Java1.8新特性 - Stream流

授课老师：学相伴飞哥

01、课程大纲

01、Stream流概述
02、Stream流的应用

02、Stream流概述

概念：Stream 是Java8 提出的一个新概念，不是输入输出的 Stream 流，而是一种用函数式编程方式在集合类上进行复杂操作的工具。简而言之，是以内部迭代的方式处理集合数据的操作，内部迭代可以将更多的控制权交给集合类。Stream 和 Iterator 的功能类似，只是 Iterator 是以外部迭代的形式处理集合数据的操作。

在Java8以前，对集合的操作需要写出处理的过程，如在集合中筛选出满足条件的数据，需要一 一遍历集合中的每个元素，再把每个元素逐一判断是否满足条件，最后将满足条件的元素保存返回。而Stream 对集合筛选的操作提供了一种更为便捷的操作，只需将实现函数接口的筛选条件作为参数传递进来，Stream会自行操作并将合适的元素同样以Stream 的方式返回，最后进行接收即可。

- 集合和数组在遍历元素的时候十分冗余，受到函数式编程以及流水线思想的启发，我们可以将常见的操作封装成比较简单的方法，比如遍历的时候直接调用一个方法即可，而不用写冗余的循环程序。这就是Stream流对象的由来。

- Stream是一个接口，有两种方式来进行创建流对象。
  一是调用 Stream.接口中的of方法。
  二是调用集合或者数组中的strain方法来获取 Stream.流对象
- Stream对象中常用的方法有：遍历元素，筛选元素，跳过元素，截取元素，计数，把流对象拼接
  对流对象中的数据元素进行转换。

03、如何学习Stream

- Stream体系非常的庞大，思维的学习方式不能够强迫自己立马都掌握，因为每个使用场景都不一样，
- 学习方案：一定总结，边用边学边积累，把能够掌握的立马掌握。

01、Stream的API

流是java API中的新的成员，它可以让你用声明式的方式处理集合，简单点说，可以看成遍历数据的一个高级点的迭代器，也可以看做一个工厂，数据处理的工厂，当然，流还天然的支持并行操作；也就不用去写复杂的多线程的代码，下面我先来看下Stream的接口定义

> *Stream*是处理集合的一套高级的*API*的解决方案

```
```

```java
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    Stream<T> filter(Predicate<? super T> predicate);

    <R> Stream<R> map(Function<? super T, ? extends R> mapper);

    IntStream mapToInt(ToIntFunction<? super T> mapper);

    LongStream mapToLong(ToLongFunction<? super T> mapper);

    DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);

    <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);

    IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);

    LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper);

    DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper);

    Stream<T> distinct();

    Stream<T> sorted();

    Stream<T> sorted(Comparator<? super T> comparator);

```

```
27      Stream<T> peek(Consumer<? super T>
    action);
28
29      Stream<T> limit(long maxSize);
30
31      Stream<T> skip(long n);
32
33      void forEach(Consumer<? super T>
    action);
34
35      void forEachOrdered(Consumer<? super T>
    action);
36
37      Object[] toArray();
38
39      <A> A[] toArray(IntFunction<A[]>
    generator);
40
41      T reduce(T identity, BinaryOperator<T>
    accumulator);
42
43      Optional<T> reduce(BinaryOperator<T>
    accumulator);
44
45      <U> U reduce(U identity, BiFunction<U,
    ? super T, U> accumulator,
    BinaryOperator<U> combiner);
46
47      <R> R collect(Supplier<R> supplier,
    BiConsumer<R, ? super T> accumulator,
    BiConsumer<R, R> combiner);
48
49      <R, A> R collect(Collector<? super T,
    A, R> collector);
50
51      Optional<T> min(Comparator<? super T>
    comparator);
52
```

```
    Optional<T> max(Comparator<? super T>
comparator);

    long count();

    boolean anyMatch(Predicate<? super T>
predicate);

    boolean allMatch(Predicate<? super T>
predicate);

    boolean noneMatch(Predicate<? super T>
predicate);

    Optional<T> findFirst();

    Optional<T> findAny();

    public static <T> Builder<T> builder()
{
        return new
Streams.streamBuilderImpl<>();
    }

    public static <T> Stream<T> empty() {
        return
StreamSupport.stream(Spliterators.<T>
emptySpliterator(), false);
    }

    public static <T> Stream<T> of(T t) {
        return StreamSupport.stream(new
Streams.streamBuilderImpl<>(t), false);
    }

    @SafeVarargs
    @SuppressWarnings("varargs") //
Creating a Stream from an array is safe
```

```
81      public static <T> Stream<T> of(T...
    values) {
82          return Arrays.stream(values);
83      }
84
85      public static <T> Stream<T>
    iterate(final T seed, final
    UnaryOperator<T> f) {
86          Objects.requireNonNull(f);
87          final Iterator<T> iterator = new
    Iterator<T>() {
88              @SuppressWarnings("unchecked")
89              T t = (T) Streams.NONE;
90
91              @Override
92              public boolean hasNext() {
93                  return true;
94              }
95
96              @Override
97              public T next() {
98                  return t = (t ==
    Streams.NONE) ? seed : f.apply(t);
99              }
100         };
101         return StreamSupport.stream(
102
    Spliterators.spliteratorUnknownSize(iterato
    r, Spliterator.ORDERED |
    Spliterator.IMMUTABLE), false);
103     }
104
105     public static <T> Stream<T>
    generate(Supplier<T> s) {
106         Objects.requireNonNull(s);
107         return StreamSupport.stream(new
    StreamSpliterators.InfiniteSupplyingSpliter
    ator.OfRef<>(Long.MAX_VALUE, s),
```

```
108                 false);
109     }
110 
111     public static <T> Stream<T>
    concat(Stream<? extends T> a, Stream<?
    extends T> b) {
112         Objects.requireNonNull(a);
113         Objects.requireNonNull(b);
114 
115         @SuppressWarnings("unchecked")
116         Spliterator<T> split = new
    Streams.ConcatSpliterator.OfRef<>
    ((Spliterator<T>) a.spliterator(),
117                 (Spliterator<T>)
    b.spliterator());
118         Stream<T> Stream =
    StreamSupport.stream(split, a.isParallel()
    || b.isParallel());
119         return
    Stream.onClose(Streams.composedClose(a,
    b));
120     }
121 
122     public interface Builder<T> extends
    Consumer<T> {
123         @Override
124         void accept(T t);
125 
126         default Builder<T> add(T t) {
127             accept(t);
128             return this;
129         }
130 
131         Stream<T> build();
132 
133     }
134 }
```

通过接口定义，可以看到，抽象方法，有30多个，里面还有一些其他的接口；后续，我会慢慢给大家介绍，每个抽象方法的作用，以及用法

02、Stream-流的常用创建方法

前面（《java8 Stream接口简介》），我们已经对Stream这个接口，做了简单的介绍，下面，我们用几个案例，来看看流的几种创建方式

1.1 使用**Collection**下的 **stream()** 和 **parallelStream()** 方法

```
List<String> list = new ArrayList<>();
Stream<String> Stream = list.stream(); //获取一个顺序流
Stream<String> parallelStream = list.parallelStream(); //获取一个并行流
```

1.2 使用**Arrays** 中的 **stream()** 方法，将数组转成流

```
Integer[] nums = new Integer[10];
Stream<Integer> Stream = Arrays.stream(nums);
```

1.3 使用**Stream**中的静态方法：**of()**、**iterate()**、**generate()**

```
Stream<Integer> Stream =
Stream.of(1,2,3,4,5,6);

Stream<Integer> Stream2 = Stream.iterate(0,
(x) -> x + 2).limit(6);
Stream2.forEach(System.out::println); // 0 2
4 6 8 10

Stream<Double> Stream3 =
Stream.generate(Math::random).limit(2);
Stream3.forEach(System.out::println);
```

1.4 使用 **BufferedReader.lines()** 方法，将每行内容转成流

```
BufferedReader reader = new
BufferedReader(new
FileReader("F:\\test_Stream.txt"));
Stream<String> lineStream = reader.lines();
lineStream.forEach(System.out::println);
```

1.5 使用 **Pattern.splitAsStream()** 方法，将字符串分隔成流

```
Pattern pattern = Pattern.compile(",");
Stream<String> stringStream =
pattern.splitAsStream("a,b,c,d");
stringStream.forEach(System.out::println);
```

03、Stream两种操作

- 中间操作

intermediate operation 中间操作：中间操作的结果是刻画、描述了一个Stream，并没有产生一个新集合，这种操作也叫做惰性求值方法。

对应的方法如下：

```
这是所有Stream中间操作的列表：
过滤()==>filter()
地图()==>map()
转换()==>flatMap()
不同()==>distinct()
排序()==>sorted()
窥视()==>peek（）
限制()==>limit()
跳跃()==>skip()
叠加()==>reduce()
```

- 终止操作

  terminal operation 终止操作：最终会从Stream中得到值。说白了：就是可以直接得到结果

```
循环()==>foreach()
总计()==>count()
收集()==>collect()---放
andMatch()
noneMatch()
allMatch()
findAny()
findFirst()
min()
max()
```

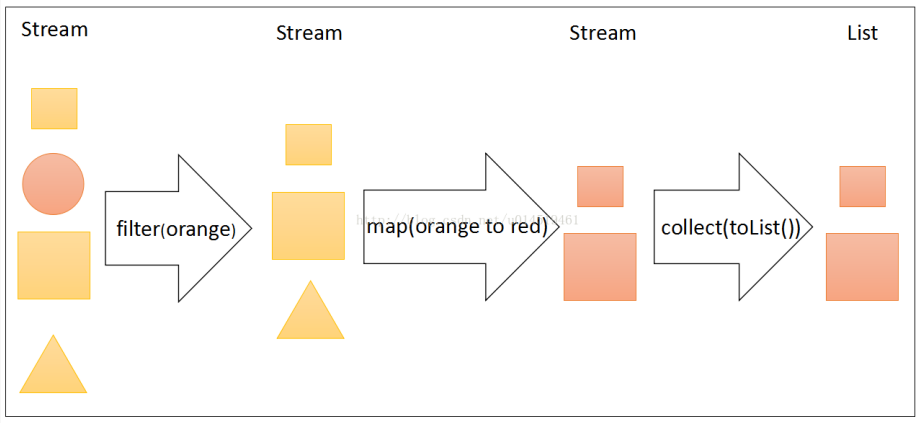
- 总结如图

表5-1 中间操作和终端操作

操 作	类 型	返回类型	使用的类型/函数式接口	函数描述符
filter	中间	Stream<T>	Predicate<T>	T -> boolean
distinct	中间 (有状态-无界)	Stream<T>		
skip	中间 (有状态-有界)	Stream<T>	long	
limit	中间 (有状态-有界)	Stream<T>	long	
map	中间	Stream<R>	Function<T, R>	T -> R
flatMap	中间	Stream<R>	Function<T, Stream<R>>	T -> Stream<R>
sorted	中间 (有状态-无界)	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	终端	boolean	Predicate<T>	T -> boolean
noneMatch	终端	boolean	Predicate<T>	T -> boolean
allMatch	终端	boolean	Predicate<T>	T -> boolean
findAny	终端	Optional<T>		
findFirst	终端	Optional<T>		
forEach	终端	void	Consumer<T>	T -> void
collect	终端	R	Collector<T, A, R>	
reduce	终端 (有状态-有界)	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	终端	long		

如何区分这2种操作呢？可以根据操作的返回值类型判断，如果返回值是Stream，则该操作是中间操作，如果返回值是其他值或者为空，则该操作是终止操作。

如下图的前2个操作是中间操作，只有最后一个操作是终止操作。



可以形象地理解Stream的操作是对一组粗糙的工艺品原型（即对应的 Stream 数据源）进行加工成颜色统一的工艺品（即最终得到的结果），第一步筛选出合适的原型（即对应Stream的filter 的方法），第二步将这些筛选出来的原型工艺品上色（对应Stream的map方法），第三步取下这些上好色的工艺品（即对应Stream的 collect(toList())方法）。在取下工艺品之前进行的操作都是中间操作，可以有多个或者0个中间操作，但每个Stream数据源只能有一次终止操作，否则程序会报错。

准备工作

```java
package com.streamdemo;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:33
 */
public class User {

    private Integer id;
    private Integer age;
    private Integer sex;
    private Double shenjia;
    private String username;
    private String password;

    public User(Integer id, String username, String password, Integer age, Integer sex, Double shenjia) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
        this.sex = sex;
        this.shenjia = shenjia;
    }

    public Integer getId() {
        return id;
    }

```

```
33      public void setId(Integer id) {
34          this.id = id;
35      }
36
37      public Integer getAge() {
38          return age;
39      }
40
41      public void setAge(Integer age) {
42          this.age = age;
43      }
44
45      public Integer getSex() {
46          return sex;
47      }
48
49      public void setSex(Integer sex) {
50          this.sex = sex;
51      }
52
53      public Double getShenjia() {
54          return shenjia;
55      }
56
57      public void setShenjia(Double shenjia) {
58          this.shenjia = shenjia;
59      }
60
61      public String getUsername() {
62          return username;
63      }
64
65      public void setUsername(String username)
    {
66          this.username = username;
67      }
68
69      public String getPassword() {
```

```
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", age=" + age +
                ", sex=" + sex +
                ", shenjia=" + shenjia +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}
```

```
package com.streamdemo;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:41
```

```
 */
public class StreamDemo {

    public static void main(String[] args) {
        List<User> userList = new
ArrayList<>();
        userList.add(new User(1, "yykk",
"111111", 34, 1, 34600d));
        userList.add(new User(2, "祈福",
"2222222", 24, 0, 883600d));
        userList.add(new User(3, "小王",
"3333333", 24, 1, 734090d));
        userList.add(new User(4, "小楠",
"4444444", 14, 0, 33400d));
        userList.add(new User(5, "小张",
"55555", 29, 1, 140000d));

    }
}

```

04、Stream的中间操作

筛选与切片

把条件满足的，过滤匹配出来。

```
filter：过滤流中的某些元素
limit(n)：获取n个元素
skip(n)：跳过n元素，配合limit(n)可实现分页
distinct：通过流中元素的 hashCode() 和 equals() 去除重复元素
```

```java
Stream<Integer> Stream = Stream.of(6, 4, 6, 7, 3, 9, 8, 10, 12, 14, 14);

Stream<Integer> newStream = Stream.filter(s -> s > 5) //6 6 7 9 8 10 12 14 14
.distinct() //6 7 9 8 10 12 14
.skip(2) //9 8 10 12 14
.limit(2); //9 8
newStream.forEach(System.out::println);
```

分析：filter语法

```java
Stream<T> filter(Predicate<? super T> predicate);
```

这个方法，传入一个Predicate的函数接口，关于Predicate函数接口定义，可以查看《JAVA8 Predicate接口》，这个接口传入一个泛型参数T，做完操作之后，返回一个boolean值；filter方法的作用，是对这个boolean做判断，返回true判断之后的对象，下面一个案例，可以看到怎么使用

案例

```java
package com.streamdemo;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
```

```
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:41
 */
public class StreamDemo {

    public static void main(String[] args) {
        List<User> userList = new ArrayList<>();
        userList.add(new User(1, "yykk", "111111", 34, 1, 34600d));
        userList.add(new User(2, "祈福", "2222222", 24, 0, 883600d));
        userList.add(new User(3, "小王", "3333333", 24, 1, 734090d));
        userList.add(new User(4, "小楠", "4444444", 14, 0, 33400d));
        userList.add(new User(5, "小张", "55555", 29, 1, 140000d));

        // 1:  filter过滤
        List<User> collect = userList.stream().filter(res -> res.getSex()==1).collect(Collectors.toList());

 collect.forEach(System.out::println);

    }
}

```

运行结果

```
User{id=1, age=34, sex=1, shenjia=34600.0, username='yykk', password='111111'}
User{id=3, age=24, sex=1, shenjia=734090.0, username='小王', password='3333333'}
User{id=5, age=29, sex=1, shenjia=140000.0, username='小张', password='55555'
```

映射方法

map：接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
flatMap：接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

```
List<String> list = Arrays.asList("a,b,c", "1,2,3");

//将每个元素转成一个新的且不带逗号的元素
Stream<String> s1 = list.stream().map(s -> s.replaceAll(",", ""));
s1.forEach(System.out::println); // abc 123

Stream<String> s3 = list.stream().flatMap(s -> {
//将每个元素转换成一个Stream
String[] split = s.split(",");
Stream<String> s2 = Arrays.stream(split);
return s2;
});
s3.forEach(System.out::println); // a b c 1 2 3
```

map方法语法

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

这个方法传入一个Function的函数式接口，接口定义可以查看《JAVA8 Function接口》，这个接口，接收一个泛型T，返回泛型R，map函数的定义，返回的流，表示的泛型是R对象，这个表示，调用这个函数后，可以改变返回的类型，先看下面的案例

案例

```
package com.streamdemo;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.function.Predicate;
import java.util.stream.Collectors;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:41
 */
public class StreamDemo {

    public static void main(String[] args) {
        List<User> userList = new ArrayList<>();
```

```
        userList.add(new User(1, "yykk", "111111", 34, 1, 34600d));
        userList.add(new User(2, "祈福", "2222222", 24, 0, 883600d));
        userList.add(new User(3, "小王", "3333333", 24, 1, 734090d));
        userList.add(new User(4, "小楠", "4444444", 14, 0, 33400d));
        userList.add(new User(5, "小张", "55555", 29, 1, 140000d));

        // 1:  使用map获取集合中username列
        List<String> userNameList = userList.stream().map(res -> res.getUsername()).collect(Collectors.toList());

 userNameList.forEach(System.out::println);

        // 2: 使用map获取集合中的id,username,age,sex,shenjia，排除password
        List<Map<String, Object>> collect = userList.stream().map(user -> {
            Map<String, Object> map = new HashMap<>();
            map.put("id",user.getId());
            map.put("age",user.getAge());
            map.put("sex",user.getSex());

 map.put("shenjia",user.getShenjia());
            return map;
        }).collect(Collectors.toList());

 collect.forEach(System.out::println);

        /*
         // 4: 使用map + min/max/count/sum快速求所有用户
```

```
45          List<Map<String, Object>> mapList =
    userList.stream().map(new Function<User,
    Map<String, Object>>() {
46              @Override
47              public Map<String, Object>
    apply(User user) {
48                  Map<String, Object> map =
    new HashMap<>();
49                  map.put("id", user.getId());
50                  map.put("age",
    user.getAge());
51                  map.put("sex",
    user.getSex());
52                  map.put("shenjia",
    user.getShenjia());
53                  return map;
54              }
55          }).collect(Collectors.toList());
56
     mapList.forEach(System.out::println);
57          */
58
59
60          // 3: 快速清空password
61          List<User> userList1 =
    userList.stream().map(user -> {
62              user.setPassword("");
63              return user;
64          }).collect(Collectors.toList());
65
     userList1.forEach(System.out::println);
66
67          // 4: 使用map + min/max/count/sum快速
    求所有用户
68          Integer maxage =
    userList.stream().map(user ->
    user.getAge()).max((a,b)->a-b).get();
```

```
69          Integer minage =
    userList.stream().map(user ->
    user.getAge()).min((a,b)->a-b).get();
70          long count =
    userList.stream().map(user ->
    user.getAge()).count();
71          long sumcount1 =
    userList.stream().map(user ->
    user.getAge()).reduce((a,b)->a+b).get();
72          long sumcount2 =
    userList.stream().mapToInt(user ->
    user.getAge()).sum();
73          System.out.println(maxage);
74          System.out.println(minage);
75          System.out.println(count);
76          System.out.println(sumcount1);
77          System.out.println(sumcount2);
78          System.out.println(sumcount1/count);
79      }
80  }
81
```

运行结果

```
1  User{id=1, age=34, sex=1, shenjia=34600.0,
   username='yykk', password='111111'}
2  User{id=3, age=24, sex=1, shenjia=734090.0,
   username='小王', password='3333333'}
3  User{id=5, age=29, sex=1, shenjia=140000.0,
   username='小张', password='55555'
```

可以看到，我们把Integer，变成了String输出，把Emp对象里的name字符串，单独输出；现在，我们只看到了一个forEach的终端操作，后面，我们会看到，更多的终端操作，把map操作后，改变的对象类型，返回各种类型的集合，或者对数字类型的，返回求和，最大，最小等的操作；

flatMap方法 语法

参考：https://www.cnblogs.com/xfyy-2020/p/13289066.html

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

这个接口，跟map一样，接收一个Fucntion的函数式接口，不同的是，Function接收的泛型参数，第二个参数是一个Stream流；方法，返回的也是泛型R，具体的作用是把两个流，变成一个流返回，下面，我们看一个案例，来详细解答，怎么把两个流的内容，变成一个流的内容

```
package com.streamdemo;

import java.util.Objects;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:33
 */
public class User {

    private Integer id;
    private Integer age;
    private Integer sex;
    private Double shenjia;
    private String username;
    private String password;

```

```
22      public User(Integer id, String
    username, String password, Integer age,
    Integer sex, Double shenjia) {
23          this.id = id;
24          this.username = username;
25          this.password = password;
26          this.age = age;
27          this.sex = sex;
28          this.shenjia = shenjia;
29      }
30
31      public Integer getId() {
32          return id;
33      }
34
35      public void setId(Integer id) {
36          this.id = id;
37      }
38
39      public Integer getAge() {
40          return age;
41      }
42
43      public void setAge(Integer age) {
44          this.age = age;
45      }
46
47      public Integer getSex() {
48          return sex;
49      }
50
51      public void setSex(Integer sex) {
52          this.sex = sex;
53      }
54
55      public Double getShenjia() {
56          return shenjia;
57      }
```

```
58
59      public void setShenjia(Double shenjia)
    {
60          this.shenjia = shenjia;
61      }
62
63      public String getUsername() {
64          return username;
65      }
66
67      public void setUsername(String
    username) {
68          this.username = username;
69      }
70
71      public String getPassword() {
72          return password;
73      }
74
75      public void setPassword(String
    password) {
76          this.password = password;
77      }
78
79      @Override
80      public String toString() {
81          return "User{" +
82                  "id=" + id +
83                  ", age=" + age +
84                  ", sex=" + sex +
85                  ", shenjia=" + shenjia +
86                  ", username='" + username +
    '\'' +
87                  ", password='" + password +
    '\'' +
88                  '}';
89      }
90
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() !=
o.getClass()) return false;
        User user = (User) o;
        return Objects.equals(id, user.id)
&&
                Objects.equals(age,
user.age) &&
                Objects.equals(sex,
user.sex) &&
                Objects.equals(shenjia,
user.shenjia) &&
                Objects.equals(username,
user.username) &&
                Objects.equals(password,
user.password);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, age, sex,
shenjia, username, password);
    }
}

```

案例

```
package com.streamdemo;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```java
import java.util.stream.stream;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/10/12 14:41
 */
public class StreamDemo2 {

    public static void main(String[] args) {
        List<User> userList = new ArrayList<>();
        userList.add(new User(1, "yykk", "111111", 34, 1, 34600d));
        userList.add(new User(2, "祈福", "2222222", 24, 0, 883600d));
        userList.add(new User(3, "小王", "3333333", 24, 1, 734090d));
        userList.add(new User(4, "小楠", "4444444", 14, 0, 33400d));
        userList.add(new User(5, "小张", "55555", 29, 1, 140000d));

        List<User> userList2 = new ArrayList<>();
        userList2.add(new User(1, "yykk", "111111", 34, 1, 34600d));
        userList2.add(new User(7, "祈福", "2222222", 24, 0, 883600d));
        userList2.add(new User(8, "小王", "3333333", 24, 1, 734090d));
        userList2.add(new User(9, "小楠", "4444444", 14, 0, 33400d));
```

```
31          userList2.add(new User(10, "小张",
"55555", 29, 1, 140000d));
32
33
34          //flatmap一般用于：
35          // 使用Java8实现集合的并、交、差操作
36          // 具体的作用是把两个流，变成一个流返回
37
38          //案例一：合并两个集合
39          System.out.println("============案例
一 合并==============");
40          List<User> collect =
userList.stream().flatMap(user-
>userList2.stream()).collect(Collectors.toLi
st());
41
 collect.forEach(System.out::println);
42
43          System.out.println("============案例
二取交集==============");
44          // 案例二取交集
45          List<User> userList1 =
userList.stream().filter(userList2::contains
).collect(Collectors.toList());
46
 userList1.forEach(System.out::println);
47
48          System.out.println("===========案例三
取并集==============");
49          // 案例三取并集
50          List<User> userList3=
Stream.of(userList,userList2).flatMap(Collec
tion::Stream).distinct().collect(Collectors.
toList());
51
 userList3.forEach(System.out::println);
52
```

```
53          System.out.println("========== 案例四取差集==============");
54          // 案例四 取差集
55          List<User> userList4= userList.stream().filter(user -> !userList2.contains(user)).collect(Collectors.toList());
56
 userList4.forEach(System.out::println);
57
58          int sum = Stream.of(userList,userList2).flatMapToInt(user -> user.stream().mapToInt(u->u.getAge())).sum();
59          System.out.println(sum);
60
61      }
62  }
63
```

运行结果

```
1
```

排序

语法

```
1  sorted()：自然排序，流中元素需实现Comparable接口
2  sorted(Comparator com)：定制排序，自定义Comparator排序器
```

案例

```
List<String> list = Arrays.asList("aa", "ff", "dd");
//String 类自身已实现Compareable接口
list.stream().sorted().forEach(System.out::println);// aa dd ff

Student s1 = new Student("aa", 10);
Student s2 = new Student("bb", 20);
Student s3 = new Student("aa", 30);
Student s4 = new Student("dd", 40);
List<Student> studentList = Arrays.asList(s1, s2, s3, s4);

//自定义排序：先按姓名升序，姓名相同则按年龄升序
studentList.stream().sorted(
(o1, o2) -> {
if (o1.getName().equals(o2.getName())) {
return o1.getAge() - o2.getAge();
} else {
return o1.getName().compareTo(o2.getName());
}
}
).forEach(System.out::println);
```

消费

概述

peek：如同于map，能得到流中的每一个元素。但map接收的是一个Function表达式，有返回值；而peek接收的是Consumer表达式，没有返回值。

案例

```
Student s1 = new Student("aa", 10);
Student s2 = new Student("bb", 20);
List<Student> studentList =
Arrays.asList(s1, s2);

studentList.stream()
.peek(o -> o.setAge(100))
.forEach(System.out::println);

//结果:
Student{name='aa', age=100}
Student{name='bb', age=100}
```

其他的中间操作-改造变换（重要）

下面，我们来看其他的剩余的一些中间操作，各自的作用，我也通过注释，做了解析，方法定义如下；

语法

```java
//去重复
Stream<T> distinct();
//排序
Stream<T> sorted();
//根据属性排序
Stream<T> sorted(Comparator<? super T>
comparator);
//对对象的进行操作
Stream<T> peek(Consumer<? super T> action);
//截断--取先maxSize个对象
Stream<T> limit(long maxSize);
//截断--忽略前N个对象
Stream<T> skip(long n);
```

下面，我们用一些案例，对这些操作，做一些综合的演示

案例

```java
package com.taihao;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.stream;

public class TestJava8 {
    public static List<Emp> list = new
ArrayList<>();
    static {
        list.add(new Emp("xiaoHong1", 20,
1000.0));
        list.add(new Emp("xiaoHong2", 25,
2000.0));
        list.add(new Emp("xiaoHong3", 30,
3000.0));
```

```
15          list.add(new Emp("xiaoHong4", 35,
    4000.0));
16          list.add(new Emp("xiaoHong5", 38,
    5000.0));
17          list.add(new Emp("xiaoHong6", 45,
    9000.0));
18          list.add(new Emp("xiaoHong7", 55,
    10000.0));
19          list.add(new Emp("xiaoHong8", 42,
    15000.0));
20      }
21
22      public static void println(Stream<Emp>
    Stream) {
23          Stream.forEach(emp -> {
24
    System.out.println(String.format("名字：%s，年
    纪：%s，薪水：%s", emp.getName(),
    emp.getAge(), emp.getSalary()));
25          });
26      }
27
28      public static void main(String[] args) {
29          // 对数组流，先过滤重复，在排序，再控制台输
    出 1，2，3
30          Arrays.asList(3, 1, 2,
    1).stream().distinct().sorted().forEach(str
    -> {
31              System.out.println(str);
32          });
33          // 对list里的emp对象，取出薪水，并对薪水进
    行排序，然后输出薪水的内容，map操作，改变了Strenm的
    泛型对象
34          list.stream().map(emp ->
    emp.getSalary()).sorted().forEach(salary ->
    {
35              System.out.println(salary);
36          });
```

```
37          // 根据emp的属性name，进行排序
38
    println(list.stream().sorted(Comparator.comp
    aring(Emp::getName)));
39
40          // 给年纪大于30岁的人，薪水提升1.5倍，并输
    出结果
41          Stream<Emp> Stream =
    list.stream().filter(emp -> {
42              return emp.getAge() > 30;
43          }).peek(emp -> {
44              emp.setSalary(emp.getSalary() *
    1.5);
45          });
46          println(Stream);
47          // 数字从1开始迭代（无限流），下一个数字，
    是上个数字+1，忽略前5个 ，并且只取10个数字
48          // 原本1-无限，忽略前5个，就是1-5数字，不
    要，从6开始，截取10个，就是6-15
49          Stream.iterate(1, x ->
    ++x).skip(5).limit(10).forEach(System.out::p
    rintln);
50      }
51
52      public static class Emp {
53          private String name;
54
55          private Integer age;
56
57          private Double salary;
58
59          public Emp(String name, Integer age,
    Double salary) {
60              super();
61              this.name = name;
62              this.age = age;
63              this.salary = salary;
64          }
```

```
65
66          public String getName() {
67              return name;
68          }
69
70          public void setName(String name) {
71              this.name = name;
72          }
73
74          public Integer getAge() {
75              return age;
76          }
77
78          public void setAge(Integer age) {
79              this.age = age;
80          }
81
82          public Double getSalary() {
83              return salary;
84          }
85
86          public void setSalary(Double salary)
    {
87              this.salary = salary;
88          }
89
90      }
91  }
```

运行结果

```
1
```

每个例子，也都加了注释，大家看例子，自己get吧

05、Stream 流的终止操作

01、匹配、聚合操作

把条件满足的，过滤匹配出来。

语法

```
allMatch：接收一个 Predicate 函数，当流中每个元素都符合该断言时才返回true，否则返回false
noneMatch：接收一个 Predicate 函数，当流中每个元素都不符合该断言时才返回true，否则返回false
anyMatch：接收一个 Predicate 函数，只要流中有一个元素满足该断言则返回true，否则返回false
findFirst：返回流中第一个元素
findAny：返回流中的任意元素
count：返回流中元素的总个数
max：返回流中元素最大值
min：返回流中元素最小值
```

案例

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

boolean allMatch = list.stream().allMatch(e -> e > 10); //false
boolean noneMatch = list.stream().noneMatch(e -> e > 10); //true
boolean anyMatch = list.stream().anyMatch(e -> e > 4); //true

Integer findFirst = list.stream().findFirst().get(); //1
Integer findAny = list.stream().findAny().get(); //1

long count = list.stream().count(); //5
Integer max = list.stream().max(Integer::compareTo).get(); //5
Integer min = list.stream().min(Integer::compareTo).get(); //1
```

运行结果

```
```

02、规约操作

Optional reduce(BinaryOperator accumulator)：第一次执行时，accumulator函数的第一个参数为流中的第一个元素，第二个参数为流中元素的第二个元素；第二次执行时，第一个参数为第一次函数执行的结果，第二个参数为流中的第三个元

素；依次类推。

T reduce(T identity, BinaryOperator accumulator)：流程跟上面一样，只是第一次执行时，accumulator函数的第一个参数为identity，而第二个参数为流中的第一个元素。

U reduce(U identity,BiFunction<U, ? super T, U> accumulator,BinaryOperator combiner)：在串行流(Stream)中，该方法跟第二个方法一样，即第三个参数combiner不会起作用。在并行流(parallelStream)中,我们知道流被fork join出多个线程进行执行，此时每个线程的执行流程就跟第二个方法reduce(identity,accumulator)一样，而第三个参数combiner函数，则是将每个线程的执行结果当成一个新的流，然后使用第一个方法reduce(accumulator)流程进行规约。

案例

```
//经过测试，当元素个数小于24时，并行时线程数等于元素个数，当大于等于24时，并行时线程数为16
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24);

Integer v = list.stream().reduce((x1, x2) -> x1 + x2).get();
System.out.println(v); // 300

Integer v1 = list.stream().reduce(10, (x1, x2) -> x1 + x2);
System.out.println(v1); //310

Integer v2 = list.stream().reduce(0,
(x1, x2) -> {
System.out.println("Stream accumulator: x1:" + x1 + " x2:" + x2);
return x1 - x2;
},
(x1, x2) -> {
```

```
System.out.println("Stream combiner: x1:" +
x1 + " x2:" + x2);
return x1 * x2;
});
System.out.println(v2); // -300

Integer v3 = list.parallelStream().reduce(0,
(x1, x2) -> {
System.out.println("parallelStream
accumulator: x1:" + x1 + " x2:" + x2);
return x1 - x2;
},
(x1, x2) -> {
System.out.println("parallelStream combiner:
x1:" + x1 + " x2:" + x2);
return x1 * x2;
});
System.out.println(v3); //197474048
```

运行结果

```
1
```

03、收集操作

```
collect：接收一个Collector实例，将流中元素收集成另外一个数据结构。
Collector<T, A, R> 是一个接口，有以下5个抽象方法：
Supplier<A> supplier()：创建一个结果容器A
BiConsumer<A, T> accumulator()：消费型接口，第一个参数为容器A，第二个参数为流中元素T。
BinaryOperator<A> combiner()：函数接口，该参数的作用跟上一个方法(reduce)中的combiner参数一样，将并行流中各 个子进程的运行结果(accumulator函数操作后的容器A)进行合并。
Function<A, R> finisher()：函数式接口，参数为：容器A，返回类型为：collect方法最终想要的结果R。
Set<Characteristics> characteristics()：返回一个不可变的Set集合，用来表明该Collector的特征。有以下三个特征：
CONCURRENT：表示此收集器支持并发。（官方文档还有其他描述，暂时没去探索，故不作过多翻译）
UNORDERED：表示该收集操作不会保留流中元素原有的顺序。
IDENTITY_FINISH：表示finisher参数只是标识而已，可忽略。
```

3.3.1 Collector 工具库：Collectors

```
Student s1 = new Student("aa", 10,1);
Student s2 = new Student("bb", 20,2);
Student s3 = new Student("cc", 10,3);
List<Student> list = Arrays.asList(s1, s2, s3);

//装成list
List<Integer> ageList = list.stream().map(Student::getAge).collect(Collectors.toList()); // [10, 20, 10]

//转成set
```

```
Set<Integer> ageSet = list.stream().map(Student::getAge).collect(Collectors.toSet()); // [20, 10]

//转成map,注:key不能相同，否则报错
Map<String, Integer> studentMap = list.stream().collect(Collectors.toMap(Student::getName, Student::getAge)); // {cc=10, bb=20, aa=10}

//字符串分隔符连接
String joinName = list.stream().map(Student::getName).collect(Collectors.joining(",", "(", ")")); // (aa,bb,cc)

//聚合操作
//1.学生总数
Long count = list.stream().collect(Collectors.counting()); // 3
//2.最大年龄 (最小的minBy同理)
Integer maxAge = list.stream().map(Student::getAge).collect(Collectors.maxBy(Integer::compare)).get(); // 20
//3.所有人的年龄
Integer sumAge = list.stream().collect(Collectors.summingInt(Student::getAge)); // 40
//4.平均年龄
Double averageAge = list.stream().collect(Collectors.averagingDouble(Student::getAge)); // 13.333333333333334
// 带上以上所有方法
```

```
28 DoubleSummaryStatistics statistics =
   list.stream().collect(Collectors.summarizing
   Double(Student::getAge));
29 System.out.println("count:" +
   statistics.getCount() + ",max:" +
   statistics.getMax() + ",sum:" +
   statistics.getSum() + ",average:" +
   statistics.getAverage());
30
31 //分组
32 Map<Integer, List<Student>> ageMap =
   list.stream().collect(Collectors.groupingBy(
   Student::getAge));
33 //多重分组,先根据类型分再根据年龄分
34 Map<Integer, Map<Integer, List<Student>>>
   typeAgeMap =
   list.stream().collect(Collectors.groupingBy(
   Student::getType,
   Collectors.groupingBy(Student::getAge)));
35
36 //分区
37 //分成两部分，一部分大于10岁，一部分小于等于10岁
38 Map<Boolean, List<Student>> partMap =
   list.stream().collect(Collectors.partitionin
   gBy(v -> v.getAge() > 10));
39
40 //规约
41 Integer allAge =
   list.stream().map(Student::getAge).collect(C
   ollectors.reducing(Integer::sum)).get();
   //40
```

3.3.2 Collectors.toList() 解析

```
1 //toList 源码
2 public static <T> Collector<T, ?, List<T>>
   toList() {
```

```
3  return new CollectorImpl<>
   ((Supplier<List<T>>) ArrayList::new,
   List::add,
4  (left, right) -> {
5  left.addAll(right);
6  return left;
7  }, CH_ID);
8  }
9
10 //为了更好地理解，我们转化一下源码中的lambda表达式
11 public <T> Collector<T, ?, List<T>> toList()
   {
12 Supplier<List<T>> supplier = () -> new
   ArrayList();
13 BiConsumer<List<T>, T> accumulator = (list,
   t) -> list.add(t);
14 BinaryOperator<List<T>> combiner = (list1,
   list2) -> {
15 list1.addAll(list2);
16 return list1;
17 };
18 Function<List<T>, List<T>> finisher = (list)
   -> list;
19 Set<Collector.Characteristics>
   characteristics =
   Collections.unmodifiableSet(EnumSet.of(Colle
   ctor.Characteristics.IDENTITY_FINISH));
20
21 return new Collector<T, List<T>, List<T>>()
   {
22 @Override
23 public Supplier supplier() {
24 return supplier;
25 }
26
27 @Override
28 public BiConsumer accumulator() {
29 return accumulator;
```

```
30  }
31
32  @Override
33  public BinaryOperator combiner() {
34  return combiner;
35  }
36
37  @Override
38  public Function finisher() {
39  return finisher;
40  }
41
42  @Override
43  public Set<Characteristics>
    characteristics() {
44  return characteristics;
45  }
46  };
47  }
```

参考文献

> 参考网址：
>
> *https://blog.csdn.net/qq_28410283/article/details/80642786*
>
> *https://www.cnblogs.com/owenma/p/12207330.html*