Используйте Stream API проще (или не используйте вообще)

Программирование, Java

С появлением Java 8 Stream API позволило программистам писать существенно короче то, что раньше занимало много строк кода. Однако оказалось, что многие даже с использованием Stream API пишут длиннее, чем надо. Причём это не только делает код длиннее и усложняет его понимание, но иногда приводит к существенному провалу производительности. Не всегда понятно, почему люди так пишут. Возможно, они прочитали только небольшой кусок документации, а про другие возможности не слышали. Или вообще документацию не читали, просто видели где-то пример и решили сделать похоже. Иногда это напоминает анекдот про «задача сведена к предыдущей».

В этой статье я собрал те примеры, с которыми столкнулся в практике. Надеюсь, после такого ликбеза код программистов станет чуточку красивее и быстрее. Большинство этих штук хорошая IDE поможет вам исправить, но IDE всё-таки не всесильна и голову не заменяет.

1. Стрим из коллекции без промежуточных операций обычно не нужен

Если у вас промежуточных операций нет, часто можно и нужно обойтись без стрима.

1.1. collection.stream().forEach()

Хотите что-то сделать для всех элементов коллекции? Замечательно. Но зачем вам стрим? Напишите просто `collection.forEach()`. В большинстве случаев это одно и то же, но короче и производит меньше мусора. Некоторые боятся, что различие в функциональности есть, но не могут толком объяснить, какое оно. Говорят, мол, `forEach` не гарантирует порядок. Как раз в стриме по спецификации не гарантирует (по факту он есть), а без стрима для упорядоченных коллекций гарантирует. Если порядок вам не нужен, вам не станет хуже, если он появится. Единственное отличие из стандартной библиотеки, которое мне известно — это синхронизированные коллекции, созданные через `Collections.synchronizedXyz()`. В этом случае `collection.forEach()` синхронизирует всю операцию, тогда как `collection.stream().forEach()` не синхронизирует ничего. Скорее всего, если вы уж используете синхронизированные коллекции, вам всё-таки синхронизация нужна, поэтому станет только лучше.

1.2. collection.stream().collect(Collectors.toList())

Собираетесь преобразовать произвольную коллекцию в список? Замечательно. Начиная с Java 1.2 у вас есть отличная возможность для этого: `new ArrayList<>(collection)` (ну хорошо, до Java 5 дженериков не было). Это не только короче, но и быстрее и опять же создаст меньше мусора в куче. Может быть значительно меньше, так как в большинстве случаев у вас выделится один массив нужного размера, тогда как стрим будет добавлять элементы по одному, растягивая по мере необходимости. Аналогично вместо `stream().collect(toSet())` создаём `new HashSet<>()`, а вместо `stream().collect(toCollection(TreeSet::new))` — `new TreeSet<>()`.

1.3. collection.stream().toArray(String[]::new)

Новый способ преобразования в массив ничем не лучше старого доброго `collection.toArray(new String[0])`. Опять же: так как на пути меньше абстракций, преобразование может оказаться более эффективно. Во всяком случае объект стрима вам не нужен.

1.4. collection.stream().max(Comparator.naturalOrder()).get()

Есть замечательный метод `Collections.max`, который почему-то незаслуженно многими забыт. Вызов `Collections.max(collection)` сделает то же самое и опять же с меньшим количеством мусора. Если у вас свой компаратор, используйте `Collections.max(collection, comparator)`. Метод `Collections.max()` подойдёт хуже, если вы хотите специально обработать пустую коллекцию, тогда стрим более оправдан. Цепочка `collection.stream().max(comparator).orElse(null)` смотрится лучше, чем `collection.isEmpty() ? null : Collections.max(collection, comparator)`.

1.5. collection.stream().count()

Это совсем ни в какие ворота не лезет: есть ведь `collection.size()`! Если в Java 9 `count()` отработает быстро, то в Java 8 этот вызов всегда пересчитывает все элементы, даже если размер очевиден. Не делайте так.

2. Поиск элемента

2.1. stream.filter(condition).findFirst().isPresent()

Такой код вижу на удивление часто. Суть его: проверить, выполняется ли условие для какого-то элемента стрима. Именно для этого есть специальный метод: `stream.anyMatch(condition)`. Зачем вам `Optional`?

2.2. !stream.anyMatch(condition)

Тут некоторые поспорят, но я считаю, что использовать специальный метод `stream.noneMatch(condition)` более выразительно. А вот если и в условии отрицание: `!stream.anyMatch(x -> !condition(x))`, то тут однозначно лучше написать `stream.allMatch(x -> condition(x))`. Тот, кто будет читать код, скажет вам спасибо.

2.3. stream.map(condition).anyMatch(b -> b)

И такой странный код иногда пишут, чтобы запутать коллег. Если увидите такое, знайте, что это просто `stream.anyMatch(condition)`. Здесь же вариации на тему вроде `stream.map(condition).noneMatch(Boolean::booleanValue)` или `stream.map(condition).allMatch(Boolean.TRUE::equals)`.

3. Создание стрима

3.1. Collections.emptyList().stream()

Нужен пустой стрим? Бывает, ничего страшного. И для этого есть специальный метод `Stream.empty()`. Производительность одинаковая, но короче и понятнее. Метод `emptySet` здесь не отличается от `emptyList`.

3.2. Collections.singleton(x).stream()

И тут можно упростить жизнь: если вам потребовался стрим из одного элемента, пишите просто `Stream.of(x)`. Опять же без разницы `singleton` или `singletonList`: когда в стриме один элемент, никого не волнует, упорядочен стрим или нет.

3.3. Arrays.asList(array).stream()

Развитие этой же темы. Люди почему-то так делают, хотя `Arrays.stream(array)` или `Stream.of(array)` отработает не хуже. Если вы указываете элементы явно (`Arrays.asList(x, y, z).stream()`), то `Stream.of(x, y, z)` тоже сработает. Аналогично с `EnumSet.of(x, y, z).stream()`. Вам же стрим нужен, а не коллекция, так и создавайте сразу стрим.

3.4. Collections.nCopies(N, "ignored").stream().map(ignored -> new MyObject())

Нужен стрим из N одинаковых объектов? Тогда `nCopies()` — ваш выбор. А вот если нужно сгенерировать стрим из N объектов, созданных одним и тем же способом, то тут красивее и оптимальнее воспользоваться `Stream.generate(() -> new MyObject()).limit(N)`.

3.5. IntStream.range(from, to).mapToObj(idx -> array[idx])

Нужен стрим из куска массива? Есть специальный метод `Arrays.stream(array, from, to)`. Опять же короче и меньше мусора, плюс так как массив больше не захвачен лямбдой, он не обязан быть effectively-final. Понятно, если from — это 0, а to — это `array.length`, тогда вам просто нужен `Arrays.stream(array)`, причём тут код станет приятнее, даже если в `mapToObj` что-то более сложное. Например, `IntStream.range(0, strings.length).mapToObj(idx -> strings[idx].trim())` легко превращается в `Arrays.stream(strings).map(String::trim)`.

Более хитрая вариация на тему — `IntStream.range(0, Math.min(array.length, max)).mapToObj(idx -> array[idx])`. Немножко подумав, понимаешь, что это `Arrays.stream(array).limit(max)`.

4. Ненужные и сложные коллекторы

Иногда люди изучают коллекторы и всё пытаются делать через них. Однако не всегда они нужны.

4.1. stream.collect(Collectors.counting())

Многие коллекторы нужны только как вторичные в сложных каскадных операциях вроде `groupingBy`. Коллектор `counting()` как раз из них. Пишите `stream.count()` и не мучайтесь. Опять же если в Java 9 `count()` может иногда выполниться за константное время, то коллектор всегда будет пересчитывать элементы. А в Java 8 коллектор `counting()` ещё и боксит зазря (я это исправил в Java 9). Из этой же оперы коллекторы `maxBy()`, `minBy()` (есть методы `max()` и `min()`), `reducing()` (используйте `reduce()`), `mapping()` (просто добавьте шаг `map()`, а затем воспользуйтесь вторичным коллектором напрямую). В Java 9 добавились `filtering()` и `flatMapping()`, которые также дублируют соответствующие промежуточные операции.

4.2. groupingBy(classifier, collectingAndThen(maxBy(comparator), Optional::get))

Частая задача: хочется сгруппировать элементы по классификатору, выбрав в каждой группе максимум. В SQL это выглядит просто `SELECT classifier, MAX(...) FROM ... GROUP BY classifier`. Видимо, пытаясь перенести опыт SQL, люди пытаются использовать тот же самый groupingBy и в Stream API. Казалось бы должно сработать `groupingBy(classifier, maxBy(comparator))`, но нет. Коллектор `maxBy` возвращает `Optional`. Но мы-то знаем, что вложенный `Optional` всегда не пуст, так как в каждой группе по крайней мере один элемент есть. Поэтому приходится добавлять некрасивые шаги вроде `collectingAndThen`, и всё начинает выглядеть совсем чудовищно.

Однако отступив на шаг назад, можно понять, что `groupingBy` тут не нужен. Есть другой замечательный коллектор — `toMap`, и это как раз то что надо. Мы просто хотим собрать элементы в `Map`, где ключом будет классификатор, а значением сам элемент. В случае же дубликата выберем больший из них. Для этого, кстати, есть `BinaryOperator.maxBy(comparator)`, который можно статически импортировать вместо одноимённого коллектора. В результате имеем: `toMap(classifier, identity(), maxBy(comparator))`.

Если вы порываетесь использовать `groupingBy`, а вторичным коллектором у вас `maxBy`, `minBy` или `reducing` (возможно, с промежуточным `mapping`), посмотрите в сторону коллектора `toMap` — может полегчать.

5. Не считайте то, что не нужно считать

5.1. listOfLists.stream().flatMap(List::stream).count()

Это перекликается с пунктом 1.5. Мы хотим посчитать суммарное число элементов во вложенных коллекциях. Казалось бы всё логично: растянем эти коллекции в один стрим с помощью `flatMap` и пересчитаем. Однако в большинстве случаев размеры вложенных списков уже посчитаны, хранятся у них в поле и легко доступны с помощью метода `size()`. Небольшая модификация существенно увеличит скорость операции: `listOfLists.stream().mapToInt(List::size).sum()`. Если боитесь, что `int` переполнится, `mapToLong` тоже сработает.

5.2. if(stream.filter(condition).count() > 0)

Опять же забавный способ записать `stream.anyMatch(condition)`. Но в отличие от довольно безобидного 2.1 вы тут теряете короткое замыкание: будут перебраны все элементы, даже если условие сработало на самом первом. Аналогично если вы проверяете `filter(condition).count() == 0`, лучше воспользоваться `noneMatch(condition)`.

5.3. if(stream.count() > 2)

Этот случай более хитрый. Вам теперь важно знать, больше двух элементов в стриме или нет. Если вас волнует производительность, возможно, стоит вставить `stream.limit(3).count()`. Вам ведь не важно, сколько их, если их больше двух.

6. Разное

6.1. stream.sorted(comparator).findFirst()

Что хотел сказать автор? Отсортируй стрим и возьми первый элемент. Это же всё равно что взять минимальный элемент: `stream.min(comparator)`. Иногда видишь даже `stream.sorted(comparator.reversed()).findFirst()`, что аналогично `stream.max(comparator)`. Реализация Stream API не соптимизирует тут (хотя могла бы), а сделает всё как вы сказали: соберёт стрим в промежуточный массив, отсортирует его весь и выдаст вам первый элемент. Вы существенно потеряете в памяти и скорости на такой операции. Ну и, конечно, замена существенно понятнее.

6.2. stream.map(x -> {counter.addAndGet(x);return x;})

Некоторые люди пытаются выполнить какой-нибудь побочный эффект в стриме. Вообще это в принципе уже звоночек, что может стрим вам вовсе и не нужен. Но так или иначе для этих целей есть специальный метод `peek`. Пишем `stream.peek(counter::addAndGet)`.

На этом у меня всё. Если вы сталкивались со странными и неэффективными способами использования Stream API, напишите о них в комментариях.

Метки: stream api, оптимизация, java 8, java 9, forEach, редукция, коллекции