2*5=10 элементов у своего источника [24] (ведь чтобы сформировать 2 буфера по 5, нужно 10 элементов).
- Операторы, создающие внутренние последовательности (например, `flatMap`) используют параметр *prefetch* (по умолчанию 32). Prefetch = 32 означает, что при подписке на каждый внутренний Publisher оператор сразу запрашивает 32 элемента из него [25]. Также при выработке ~75% от этого количества, оператор дозапрашивает ещё (так называемый *replenishing strategy*) [26].
- С помощью метода `limitRate(int highTide, int lowTide)` можно ограничить скорость поступления: например, `limitRate(100)` разобьёт большой запрос на партии по 100, запрашивая следующую партию лишь когда 75% (по умолчанию) элементов от предыдущей получено [27]. Это тоже способ сглаживания нагрузки.

- Ещё есть `limitRequest(n)` – просто обрезает стрим после n элементов, посылая cancel вверх после получения n элементов [28].

- **Стратегии переполнения.** Если поток **горячий** (то есть выдаёт данные независимо от запросов, см. раздел про Hot vs Cold) или внешний ресурс может лить данные быстрее потребителя, обычно используют стратегии: буферизация, пропуск (drop), новейший (keep latest) и т.п. В Reactor они представлены через `onBackpressureBuffer(...)`, `onBackpressureDrop()`, `onBackpressureLatest()` операторы – они определяют, что делать с элементами, пришедшими сверх запрошенного:

- `onBackpressureBuffer` – складывать «лишние» элементы в буфер (размер буфера можно ограничить, тогда переполнение вызовет ошибку или другую реакцию).
- `onBackpressureDrop` – отбрасывать новые элементы, если у подписчика нет спроса.
- `onBackpressureLatest` – держать только самый последний элемент, остальное сбрасывать.

Эти операторы нужны сравнительно редко – в основном, если вы имеете дело с **источником, который игнорирует backpressure**. В идеале же все участники Reactive Streams соблюдают протокол и **не посылают больше, чем запрошено**.

> **Совет:** На собеседовании практически гарантированно спросят про **backpressure**. Стоит чётко объяснить, что **подписчик запрашивает `N` элементов через Subscription, и издатель обязан не превышать эту скорость** [29]. Важные детали: *по умолчанию Reactor запрашивает бесконечно (Long.MAX_VALUE)*, *подписчик может отменить подписку (`cancel()`) чтобы прервать поток*, *backpressure особенно важен для медленных подписчиков или быстрых бесконечных источников*. Упоминание операторов вроде `limitRate` или `onBackpressureBuffer` будет плюсом, показывая глубокое понимание.

Операторы преобразования и комбинирования

Одно из сильнейших преимуществ Reactor – богатый набор **операторов**, позволяющих удобно преобразовывать, фильтровать, комбинировать и обрабатывать поток данных. Работа с операторами напоминает работу со Stream API, но в **асинхронном** режиме. Ниже – ключевые категории операторов и несколько примеров.

Преобразование элементов (map, filter, etc.)

- `map` – применяет функцию к каждому элементу, преобразуя элемент типа T в элемент типа R (аналогично `Stream.map`). Пример: `flux.map(x -> x * 2)` – умножит каждый числовой элемент на 2.
- `filter` – пропускает только те элементы, для которых заданный предикат возвращает true. Пример: `flux.filter(x -> x % 2 == 0)` – оставит только чётные числа.
- `flatMap` – асинхронное преобразование: функция для каждого элемента возвращает Publisher (Mono или Flux), а flatMap сливает (merge) эти внутренние потоки в один внешний Flux. Очень мощный оператор для **композиции асинхронных запросов**. Например, `userIdsFlux.flatMap(id -> service.getUser(id))` – для каждого `id` вызывает сервис, возвращающий `Mono<User>`, и затем собирает все результаты в единый Flux пользователей. **Важно:** flatMap не гарантирует сохранения исходного порядка элементов, так как запросы выполняются конкурентно.

- `concatMap` – похож на flatMap, но **гарантирует порядок**, выполняя вложенные издатели последовательно, один за другим. Используется, когда порядок критичен.
- `flatMapSequential` – компромисс: запускает внутренние Publishers параллельно, но собирает результаты в исходном порядке.
- `buffer` / `window` – накопление элементов в коллекции или под-потоки:
- `buffer(n)` собирает батч из n элементов и выдаёт их списком (List) вниз по потоку.
- `window(n)` разбивает Flux на **под-Flux** размером n элементов каждый (возвращает Flux<Flux<T>>).
- `take` / `takeUntil` – позволяют ограничить поток по количеству или условию. `take(n)` возьмёт не более n элементов и завершит поток. `take(Duration)` – возьмёт элементы в течение заданного времени.
- `delayElements(Duration)` – вводит задержку перед каждым `onNext` элементом (например, чтобы замедлить поток).
- `timeout(Duration)` – устанавливает тайм-аут: если следующий элемент не появляется вовремя, вызывает ошибку (TimeoutException) [30] . Можно сочетать с `onErrorResume` для перехода на запасной источник (например, кэш) в случае таймаута [31] .

Пример: использование нескольких операторов в цепочке – императивный «callback hell» vs декларативный Reactor:

```
// Императивная вложенная логика (условный пример, как НЕ надо)
userService.getFavorites(userId, new Callback<List<String>>() {
    public void onSuccess(List<String> favIds) {
        if (favIds.isEmpty()) {
            suggestionService.getSuggestions(...).onSuccess(...);
            // ... далее вложенные callbacks ...
        } else {
            favIds.forEach(id -> favoriteService.getDetails(id, new
Callback<Favorite>() { ... }));
        }
    }
    public void onError(Throwable e) { ... }
});

// Декларативно с Reactor:
userService.getFavorites(userId)
    .flatMapMany(favIds -> {
        if (favIds.isEmpty())
            return suggestionService.getSuggestions();    // Flux<Favorite>
        else
            return Flux.fromIterable(favIds)
                       .flatMap(favoriteService::getDetails);  //
Flux<Favorite>
    })
    .take(5)
    .publishOn(UiUtils.uiThreadScheduler())
    .subscribe(
        fav -> uiList.show(fav),
```

```
        error -> UiUtils.errorPopup(error)
    );
```

В реактивном варианте: 1. Получаем избранные `favIds` (`Mono<List<String>>` от сервиса). 2. Преобразуем через `flatMapMany` в **Flux**: если список пуст, запрашиваем рекомендации (fallback) через другой сервис; иначе для каждого ID запрашиваем детали через `favoriteService.getDetails` (который возвращает, предположим, Mono<Favorite>, а flatMapMany/flatMap собирает их в Flux<Favorite>). 3. `take(5)` – берем только первые 5 результатов. 4. `publishOn(UiThread)` – переключаем поток выполнения на поток UI для последующей обработки (отображения). 5. `subscribe` – инициируем выполнение, указывая действия при выводе результата на UI и обработке ошибки.

Этот пример иллюстрирует, как **цепочка операторов** позволяет выразить асинхронную логику последовательно, без вложенных коллбэков, улучшая читаемость и управление потоками данных [32] [33].

Комбинирование потоков

Reactor также предоставляет операторы для объединения нескольких Publishers: - `mergeWith` / `merge` – конкурентное слияние нескольких Flux. Элементы из разных источников могут чередоваться по мере поступления. - `concatWith` / `concat` – последовательная конкатенация: второй поток начнётся только после завершения первого. - `zip` – объединение элементов по индексу из двух (или более) Publisher-ов. Например, `Flux.zip(flux1, flux2, (a,b) -> combine(a,b))` будет ожидать соответствующие пары элементов и выдавать новый элемент-комбинацию. Если один поток закончится, zip завершится. - `combineLatest` – похож на zip, но каждый раз, когда какой-либо источник выдаёт новый элемент, комбинирует его с *последними* известными элементами других источников. - `firstWithSignal` / `firstWithValue` – выбирают первый выдавший сигнал Publisher (полезно для организации гонки запросов, например, к основному и резервному источнику данных).

Для типа Mono существуют удобные методы: - `Mono.zip(mono1, mono2, BiFunction)` – параллельный запуск двух Mono и сбор результатов в один объект (например, Tuple2 или используя BiFunction). Аналогично есть перегрузки для 3, 4, ... Mono. - `Mono.when(mono1, mono2, ...)` – выполняет несколько Mono параллельно и возвращает Mono<Void>, которое завершается, когда все исходные завершились (идеально для параллельного выполнения независимых задач без комбинирования результата).

Пример: параллельный запрос двух сервисов и комбинирование результатов:

```
Mono<User> userMono = userService.getUser(userId);
Mono<Account> accountMono = accountService.getAccount(userId);

// Запустить оба запроса параллельно и объединить результаты, когда оба
готовы:
Mono<UserProfile> profileMono = Mono.zip(userMono, accountMono,
    (user, account) -> new UserProfile(user, account)
);
```

Здесь `UserProfile` – предположим, класс, агрегирующий информацию пользователя и его аккаунта. `Mono.zip` запустит `userMono` и `accountMono` одновременно. Как только оба Mono испустят значение, BiFunction соберёт их в `UserProfile` и mono завершится.

Другие полезные операторы

- `doOnNext`, `doOnError`, `doOnComplete`, `doFinally` – *побочные действия* (side effects) для отладки или вспомогательной логики. Они не изменяют сами элементы, но позволяют, например, залогировать получение элемента, факт ошибки или завершения. `doFinally` вызывается в конце в любом случае (при cancel, ошибке или обычном завершении) – аналог блока `finally` [34].
- `delaySubscription(Duration)` – откладывает момент подписки (и соответственно начала выполнения) на заданное время.
- `cache` – кэширует результаты Flux, чтобы повторные подписчики получали уже вычисленные элементы (похож на replay(...) + autoConnect).
- `repeat` – для Flux: повторяет последовательность заново после завершения (можно указать число повторов или условие).
- `retry` – повторная подписка при ошибке (см. раздел об ошибках ниже).

Совет: Знание нескольких ключевых операторов – обязательный минимум. На собеседовании часто просят объяснить различие между `map` **и** `flatMap` (в Reactor flatMap возвращает Publisher и объединяет результаты асинхронно, а map – простое преобразование 1:1) или между `merge` **и** `concat` (первый – параллельно, второй – последовательно). Также полезно упомянуть `zip` **для параллельного выполнения Mono** и `filter` для отбора элементов. Это демонстрирует уверенное владение реактивной обработкой данных.

Планировщики (Schedulers) и управление потоками выполнения

По умолчанию Reactor не навязывает конкретный поток выполнения – если не указано иначе, **все операторы выполняются в том потоке, где произошла подписка** [35]. В этом смысле Reactor *“конкуррентно-агностичен”* – вы сами решаете, на каких потоках выполнять ту или иную часть реактивной цепочки. Для управления потоками используются объекты `Scheduler` из Reactor.

Scheduler – аналог пула потоков или планировщика задач. Reactor предоставляет несколько готовых реализаций, доступных через фабричные методы класса `Schedulers` [36] [37]:

- `Schedulers.immediate()` – не переключает поток: задачи выполняются немедленно в текущем потоке (по сути no-op Scheduler).
- `Schedulers.single()` – возвращает планировщик, использующий **один** переиспользуемый поток (подойдёт, когда нужен гарантированно один фоновой поток для выполнения).
- `Schedulers.newSingle()` – создаёт новый отдельный поток для каждого вызова (не shared).
- `Schedulers.parallel()` – пул фиксированного размера, оптимизированный под параллельные задачи (по умолчанию размер = число ядер CPU). Используется для CPU-bound операций.

- `Schedulers.boundedElastic()` – **эластичный пул потоков** для блокирующих задач (I/O-bound). Создаёт новые потоки по мере необходимости, но ограничивает их число (по умолчанию не более 10 × число ядер). Неактивные потоки закрываются спустя таймаут. Предназначен для задач, которые могут блокировать (например, обращения к БД, файловой системе) [37] [38] . **Важно:** начиная с Reactor 3.4+, `boundedElastic` предпочтителен вместо устаревшего `elastic()`, т.к. лучше контролирует рост потоков.
- `Schedulers.elastic()` – (устаревающий) неограниченный пул, создающий бесконечно новые потоки по мере нагрузки. Может скрывать проблемы backpressure и приводить к слишком большому числу потоков [37] . Вместо него лучше `boundedElastic`.
- `Schedulers.fromExecutorService(...)` – адаптирует существующий `ExecutorService` или `Executor` к Scheduler. Можно использовать, чтобы интегрировать Reactor с вашим кастомным пулом.

  Начиная с Java 21 Reactor умеет интегрироваться с виртуальными потоками: при включении соответствующего свойства `reactor.schedulers.defaultBoundedElasticOnVirtualThreads=true` `Schedulers.boundedElastic()` будет порождать задачи на **виртуальных потоках** вместо платформенных [39] .

Операторы переключения контекста: publishOn vs subscribeOn

Reactor предлагает два основных оператора для переключения выполнения на другой Scheduler: - `publishOn(Scheduler)` - `subscribeOn(Scheduler)`

Оба планируют последующее выполнение на заданном Scheduler, но есть ключевые различия:

publishOn:
Вставленный в цепочку оператор `publishOn` влияет на **все последующие** операторы *вниз* по цепочке после него [40] . То есть, как только поток дойдёт до этого места, дальнейшие операции (`map`, `flatMap`, `filter` и пр. после publishOn) будут выполняться в потоках того Scheduler, который передан в publishOn.

Можно использовать несколько `publishOn` в разных местах цепочки, чтобы распределить разные части обработки по разным пулам. Например, можно читать данные в IO-пуле, а затем переключиться на вычислительный пул для обработки:

```
Flux<Data> flux = dataSourceFlux   // предположим, изначально исполняется на одном потоке
    .publishOn(Schedulers.boundedElastic())    // переключаем на I/O пул для следующей операции
    .flatMap(this::blockingIoOperation)        // выполняется в boundedElastic
    .publishOn(Schedulers.parallel())          // переключаем на CPU пул
    .map(this::compute)                        // выполняется в parallel
    .publishOn(Schedulers.immediate())         // вернуться в текущий поток (если нужно)
    .filter(...) ... ;
```

Поскольку publishOn влияет только на **последующую** обработку, его позиция в цепочке имеет значение.

subscribeOn:
В отличие от publishOn, оператор `subscribeOn` обычно ставят *один раз* и он влияет на **весь цепочку от самого источника** [40] [41] . `subscribeOn(Scheduler)` заставляет начальный этап подписки и исполнение *источника* (а также всего, что *выше* по цепочке) произойти на указанном Scheduler, вне зависимости от того, где в цепочке он был вызван.

Проще говоря, `subscribeOn` "поднимается" вверх до самого первого Publisher и заставляет его emit-ить данные на указанном Scheduler. Например:

```
Flux<Integer> flux = Flux.range(1, 5)
    .map(i -> i * 2);             // умножение (пока без переключения)

flux
    .subscribeOn(Schedulers.parallel())
    .subscribe(i -> System.out.println(Thread.currentThread().getName() + ":
" + i));
```

В этом случае даже оператор `map` и генерация `range` произойдут в пуле `Schedulers.parallel()` – поскольку subscribeOn перенёс выполнение источника и всей цепочки на этот Scheduler.

Важно: Если в цепочке указано несколько subscribeOn, будет действовать **только первый из них (ближайший к источнику)**, остальные будут игнорированы или, точнее, перепишут друг друга без эффекта [42] [43] . Поэтому нет смысла иметь более одного subscribeOn на одну цепочку.

Пример взаимодействия publishOn и subscribeOn:

```
Flux<Integer> flux = Flux.range(1, 4)
    .map(i -> { // (1)
        System.out.println("map1: " + Thread.currentThread().getName());
        return i;
    })
    .publishOn(Schedulers.parallel())
    .map(i -> { // (2)
        System.out.println("map2: " + Thread.currentThread().getName());
        return i;
    })
    .subscribeOn(Schedulers.boundedElastic());

flux.subscribe(i -> {
    System.out.println("subscribe: " + Thread.currentThread().getName());
});
```

Разберём выполнение: 1. `subscribeOn(Schedulers.boundedElastic())` пытается выполнить *источник* (Flux.range и далее) на `boundedElastic` . Однако... 2. ...когда поток дойдёт до `publishOn(Schedulers.parallel())` , все последующие операции переключатся на `parallel` . 3. Таким образом: - Первый `map1` выполняется **на потоке boundedElastic** (так как он *до* publishOn, но subscribeOn переместил исполнение источника на boundedElastic). - Второй

`map2` выполняется **на потоке parallel**, т.к. после publishOn. - Также и Consumer в subscribe (печатает "subscribe: ...") выполнится на parallel, ведь Subscription уже перешёл на parallel после publishOn.

Итоговый вывод может быть примерно:

```
map1: boundedElastic-1
map2: parallel-1
subscribe: parallel-1
```

Это подтверждает: **subscribeOn задаёт контекст для начала цепочки**, а **publishOn – для продолжения с определённого места** [40] .

> **Совет:** Отличие между `subscribeOn` **и** `publishOn` – один из самых популярных вопросов. Ответ: *subscribeOn влияет на весь поток и исходный Publisher (обычно ставится один раз, "вверху"), а publishOn – переключает поток выполнения начиная с места вызова вниз по цепочке* [40] . Также можно упомянуть, что обычно **subscribeOn применяют для источников (например, выполнять блокирующий источник на отдельном Scheduler)** – типичный пример: `Mono.fromCallable(this::blockingCall).subscribeOn(Schedulers.boundedElastic())`, а **publishOn – для разделения этапов обработки** (например, получить данные на IO-пуле, а обработать на CPU-пуле).

Применение Scheduler в практике

Пример 1: переключение контекста для блокирующей операции:

```
Mono<String> fileContentMono = Mono.fromCallable(() -> {
        // чтение файла (блокирующая операция)
        return Files.readString(path);
    })
    .subscribeOn(Schedulers.boundedElastic()); // выполнить чтение на пуле
для блокирующих задач
```

Здесь мы используем `boundedElastic()` – Reactor сам выделит поток из пула (или создаст новый, если все заняты) для выполнения чтения файла, чтобы не блокировать основной поток.

Пример 2: разделение этапов с publishOn:

```
Flux<Item> items = itemSourceFlux   // допустим, генерирует события быстро
    .publishOn(Schedulers.parallel())    // обрабатываем на параллельном пуле
    .map(this::transformItem)           // тяжелое CPU-преобразование
    .publishOn(Schedulers.boundedElastic())
    .flatMap(this::storeItemToDatabase); // сохранение в базе (I/O операция)
```

Первый publishOn(parallel) – чтобы параллельно обработать элементы (возможно, на нескольких ядрах). Второй publishOn(boundedElastic) – переключаемся на пул для блокирующих операций перед сохранением в БД (чтобы при сохранении не блокировать parallel-пул).

Таким образом, Reactor даёт очень гибкие возможности по **настройке потоков выполнения**, но ответственность за правильное применение ложится на разработчика.

> **Примечание:** Если *не использовать никакие Schedulers*, весь код Flux/Mono от начала до конца выполнится в том потоке, где вызван `.subscribe()`. Это упрощает отладку, но в реальных приложениях обычно необходимо задействовать фоновые потоки для ввода-вывода и распараллеливания. Reactor при этом не создаёт скрытых потоков – всё под контролем разработчика.

Обработка ошибок

В Reactive Streams ошибки обрабатываются особым образом: **ошибка считается терминальным событием**, прерывающим последовательность элементов [44] [45]. Если где-либо в цепочке операторов возникает исключение, оно преобразуется в сигнал `onError` и передаётся подписчику, после чего никакие `onNext` больше не происходят.

Однако Reactor позволяет перехватывать ошибки *до того, как они дойдут до конечного подписчика*, и предоставлять **альтернативный поток** или значение. Существует несколько стратегий обработки ошибок с помощью операторов:

1. **Вернуть запасное значение (статический фолбэк)** – оператор `onErrorReturn`. При ошибке продолжает поток одним заданным значением и завершает его успешно [46]. Аналог `catch` с возвращением значения по умолчанию.

   ```
   Flux<Integer> result = source.map(x -> 100 / x)
                               .onErrorReturn(-1);
   ```

   Здесь, если в `source` встретится 0 (деление на ноль вызовет ошибку), мы перехватим её и вместо падения вернём значение -1 как результат и завершение. Все последующие операторы после onErrorReturn уже не выполняются для ошибочного случая, поток просто завершается нормально с этим значением.

2. **Вернуть запасной поток (fallback-поток)** – оператор `onErrorResume`. При возникновении ошибки можно переключиться на другой Publisher (Flux или Mono), возможно в зависимости от типа/содержимого ошибки [47].

   ```
   Mono<Item> item = remoteService.getItem(id)
                   .onErrorResume(Exception.class, e ->
   localCache.getItem(id));
   ```

   Если `remoteService.getItem` вернул ошибку, мы обращаемся к `localCache` как запасному варианту. Функция внутри onErrorResume может проанализировать

`Throwable` и вернуть разный Publisher для разных исключений (или вообще re-throw, вернув `Mono.error(e)`). [47]

Также есть перегрузки onErrorResume и onErrorReturn, позволяющие фильтровать по классу исключения или предикату, чтобы выборочно обрабатывать только определённые ошибки [48].

1. **Игнорировать ошибку и завершить пустым** – оператор `onErrorComplete`. Редко используемый, но на случай, если ошибку можно считать некритичной: он просто переводит `onError` в `onComplete`, завершает поток без данных [49]. Можно сочетать с фильтром по типу ошибки, чтобы игнорировать только определённые несущественные исключения.

2. **Преобразовать ошибку и выбросить другой** – оператор `onErrorMap`. Перехватывает ошибку и заменяет её другой (например, обернуть в бизнес-исключение). Исходный поток завершится ошибкой нового типа [50]. Это как `catch (Exception e) { throw new OtherException(e) }` в императивном коде.

3. **Выполнить побочное действие при ошибке** – оператор `doOnError`. Он позволяет залогировать или выполнить какую-то логику при возникновении ошибки, но **не перехватывает** её (поток всё равно завершится с onError, если далее не стоит другой обработчик). Аналогично `doFinally` можно использовать для действий как при ошибке, так и при нормальном завершении, например, освобождение ресурсов.

4. **Использование ресурсов и гарантированное освобождение** – оператор `using`. Позволяет задать ресурс (например, открытое соединение, файл), поток на основе этого ресурса и функцию освобождения ресурса. Reactor гарантирует, что даже при ошибке ресурс будет освобождён (аналог try-with-resources в реактивном стиле) [51]. Более современный вариант – `usingWhen` (принимает Mono, сигнализирующий завершение работы с ресурсом, чтобы вернуть ресурс в пул, например).

5. **Повтор попытки (retry)** – это не совсем *обработка* ошибки, а способ *реакции* на неё: при ошибке автоматически заново подписаться на исходный Publisher, пытаясь повторить операцию. В Reactor есть:

6. `retry(long numRetries)` – повторить указанное число раз (при любой ошибке) [52].
7. `retryWhen(Function<Flux<Throwable>, Publisher<?>>)` – более гибко, с возможностью задержек между попытками, условием остановки и т.д. (получает поток ошибок и должен вернуть некий сигнал для повторной подписки или завершить без сигнала для прекращения).
8. `retryBackoff` (в старых версиях Reactor, сейчас deprecated, заменён на `retryWhen` с `Retry.backoff`).

Важно: `retry` перезапускает *весь* поток заново, начиная с подписки на источник. Поэтому стоит быть осторожным с побочными эффектами (могут выполниться несколько раз).

Пример – использование onErrorResume и retry:

```
Flux<String> data = fetchDataFromServer()              // может выбросить
ошибки сети
```

```
    .onErrorResume(IOException.class, e -> {
        // при сетевой ошибке берем данные из кэша
        return fetchDataFromCache();
    })
    .retry(1);  // если любая другая ошибка – попробуем заново один раз
```

Здесь: - Если произошёл `IOException` (сеть недоступна), переключаемся на `fetchDataFromCache()` – альтернативный Flux. - Для прочих ошибок (не `IOException`) – используем `.retry(1)`. Это заставит повторно подписаться на `fetchDataFromServer` один раз. Если со второго раза опять ошибка – тогда уже она пойдёт вниз по цепочке (или к финальному подписчику, если нет других обработчиков).

Замечание: Если поставить `retry` *до* `onErrorResume`, то он будет перехватывать и повторять также и IOException. Расположение операторов важно: обработчики ошибок обычно ставятся ближе к концу цепочки (после основных операторов), чтобы охватить все возможные исключения.

Terminal ошибки: После срабатывания любого из onErrorX операторов или окончания попыток retry, поток либо продолжает работу как новый (fallback) поток, либо окончательно завершается. Главное помнить, что **исходный поток после** `onError` **считается прерванным** – error-handling оператор как бы заменяет его новым потоком (либо значением). Нельзя просто «пропустить ошибку и идти дальше» по тому же потоку, не используя специальные средства. (В RxJava есть `onErrorContinue`, в Reactor 3.4+ тоже появился экспериментальный `onErrorContinue`, но он не рекомендуется к широкому применению, так как нарушает основную гарантию – терминальность ошибки).

> **Совет:** При обсуждении обработки ошибок отметьте, что **ошибки в реактивных потоках – терминальны** [44]. Частый вопрос: *как реализовать аналог try-catch?* – Ответ: с помощью операторов `onErrorReturn` (как `catch` с default-значением) или `onErrorResume` (как `catch` с альтернативным методом/источником) [46] [53]. Также можете упомянуть `retry` для повторных попыток. Важно показать понимание, что после ошибки **нельзя продолжить тот же Flux**, можно лишь переключиться на другой.

Горячие и холодные издатели (Hot vs Cold)

Все рассмотренные до сих пор примеры предполагали, что поток **ничего не делает до подписки**, и при каждом новом подписчике заново генерирует данные. Это так называемые **холодные (cold) издатели** [54]. Противоположный тип – **горячие (hot) издатели**, которые могут испускать данные независимо от наличия подписчиков или делиться событиями между подписчиками.

Холодный Publisher (Cold): каждый подписчик получает **полную собственную** последовательность данных с самого начала. Данные *генерируются заново* для каждого подписчика. Если подписчика нет – данные вообще не генерируются. Пример: `Flux.fromIterable(list)` – каждый новый подписчик переберёт весь список заново. Многие источники, зависящие от запроса, – холодные. Например, `Mono.fromCallable(() -> httpRequest())` – при подписке выполнит HTTP-запрос; две подписки сделают два независимых HTTP-запроса.

Горячий Publisher (Hot): данные генерируются независимо от подписчиков или один раз на всех. Подписчики получают элементы, которые *произошли после момента их подписки*. Если подписчика не было в момент эмиссии элементов – он их пропустил. Примеры: потоки от аппаратных датчиков, системные события – они возникают независимо от того, слушаете вы их или нет. В Reactor, например, `Flux.interval(Duration.ofSeconds(1))` – горячий поток тиков: если вы подпишетесь позже, то пропустите уже прошедшие тики и начнёте получать текущие.

> Тонкий момент: `Flux.interval` на самом деле **не начнёт** тикать, пока на него не подпишутся (то есть ленивый старт), но после подписки он выдаёт данные по времени – это бесконечный **горячий** источник, т.к. второй подписчик не получит те тики, что прошли до его подписки.

Рассмотрим простой пример:

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

coldFlux.subscribe(color -> System.out.println("Subscriber1: " + color));
coldFlux.subscribe(color -> System.out.println("Subscriber2: " + color));
```

Вывод:

```
Subscriber1: BLUE
Subscriber1: GREEN
Subscriber1: ORANGE
Subscriber2: BLUE
Subscriber2: GREEN
Subscriber2: ORANGE
```

Оба подписчика получили все элементы. При втором подписчике последовательность **началась сначала** (список был снова итерарован) [55] [56] . Это поведение **cold**: каждый подписчик независим.

Теперь сделаем **горячий** поток с помощью Sinks (императивно) или оператора share():

```
// Горячий источник через Sinks.Many (multicast)
Sinks.Many<String> sink = Sinks.many().multicast().directAllOrNothing();
Flux<String> hotFlux = sink.asFlux().map(String::toUpperCase);

hotFlux.subscribe(d -> System.out.println("Subscriber1: " + d));
sink.tryEmitNext("blue");
sink.tryEmitNext("green");

hotFlux.subscribe(d -> System.out.println("Subscriber2: " + d)); //
подключается позже
sink.tryEmitNext("orange");
```

```
sink.tryEmitNext("purple");
sink.tryEmitComplete();
```

Вывод:

```
Subscriber1: BLUE
Subscriber1: GREEN
Subscriber1: ORANGE
Subscriber2: ORANGE
Subscriber1: PURPLE
Subscriber2: PURPLE
```

Здесь второй подписчик подключился после того, как "blue" и "green" уже были отправлены через sink. Поэтому он получил только "orange" и "purple", а первых двух не увидел [57] [58] . Первый же подписчик получил все четыре цвета. Это типичное поведение **hot** источника: поздние подписчики начинают слушать **с текущего момента**.

В Reactor есть несколько способов получить горячее поведение: - Оператор `publish()` – преобразует обычный холодный Flux в `ConnectableFlux` (подробно ниже). `ConnectableFlux` позволяет запустить источник в работу отдельно от подписчиков. - Метод `connect()` у ConnectableFlux – запускает эмиссию (до вызова connect данные не текут). Это даёт возможность нескольким подписчикам *подождать* запуска, а потом одновременно начать получать поток [59] [60] . - `autoConnect(n)` – разновидность, автоматически запускающая подключение к источнику после появления n подписчиков [61] . - `refCount(n)` – ещё умнее: подключает при n подписчиках, и отключает (отменяет subscription к исходнику), когда число подписчиков падает до ниже n [62] . Также есть перегрузка с таймаутом отключения, чтобы дать возможность быстро переподписавшимся клиентам не прерывать поток [63] . - Упрощённый вариант: оператор `share()` – эквивалент `publish().refCount(1)`. То есть делает Flux горячим, подключаясь к источнику при первом подписчике и оставаясь подключенным пока хоть один подписчик есть. Подписчики, присоединившиеся позже, не получат прошлых элементов. `share()` удобно для трансформации холодного потока в горячий с минимальным кодом.

- `replay(n)` – возвращает ConnectableFlux, который **кэширует n последних элементов** и при новом подписчике воспроизводит их перед тем, как переключиться на live-данные [64] . Если использовать `replay(Duration)` – кэш по времени (последние элементы за некоторый промежуток). У ConnectableFlux, полученного через replay, тоже можно вызвать autoConnect или refCount, чтобы не делать вручную connect. **Важно:** replay превращает поток в горячий, но с **ретроспекцией** – новые подписчики получают часть истории.

- **Использование** `Sinks` – как в примере выше, даёт полный контроль: Sinks.many().multicast().directXxx() создаёт hot-поток, где вы вручную эмитите события. Этот способ применим для интеграции с внешними лисенерами/колбэками (например, обёрнуть слушатель системных событий в Flux).

Когда нужен hot? Обычно, когда вы имеете *единый источник событий*, который должны слушать несколько потребителей, и не хотите повторно создавать источник для каждого. Например, есть одна очередь сообщений RabbitMQ, и несколько обработчиков – можно обернуть её в Flux и раздать (multicast) всем обработчикам. Либо периодически обновляемый кеш – раздать обновления нескольким подписчикам.

Когда нужен cold? Когда данные должны быть **раздельно вычислены для каждого подписчика**, или чтобы изоляция подписчиков была полной. Например, `Mono.fromCallable(dbQuery)` – каждый подписчик выполнит свой запрос к БД (что логично, т.к. запросы могут быть разными или для разных пользователей).

> **Совет:** Тема **Hot vs Cold** может всплыть в вопросах типа: *“что произойдёт, если два раза подписаться на один Flux?”*. Хороший ответ: *“Если Flux холодный – каждый подпишется на свой поток данных (пример: Flux.fromIterable всегда начнёт с начала для нового подписчика). Если Flux горячий (например, после share()), то второй подписчик присоединится к уже идущему потоку и пропустит уже прошедшие элементы”* [65] [66]. Можно упомянуть `ConnectableFlux` и `publish().refCount()`, если уверенно в них ориентируетесь. Это покажет глубину знаний.

Тестирование реактивных потоков

(Кратко, так как не напрямую спрашивалось, но на всякий случай):

Для тестирования Flux/Mono Reactor предоставляет утилиту **StepVerifier** (в модуле reactor-test). С помощью `StepVerifier.create(flux)` можно декларировать ожидаемые события (`expectNext`, `expectError`, `expectComplete`) и затем вызвать `verify()` или `verifyComplete()` для запуска проверки. Это удобнее, чем вручную подписываться в тестах и собирать результаты, особенно для асинхронных потоков.

Пример:

```
Flux<String> flux = Flux.just("a", "b").concatWith(Mono.error(new
RuntimeException("boom")));
StepVerifier.create(flux)
    .expectNext("a", "b")
    .expectErrorMessage("boom")
    .verify();
```

StepVerifier автоматически будет ждать асинхронные эмиссии и сравнивать с ожидаемым сценарием.

> **Совет:** Если вас спросят “как протестировать Flux/Mono”, упомяните StepVerifier. Это покажет, что вы знакомы с экосистемой Reactor и умеете писать unit-тесты для реактивного кода.

Заключение

Project Reactor – мощная библиотека для реактивного программирования на JVM. В этом конспекте мы рассмотрели: - **Базовые интерфейсы Reactive Streams**: Publisher, Subscriber, Subscription, и как они реализованы в Reactor. - **Типы Flux и Mono** и их предназначение. - **Создание реактивных цепочек** и важность подписки (`subscribe()`). - **Механизм Backpressure (обратного давления)** и управление запросами. - **Операторы преобразования, фильтрации, объединения** – как строить конвейеры обработки данных. - **Потоки и Schedulers** –

как управлять, где выполняется реактивный код, и правильно использовать `publishOn`/`subscribeOn`. - **Обработка ошибок** – различные подходы дляFallback, ретраев и пр. - **Hot vs Cold** источники – отличие в поведении при многократной подписке и способы multicast.

В процессе были даны примеры кода и пояснения, иллюстрирующие каждую концепцию на практике.

Подводя итог, для успешного прохождения собеседования по Reactor стоит акцентировать внимание на следующих пунктах: - Понимание **контракта Reactive Streams** (onNext/onError/onComplete, единственность терминальных событий, роль Subscription). - Умение объяснить **backpressure** и как в Reactor задаётся спрос через request. - Различие и применение **Flux vs Mono**. - **Ключевые операторы** (map, flatMap, filter, take, etc.) и примеры их использования. - Разница между **subscribeOn и publishOn** – где и как они воздействуют на поток выполнения. - Способы **обработки ошибок** – onErrorReturn, onErrorResume, retry. - Понимание **горячих и холодных потоков**, умение использовать `publish()/connect()` или `share()` при необходимости. - (Опционально) Использование **Sinks**/Processor для императивной эмиссии и интеграции с внешними источниками событий. - (Опционально) Как тестировать реактивные последовательности (StepVerifier).

Хорошее владение этими темами позволит уверенно отвечать на вопросы и показывать практический опыт работы с Project Reactor и реактивными потоками в Java. Успехов в подготовке!

[3] [5] [6] [29] [40] [66]

---

[1] [2] [3] [4] [15] [30] [31] [32] [33] Introduction to Reactive Programming :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/reactiveProgramming.html

[5] [14] Flux, an Asynchronous Sequence of 0-N Items :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/flux.html

[6] [16] [17] Mono, an Asynchronous 0-1 Result :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/mono.html

[7] [34] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] Handling Errors :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/error-handling.html

[8] [9] [18] [19] [20] [21] [22] [24] [25] [26] [27] [28] [29] Simple Ways to Create a Flux or Mono and Subscribe to It :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/simple-ways-to-create-a-flux-or-mono-and-subscribe-to-it.html

[10] [11] [12] [13] Sinks :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/sinks.html

[23] Java, реактивное программирование, Reactor, Spring Cloud Function, Streams, etc... / Хабр
https://habr.com/ru/articles/679750/

[35] [36] [37] [38] [39] [40] [41] [42] [43] Threading and Schedulers :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/coreFeatures/schedulers.html

[54] [55] [56] [57] [58] [65] [66] Hot Versus Cold :: Reactor Core Reference Guide
https://projectreactor.io/docs/core/release/reference/advancedFeatures/reactor-hotCold.html

59 60 61 62 63 64 Broadcasting to Multiple Subscribers with ConnectableFlux :: Reactor Core Reference Guide

https://projectreactor.io/docs/core/release/reference/advancedFeatures/advanced-broadcast-multiple-subscribers-connectableflux.html