Stream

1.Stream概述

1.1.一个问题

假如我们拥有一个 学生列表

```
public class Person {
    public enum Address{
        BJ,
        SH,
        WH,
        SZ
    }
    private String name;
    private int age;
    private int height;
    private Address address;
}
```

```
public class StudentList {
    public List<Person> personList ;

    public StudentList(){
        this.personList = new ArrayList<>();

        personList.add(new Person("aa", 18, 170, Person.Address.BJ));
        personList.add(new Person("bb", 20, 163, Person.Address.SH));
        personList.add(new Person("cc", 30, 182, Person.Address.WH));
        personList.add(new Person("dd", 16, 190, Person.Address.BJ));
        personList.add(new Person("ee", 15, 210, Person.Address.SH));
        personList.add(new Person("ff", 17, 160, Person.Address.WH));
        personList.add(new Person("gg", 18, 169, Person.Address.BJ));
        personList.add(new Person("hh", 20, 173, Person.Address.WH));
        personList.add(new Person("ii", 22, 192, Person.Address.SH));
        personList.add(new Person("jj", 25, 172, Person.Address.SH));
        personList.add(new Person("kk", 24, 188, Person.Address.BJ));
        personList.add(new Person("ll", 17, 161, Person.Address.WH));
        personList.add(new Person("mm", 18, 169, Person.Address.SH));
        personList.add(new Person("nn", 20, 162, Person.Address.BJ));
        personList.add(new Person("oo", 22, 166, Person.Address.SH));
        personList.add(new Person("pp", 24, 176, Person.Address.WH));
        personList.add(new Person("qq", 22, 173, Person.Address.BJ));
        personList.add(new Person("rr", 24, 177, Person.Address.BJ));
        personList.add(new Person("ss", 17, 169, Person.Address.SH));
        personList.add(new Person("tt", 18, 170, Person.Address.SH));
        personList.add(new Person("uu", 20, 171, Person.Address.WH));
        personList.add(new Person("vv", 22, 172, Person.Address.WH));
        personList.add(new Person("ww", 24, 181, Person.Address.BJ));
        personList.add(new Person("xx", 18, 188, Person.Address.SH));
```

```
        personList.add(new Person("yy", 20, 183, Person.Address.BJ));
        personList.add(new Person("zz", 22, 191, Person.Address.WH));

    }
}
```

我们要对这个学生列表进行处理: 得到来自北京同学, 并且高度最高的三个同学

应该怎么做

```
        TreeSet<Person> peoples = new TreeSet<>(new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                int com = o2.getHeight() - o1.getHeight();
                return com;
            }
        });

        for (Person person : personList) {
            if (person.getAddress() == Person.Address.BJ){
                peoples.add(person);
            }
        }

        System.out.println(peoples.first());
        peoples.pollFirst();
        System.out.println(peoples.first());
        peoples.pollFirst();
        System.out.println(peoples.first());
        peoples.pollFirst();
```

那么对于上述代码有没有更好的方法(类似于我们对数据库的操作一样)?

1.2.使用流解决上述问题

我们可以使用JAVA中Stream流来解决上述问题

```
List<Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .sorted(Comparator.comparing(Person::getHeight).reversed())
                .limit(3)
                .collect(Collectors.toList());

         System.out.println(collect);
```

1.3.流的概述

什么是流?

流的思想是什么?

流用来解决什么问题?

流有哪些优点?

怎么使用流?

1, 什么是流?

流是Java API, 它允许我们以声明的方式来处理集合数据, 也就是说我们不再需要编写一个实现来操作要处理的数据, 而是类似于数据库那样通过查询语句来表达.

Java中集合类数据处理的一种简化方式;

```
// 通俗的讲: 也就是说, Stream流是Java在jdk1,8提供的对集合数据进行优化/简化操作的一种数据处理方式
```

2, 流的思想是什么?: 重要

和Collection操作不同的是Stream操作有两个基础的特征:
Pipelining: 中间操作都会返回流对象本身. 这样多个操作可以串联成一个管道, 如同流式风格(fluent style).
这样做可以对操作进行优化.

内部迭代: 以前对集合遍历都是通过Iterator或者增强for的方式, 显式的在集合外部进行迭代, 这叫做外部迭代.
Stream提供了内部迭代的方式, 流可以直接调用遍历方法(这个遍历在语法底层是不可直接见的).

3, 流用来解决什么问题?

Stream流一般用来处理Java中的集合类数据, 进以避免在日常代码书写中的对集合数据操作的性能以及代码冗长问题.

4, 流有哪些优点?

1,Stream流是一个集合元素的函数模型，它并不是集合，也不是数据结构，其本身并不存储任何元素.

2,Stream流是在对函数模型进行操作，(在终结触发之前)集合元素并没有真正被处理.

```
只有当终结方法执行的时候，整个模型才会按照指定策略执行操作.
```

3,对集合数据操作的性能优化,解决代码冗长问题.

5, 怎么使用流?

使用一个流的时候，通常包括三个基本步:
1, 一个数据源, 创建一个流

2, 多个中间操作, 形成一条流水线

3, 一个终止/终端操作, 执行流水线,并生成结果

```
List<Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .sorted(Comparator.comparing(Person::getHeight).reversed())
                .limit(3)
                .collect(Collectors.toList());

         System.out.println(collect);
```

2.创建流

1, 调用集合类的stream方法,生成一个流(最常用/常见)

```
        Collection collection = new ArrayList();
        Stream stream = collection.stream();
```

2, 由值创建

```
        Stream<String> zs = Stream.of("zs", "ls", "wu", "zl");
```

3,由数组创建

```
        String [] strs = {"zs", "ls", "wu"};
        Stream<String> stream = Arrays.stream(strs);
```

3.中间操作

3.1 filter

filter 方法用于通过设置的条件过滤出元素。

```
// Stream<T> filter(Predicate<? super T> predicate);
// TODO: filter方法参数----  要接收的是布尔类型的值  or  返回布尔类型值的表达式
        List<Person> personList = StudentList.personList;
        // 获取所有北京地区的同学
        List<Person> collect =  personList.stream()
                .filter(d -> d.getAddress().equals(Person.Address.BJ))
                .collect(Collectors.toList());
        System.out.println(collect);
```

注意: 每次中间操作原有 Stream 对象不改变,返回一个新的 Stream (可以有多次中间操作),这就允许对其操作可以像链条一样排列,变成一个管道.

3.2 distinct

distinct方法用于筛选元素(相当于去除重复元素)

```
//  Stream<T> distinct();
//  distinct方法----筛选元素, 筛选的机制是根据元素的hashCode和equals判断重复
        List<Person> personList = StudentList.personList;

        // 创建一个新的集合类
        ArrayList<Person> people = new ArrayList<>(personList);
        // 添加一个重复元素
        people.add(personList.get(0));

        // 不去重, 计算总共有多少个北京同学
        long count1 = people.stream()
                .filter(d -> d.getAddress().equals(Person.Address.BJ))
                .count();
        // 去重, 计算总共有多少个北京同学
        long count2 = people.stream()
                .filter(d -> d.getAddress().equals(Person.Address.BJ))
                .distinct()
                .count();

        System.out.println(count1);
        System.out.println(count2);
```

3.3 limit

limit 方法用于获取指定数量的流。

```
// Stream<T> limit(long maxSize);
// TODO: limit(n)方法, 返回前n个元素.

        // 获取三个年龄大于22岁的同学
        List<Person> collect = personList.stream()
                        .filter(d -> d.getAge() > 22)
                        .limit(3)
                        .collect(Collectors.toList());

        System.out.println(collect);
```

3.4 skip

skip(n)方法, 跳过前n个元素

```
        List<Person> personList = StudentList.personList;
// Stream<T> skip(long n);
// TODO: skip(n)方法, 跳过前n个元素, 返回之后的元素.  (如果整体不够n个, 返回空流)

        List<Person> collect = personList.stream()
                        .filter(d -> d.getAge() > 22)
                        .collect(Collectors.toList());

        List<Person> collect2 = personList.stream()
                        .filter(d -> d.getAge() > 22)
                        .skip(1)
                        .collect(Collectors.toList());

        System.out.println(collect);
        System.out.println(collect2);
```

3.5 map

map 方法用于映射每个元素到对应的结果

```
// <R> Stream<R> map(Function<? super T, ? extends R> mapper);
// TODO: map映射返回新的数据,  map的参数是一个方法

        // 获取所有学生姓名
        List<String> collect = personList.stream()
                        .map(a -> a.getName())
                        .collect(Collectors.toList());
        System.out.println(collect);
```

```
        // 获取所有学生姓名的首字母
        List<String> collect = personList.stream()
                .map(a -> a.getName().substring(0, 1))
                .collect(Collectors.toList());
        System.out.println(collect);
```

```
        // 获取非常高的学生(超过190)
        List<SuperPerson> collect = personList.stream()
                .filter(a -> a.getHeight() > 190)
                .map(a -> new SuperPerson(a.getName(), a.getHeight()))
                .collect(Collectors.toList());
        System.out.println(collect);

class SuperPerson{
    String name;
    int height;
}
```

3.7 sorted

sorted 方法用于对流进行排序

```
//     Stream<T> sorted();: 自然顺序排序
//     Stream<T> sorted(Comparator<? super T> comparator);: 提供一个比较器

        // 对高于180的同学根据身高进行排序
        List<Person> collect = personList.stream()
                .filter(a -> a.getHeight() > 180)
                .sorted(Comparator.comparing(Person::getHeight))
                .collect(Collectors.toList());

        System.out.println(collect);
```

```
        // 对高于180的同学根据身高进行排序(从高到低)
        List<Person> collect = personList.stream()
                .filter(a -> a.getHeight() > 180)
                .sorted(Comparator.comparing(Person::getHeight).reversed())
                .collect(Collectors.toList());

        System.out.println(collect);
```

4.终止操作

4.1 anyMatch

anyMatch:检查流到最后的数据，是否有一个/多个数据匹配某种情况

```
//  boolean anyMatch(Predicate<? super T> predicate);
//  anyMatch: 判断该stream中的所有元素， 是否存在某个/某些元素,可以根据某个条件处理之后， 满足
true
        //  判断是否存在北京的同学
        boolean b1 = personList.stream()
                .anyMatch(a -> {
                    return a.getAddress() == Person.Address.BJ;
                });
        //  判断高于190的是否存在北京的同学
        boolean b2 = personList.stream()
                .filter(a -> a.getHeight() > 190)
                .anyMatch(a -> {
                    return a.getAddress() == Person.Address.BJ;
                });

        System.out.println(b1 + " " + b2);
```

4.2 allMatch

allMatch:检查是否所有所有元素都匹配

```
// boolean allMatch(Predicate<? super T> predicate);
// allMatch: 判断该stream中的所有元素, 是否所有元素 可以根据某个条件处理之后, 满足true
        //  判断是否都是北京的同学
        boolean b1 = personList.stream()
                .allMatch(a -> {
                    return a.getAddress() == Person.Address.BJ;
                });
        //  判断高于200的是否都是上海的同学
        boolean b2 = personList.stream()
                .filter(a -> a.getHeight() > 200)
                .allMatch(a -> {
                    return a.getAddress() == Person.Address.SH;
                });

        System.out.println(b1 + " " + b2);
```

4.3 nonematch

nonematch: 检查是否没有匹配元素

```
// boolean noneMatch(Predicate<? super T> predicate);
// noneMatch: 判断该stream中的所有元素, 是否所有元素 可以根据某个条件处理之后, 满足false

        //  判断是否不存在深圳的同学
        boolean b1 = personList.stream()
                .noneMatch(a -> {
                    return a.getAddress() == Person.Address.SZ;
                });
        System.out.println(b1);
```

4.4 findAny

findAny:返回流中任意元素: 默认第一个

```
// Optional<T> findAny();
// findAny: 返回任意元素(默认第一个)

        //  返回任意一个同学
        Optional<Person> any = personList.stream()
                .findAny();

        //TODO: 注意, Optional作为一个容器代表一个值存在或者不存在
        //TODO: Optional中存在几个方法, 可以让使用者显式的检查值存在或者不存在
        // <1>: isPresent()方法:  如果 Optional包含值返回true, 否则返回false
        // <2>: ifPresent(代码块)方法: 会将Optional包含的值, 传给指定的代码块
        // <3>: get()方法: 如果Optional包含值, 返回包含的值, 否则抛出异常
        // <4>: orElse(默认值):  如果Optional包含值, 返回包含的值, 否则返回默认值
        any.isPresent();
        any.ifPresent(a ->{});
```

```
        any.get();
        any.orElse(new Person("默认值", 18, 200, Person.Address.SH));

        System.out.println(any);
```

```
        // 返回任意一个身高小于170同学
        Optional<Person> any = personList.stream()
                .filter(a -> a.getHeight() < 170)
                .findAny();

        System.out.println(any);
```

4.5 findFirst

findFirst:返回第一个元素

```
//  Optional<T> findFirst();
//  findFirst: 返回任第一个元素

        //  获得年龄最小的同学
        Optional<Person> first = personList.stream()
                .sorted(Comparator.comparing(Person::getAge))
                .findFirst();

        System.out.println(first);
```

4.6 forEach

forEach: 遍历流

```
//  void forEach(Consumer<? super T> action);
//  forEach: 遍历元素(void方法)

        //  遍历列表
        personList.stream()
                .sorted(Comparator.comparing(Person::getAge))
                .forEach(a -> System.out.println(a.getName()));
        personList.stream()
                .sorted(Comparator.comparing(Person::getAge))
                .forEach(System.out::println);
```

4.7 count

count: 返回元素中数量

```
//  long count();
//  count: 计算元素个数

        //  北京同学的数量
        long count = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .count();
```

4.8 reduce

reduce: 计算元素

```
 List<Person> personList = StudentList.personList;
// TODO: 规约 reduce
//  <1>一参情况: Optional<T> reduce(BinaryOperator<T> accumulator)
//  <2>二参情况: T reduce(T identity, BinaryOperator<T> accumulator);

//  1参数:
//  返回值类型为Optional, 是应对如果流中没有任何元素情况(这种情况没有初始值就无法返回结果)
//  所以1参是把结果包裹在一个Optional对象里(可以通过get方法获取),用以表明/处理结果可能不存在情况

//  2参数:
//  BinaryOperator: 将两个元素合起来产生一个新值
//  identity: 计算的初始值/起始值(用来和第一个元素计算结果)


        //  TODO:班级同学年龄总和
        Optional<Integer> reduce1 = personList.stream()
                .map(Person::getAge)
                .reduce((a, b) -> {
                    return a + b;
                });
        Optional<Integer> reduce2 = personList.stream()
                .map(Person::getAge)
                .reduce(Integer::sum);

        System.out.println(reduce1 + "  " + reduce2);
```

```
//  TODO:返回所有同学中最大的年龄
        Optional<Integer> reduce3 = personList.stream()
                .map(Person::getAge)
                .reduce( (a, b) -> {
                    if (a > b){
                        return a;
                    }else {
                        return b;
                    }
                });

        Optional<Integer> reduce4 = personList.stream()
```

```
                .map(Person::getAge)
                .reduce( Integer::max);

        //  TODO:返回所有同学中最小的年龄
        Optional<Integer> reduce5 = personList.stream()
                .map(Person::getAge)
                .reduce( (a, b) -> {
                    if (a > b){
                        return b;
                    }else {
                        return a;
                    }
                });

        Optional<Integer> reduce6 = personList.stream()
                .map(Person::getAge)
                .reduce( Integer::min);

        System.out.println(reduce3 + " " + reduce4 + " " + reduce5 + " " +
reduce6);
```

4.9 collect

collect: 收集器, 用于收集数据经过流计算的结果

4.9.1 收集

作用是将元素分别归纳进可变容器 `List`、`Map`、`Set`、`Collection` 或者`ConcurrentMap`

```
// Collectors.toList()
// Collectors.toCollection()
// Collectors.toSet()
// Collectors.toMap()
```

```
 // 获得北京同学集合: toList
        List<Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toList());
```

```
// 获得北京同学集合: toCollection
        LinkedList<Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toCollection(LinkedList::new));
```

```
 // 获得北京同学集合: toSet
        Set<Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toSet());
```

```
// 获得北京同学集合(姓名和地址): toMap
        Map<String, Person.Address> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toMap(Person::getName, Person::getAddress));
// 获得北京同学集合(姓名和地址)
        Map<String, Person.Address> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toMap(Person::getName, a ->
a.getAddress()));
// 获得北京同学集合(姓名和对象本身)
        Map<String, Person> collect = personList.stream()
                .filter(a -> a.getAddress() == Person.Address.BJ)
                .collect(Collectors.toMap(Person::getName, a -> a));
```