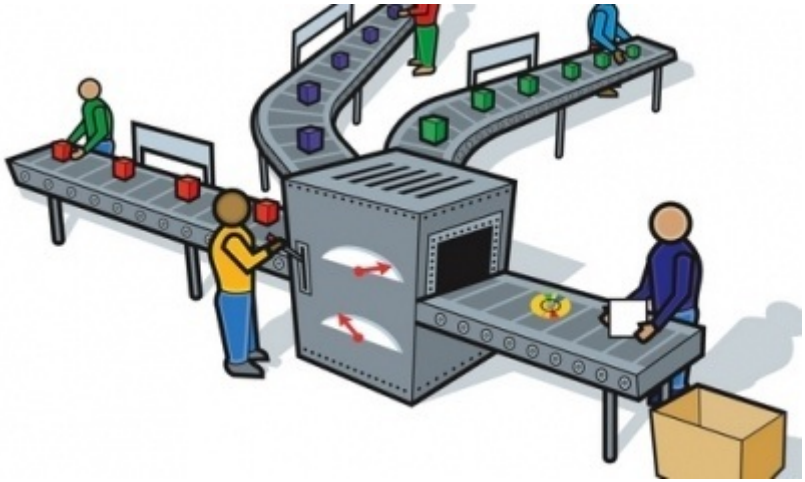
vedenin1980 18 ноября 2015 в 16:43

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

Stream API это новый способ работать со структурами данных в функциональном стиле. Чаще всего с помощью stream в Java 8 работают с коллекциями, но на самом деле этот механизм может использоваться для самых различных данных.

Stream Api позволяет писать обработку структур данных в стиле SQL, то если раньше задача получить сумму всех нечетных чисел из коллекции решалась следующим кодом:

```
Integer sumOddOld = 0;
for(Integer i: collection) {
    if(i % 2 != 0) {
        sumOddOld += i;
    }
}
```

То с помощью Stream Api можно решить такую задачу в функциональном стиле:

```
Integer sumOdd = collection.stream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0);
```

Более того, Stream Api позволяет решать задачу параллельно лишь изменив stream() на parallelStream() без всякого лишнего кода, т.е.

```
Integer sumOdd = collection.parallelStream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0);
```

Уже делает код параллельным, без всяких семафоров, синхронизаций, рисков взаимных блокировок и т.п.

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

Выражение n -> n + 1, это просто аналог выражения Integer func(Integer n) { return n+1;}, а выражение () -> «a1» аналог выражения String func() { return «a1»;} обернутых в анонимный класс.

Более подробные примеры

Так же этот пример можно найти на github'е

```
        System.out.println("Test buildStream start");

        // Создание стрима из значений
        Stream<String> streamFromValues = Stream.of("a1", "a2", "a3");
        System.out.println("streamFromValues = " + streamFromValues.collect(Collectors.toList())); // напечатает stream
FromValues = [a1, a2, a3]

        // Создание стрима из массива
        String[] array = {"a1","a2","a3"};
        Stream<String> streamFromArrays = Arrays.stream(array);
        System.out.println("streamFromArrays = " + streamFromArrays.collect(Collectors.toList())); // напечатает stream
FromArrays = [a1, a2, a3]

        Stream<String> streamFromArrays1 = Stream.of(array);
        System.out.println("streamFromArrays1 = " + streamFromArrays1.collect(Collectors.toList())); // напечатает стре
amFromArrays = [a1, a2, a3]

        // Создание стрима из файла (каждая запись в файле будет отдельной строкой в стриме)
        File file = new File("1.tmp");
        file.deleteOnExit();
        PrintWriter out = new PrintWriter(file);
```

```
        out.println("a1");
        out.println("a2");
        out.println("a3");
        out.close();

        Stream<String> streamFromFiles = Files.lines(Paths.get(file.getAbsolutePath()));
        System.out.println("streamFromFiles = " + streamFromFiles.collect(Collectors.toList())); // напечатает streamFr
omFiles = [a1, a2, a3]

        // Создание стрима из коллекции
        Collection<String> collection = Arrays.asList("a1", "a2", "a3");
        Stream<String> streamFromCollection = collection.stream();
        System.out.println("streamFromCollection = " + streamFromCollection.collect(Collectors.toList())); // напечатае
m streamFromCollection = [a1, a2, a3]

        // Создание числового стрима из строки
        IntStream streamFromString = "123".chars();
        System.out.print("streamFromString = ");
        streamFromString.forEach((e)->System.out.print(e + " , ")); // напечатает streamFromString = 49 , 50 , 51 ,
        System.out.println();

        // С помощью Stream.builder
        Stream.Builder<String> builder = Stream.builder();
        Stream<String> streamFromBuilder = builder.add("a1").add("a2").add("a3").build();
        System.out.println("streamFromBuilder = " + streamFromBuilder.collect((Collectors.toList()))); // напечатает st
reamFromFiles = [a1, a2, a3]

        // Создание бесконечных стримов
        // С помощью Stream.iterate
        Stream<Integer> streamFromIterate = Stream.iterate(1, n -> n + 2);
        System.out.println("streamFromIterate = " + streamFromIterate.limit(3).collect(Collectors.toList())); // напеча
тает streamFromIterate = [1, 3, 5]

        // С помощью Stream.generate
```

```
        Stream<String> streamFromGenerate = Stream.generate(() -> "a1");
        System.out.println("streamFromGenerate = " + streamFromGenerate.limit(3).collect(Collectors.toList())); // напе
чатает streamFromGenerate = [a1, a1, a1]

        // Создать пустой стрим
        Stream<String> streamEmpty = Stream.empty();
        System.out.println("streamEmpty = " + streamEmpty.collect(Collectors.toList())); // напечатает streamEmpty = []

        // Создать параллельный стрим из коллекции
        Stream<String> parallelStream = collection.parallelStream();
        System.out.println("parallelStream = " + parallelStream.collect(Collectors.toList())); // напечатает parallelSt
ream = [a1, a2, a3]
```

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

Общее правило: у stream'а может быть сколько угодно вызовов конвейерных вызовов и в конце один терминальный, при этом все конвейерные методы выполняются лениво и пока не будет вызван терминальный метод никаких действий на самом деле не происходит, так же как создать объект Thread или Runnable, но не вызвать у него start.

В целом, этот механизм похож на конструирования SQL запросов, может быть сколько угодно вложенных Select'ов и только

один результат в итоге. Например, в выражении collection.stream().filter((s) -> s.contains(«1»)).skip(2).findFirst(), filter и skip — конвейерные, а findFirst — терминальный, он возвращает объект Optional и это заканчивает работу со stream'ом.

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

	Comparator	
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 выполняется ни для одного элемента	collection.stream().noneMatch(«a8»::equals)
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

Выражение «a3»::equals это аналог boolean func(s) { return «a3».equals(s);}, а (s) -> s.contains(«1») это аналог boolean func(s) { return s.contains(«1»);} обернутых в анонимный класс.

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

Также этот пример можно найти на github'e: первый класс и второй класс

```
    // filter - возвращает stream, в котором есть только элементы, соответствующие условию фильтра
    // count - возвращает количество элементов в стриме
    // collect - преобразует stream в коллекцию или другую структуру данных
    // mapToInt - преобразовать объект в числовой стрим (стрим, содержащий числа)
    private static void testFilterAndCount() {
        System.out.println();
```

```
        System.out.println("Test filter and count start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");
        Collection<People> peoples = 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)
        );

        // Вернуть количество вхождений объекта
        long count = collection.stream().filter("a1"::equals).count();
        System.out.println("count = " + count); // напечатает count = 2

        // Выбрать все элементы по шаблону
        List<String> select = collection.stream().filter((s) -> s.contains("1")).collect(Collectors.toList());
        System.out.println("select = " + select); // напечатает select = [a1, a1]

        // Выбрать мужчин-военнообязанных
        List<People> militaryService = peoples.stream().filter((p)-> p.getAge() >= 18 && p.getAge() < 27
                && p.getSex() == Sex.MAN).collect(Collectors.toList());
        System.out.println("militaryService = " + militaryService); // напечатает militaryService = [{name='Петя', age=
23, sex=MAN}]

        // Найти средний возраст среди мужчин
        double manAverageAge = peoples.stream().filter((p) -> p.getSex() == Sex.MAN).
                mapToInt(People::getAge).average().getAsDouble();
        System.out.println("manAverageAge = " + manAverageAge); // напечатает manAverageAge = 36.0

        // Найти кол-во потенциально работоспособных людей в выборке (т.е. от 18 лет и учитывая что женщины выходят в 5
5 лет, а мужчина в 60)
        long peopleHowCanWork = peoples.stream().filter((p) -> p.getAge() >= 18).filter(
                (p) -> (p.getSex() == Sex.WOMEN && p.getAge() < 55) || (p.getSex() == Sex.MAN && p.getAge() < 60)).coun
t();
        System.out.println("peopleHowCanWork = " + peopleHowCanWork); // напечатает manAverageAge = 2
```

```
    }

    // findFirst - возвращает первый Optional элемент из стрима
    // skip - пропускает N первых элементов (где N параметр метода)
    // collect преобразует stream в коллекцию или другую структуру данных
    private static void testFindFirstSkipCount() {
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

        System.out.println("Test findFirst and skip start");
        // вернуть первый элемент коллекции
        String first = collection.stream().findFirst().orElse("1");
        System.out.println("first = " + first); // напечатает first = a1

        // вернуть последний элемент коллекции
        String last = collection.stream().skip(collection.size() - 1).findAny().orElse("1");
        System.out.println("last = " + last ); // напечатает last = a1

        // найти элемент в коллекции
        String find = collection.stream().filter("a3"::equals).findFirst().get();
        System.out.println("find = " + find); // напечатает find = a3

        // вернуть третий элемент коллекции по порядку
        String third = collection.stream().skip(2).findFirst().get();
        System.out.println("third = " + third); // напечатает third = a3

        System.out.println();
        System.out.println("Test collect start");
        // выбрать все элементы по шаблону
        List<String> select = collection.stream().filter((s) -> s.contains("1")).collect(Collectors.toList());
        System.out.println("select = " + select); // напечатает select = [a1, a1]
    }

    // Метод Limit позволяет ограничить выборку определенным количеством первых элементов
```

```
    private static void testLimit() {
        System.out.println();
        System.out.println("Test limit start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

        // Вернуть первые два элемента
        List<String> limit = collection.stream().limit(2).collect(Collectors.toList());
        System.out.println("limit = " + limit); // напечатает limit = [a1, a2]

        // Вернуть два элемента начиная со второго
        List<String> fromTo = collection.stream().skip(1).limit(2).collect(Collectors.toList());
        System.out.println("fromTo = " + fromTo); // напечатает fromTo = [a2, a3]

        // вернуть последний элемент коллекции
        String last = collection.stream().skip(collection.size() - 1).findAny().orElse("1");
        System.out.println("last = " + last ); // напечатает last = a1
    }

    private enum Sex {
        MAN,
        WOMEN
    }

    private static class People {
        private final String name;
        private final Integer age;
        private final Sex sex;

        public People(String name, Integer age, Sex sex) {
            this.name = name;
            this.age = age;
            this.sex = sex;
        }
```

```
        public String getName() {
            return name;
        }

        public Integer getAge() {
            return age;
        }

        public Sex getSex() {
            return sex;
        }

        @Override
        public String toString() {
            return "{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    ", sex=" + sex +
                    '}';
        }
    }
```

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

Так же этот пример можно найти на github'e

```
// Метод distinct возвращает stream без дубликатов, при этом для упорядоченного стрима (например, коллекция на основе l
ist) порядок стабилен , для неупорядоченного - порядок не гарантируется
    // Метод collect преобразует stream в коллекцию или другую структуру данных
    private static void testDistinct() {
        System.out.println();
        System.out.println("Test distinct start");
```

```
        Collection<String> ordered = Arrays.asList("a1", "a2", "a2", "a3", "a1", "a2", "a2");
        Collection<String> nonOrdered = new HashSet<>(ordered);

        // Получение коллекции без дубликатов
        List<String> distinct = nonOrdered.stream().distinct().collect(Collectors.toList());
        System.out.println("distinct = " + distinct); // напечатает distinct = [a1, a2, a3] - порядок не гарантируется

        List<String> distinctOrdered = ordered.stream().distinct().collect(Collectors.toList());
        System.out.println("distinctOrdered = " + distinctOrdered); // напечатает distinct = [a1, a2, a3] - порядок гар
антируется
    }
```

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

Также этот пример можно найти на github'e

```
// Метод anyMatch - возвращает true, если условие выполняется хотя бы для одного элемента
// Метод noneMatch - возвращает true, если условие не выполняется ни для одного элемента
// Метод allMatch - возвращает true, если условие выполняется для всех элементов
private static void testMatch() {
    System.out.println();
    System.out.println("Test anyMatch, allMatch, noneMatch  start");
    Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

    // найти существуют ли хоть одно совпадение с шаблоном в коллекции
    boolean isAnyOneTrue = collection.stream().anyMatch("a1"::equals);
    System.out.println("anyOneTrue " + isAnyOneTrue); // напечатает true
    boolean isAnyOneFalse = collection.stream().anyMatch("a8"::equals);
    System.out.println("anyOneFlase " + isAnyOneFalse); // напечатает false

    // найти существуют ли все совпадения с шаблоном в коллекции
    boolean isAll = collection.stream().allMatch((s) -> s.contains("1"));
    System.out.println("isAll " + isAll); // напечатает false

    // сравнение на неравенство
    boolean isNotEquals = collection.stream().noneMatch("a7"::equals);
    System.out.println("isNotEquals " + isNotEquals); // напечатает true
}
```

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

Так же этот пример можно найти на github'e

```
    // Метод Map изменяет выборку по определенному правилу, возвращает stream с новой выборкой
    private static void testMap() {
        System.out.println();
        System.out.println("Test map start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");
        // Изменение всех элементов коллекции
        List<String> transform = collection.stream().map((s) -> s + "_1").collect(Collectors.toList());
        System.out.println("transform = " + transform); // напечатает transform = [a1_1, a2_1, a3_1, a1_1]

        // убрать первый символ и вернуть числа
        List<Integer> number = collection.stream().map((s) -> Integer.parseInt(s.substring(1))).collect(Collectors.toLi
st());
        System.out.println("number = " + number); // напечатает transform = [1, 2, 3, 1]

    }

    // Метод MapToInt - изменяет выборку по определенному правилу, возвращает stream с новой числовой выборкой
    private static void testMapToInt() {
        System.out.println();
        System.out.println("Test mapToInt start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");
        // убрать первый символ и вернуть числа
        int[] number = collection.stream().mapToInt((s) -> Integer.parseInt(s.substring(1))).toArray();
        System.out.println("number = " + Arrays.toString(number)); // напечатает number = [1, 2, 3, 1]

    }

    // Метод FlatMap - похоже на Map - только вместо одного значения, он возвращает целый stream значений
    private static void testFlatMap() {
        System.out.println();
```

```
        System.out.println("Test flat map start");
        Collection<String> collection = Arrays.asList("1,2,0", "4,5");
        // получить все числовые значения, которые хранятся через запятую в collection
        String[] number = collection.stream().flatMap((p) -> Arrays.asList(p.split(",")).stream()).toArray(String[]::ne
w);
        System.out.println("number = " + Arrays.toString(number)); // напечатает number = [1, 2, 0, 4, 5]
    }

    // Метод FlatMapToInt - похоже на MapToInt - только вместо одного значения, он возвращает целый stream значений
    private static void testFlatMapToInt() {
        System.out.println();
        System.out.println("Test flat map start");
        Collection<String> collection = Arrays.asList("1,2,0", "4,5");
        // получить сумму всех числовые значения, которые хранятся через запятую в collection
        int sum = collection.stream().flatMapToInt((p) -> Arrays.asList(p.split(",")).stream().mapToInt(Integer::parseI
nt)).sum();
        System.out.println("sum = " + sum); // напечатает sum = 12
    }
```

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

Детальные примеры

Так же этот пример можно найти на github'e

```
    // Метод Sorted позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator
    private static void testSorted() {
        System.out.println();
```

```
        System.out.println("Test sorted start");

        // ************ Работа со строками
        Collection<String> collection = Arrays.asList("a1", "a4", "a3", "a2", "a1", "a4");

        // отсортировать значения по алфавиту
        List<String> sorted = collection.stream().sorted().collect(Collectors.toList());
        System.out.println("sorted = " + sorted); // напечатает sorted = [a1, a1, a2, a3, a4, a4]

        // отсортировать значения по алфавиту и убрать дубликаты
        List<String> sortedDistinct = collection.stream().sorted().distinct().collect(Collectors.toList());
        System.out.println("sortedDistinct = " + sortedDistinct); // напечатает sortedDistinct = [a1, a2, a3, a4]

        // отсортировать значения по алфавиту в обратном порядке
        List<String> sortedReverse = collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).collect(Collectors.toLis
t());
        System.out.println("sortedReverse = " + sortedReverse); // напечатает sortedReverse = [a4, a4, a3, a2, a1, a1]

        // отсортировать значения по алфавиту в обратном порядке  и убрать дубликаты
        List<String> distinctReverse = collection.stream().sorted((o1, o2) -> -o1.compareTo(o2)).distinct().collect(Col
lectors.toList());
        System.out.println("distinctReverse = " + distinctReverse); // напечатает sortedReverse = [a4, a3, a2, a1]

        // ************ Работа с объектами
        // Зададим коллекцию людей
        Collection<People> peoples = 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)
        );

        // Отсортировать по имени в обратном алфавитном порядке
        Collection<People> byName = peoples.stream().sorted((o1,o2) -> -o1.getName().compareTo(o2.getName())).collect(C
```

```
ollectors.toList());
        System.out.println("byName = " + byName); // byName = [{name='Петя', age=23, sex=MAN}, {name='Иван Иванович', a
ge=69, sex=MAN}, {name='Елена', age=42, sex=WOMEN}, {name='Вася', age=16, sex=MAN}]

        // Отсортировать сначала по полу, а потом по возрасту
        Collection<People> bySexAndAge = peoples.stream().sorted((o1, o2) -> o1.getSex() != o2.getSex() ? o1.getSex().
                compareTo(o2.getSex()) : o1.getAge().compareTo(o2.getAge())).collect(Collectors.toList());
        System.out.println("bySexAndAge = " + bySexAndAge); // bySexAndAge = [{name='Вася', age=16, sex=MAN}, {name='Пе
тя', age=23, sex=MAN}, {name='Иван Иванович', age=69, sex=MAN}, {name='Елена', age=42, sex=WOMEN}]
    }

    private enum Sex {
        MAN,
        WOMEN
    }

    private static class People {
        private final String name;
        private final Integer age;
        private final Sex sex;

        public People(String name, Integer age, Sex sex) {
            this.name = name;
            this.age = age;
            this.sex = sex;
        }

        public String getName() {
            return name;
        }

        public Integer getAge() {
            return age;
        }
```

```
    public Sex getSex() {
        return sex;
    }

    @Override
    public String toString() {
        return "{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof People)) return false;
        People people = (People) o;
        return Objects.equals(name, people.name) &&
                Objects.equals(age, people.age) &&
                Objects.equals(sex, people.sex);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, sex);
    }
}
```

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

Так же этот пример можно найти на github'e

```
// Метод max вернет максимальный элемент, в качестве условия использует компаратор
// Метод min вернет минимальный элемент, в качестве условия использует компаратор
private static void testMinMax() {
    System.out.println();
    System.out.println("Test min and max start");
    // ************ Работа со строками
    Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");

    // найти максимальное значение
    String max = collection.stream().max(String::compareTo).get();
    System.out.println("max " + max); // напечатает a3

    // найти минимальное значение
    String min = collection.stream().min(String::compareTo).get();
    System.out.println("min " + min); // напечатает a1

    // ************ Работа со сложными объектами

    // Зададим коллекцию людей
    Collection<People> peoples = 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)
    );

    // найти человека с максимальным возрастом
    People older = peoples.stream().max((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get();
    System.out.println("older " + older); // напечатает {name='Иван Иванович', age=69, sex=MAN}

    // найти человека с минимальным возрастом
    People younger = peoples.stream().min((p1, p2) -> p1.getAge().compareTo(p2.getAge())).get();
    System.out.println("younger " + younger); // напечатает {name='Вася', age=16, sex=MAN}
```

```
    }


    private enum Sex {
        MAN,
        WOMEN
    }

    private static class People {
        private final String name;
        private final Integer age;
        private final Sex sex;

        public People(String name, Integer age, Sex sex) {
            this.name = name;
            this.age = age;
            this.sex = sex;
        }

        public String getName() {
            return name;
        }

        public Integer getAge() {
            return age;
        }

        public Sex getSex() {
            return sex;
        }

        @Override
        public String toString() {
            return "{" +
```

```
                "name='" + name + '\'' +
                ", age=" + age +
                ", sex=" + sex +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof People)) return false;
        People people = (People) o;
        return Objects.equals(name, people.name) &&
                Objects.equals(age, people.age) &&
                Objects.equals(sex, people.sex);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, sex);
    }
}
```

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 StringBuilder
("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()); // и list и newL
ist содержат [a1_new, a2_new, a3_new]
```

Детальные примеры

Так же этот пример можно найти на github'e

```
    // Метод ForEach применяет указанный метод к каждому элементу стрима и заканчивает работу со стримом
    private static void testForEach() {
        System.out.println();
        System.out.println("For each start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");
        // Напечатать отладочную информацию по каждому элементу стрима
        System.out.print("forEach = ");
        collection.stream().map(String::toUpperCase).forEach((e) -> System.out.print(e + ",")); // напечатает forEach =
 A1,A2,A3,A1,
        System.out.println();

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

```
der("a3"));
        list.stream().forEachOrdered((p) -> p.append("_new"));
        System.out.println("forEachOrdered = " + list); // напечатает forEachOrdered = [a1_new, a2_new, a3_new]
    }

    // Метод Peek возвращает тот же стрим, но при этом применяет указанный метод к каждому элементу стрима
    private static void testPeek() {
        System.out.println();
        System.out.println("Test peek start");
        Collection<String> collection = Arrays.asList("a1", "a2", "a3", "a1");
        // Напечатать отладочную информацию по каждому элементу стрима
        System.out.print("peak1 = ");
        List<String> peek = collection.stream().map(String::toUpperCase).peek((e) -> System.out.print(e + ",")).
                collect(Collectors.toList());
        System.out.println(); // напечатает peak1 = A1,A2,A3,A1,
        System.out.println("peek2 = " + peek); // напечатает peek2 = [A1, A2, A3, A1]

        Collection<StringBuilder> list = Arrays.asList(new StringBuilder("a1"), new StringBuilder("a2"), new StringBuil
der("a3"));
        List<StringBuilder> newList = list.stream().peek((p) -> p.append("_new")).collect(Collectors.toList());
        System.out.println("newList = " + newList); // напечатает newList = [a1_new, a2_new, a3_new]
    }
```

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

Также этот пример можно найти на github'e

```
    // Метод reduce позволяет выполнять агрегатные функции на всей коллекцией (такие как сумма, нахождение минимального
или максимального значение и т.п.)
    // Он возвращает одно Optional значение
    // map - преобразует один объект в другой (например, класс одного тип в другой)
    // mapToInt - преобразование объектов в числовой стрим (стрим, состоящий из значений int)
    private static void testReduce() {
        System.out.println();
        System.out.println("Test reduce start");

        // ************ Работа с числовыми объектами
        Collection<Integer> collection = Arrays.asList(1, 2, 3, 4, 2);

        // Вернуть сумму
        Integer sum = collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0); // через stream Api
```

```
Integer sumOld = 0; // по старому методу
for(Integer i: collection) {
    sumOld += i;
}
System.out.println("sum = " + sum + " : " + sumOld); // напечатает sum = 12 : 12


// Вернуть максимум
Integer max1 = collection.stream().reduce((s1, s2) -> s1 > s2 ? s1 : s2).orElse(0); // через stream Api
Integer max2 = collection.stream().reduce(Integer::max).orElse(0); // через stream Api используя Integer::max
Integer maxOld = null; // по старому методу
for(Integer i: collection) {
    maxOld = maxOld != null && maxOld > i? maxOld: i;
}
maxOld = maxOld == null? 0 : maxOld;
System.out.println("max = " + max1 + " : " + max2 + " : " + maxOld); // напечатает max1 = 4 : 4 : 4

// Вернуть минимум
Integer min = collection.stream().reduce((s1, s2) -> s1 < s2 ? s1 : s2).orElse(0); // через stream Api
Integer minOld = null; // по старому методу
for(Integer i: collection) {
    minOld = minOld != null && minOld < i? minOld: i;
}
minOld = minOld == null? 0 : minOld;
System.out.println("min = " + min+ " : " + minOld); // напечатает min = 1 : 1

// Вернуть последний элемент
Integer last = collection.stream().reduce((s1, s2) -> s2).orElse(0); // через stream Api
Integer lastOld = null; // по старому методу
for(Integer i: collection) {
    lastOld = i;
}
lastOld = lastOld == null? 0 : lastOld;
System.out.println("last = " + last + " : " + lastOld); // напечатает last = 2 : 2
```

```
        // Вернуть сумму чисел, которые больше 2
        Integer sumMore2 = collection.stream().filter(o -> o > 2).reduce((s1, s2) -> s1 + s2).orElse(0);     // через s
tream Api
        Integer sumMore2Old = 0; // по старому методу
        for(Integer i: collection) {
            if(i > 2) {
                sumMore2Old += i;
            }
        }
        System.out.println("sumMore2 = " + sumMore2 + " : " + sumMore2Old); // напечатает sumMore2 = 7 : 7

        // Вернуть сумму нечетных чисел
        Integer sumOdd = collection.stream().filter(o -> o % 2 != 0).reduce((s1, s2) -> s1 + s2).orElse(0); // через st
ream Api
        Integer sumOddOld = 0; // по старому методу
        for(Integer i: collection) {
            if(i % 2 != 0) {
                sumOddOld += i;
            }
        }
        System.out.println("sumOdd = " + sumOdd + " : " + sumOddOld); // напечатает sumOdd = 4 : 4

        // ************ Работа со сложными объектами

        // Зададим коллекцию людей
        Collection<People> peoples = 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)
        );

        // Найдем самого старшего мужчину
```

```
        int oldMan = peoples.stream().filter((p) -> p.getSex() == Sex.MAN).map(People::getAge).reduce((s1, s2) -> s1 >
 s2 ? s1 : s2).get();
        System.out.println("oldMan = " + oldMan); // напечатает 69

        // Найдем самого минимальный возраст человека у которого есть бука е в имени
        int younger = peoples.stream().filter((p) -> p.getName().contains("e")).mapToInt(People::getAge).reduce((s1, s
2) -> s1 < s2 ? s1 : s2).orElse(0);
        System.out.println("younger = " + younger); // напечатает 23
    }

    private enum Sex {
        MAN,
        WOMEN
    }

    private static class People {
        private final String name;
        private final Integer age;
        private final Sex sex;

        public People(String name, Integer age, Sex sex) {
            this.name = name;
            this.age = age;
            this.sex = sex;
        }

        public String getName() {
            return name;
        }

        public Integer getAge() {
            return age;
        }
```

```
        public Sex getSex() {
            return sex;
        }

        @Override
        public String toString() {
            return "{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    ", sex=" + sex +
                    '}';
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof People)) return false;
            People people = (People) o;
            return Objects.equals(name, people.name) &&
                    Objects.equals(age, people.age) &&
                    Objects.equals(sex, people.sex);
        }

        @Override
        public int hashCode() {
            return Objects.hash(name, age, sex);
        }
    }
```

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	strings.stream().distinct().map(String::toUpperCase).toArray(String[]::new)	{A1, B2, C3}

дубликатов и в верхнем регистре		
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

Также примеры можно найти на github'e

```
// Метод collect преобразует stream в коллекцию или другую структуру данных
// Полезные статические методы из Collectors:
// toList, toCollection, toSet - представляют стрим в виде списка, коллекции или множества
// toConcurrentMap, toMap - позволяют преобразовать стрим в map, используя указанные функции
// averagingInt, averagingDouble, averagingLong - возвращают среднее значение
// summingInt, summingDouble, summingLong - возвращает сумму
// summarizingInt, summarizingDouble, summarizingLong - возвращают SummaryStatistics с разными агрегатными значения
```

```
ми
    // partitioningBy - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>
    // groupingBy - разделить коллекцию по условию и вернуть Map<N, List<T>>, где T - тип последнего стрима, N - значен
ие разделителя
    // mapping - дополнительные преобразования значений для сложных Collector'ов
    private static void testCollect() {
        System.out.println();
        System.out.println("Test distinct start");

        // ******** Работа со строками
        Collection<String> strings = Arrays.asList("a1", "b2", "c3", "a1");

        // Получение списка из коллекции строк без дубликатов
        List<String> distinct = strings.stream().distinct().collect(Collectors.toList());
        System.out.println("distinct = " + distinct); // напечатает distinct = [a1, b2, c3]

        // Получение массива уникальных значений из коллекции строк
        String[] array = strings.stream().distinct().map(String::toUpperCase).toArray(String[]::new);
        System.out.println("array = " + Arrays.asList(array)); // напечатает array = [A1, B2, C3]

        // Объединить все элементы в одну строку через разделитель : и обернуть тегами <b> ... </b>
        String join = strings.stream().collect(Collectors.joining(" : ", "<b> ", " </b>"));
        System.out.println("join = " + join); // напечатает <b> a1 : b2 : c3 : a1 </b>

        // Преобразовать в map, где первый символ ключ, второй символ значение
        Map<String, String> map = strings.stream().distinct().collect(Collectors.toMap((p) -> p.substring(0, 1), (p) ->
p.substring(1, 2)));
        System.out.println("map = " + map); // напечатает map = {a=1, b=2, c=3}

        // Преобразовать в map, сгруппировав по первому символу строки
        Map<String, List<String>> groups = strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1)));
        System.out.println("groups = " + groups); // напечатает groups = {a=[a1, a1], b=[b2], c=[c3]}

        // Преобразовать в map, сгруппировав по первому символу строки и в качестве значения взять второй символ объеди
```

```
ним через :
        Map<String, String> groupJoin = strings.stream().collect(Collectors.groupingBy((p) -> p.substring(0, 1), Collec
tors.mapping((p) -> p.substring(1, 2), Collectors.joining(":"))));
        System.out.println("groupJoin = " + groupJoin); // напечатает groupJoin = groupJoin = {a=1/1, b=2, c=3}

        // ******** Работа с числами
        Collection<Integer> numbers = Arrays.asList(1, 2, 3, 4);

        // Получить сумму нечетных чисел
        long sumOdd = numbers.stream().collect(Collectors.summingInt(((p) -> p % 2 == 1 ? p : 0)));
        System.out.println("sumOdd = " + sumOdd); // напечатает sumEven = 4

        // Вычесть к каждого элемента 1 и получить среднее
        double average = numbers.stream().collect(Collectors.averagingInt((p) -> p - 1));
        System.out.println("average = " + average); // напечатает average = 1.5

        // Прибавить к числам 3 и получить статистику
        IntSummaryStatistics statistics = numbers.stream().collect(Collectors.summarizingInt((p) -> p + 3));
        System.out.println("statistics = " + statistics); // напечатает statistics = IntSummaryStatistics{count=4, sum=
22, min=4, average=5.500000, max=7}

        // Получить сумму четных чисел через IntSummaryStatistics
        long sumEven = numbers.stream().collect(Collectors.summarizingInt((p) -> p % 2 == 0 ? p : 0)).getSum();
        System.out.println("sumEven = " + sumEven); // напечатает sumEven = 6

        // Разделить числа на четные и нечетные
        Map<Boolean, List<Integer>> parts = numbers.stream().collect(Collectors.partitioningBy((p) -> p % 2 == 0));
        System.out.println("parts = " + parts); // напечатает parts = {false=[1, 3], true=[2, 4]}
    }
```

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 выполнен
ии
        );
// Используем его для получение списка строк без дубликатов из стрима
        List<String> distinct1 = strings.stream().distinct().collect(toList);
```

Детальные примеры

Так же примеры можно найти на github'e

```
        // Напишем собственный Collector, который будет выполнять объединение строк с помощью StringBuilder
        Collector<String,StringBuilder, String> stringBuilderCollector =  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 при параллельном выполн
 ении
                StringBuilder::toString // метод, выполняющийся в самом конце
        );
        String joinBuilder = strings.stream().collect(stringBuilderCollector);
        System.out.println("joinBuilder = " + joinBuilder); // напечатает joinBuilder = a1 , b2 , c3 , a1 ,

        // Аналог Collector'а выше стилем JDK7 и ниже
```

```
        StringBuilder b = new StringBuilder(); // метод инициализации аккумулятора
        for(String s: strings) {
            b.append(s).append(" , "); // метод обработки каждого элемента
        }
        String joinBuilderOld = b.toString(); // метод, выполняющийся в самом конце
        System.out.println("joinBuilderOld = " + joinBuilderOld); // напечатает joinBuilderOld = a1 , b2 , c3 , a1 ,

        // Напишем свой аналог toList для получение списка из коллекции строк без дубликатов
        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
        );
        List<String> distinct1 = strings.stream().distinct().collect(toList);
        System.out.println("distinct1 = " + distinct1); // напечатает distinct1 = [a1, b2, c3]
```

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?

- Да, уже в production
- Да, но только в личных проектах
- Очень ограничено