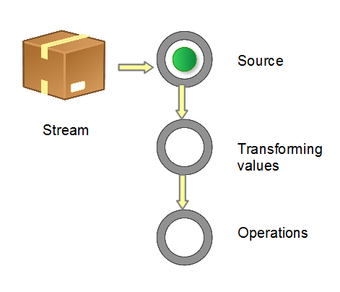
如果说Runnable接口是将**执行逻辑**从Thread中剥离了的话, 那Stream则是将**数据计算逻辑**从Collection中抽离了出来, 使Collection只专注于数据的存储, 而不用分心计算.

Stream允许以声明方式处理集合等可以转换为Stream<T>的数据, 他有很多特点:

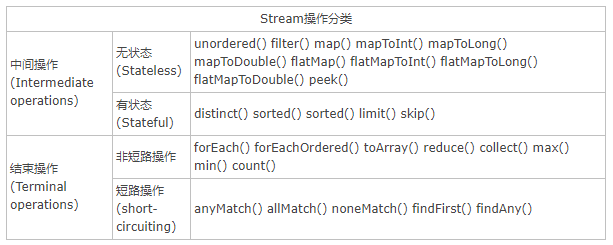
* 内部迭代   
  与原有的Iterator不同, Stream将迭代操作(类似for/for-each)全部固化到了Api内部实现, 用户只需传入表达计算逻辑的**lambda**表达式(可以理解为Supplier、Function这些的@FunctionalInterface的实现), Stream便会自动迭代数据触发计算逻辑并生成结果. 内部迭代主要解决了两方面的问题: **避免集合处理时的套路和晦涩**; **便于库内部实现的多核并行优化**.
* 流水线   
  很多Stream操作会再返回一个Stream, 这样多个操作就可以链接起来, 形成一个大的流水线, 使其看起来像是***对数据源进行数据库式查询***, 这也就让自动优化成为可能, 如**隐式并行**.
* 隐式并行   
  如将.stream()替换为.parallelStream(), Stream则会自动启用Fork/Join框架, 并行执行各条流水线, 并最终自动将结果进行合并.
* 延迟计算   
  由于Stream大部分的操作(如filter()、generate()、map()…)都是接受一段**lambda**表达式, 逻辑类似接口实现(可以看成是**回调**), 因此代码并不是立即执行的, 除非流水线上触发一个终端操作, 否则中间操作不会执行任何处理.
* 短路求值   
  有些操作不需要处理整个流就能够拿到结果, 很多像anyMatch()、allMatch()、limit(), 只要找到一个元素他们的工作就可以结束, 也就没有必要执行后面的操作, 因此如果后面有大量耗时的操作, 此举可大大节省性能.

**Stream 构成**

一个流管道(Stream pipeline)通常由3部分构成: ***数据源(Source)*** -> ***中间操作/转换(Transforming)*** -> ***终端操作/执行(Operations)***: Stream由数据源生成, 经由中间操作串联起来的一条流水线的转换, 最后由终端操作触发执行拿到结果.   


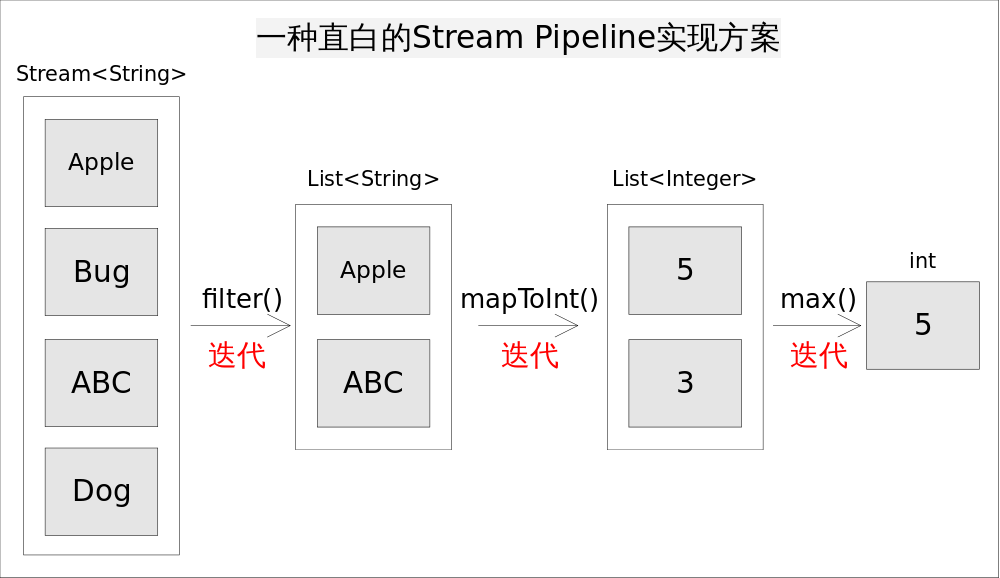
1. ***Source*** - 对应Stream的生成: -> 如何生成一个Stream;
2. ***Transforming*** - 对应Stream的转换: -> 如前面的map()、filter()、limit(), 将原Stream转换为另一形态;
3. ***Operations*** - 对应Stream的执行: -> 他会真正引发前面一系列***Transforming***的执行, 并生成一个结果(如List、Array、Optional<T>), 或一个*side effect*.

 Stream的常用操作分类



所有的中间操作都会返回另一个Stream, 这让多个操作可以链接起来组成中间操作链, 从而形成一条流水线, 因此它的特点就是前面提到的**延迟执行**: 触发流水线上触发一个终端操作, 否则中间操作不执行任何处理.

如表所示，**Stream中的操作可以分为两大类**：中间操作与结束操作，中间操作只是对操作进行了记录，只有结束操作才会触发实际的计算（即惰性求值），这也是Stream在迭代大集合时高效的原因之一。中间操作又可以分为无状态（Stateless）操作与有状态（Stateful）操作，前者是指元素的处理不受之前元素的影响；后者是指该操作只有拿到所有元素之后才能继续下去。结束操作又可以分为短路与非短路操作，这个应该很好理解，前者是指遇到某些符合条件的元素就可以得到最终结果；而后者是指必须处理所有元素才能得到最终结果。



这样做实现起来非常简单直观，但有两个明显的弊端：

1. 迭代次数多。迭代次数跟函数调用的次数相等。
2. 频繁产生中间结果。每次函数调用都产生一次中间结果，存储开销无法接受。

这些弊端使得效率底下，根本无法接受。如果不使用Stream API我们都知道上述代码该如何在一次迭代中完成，大致是如下形式：

int longest = 0;

for(String str : strings){

// 1. filter(), 保留以A开头的字符串

if(str.startsWith("A")){

// 2. mapToInt(), 转换成长度

int len = str.length();

//3. max(), 保留最长的长度

longest = Math.max(len, longest);

}

}

Stream API中大量使用Lambda表达式作为回调方法，但这并不是关键。理解Stream我们更关心的是另外两个问题：**流水线**和**自动并行**。使用Stream或许很容易写入如下形式的代码：

int longestStringLengthStartingWithA = strings.stream()

.filter(s -> s.startsWith("A"))

.mapToInt(String::length)

.max();

类库的实现着使用流水线（Pipeline）的方式巧妙的避免了多次迭代，其基本思想是在一次迭代中尽可能多的执行用户指定的操作。

我们分别来介绍这些Stream的构成部分:

## 数据源-Stream生成

除了前面介绍过的collection.stream(), 流的生成方式多种多样, 可简单概括为3类: **通用流**、**数值流**、**其他**, 其中以**通用流**最为常用, **数值流**是Java为int、long、double三种数值类型防**拆装箱**成本所做的优化:

### 1. 通用流

| **API** | **description** |
| --- | --- |
| Arrays.stream(T[] array) | Returns a sequential Stream with the specified array as its source. |
| Stream.empty() | Returns an empty sequential Stream. |
| Stream.generate(Supplier<T> s) | Returns an infinite sequential **unordered** stream where each element is generated by the provided Supplier<T>. |
| Stream.iterate(T seed, UnaryOperator<T> f) | Returns an infinite sequential **ordered** Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of **seed**, **f(seed)**, **f(f(seed))**, etc. |
| Stream.of(T... values) | Returns a sequential ordered stream whose elements are the specified values. |
| Stream.concat(Stream<? extends T> a, Stream<? extends T> b) | Creates a lazily concatenated stream whose elements are **all the elements of the first stream followed by all the elements of the second stream**. |
| StreamSupport.stream(Spliterator<T> spliterator, boolean parallel) | Creates a new **sequential** or **parallel** Stream from a Spliterator. |

### 2. 数值流

| **API** | **description** |
| --- | --- |
| Arrays.stream(Xxx[] array) | Returns a sequential Int/Long/DoubleStream with the specified array as its source. |
| XxxStream.empty() | Returns an empty sequential Int/Long/DoubleStream. |
| XxxStream.generate(XxxSupplier s) | Returns an infinite sequential **unordered** stream where each element is generated by the provided Int/Long/DoubleSupplier. |
| XxxStream.iterate(Xxx seed, XxxUnaryOperator f) | Returns an infinite sequential **ordered** Int/Long/DoubleStream like as Stream.iterate(T seed, UnaryOperator<T> f) |
| XxxStream.of(Xxx... values) | Returns a sequential ordered stream whose elements are the specified values. |
| XxxStream.concat(XxxStream a, XxxStream b) | Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. |
| Int/LongStream.range(startInclusive, endExclusive) | Returns a sequential ordered Int/LongStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1. |
| Int/LongStream.rangeClosed(startInclusive, endInclusive) | Returns a sequential ordered Int/LongStream from startInclusive (inclusive) to endInclusive (inclusive) by an incremental step of 1. |

### 3. 其他

* I/O Stream   
  + BufferedReader.lines()
* File Stream   
  + Files.lines(Path path)
  + Files.find(Path start, int maxDepth, BiPredicate<Path,BasicFileAttributes> matcher, FileVisitOption... options)
  + DirectoryStream<Path> newDirectoryStream(Path dir)
  + Files.walk(Path start, FileVisitOption... options)
* Jar   
  + JarFile.stream()
* Random   
  + Random.ints()
  + Random.longs()
  + Random.doubles()
* Pattern   
  + splitAsStream(CharSequence input)   
    …

另外, 三种数值流之间, 以及数值流与通用流之间都可以相互转换:   
1. 数值流转换: doubleStream.mapToInt(DoubleToIntFunction mapper)、intStream.asLongStream() …   
2. 数值流转通用流: longStream.boxed()、intStream.mapToObj(IntFunction<? extends U> mapper) …   
3. 通用流转数值流: stream.flatMapToInt(Function<? super T,? extends IntStream> mapper)、stream.mapToDouble(ToDoubleFunction<? super T> mapper)…

## 中间操作-Stream转换

所有的中间操作都会返回另一个Stream, 这让多个操作可以链接起来组成中间操作链, 从而形成一条流水线, 因此它的特点就是前面提到的**延迟执行**: 触发流水线上触发一个终端操作, 否则中间操作不执行任何处理.

| **API** | **Description** |
| --- | --- |
| filter(Predicate<? super T> predicate) | Returns a stream consisting of the elements of this stream that match the given predicate. |
| distinct() | Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| limit(long maxSize) | Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. |
| skip(long n) | Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. |
| sorted(Comparator<? super T> comparator) | Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator. |
| map(Function<? super T,? extends R> mapper) | Returns a stream consisting of the results of applying the given function to the elements of this stream. |
| flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | Returns a stream consisting of the results of **replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.** |
| peek(Consumer<? super T> action) | Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. |

这里着重讲解下flatMap(), 因为我在第一次接触他时也没明白他到底能做什么:   
假设我们有这样一个字符串list:List<String> strs = Arrays.asList("hello", "alibaba", "world");如何列出里面各不相同的字符呢?   
首先我们想到的是String包含一个split()方法, 将字符串分解为子串, 于是我们这样写:

Stream<Stream<String>> streamStream = strs.stream()

.map(str -> Arrays.stream(str.split("")));

* 1
* 2

我们将String分解成String[]后再由Arrays.stream()将String[]映射成Stream<String>, 但这个结果是我们不想看到的: 我们明明想要的是Stream<String>却得到的是Stream<Stream<String>>, 他把我们想要的结果包到Stream里面了. 这时候就需要我们的flatMap()出场了:

Stream<String> stringStream = strs.stream()

.flatMap(str -> Arrays.stream(str.split("")));

* 1
* 2

flatMap()把Stream中的层级结构扁平化了, 将内层Stream内的元素抽取出来, 最终新的Stream就没有内层Stream了.

可以简单概括为: flatMap()方法让你把一个流中的每个值都换成另一个Stream, 然后把所有的Stream连接起来成为一个Stream.

## 终端操作-Stream执行

终端操作不仅担负着触发流水线执行的任务, 他还需要拿到流水线执行的结果, 其结果为任何不是流的值, 如List、**Array**、boolean、Optional<T>, 甚至是void(forEach()):

| **Api** | **Description** |
| --- | --- |
| count() | Returns the count of elements in this stream. |
| max(Comparator<? super T> comparator) | Returns the maximum element of this stream according to the provided Comparator. |
| min(Comparator<? super T> comparator) | Returns the minimum element of this stream according to the provided Comparator. |
| allMatch(Predicate<? super T> predicate) | Returns whether all elements of this stream match the provided predicate. |
| anyMatch(Predicate<? super T> predicate) | Returns whether any elements of this stream match the provided predicate. |
| noneMatch(Predicate<? super T> predicate) | Returns whether no elements of this stream match the provided predicate. |
| findAny() | Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| findFirst() | Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. |
| reduce(BinaryOperator<T> accumulator) | Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. |
| toArray() | Returns an array containing the elements of this stream. |
| forEach(Consumer<? super T> action) | Performs an action for each element of this stream. |
| forEachOrdered(Consumer<? super T> action) | Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order. |
| collect(Collector<? super T,A,R> collector) | Performs a mutable reduction operation on the elements of this stream using a Collector. |

像IntStream/LongStream/DoubleStream还提供了average()、sum()、summaryStatistics()这样的操作, 拿到一个对Stream进行汇总了的结果.

### other

java.util.stream.Stream接口集成自java.util.stream.BaseStream接口, 而BaseStream接口也提供了很多工具方法(如将串行流转换为并行流的parallel()方法)供我们使用:

| **Api** | **Description** |
| --- | --- |
| S onClose(Runnable closeHandler) | Returns an equivalent stream with **an additional close handler**. |
| void close() | Closes this stream, **causing all close handlers** for this stream pipeline to be called. |
| S unordered() | Returns an equivalent stream that is unordered. |
| Iterator<T> iterator() | Returns an iterator for the elements of this stream. |
| Spliterator<T> spliterator() | Returns a spliterator for the elements of this stream. |
| S sequential() | Returns an equivalent stream that is **sequential**. |
| S parallel() | Returns an equivalent stream that is **parallel**. |
| boolean isParallel() | Returns whether this stream, if a terminal operation were to be executed, would execute in parallel. |

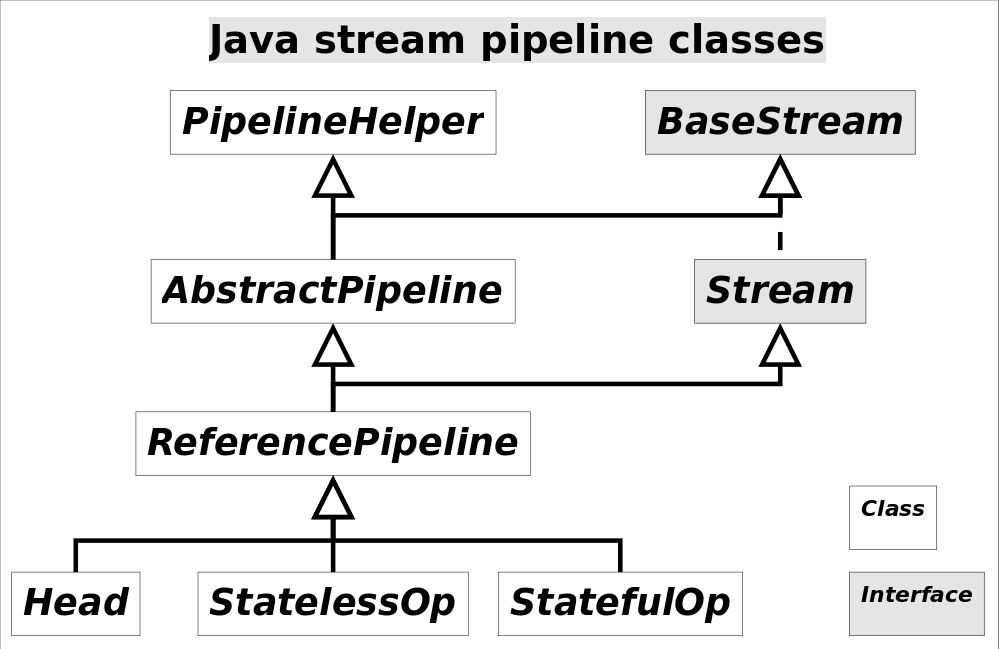
## Stream流水线解决方案

我们大致能够想到，应该采用某种方式记录用户每一步的操作，当用户调用结束操作时将之前记录的操作叠加到一起在一次迭代中全部执行掉。沿着这个思路，有几个问题需要解决：

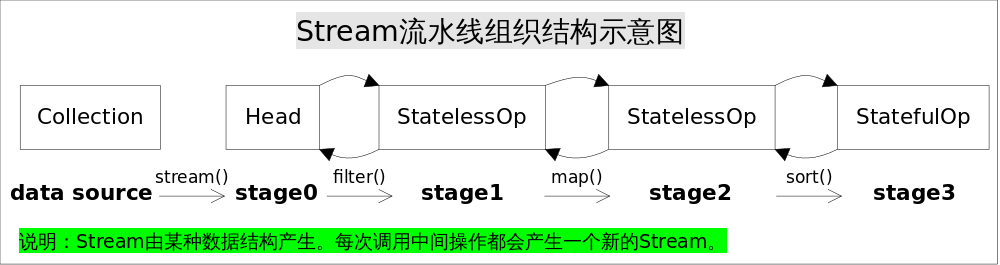
1. 用户的操作如何记录？
2. 操作如何叠加？
3. 叠加之后的操作如何执行？
4. 执行后的结果（如果有）在哪里？

### >> 操作如何记录

注意这里使用的是“操作(operation)”一词，指的是“Stream中间操作”的操作，很多Stream操作会需要一个回调函数（Lambda表达式），因此一个完整的操作是<数据来源，操作，回调函数>构成的三元组。Stream中使用Stage的概念来描述一个完整的操作，并用某种实例化后的PipelineHelper来代表Stage，将具有先后顺序的各个Stage连到一起，就构成了整个流水线。跟Stream相关类和接口的继承关系图示。

还有IntPipeline, LongPipeline, DoublePipeline没在图中画出，这三个类专门为三种基本类型（不是包装类型）而定制的，跟ReferencePipeline是并列关系。图中Head用于表示第一个Stage，即调用调用诸如Collection.stream()方法产生的Stage，很显然这个Stage里不包含任何操作；StatelessOp和StatefulOp分别表示无状态和有状态的Stage，对应于无状态和有状态的中间操作。

Stream流水线组织结构示意图如下：



图中通过Collection.stream()方法得到Head也就是stage0，紧接着调用一系列的中间操作，不断产生新的Stream。**这些Stream对象以双向链表的形式组织在一起，构成整个流水线，由于每个Stage都记录了前一个Stage和本次的操作以及回调函数，依靠这种结构就能建立起对数据源的所有操作**。这就是Stream记录操作的方式。

### >> 操作如何叠加

以上只是解决了操作记录的问题，要想让流水线起到应有的作用我们需要一种将所有操作叠加到一起的方案。你可能会觉得这很简单，只需要从流水线的head开始依次执行每一步的操作（包括回调函数）就行了。这听起来似乎是可行的，但是你忽略了前面的Stage并不知道后面Stage到底执行了哪种操作，以及回调函数是哪种形式。换句话说，只有当前Stage本身才知道该如何执行自己包含的动作。这就需要有某种协议来协调相邻Stage之间的调用关系。

这种协议由Sink接口完成，Sink接口包含的方法如下表所示：

|  |  |
| --- | --- |
| 方法名 | 作用 |
| void begin(long size) | 开始遍历元素之前调用该方法，通知Sink做好准备。 |
| void end() | 所有元素遍历完成之后调用，通知Sink没有更多的元素了。 |
| boolean cancellationRequested() | 是否可以结束操作，可以让短路操作尽早结束。 |
| void accept(T t) | 遍历元素时调用，接受一个待处理元素，并对元素进行处理。Stage把自己包含的操作和回调方法封装到该方法里，前一个Stage只需要调用当前Stage.accept(T t)方法就行了。 |

有了上面的协议，相邻Stage之间调用就很方便了，每个Stage都会将自己的操作封装到一个Sink里，前一个Stage只需调用后一个Stage的accept()方法即可，并不需要知道其内部是如何处理的。当然对于有状态的操作，Sink的begin()和end()方法也是必须实现的。比如Stream.sorted()是一个有状态的中间操作，其对应的Sink.begin()方法可能创建一个乘放结果的容器，而accept()方法负责将元素添加到该容器，最后end()负责对容器进行排序。对于短路操作，Sink.cancellationRequested()也是必须实现的，比如Stream.findFirst()是短路操作，只要找到一个元素，cancellationRequested()就应该返回true，以便调用者尽快结束查找。Sink的四个接口方法常常相互协作，共同完成计算任务。**实际上Stream API内部实现的的本质，就是如何重载Sink的这四个接口方法**。

有了Sink对操作的包装，Stage之间的调用问题就解决了，执行时只需要从流水线的head开始对数据源依次调用每个Stage对应的Sink.{begin(), accept(), cancellationRequested(), end()}方法就可以了。一种可能的Sink.accept()方法流程是这样的：

void accept(U u){

1. 使用当前Sink包装的回调函数处理u

2. 将处理结果传递给流水线下游的Sink

}

Sink接口的其他几个方法也是按照这种[处理->转发]的模型实现。

实战

下面, 我们针对一系列**交易**提出一些问题综合实践上面列举的Api:

* DO定义

/\*\*

\* 交易员

\*/

private class Trader {

private String name;

private String city;

public Trader(String name, String city) {

this.name = name;

this.city = city;

}

public String getName() {

return name;

}

public String getCity() {

return city;

}

@Override

public String toString() {

return "Trader{" +

"name='" + name + '\'' +

", city='" + city + '\'' +

'}';

}

}

/\*\*

\* 交易

\*/

private class Transaction {

private Trader trader;

private int year;

private int value;

public Transaction(Trader trader, int year, int value) {

this.trader = trader;

this.year = year;

this.value = value;

}

public Trader getTrader() {

return this.trader;

}

public int getYear() {

return this.year;

}

public int getValue() {

return this.value;

}

@Override

public String toString() {

return "Transaction{" +

"trader=" + trader +

", year=" + year +

", value=" + value +

'}';

}

}

* Stream操作

/\*\*

\* @author jifang.zjf

\* @since 2017/7/3 下午4:05.

\*/

public class StreamLambda {

private List<Transaction> transactions;

@Before

public void setUp() {

Trader raoul = new Trader("Raoul", "Cambridge");

Trader mario = new Trader("Mario", "Milan");

Trader alan = new Trader("Alan", "Cambridge");

Trader brian = new Trader("Brian", "Cambridge");

transactions = Arrays.asList(

new Transaction(brian, 2011, 300),

new Transaction(raoul, 2012, 1000),

new Transaction(raoul, 2011, 400),

new Transaction(mario, 2012, 710),

new Transaction(mario, 2012, 700),

new Transaction(alan, 2012, 950)

);

}

@Test

public void action() {

// 1. 打印2011年发生的所有交易, 并按交易额排序(从低到高)

transactions.stream()

.filter(transaction -> transaction.getYear() == 2011)

.sorted(Comparator.comparing(Transaction::getValue))

.forEachOrdered(System.out::println);

// 2. 找出交易员都在哪些不同的城市工作过

Set<String> distinctCities = transactions.stream()

.map(transaction -> transaction.getTrader().getCity())

.collect(Collectors.toSet()); // or .distinct().collect(Collectors.toList())

System.out.println(distinctCities);

// 3. 找出所有来自于剑桥的交易员, 并按姓名排序

Trader[] traders = transactions.stream()

.map(Transaction::getTrader)

.filter(trader -> trader.getCity().equals("Cambridge"))

.distinct()

.sorted(Comparator.comparing(Trader::getName))

.toArray(Trader[]::new);

System.out.println(Arrays.toString(traders));

// 4. 返回所有交易员的姓名字符串, 并按字母顺序排序

String names = transactions.stream()

.map(transaction -> transaction.getTrader().getName())

.distinct()

.sorted(Comparator.naturalOrder())

.reduce("", (str1, str2) -> str1 + " " + str2);

System.out.println(names);

// 5. 返回所有交易员的姓名字母串, 并按字母顺序排序

String letters = transactions.stream()

.map(transaction -> transaction.getTrader().getName())

.distinct()

.map(name -> name.split(""))

.flatMap(Arrays::stream)

.sorted()

.collect(Collectors.joining());

System.out.println(letters);

// 6. 有没有交易员是在米兰工作

boolean workMilan = transactions.stream()

.anyMatch(transaction -> transaction.getTrader().getCity().equals("Milan"));

System.out.println(workMilan);

// 7. 打印生活在剑桥的交易员的所有交易额总和

long sum = transactions.stream()

.filter(transaction -> transaction.getTrader().getCity().equals("Cambridge"))

.mapToLong(Transaction::getValue)

.sum();

System.out.println(sum);

// 8. 所有交易中，最高的交易额是多少

OptionalInt max = transactions.stream()

.mapToInt(Transaction::getValue)

.max();

// or transactions.stream().map(Transaction::getValue).max(Comparator.naturalOrder());

System.out.println(max.orElse(0));

// 9. 找到交易额最小的交易

Optional<Transaction> min = transactions.stream()

.min(Comparator.comparingInt(Transaction::getValue));

System.out.println(min.orElseThrow(IllegalArgumentException::new));

}

}