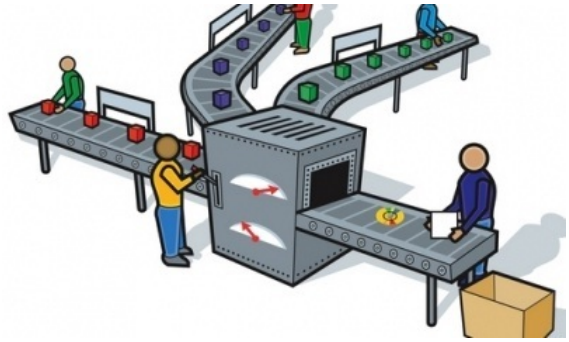
vedenin1980 18 ноября 2015 в 16:43

Шпаргалка Java программиста 4. Java Stream API

Блог компании Luxoft, Разработка веб-сайтов, Программирование, Java, Функциональное программирование

Tutorial



Несмотря на то, что Java 8 вышла уже достаточно давно, далеко не все программисты используют её новые возможности, кого-то останавливает то, что рабочие проекты слишком сложно перевести с Java 7 или даже Java 6, кого-то использование в своих проектах GWT, кто-то делает проекты под Android и не хочет или не может использовать сторонние библиотеки для реализации лямбд и Stream Api. Однако знание лямбд и Stream Api для программиста Java зачастую требуют на собеседованиях, ну и просто будет полезно при переходе на проект где используется Java 8. Я хотел бы предложить вам краткую шпаргалку по Stream Api с практическими примерами реализации различных задач с новым функциональным подходом. Знания лямбд и функционального программирования не потребуется (я постарался дать примеры так, чтобы все было понятно), уровень от самого базового знания Java и выше.

Также, так как это шпаргалка, статья может использоваться, чтобы быстро вспомнить как работает та или иная особенность Java Stream Api. Краткое перечисление возможностей основных функций дано в начале статьи.

Для тех кто совсем не знает что такое Stream Api

Общее оглавление 'Шпаргалок'

Давайте начнем с начала, а именно с создания объектов stream в Java 8.

I. Способы создания стримов

Перечислим несколько способов создать стрим

Способ создания стрима	Шаблон создания	Пример
1. Классический: Создание стрима из коллекции	collection.**stream**()	`Collection<String> collection = Arrays.asList("a1", "a2", "a3");` `Stream<String> streamFromCollection = collection.stream();`

2. Создание стрима из значений	**Stream.of**(*значение1,... значениеN*)	`Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");`
3. Создание стрима из массива	**Arrays.stream**(*массив*)	`String[] array = {"a1", "a2","a3"}; Stream<String> streamFromArrays = Arrays.stream(array);`
4. Создание стрима из файла (каждая строка в файле будет отдельным элементом в стриме)	**Files.lines**(*путь_к_файлу*)	`Stream<String> streamFromFiles = Files.lines(Paths.get("file.txt"))`
5. Создание стрима из строки	«строка».**chars**()	`IntStream streamFromString = "123".chars()`
6. С помощью Stream.builder	Stream.**builder**().**add**(...)....**build**()	`Stream.builder().add("a1").add("a2").add("a3").build()`
7. Создание параллельного стрима	collection.**parallelStream**()	`Stream<String> stream = collection.parallelStream();`
8. Создание бесконечных стрима с помощью Stream.iterate	**Stream.iterate**(*начальное_условие, выражение_генерации*)	`Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 1)`
9. Создание бесконечных стрима с помощью Stream.generate	**Stream.generate**(*выражение_генерации*)	`Stream<String> streamFromGenerate = Stream.generate(() -> "a1")`

В принципе, кроме последних двух способов создания стрима, все не отличается от обычных способов создания коллекций. Последние два способа служат для генерации бесконечных стримов, в iterate задается начальное условие и выражение получение следующего значения из предыдущего, то есть Stream.iterate(1, n -> n + 1) будет выдавать значения 1, 2, 3, 4,... N. Stream.generate служит для генерации константных и случайных значений, он просто выдает значения соответствующие выражению, в данном примере, он будет выдавать бесконечное количество значений «a1».

Для тех кто не знает лямбды

Более подробные примеры

II. Методы работы со стримами

Java Stream API предлагает два вида методов:
1. Конвейерные — возвращают другой stream, то есть работают как builder,
2. Терминальные — возвращают другой объект, такой как коллекция, примитивы, объекты, Optional и т.д.

О том чем отличаются конвейерные и терминальные методы

2.1 Краткое описание конвейерных методов работы со стримами

Метод stream	Описание	Пример
filter	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(«a1»::equals).count()
skip	Позволяет пропустить N первых элементов	collection.stream().skip(collection.size() — 1).findFirst().orElse(«1»)
distinct	Возвращает стрим без дубликатов (для метода equals)	collection.stream().distinct().collect(Collectors.toList())
map	Преобразует каждый элемент стрима	collection.stream().map((s) -> s + "_1").collect(Collectors.toList())
peek	Возвращает тот же стрим, но применяет функцию к каждому элементу стрима	collection.stream().map(String::toUpperCase).peek((e) -> System.out.print("," + e)). collect(Collectors.toList())
limit	Позволяет ограничить выборку определенным количеством первых элементов	collection.stream().limit(2).collect(Collectors.toList())
sorted	Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator	collection.stream().sorted().collect(Collectors.toList())
mapToInt, **mapToDouble**, **mapToLong**	Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)	collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()
flatMap, **flatMapToInt**, **flatMapToDouble**, **flatMapToLong**	Похоже на map, но может создавать из одного элемента несколько	collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new)

2.2 Краткое описание терминальных методов работы со стримами

Метод stream	Описание	Пример
findFirst	Возвращает первый элемент из стрима (возвращает Optional)	collection.stream().findFirst().orElse(«1»)
findAny	Возвращает любой подходящий элемент из стрима (возвращает Optional)	collection.stream().findAny().orElse(«1»)
collect	Представление результатов в виде коллекций и других структур данных	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())
count	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()
anyMatch	Возвращает true, если условие выполняется хотя бы для одного элемента	collection.stream().anyMatch(«a1»::equals)
noneMatch	Возвращает true, если условие не	collection.stream().noneMatch(«a8»::equals)

	выполняется ни для одного элемента	
allMatch	Возвращает true, если условие выполняется для всех элементов	collection.stream().allMatch((s) -> s.contains(«1»))
min	Возвращает минимальный элемент, в качестве условия использует компаратор	collection.stream().min(String::compareTo).get()
max	Возвращает максимальный элемент, в качестве условия использует компаратор	collection.stream().max(String::compareTo).get()
forEach	Применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется	set.stream().forEach((p) -> p.append("_1"));
forEachOrdered	Применяет функцию к каждому объекту стрима, сохранение порядка элементов гарантирует	list.stream().forEachOrdered((p) -> p.append("_new"));
toArray	Возвращает массив значений стрима	collection.stream().map(String::toUpperCase).toArray(String[]::new);
reduce	Позволяет выполнять агрегатные функции на всей коллекцией и возвращать один результат	collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)

Обратите внимание методы findFirst, findAny, anyMatch это short-circuiting методы, то есть обход стримов организуется таким образом чтобы найти подходящий элемент максимально быстро, а не обходить весь изначальный стрим.

2.3 Краткое описание дополнительных методов у числовых стримов

Метод stream	Описание	Пример
sum	Возвращает сумму всех чисел	collection.stream().mapToInt((s) -> Integer.parseInt(s)).sum()
average	Возвращает среднее арифметическое всех чисел	collection.stream().mapToInt((s) -> Integer.parseInt(s)).average()
mapToObj	Преобразует числовой стрим обратно в объектный	intStream.mapToObj((id) -> new Key(id)).toArray()

2.4 Несколько других полезных методов стримов

Метод stream	Описание
isParallel	Узнать является ли стрим параллельным
parallel	Вернуть параллельный стрим, если стрим уже параллельный, то может вернуть самого себя
sequential	Вернуть последовательный стрим, если стрим уже последовательный, то может вернуть самого себя

С помощью, методов parallel и sequential можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот, то есть:

```
collection.stream().
peek(...). // операция последовательна
parallel().
map(...). // операция может выполняться параллельно,
sequential().
reduce(...) // операция снова последовательна
```

Внимание: крайне не рекомендуется использовать параллельные стримы для сколько-нибудь долгих операций (получение данных из базы, сетевых соединений), так как все параллельные стримы работают с одним пулом fork/join и такие долгие операции могут остановить работу всех параллельных стримов в JVM из-за того отсутствия доступных потоков в пуле, т.е. параллельные стримы стоит использовать лишь для коротких операций, где счет идет на миллисекунды, но не для тех где счет может идти на секунды и минуты.

III. Примеры работы с методами стримов

Рассмотрим работу с методами на различных задачах, обычно требующихся при работе с коллекциями.

3.1 Примеры использования filter, findFirst, findAny, skip, limit и count

Условие: дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), давайте посмотрим как её можно обрабатывать используя методы filter, findFirst, findAny, skip и count:

Задача	Код примера	Результат
Вернуть количество вхождений объекта «a1»	collection.stream().filter(«a1»::equals).count()	2
Вернуть первый элемент коллекции или 0, если коллекция пуста	collection.stream().findFirst().orElse(«0»)	a1
Вернуть последний элемент коллекции или «empty», если коллекция пуста	collection.stream().skip(collection.size() — 1).findAny().orElse(«empty»)	a1
Найти элемент в коллекции равный «a3» или кинуть ошибку	collection.stream().filter(«a3»::equals).findFirst().get()	a3
Вернуть третий элемент коллекции по порядку	collection.stream().skip(2).findFirst().get()	a3
Вернуть два элемента начиная со второго	collection.stream().skip(1).limit(2).toArray()	[a2, a3]
Выбрать все элементы по шаблону	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())	[a1, a1]

Обратите внимание, что методы findFirst и findAny возвращают новый тип Optional, появившийся в Java 8, для того чтобы избежать NullPointerException. Метод filter удобно использовать для выборки лишь определенного множества значений, а метод skip позволяет пропускать определенное количество элементов.

Если вы не знаете лямбды

Условие: дана коллекция класс People (с полями name — имя, age — возраст, sex — пол), вида Arrays.asList(new People(«Вася», 16, Sex.MAN), new People(«Петя», 23, Sex.MAN), new People(«Елена», 42, Sex.WOMEN), new People(«Иван Иванович», 69, Sex.MAN)). Давайте посмотрим примеры как работать с таким классом:

Задача	Код примера	Результат
Выбрать мужчин-военнообязанных (от 18 до 27 лет)	peoples.stream().filter((p)-> p.getAge() >= 18 && p.getAge() < 27 && p.getSex() == Sex.MAN).collect(Collectors.toList())	[{name='Петя', age=23, sex=MAN}]

Найти средний возраст среди мужчин	peoples.stream().filter((p) -> p.getSex() == Sex.MAN). mapToInt(People::getAge).average().getAsDouble()	36.0
Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 55 лет, а мужчина в 60)	peoples.stream().filter((p) -> p.getAge() >= 18).filter((p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55) || (p.getSex() == Sex.MAN && p.getAge() < 60)).count()	2

Детальные примеры

3.2 Примеры использования distinct

Метод distinct возвращает stream без дубликатов, при этом для упорядоченного стрима (например, коллекция на основе list) порядок стабилен, для неупорядоченного — порядок не гарантируется. Рассмотрим результаты работы над коллекцией Collection ordered = Arrays.asList(«a1», «a2», «a2», «a3», «a1», «a2», «a2») и Collection nonOrdered = new HashSet<>(ordered).

Задача	Код примера	Результат
Получение коллекции без дубликатов из неупорядоченного стрима	nonOrdered.stream().distinct().collect(Collectors.toList())	[a1, a2, a3] — порядок не гарантируется
Получение коллекции без дубликатов из упорядоченного стрима	ordered.stream().distinct().collect(Collectors.toList());	[a1, a2, a3] — порядок гарантируется

Обратите внимание:
1. Если вы используете distinct с классом, у которого переопределен equals, обязательно так же корректно переопределить hashCode в соответствие с контрактом equals/hashCode (самое главное чтобы hashCode для всех equals объектов, возвращал одинаковое значение), иначе distinct может не удалить дубликаты (аналогично, как при использовании HashSet/HashMap),
2. Если вы используете параллельные стримы и вам не важен порядок элементов после удаления дубликатов — намного лучше для производительности сделать сначала стрим неупорядоченным с помощь unordered(), а уже потом применять distinct(), так как подержание стабильности сортировки при параллельном стриме довольно затратно по ресурсам и distinct() на упорядоченным стриме будет выполнятся значительно дольше чем при неупорядоченном,

Детальные примеры

3.3 Примеры использования Match функций (anyMatch, allMatch, noneMatch)

Условие: дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), давайте посмотрим, как её можно обрабатывать используя Match функции

Задача	Код примера	Результат
Найти существуют ли хоть один «a1» элемент в коллекции	collection.stream().anyMatch(«a1»::equals)	true
Найти существуют ли хоть один «a8» элемент в коллекции	collection.stream().anyMatch(«a8»::equals)	false
Найти есть ли символ «1» у всех элементов коллекции	collection.stream().allMatch((s) -> s.contains(«1»))	false
Проверить что не существуют ни одного «a7» элемента в коллекции	collection.stream().noneMatch(«a7»::equals)	true

Детальные примеры

3.4 Примеры использования Map функций (map, mapToInt, FlatMap, FlatMapToInt)

Условие: даны две коллекции collection1 = Arrays.asList(«a1», «a2», «a3», «a1») и collection2 = Arrays.asList(«1,2,0», «4,5»), давайте посмотрим как её можно обрабатывать используя различные map функции

Задача	Код примера	Результат
Добавить "_1" к каждому элементу первой коллекции	collection1.stream().map((s) -> s + "_1").collect(Collectors.toList())	[a1_1, a2_1, a3_1, a1_1]
В первой коллекции убрать первый символ и вернуть массив чисел (int[])	collection1.stream().mapToInt((s) -> Integer.parseInt(s.substring(1))).toArray()	[1, 2, 3, 1]
Из второй коллекции получить все числа, перечисленные через запятую из всех элементов	collection2.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::new)	[1, 2, 0, 4, 5]
Из второй коллекции получить сумму всех чисел, перечисленных через запятую	collection2.stream().flatMapToInt((p) -> Arrays.asList(p.split(",")).stream().mapToInt(Integer::parseInt)).sum()	12

Обратите внимание: все map функции могут вернуть объект другого типа (класса), то есть map может работать со стримом строк, а на выходе дать Stream из значений Integer или получать класс людей People, а возвращать класс Office, где эти люди работают и т.п., flatMap (flatMapToInt и т.п.) на выходе должны возвращать стрим с одним, несколькими или ни одним элементов для каждого элемента входящего стрима (см. последние два примера).

Детальные примеры

3.5 Примеры использования Sorted функции

Условие: даны две коллекции коллекция строк Arrays.asList(«a1», «a4», «a3», «a2», «a1», «a4») и коллекция людей класса People (с полями name — имя, age — возраст, sex — пол), вида Arrays.asList(new People(«Вася», 16, Sex.MAN), new People(«Петя», 23, Sex.MAN), new People(«Елена», 42, Sex.WOMEN), new People(«Иван Иванович», 69, Sex.MAN)). Давайте посмотрим примеры как их можно сортировать:

Задача	Код примера	Результат
Отсортировать коллекцию строк по алфавиту	collection.stream().sorted().collect(Collectors.toList())	[a1, a1, a2, a3, a4, a4]
Отсортировать коллекцию строк по алфавиту в обратном порядке	collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).collect(Collectors.toList())	[a4, a4, a3, a2, a1, a1]
Отсортировать коллекцию строк по алфавиту и убрать дубликаты	collection.stream().sorted().distinct().collect(Collectors.toList())	[a1, a2, a3, a4]
Отсортировать коллекцию строк по алфавиту в обратном порядке и убрать дубликаты	collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).distinct().collect(Collectors.toList())	[a4, a3, a2, a1]
Отсортировать коллекцию людей по имени в обратном алфавитном порядке	peoples.stream().sorted((o1,o2) -> -o1.getName().compareTo(o2.getName())).collect(Collectors.toList())	[{'Петя'}, {'Иван Иванович'}, {'Елена'}, {'Вася'}]
Отсортировать коллекцию людей сначала по полу, а потом по возрасту	peoples.stream().sorted((o1, o2) -> o1.getSex() != o2.getSex()? o1.getSex().	[{'Вася'}, {'Петя'}, {'Иван

	compareTo(o2.getSex()): o1.getAge().compareTo(o2.getAge())).collect(Collectors.toList())	Иванович'}, {'Елена'}]

Детальные примеры

3.6 Примеры использования Max и Min функций

Условие: дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), и коллекция класса Peoples из прошлых примеров про Sorted и Filter функции.

Задача	Код примера	Результат
Найти максимальное значение среди коллекции строк	collection.stream().max(String::compareTo).get()	a3
Найти минимальное значение среди коллекции строк	collection.stream().min(String::compareTo).get()	a1
Найдем человека с максимальным возрастом	peoples.stream().max((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get()	{name='Иван Иванович', age=69, sex=MAN}
Найдем человека с минимальным возрастом	peoples.stream().min((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get()	{name='Вася', age=16, sex=MAN}

Детальные примеры

3.7 Примеры использования ForEach и Peek функций

Обе ForEach и Peek по сути делают одно и тоже, меняют свойства объектов в стриме, единственная разница между ними в том что ForEach терминальная и она заканчивает работу со стримом, в то время как Peek конвейерная и работа со стримом продолжается. Например, есть коллекция:

```
        Collection<StringBuilder> list = Arrays.asList(new StringBuilder("a1"), new StringBuilder("a2"), new StringBuil
der("a3"));
```

И нужно добавить к каждому элементу "_new", то для ForEach код будет

```
        list.stream().forEachOrdered((p) -> p.append("_new")); // list - содержит [a1_new, a2_new, a3_new]
```

а для peek код будет

```
   List<StringBuilder> newList = list.stream().peek((p) -> p.append("_new")).collect(Collectors.toList()); // u list u n
ewList содержат [a1_new, a2_new, a3_new]
```

Детальные примеры

3.8 Примеры использования Reduce функции

Метод reduce позволяет выполнять агрегатные функции на всей коллекцией (такие как сумма, нахождение минимального или максимального значение и т.п.), он возвращает одно значение для стрима, функция получает два аргумента — значение полученное на прошлых шагах и текущее значение.

Условие: Дана коллекция чисел Arrays.asList(1, 2, 3, 4, 2) выполним над ними несколько действий используя reduce.

Задача	Код примера	Результат
Получить сумму чисел или вернуть 0	collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)	12
Вернуть максимум или -1	collection.stream().reduce(Integer::max).orElse(-1)	4
Вернуть сумму нечетных чисел или 0	collection.stream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0)	4

Детальные примеры

3.9 Примеры использования toArray и collect функции

Если с toArray все просто, можно либо вызвать toArray() получить Object[], либо toArray(T[]::new) — получив массив типа T, то collect позволяет много возможностей преобразовать значение в коллекцию, map'у или любой другой тип. Для этого используются статические методы из Collectors, например преобразование в List будет stream.collect(Collectors.toList()).

Давайте рассмотрим статические методы из Collectors:

Метод	Описание
toList, toCollection, toSet	представляют стрим в виде списка, коллекции или множества
toConcurrentMap, toMap	позволяют преобразовать стрим в map
averagingInt, averagingDouble, averagingLong	возвращают среднее значение
summingInt, summingDouble, summingLong	возвращает сумму
summarizingInt, summarizingDouble, summarizingLong	возвращают SummaryStatistics с разными агрегатными значениями
partitioningBy	разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>
groupingBy	разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>
mapping	дополнительные преобразования значений для сложных Collector'ов

Теперь давайте рассмотрим работу с collect и toArray на примерах:
Условие: Дана коллекция чисел Arrays.asList(1, 2, 3, 4), рассмотрим работу collect и toArray с ней

Задача	Код примера	Результат
Получить сумму нечетных чисел	numbers.stream().collect(Collectors.summingInt(((p) -> p % 2 == 1? p: 0)))	4

Вычесть от каждого элемента 1 и получить среднее	numbers.stream().collect(Collectors.averagingInt((p) -> p — 1))	1.5
Прибавить к числам 3 и получить статистику	numbers.stream().collect(Collectors.summarizingInt((p) -> p + 3))	IntSummaryStatistics{count=4, sum=22, min=4, average=5.5, max=7}
Разделить числа на четные и нечетные	numbers.stream().collect(Collectors.partitioningBy((p) -> p % 2 == 0))	{false=[1, 3], true=[2, 4]}

Условие: Дана коллекция строк Arrays.asList(«a1», «b2», «c3», «a1»), рассмотрим работу collect и toArray с ней

Задача	Код примера	Результат
Получение списка без дубликатов	strings.stream().distinct().collect(Collectors.toList())	[a1, b2, c3]
Получить массив строк без дубликатов и в верхнем регистре	strings.stream().distinct().map(String::toUpperCase).toArray(String[]::new)	{A1, B2, C3}
Объединить все элементы в одну строку через разделитель: и обернуть тегами …	strings.stream().collect(Collectors.joining(": ", " ", " "))	a1: b2: c3: a1
Преобразовать в map, где первый символ ключ, второй символ значение	strings.stream().distinct().collect(Collectors.toMap((p) -> p.substring(0, 1), (p) -> p.substring(1, 2)))	{a=1, b=2, c=3}
Преобразовать в map, сгруппировав по первому символу строки	strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1)))	{a=[a1, a1], b=[b2], c=[c3]}
Преобразовать в map, сгруппировав по первому символу строки и объединим вторые символы через :	strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1), Collectors.mapping((p) -> p.substring(1, 2), Collectors.joining(":"))))	{a=1:1, b=2, c=3}

Детальные примеры

3.10 Пример создания собственного Collector'a

Кроме Collector'ов уже определенных в Collectors можно так же создать собственный Collector, Давайте рассмотрим пример как его можно создать.

Метод определения пользовательского Collector'a:

```
Collector<Тип_источника, Тип_аккумулятора, Тип_результата> collector =  Collector.of(
                метод_инициализации_аккумулятора,
                метод_обработки_каждого_элемента,
                метод_соединения_двух_аккумуляторов,
                [метод_последней_обработки_аккумулятора]
        );
```

Как видно из кода выше, для реализации своего Collector'a нужно определить три или четыре метода (метод_последней_обработки_аккумулятора не обязателен). Рассмотрим следующий кода, который мы писали до Java 8, чтобы объединить все строки коллекции:

```
        StringBuilder b = new StringBuilder(); // метод_инициализации_аккумулятора
        for(String s: strings) {
            b.append(s).append(" , "); // метод_обработки_каждого_элемента,
        }
        String joinBuilderOld = b.toString(); // метод_последней_обработки_аккумулятора
```

И аналогичный код, который будет написан в Java 8

```
String joinBuilder = strings.stream().collect(
   Collector.of(
                StringBuilder::new, // метод_инициализации_аккумулятора
                (b ,s) -> b.append(s).append(" , "), // метод_обработки_каждого_элемента,
                (b1, b2) -> b1.append(b2).append(" , "), // метод_соединения_двух_аккумуляторов
                StringBuilder::toString // метод_последней_обработки_аккумулятора
        )
);
```

В общем-то, три метода легко понять из кода выше, их мы писали практически при каждой обработки коллекций, но вот что такое метод_соединения_двух_аккумуляторов? Это метод который нужен для параллельной обработки Collector'a, в данном случае при параллельном стриме коллекция может быть разделенной на две части (или больше частей), в каждой из которых будет свой аккумулятор StringBuilder и потом необходимо будет их объединить, то код до Java 8 при 2 потоках будет таким:

```
        StringBuilder b1 = new StringBuilder(); // метод_инициализации_аккумулятора_1
        for(String s: stringsPart1) { //  stringsPart1 - первая часть коллекции strings
            b1.append(s).append(" , "); // метод_обработки_каждого_элемента,
        }

        StringBuilder b2 = new StringBuilder(); // метод_инициализации_аккумулятора_2
        for(String s: stringsPart2) { //  stringsPart2 - вторая часть коллекции strings
            b2.append(s).append(" , "); // метод_обработки_каждого_элемента,
        }

        StringBuilder b = b1.append(b2).append(" , "), // метод_соединения_двух_аккумуляторов

        String joinBuilderOld = b.toString(); // метод_последней_обработки_аккумулятора
```

Напишем свой аналог Collectors.toList() для работы со строковым стримом:

```
 // Напишем свой аналог toList
        Collector<String, List<String>, List<String>> toList = Collector.of(
                ArrayList::new, // метод инициализации аккумулятора
                List::add, // метод обработки каждого элемента
                (l1, l2) -> { l1.addAll(l2); return l1; } // метод соединения двух аккумуляторов при параллельном выпол
нении
        );
// Используем его для получение списка строк без дубликатов из стрима
        List<String> distinct1 = strings.stream().distinct().collect(toList);
```

Детальные примеры

IV. Заключение

Вот и все. Надеюсь, моя небольшая шпаргалка по работе со stream api была для вас полезной. Все исходники есть на github'e, удачи в написании хорошего кода.

P.S. Список других статей, где можно прочитать дополнительно про Stream Api:

1. Processing Data with Java SE 8 Streams, Part 1 от Oracle,
2. Processing Data with Java SE 8 Streams, Part 2 от Oracle,
3. Полное руководство по Java 8 Stream

P.P.S. Так же советую посмотреть мой opensource проект useful-java-links — возможно, наиболее полная коллекция полезных Java библиотек, фреймворков и русскоязычного обучающего видео. Так же есть аналогичная английская версия этого проекта и начинаю opensource подпроект Hello world по подготовке коллекции простых примеров для разных Java библиотек в одном maven проекте (буду благодарен за любую помощь).

Общее оглавление 'Шпаргалок'

Только зарегистрированные пользователи могут участвовать в опросе. Войдите, пожалуйста.

Используете ли вы уже Stream Api?

37,5%	Да, уже в production	385
16,8%	Да, но только в личных проектах	172
8,7%	Очень ограничено	89
8,5%	Лишь планируем использовать	87
3,2%	Нет, предпочитаю старый подход	33
18,7%	Нет, но хотел бы	192
1,8%	Нет, и мне все равно использовать или нет	18
4,9%	Я вообще не пишу на Java	50

Проголосовали 1026 пользователей. Воздержались 149 пользователей.

Теги: java, stream, stream api, лямбы, функциональное программирование, многопоточность, java 8

Хабы: Блог компании Luxoft, Разработка веб-сайтов, Программирование, Java, Функциональное программирование

+25 1156 637k 20 Поделиться

Slava Vedenin @vedenin1980
Java developer

Luxoft
think. create. accelerate.

Сайт

ПОХОЖИЕ ПУБЛИКАЦИИ

14 апреля 2016 в 21:35

Шпаргалка Java программиста 7.1 Типовые задачи: Оптимальный путь преобразования InputStream в строку

+21 74,3k 451 35

10 декабря 2015 в 14:38

Шпаргалка Java-программиста 5. Двести пятьдесят русскоязычных обучающих видео докладов и лекций о Java

+29 149k 1499 27

27 октября 2015 в 23:03

Шпаргалка Java программиста 3. Коллекции в Java (стандартные, guava, apache, trove, gs-collections и другие)

+57 206k 1334 40

Комментарии 20

burjui 19 ноября 2015 в 00:10 +2

Проголосовал за production, но использую не Java 8 Stream, а библиотеку totallylazy (https://github.com/bodar/totallylazy) в сочетании с gradle-retrolambda, т.к. пишу под Android. Помимо Stream, которые там называются Sequence, там ещё много полезностей — в том числе, Option и Either.

NonGrate 19 ноября 2015 в 14:55 +1

Я пользуюсь Lightweight-Stream-API для андроида. Вы пользовались им? Если да, не могли бы вы описать, чем totallylazy отличается от урезанного Stream API?

Я первый раз слышу о totallylazy, раньше думал, что Lightweight-Stream-API единственная библиотека. К тому же, в репозитории написано, что это то же Stream API из джавы, только переписанное, чтобы работало на семёрке. Что может быть лучше, чем код из самой джавы.

burjui 19 ноября 2015 в 16:49 +1

Нет, не пользовался, но, судя по примерам кода, работа с «потоками» очень похожа. Но totallylazy ориентирована на функциональное программирование вообще, а Lightweight-Stream-API — именно на «потоки». Поэтому в Lightweight-Stream-API вы не найдёте всяких функторов, монад и т.п.

NonGrate 19 ноября 2015 в 17:12 +1

Подскажите, пожалуйста, какие-нибудь туториалы по totallylazy, если знаете? Конкретнее, именно по функциональному программированию. Можно ли с помощью totallylazy сохранять функцию в переменную? Обязательно посмотрю, что эта ленивая библиотека из себя представляет.

Спасибо!

burjui 19 ноября 2015 в 18:26 +1

По функциональному программированию вообще я бы рекомендовал бесплатную книгу «Learn You a Haskell for Great Good!» — http://learnyouahaskell.com/chapters
Первые несколько глав дают общее представление о ФП, дальше начинается самый сок и нарастает хардкорность. Книга забавно иллюстрирована и вообще написана интересно и с юмором. Сразу скажу, после долгого использования императивного стиля функциональный даётся не сразу, мешает инертность мышления. Не жалейте времени, перечитайте непонятные вещи несколько раз, если нужно. Как писал автор одной книги по теории чисел: «Если вы читаете больше одной страницы в час, возможно, вы слишком спешите». Поищите туториалы на Youtube. Разнообразнее источники информации только улучшат понимание темы.

Когда придёт понимание концепции ФП, для вас уже не будет проблемой использовать ту или иную библиотеку для этого — вы уже будете знать, что в ней искать и зачем. А по totallylazy я сам искал, но не нашёл — даже полноценных доков, не то что туториалов. Мои задачи она решает.

Что касается сохранения функции в переменную, вам это позволяет и Java 8 с её Lambda, functional interfaces и method

references. Пример для Android:

```
final DialogInterface.OnClickListener clickListener = (dialog, which) -> {
    switch (which) {
        ...
    }
};

final AlertDialog.Builder builder = new AlertDialog.Builder(context);
builder.setPositiveButton(R.string.yes, clickListener);
builder.setNegativeButton(R.string.no, clickListener);
```

или вот ещё:

```
class Example {
    static void doSomething() {
        ...
    }
}

final Runnable action = Example::doSomething;
```

Работает это потому, OnClickListener — интерфейс с одним методом, или «функциональный» интерфейс. Разумеется, Java в этом плане и рядом с Haskell не валялась, т.к. в последнем не нужны всякие специальные интерфейсы, и можно просто указать сигнатуру функции на месте типа, или вообще ничего не указывать, положившись на автоматическое выведение типов.

orthanner 19 ноября 2015 в 06:01 +1

Ну Option и здесь есть (aka Optional), но лично мне не шибко нравится то, как это реализовано, в принципе. Если уж сделали в интерфейсах реализации по умолчанию, так можно было и стандартный Collections API доработать (для ленивых — интерфейс Iterator). Нет же, породили новую сущность. И всё равно реализация по умолчанию не является потокобезопасной. Казалось бы — зачем? и тут мы вспоминаем, что Java Collections API мутабельно чуть менее, чем полностью (а меньшая часть просто не даёт себя модифицировать).
И да, Either здесь нет, но реализуется достаточно просто (правда, без sealed traits придётся повозиться, чтобы защитить иерархию классов от нежелательного расширения).

NCNecros 19 ноября 2015 в 10:33 +1

Насчет distinct(), мне кажется вы ошиблись. Я думаю «For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.» означает что distinct() гарантировано работает только если список отсортирован. На синтетических примерах с примитивами это не имеет значения, но я столкнулся с проблемой когда у меня список содержал кучу объектов с переопределенным equals(), но нереализованым компаратором для сортировки. Применение distinct() к такому списку не давала ничего. Из нескольких миллионов записей ни одна не убиралась.

vedenin1980 19 ноября 2015 в 11:40 +1

Спасибо за замечание, но мне кажется в фразе «For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.» речь идет о таком свойстве сортировки как «Устойчивость (англ. stability) — устойчивая сортировка не меняет взаимного расположения элементов с одинаковыми ключами» вики.

Это подтверждает и дальнейшее замечание в javadoc:
API Note:
Preserving **stability** for distinct() in parallel pipelines is relatively expensive (requires that the operation act as a full barrier, with substantial buffering overhead), and **stability** is often not needed. Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with BaseStream.unordered() may result in significantly more efficient execution for distinct() in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with distinct() in parallel pipelines, switching to sequential execution with BaseStream.sequential() may improve performance.

Очень странно что у вас distinct не работал, а можете дать пример вашего класса и несколько значений в личку?

vedenin1980 19 ноября 2015 в 11:53 +1

Да, у вас hashCode тоже переопределен в соответствии с контрактами (в частности, что все equals объекты всегда имеют один и тот же hashCode)? Так как если я правильно понимаю работу distinct он работает по принципу HashSet, то есть сначала сортирует по hashCode, а уже потом среди объектов с одинаковым hashCode начинает проверять на equals, соответственно при нарушении контракта hashCode/equals ничего работать не будет.

NCNecros 19 ноября 2015 в 14:02 +1

Правда ваша. Я вышел из этой ситуации обернув класс в обертку перед distinct(), а обертка как раз добавляет equals и hashcode. Косяк конечно мой, но этот камень можете указать где-нибудь в примечаниях, что distict() работает на классах при наличии hashcode и equals. Хорошая у Вас статья, stream() очень приятная штука.

vedenin1980 19 ноября 2015 в 14:36 +1

Ок, добавил примечание к distinct

lany 17 декабря 2015 в 15:41 +2

А почему reduce() не в терминальных операциях?

vedenin1980 17 декабря 2015 в 16:20 +1

Спасибо, поправил. Изначально я вообще забыл добавить recude в таблицу, а когда обнаружил вставил не в ту таблицу.

MaximChistov 26 апреля 2016 в 18:40 0

Важный момент:
allMatch(условие) вернет true для пустого стрима!
а anyMatch(то же условие) для пустого стрима вернет false!

dougrinch 26 апреля 2016 в 20:06 0

Вообще, было бы странно, если бы оно работало иначе.

НЛО прилетело и опубликовало эту надпись здесь

vedenin1980 20 октября 2016 в 11:56 0

Только в одном месте, где пример про список строк, действительно должно быть orElse(«0»). Поправил

vedenin1980 26 декабря 2017 в 12:28 0

Добавил новую часть шпаргалок — шпаргалка по Java SE 8

softwind 22 октября 2018 в 16:51 0

Наткнулся на фичу, которая может создать неочевидную проблему...
В общем Stream.of(...).peek(s->System.out.println(«PEEK:S»)).X
В случае X=.forEach(System.out.println(«FOR:S»)) выдаст PEEK:S\nFOR:S...
А в случае X=.count() — ничего не выдаст, что неочевидно, так как ожидается PEEK:S...

artelektrik 27 июня 2019 в 14:31 0

Статья просто супер! Спасибо громадное!!!

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Пользователь нашел простой способ просмотра видеороликов на YouTube без рекламы, а также онлайн-статей без подписки

+40 32,9k 21 93

Отличия мужского мозга от женского: женщины кодят лучше?

+52 23,6k 78 161

Принимаем и анализируем радиосигнал платежного терминала с помощью SDR

+25 10,3k 32 52

Илон Маск: «Лидар это потеря времени. Все, кто полагаются на лидар, обречены»

+19 34,2k 40 122

Lynwood Investments пытается помешать Nginx зарегистрировать в США товарный знак NGINX APP PROTECT

+27 8k 6 57

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.