java8新特性

java8新特性

> 基于尚硅谷java8新特性视频

1. 速度更快

对底层数据结构进行了改动，优化了性能

1.1 HashMap

1.1.1 不用hash表存储

如果HashMap存储元素时不采用hash表。由于HashMap是无序存储，当有新元素要添加进来时，新元素必须通过equal方法和集合中的每个元素一一比较，如果没有与之相同的元素则存入，否则就替换。显然，在集合中元素很大的情况下这种比较方法的效率是极慢的。

1.1.2 1.8之前的HashMap

1.8之前的HashMap采用**数组+链表+hash表**的方式来优化操作。默认为16长的数组，每个数组元素中都存放一个entry。当添加元素时，会调用对象的hashcode方法，该方法会根据hash算法生成一个数组的索引值；然后判断该下标的数组中是否有对象，如果没有则直接存储；如果有，则再通过两个对象的equals方法比较内容，如果内容一样则替换该key的value值，如果内容不一样，则形成链表，**且后加入的元素被放在前面**，这种情况又叫做hash碰撞。

显然，hash碰撞是需要尽量避免的一种现象。否则会使链表的长度不断增长，比较的次数不断增大。所以在自定义类时都会重写hashcode和equals方法，且保证对象的equals值一样，则生成的hashcode值也一样。

但是由于数组的长度是有限的，而元素是无限的。所以hash碰撞是无法完全避免的。为了尽量减小hash碰撞的次数，HashMap提供了一个**0.75的扩容因子**，该扩容因子的意思是在存放新值时如果发生了hash碰撞，且此时容器中当前已有元素的个数大于阈值时数组才会进行2倍扩容。扩容后，原数组中的每个元素都会被重新计算hash值后再分配到相应的位置。

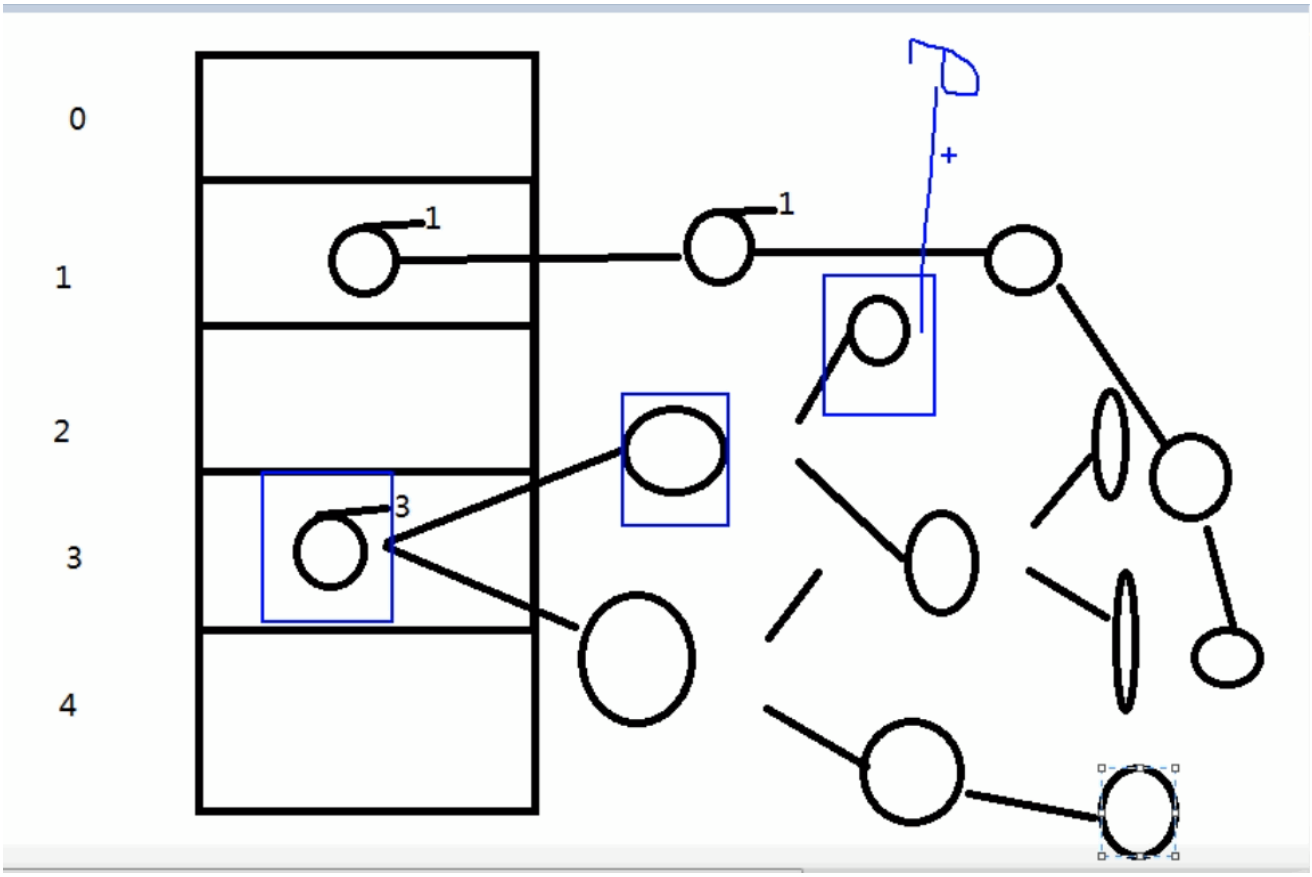
> （1）hashmap在存值的时，可能达到最后存满16个值的时候，再存入第17个值才会发生扩容现象，因为前16个值，每个值在底层数组中分别占据一个位置，并没有发生hash碰撞。
>
> （2）当然也有可能存储更多值（超多16个值，最多可以存26个值）都还没有扩容。原理：前11个值全部hash碰撞，存到数组的同一个位置（这时元素个数小于阈值12，不会扩容），后面所有存入的15个值全部分散到数组剩下的15个位置（这时元素个数大于等于阈值，但是每次存入的元素并没有发生hash碰撞，所以不会扩容），前面11+15=26，所以在存入第27个值的时候才同时满足上面两个条件，这时候才会发生扩容现象。

1.1.3 1.8之后的HashMap

1.8之前发生碰撞的极端情况是需要和链表中的每个元素比较。1.8之后使用数组+链表+红黑树(二叉树的一种)来优化HashMap。

其他条件不变，在发生hash碰撞次数超过8时（链表长度大于8，且添加方式改为末尾添加），且集合中总元素大于64时，会将原本的链表结构转为红黑树结构。红黑树除了添加操作外，其他操作都比链表快。

如下图，在极端情况下进行插入，如果是链表结构，则要判断8次，然后插入到末尾；如果是红黑树结构，则可能只需要判断3次即可。



而且在数组扩容后，红黑树中的某些元素无需再重新计算hash值，直接移动到原来数组总长度+原来的下标值处即可。比如图中数组进行扩容后，原来下标为3的红黑树直接移动到下标为5+3=8的位置即可。

1.2 HashSet

同hashmap

1.3 ConcurrentHashMap

1.8之前采用锁分段机制使得ConcurrentHashMap默认支持16个并发。由于并发隔离级别（默认为16）不是很好评定，**1.8之后废除了锁分段机制，改成了CAS算法。**

1.4 JVM

Hotspot虚拟机取消了永久区（PremGen），使用Metaspace(元空间)来替代。Metaspace的大小等于物理内存，机器的内存多大，元空间的空间就有多大，这样使得JVM进行GC的概率大大降低，OOM异常的概率大大降低。

原先的PremGenSize参数和MaxPremGenSize参数也无效了，取而代之是MetaspaceSize和MaxMetaspaceSize。默认和物理内存大小相同。

2. 代码更少（Lambda表达式）

Lambda是一个匿名函数，可以把Lambda表达式理解为是一段可以传递的代码（将代码像数据一样进行传递）。可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java的语言表达能力得到了提升。

```
public class TestLambda {
     List<Employee> employees=Arrays.asList(
             new Employee("张三",18,9496.2),
             new Employee("李四",52,2396.2),
             new Employee("王五",56,996.2),
             new Employee("赵六",8,94.2)
     );

     //原来的匿名内部类
     @Test
     public void test1(){
         Comparator<Integer> com=new Comparator<Integer>(){
             @Override
             public int compare(Integer o1, Integer o2) {
                 return Integer.compare(o1, o2);
             }
         };
         TreeSet<Integer> ts=new TreeSet<>(com);
     }

     //Lambda表达式
     @Test
     public void test2(){
         Comparator<Integer> com=(x,y)->Integer.compare(x,y);
         TreeSet<Integer> ts=new TreeSet<>(com);
     }

     @Test
     public void test3(){
         //需求：获取当前公司中员工年龄大于35的员工信息
         List<Employee> emps=filterEmplyees1(employees);
         for(Employee e:emps){
             System.out.println(e);
         }
         System.out.println("----------------------");

         //需求：获取当前公司中员工工资大于2000的员工信息
         List<Employee> emps2=filterEmplyees2(employees);
         for(Employee e:emps2){
             System.out.println(e);
         }
     }

     public List<Employee> filterEmplyees1(List<Employee> list){
         List<Employee> emps=new ArrayList<Employee>();
         for (Employee emp : list) {
             if(emp.getAge()>=35){
```

```
                emps.add(emp);
            }
        }
        return emps;
    }

    public List<Employee> filterEmplyees2(List<Employee> list){
        List<Employee> emps=new ArrayList<Employee>();
        for (Employee emp : list) {
            if(emp.getSalary()>=2000){
                emps.add(emp);
            }
        }
        return emps;
    }

    @Test
    public void test4(){
        List<Employee> emps=filterEmplyees(employees,new FilterEmployeeByAge());
        for(Employee e:emps){
            System.out.println(e);
        }
        System.out.println("----------------------");
        List<Employee> emps2=filterEmplyees(employees,new
FilterEmployeeBySalary());
        for(Employee e:emps2){
            System.out.println(e);
        }
    }

    //优化方式一：策略设计模式
    public List<Employee> filterEmplyees(List<Employee> list,MyPredicate<Employee>
myPredicate){
        List<Employee> emps=new ArrayList<Employee>();
        for (Employee emp : list) {
            if(myPredicate.test(emp)){
                emps.add(emp);
            }
        }
        return emps;
    }

    //优化方式二：匿名内部类
    @Test
    public void test5(){
        List<Employee> list=filterEmplyees(employees, new MyPredicate<Employee>()
{
            @Override
            public boolean test(Employee t) {
                return t.getSalary()>=2000;
            }
        });

```

```
        for (Employee employee : list) {
            System.out.println(employee);
        }
    }

    //优化方式三：Lambda表达式
    @Test
    public void test6(){
        List<Employee> list=filterEmplyees(employees, (e)->e.getSalary()>=2000);
        list.forEach(System.out::println);
    }

    //优化方式四：stream API
    @Test
    public void test7(){
        employees.stream()
                .filter((e)->e.getSalary()>=2000)
                .forEach(System.out::println);

        System.out.println("------------------");

        employees.stream()
                .map(Employee::getName)
                .forEach(System.out::println);
    }
}
```

MyPredicate.java

```
public interface MyPredicate<T> {
    public boolean test(T t);
}
```

FilterEmployeeBySalary.java

```
public class FilterEmployeeBySalary implements MyPredicate<Employee>{
    @Override
    public boolean test(Employee t) {
        return t.getSalary()>=2000;
    }
}
```

> “**语法糖**”是指使用更加方便，但是原理不变的代码语法。例如在遍历集合时使用的for-each语法，其实底层的实现原理仍然是迭代器，这便是“语法糖”。从应用层面来讲，Java中的Lambda可以被当做是匿名内部类的“语法糖”，但是二者在原理上是不同的。

2.1 基础语法

Java8中引入了一个新的操作符”->” 该操作符称为箭头操作符或Lambda操作符，箭头操作符将Lambda表达式拆分成两部分： 左侧：Lambda 表达式的**参数列表** 右侧：Lambda 表达式中所需执行的功能，即 **Lambda 体**

2.1.1 无参数，无返回值

()->System.out.println(“Hello Lambda!”);

```
@Test
public void test1(){
    //通过匿名内部类的方式实现接口
    Runnable r=new Runnable() {
        @Override
        public void run() {
            System.out.println("Hello World!");
        }
    };
    r.run();

    System.out.println("----------------------");
    //匿名内部类用代替匿名内部类
    Runnable r1=()->System.out.println("Hello Lambda!");
    r1.run();
}
```

2.1.2 有一个参数，并且无返回值

(x)->System.out.println(x);

```
@Test
public void test2(){
  //对Consumer接口中有一个参数的accept方法的实现
  Consumer<String> con=(x)->System.out.println(x);
  con.accept("啦啦啦");
}
```

2.1.3 若只有一个参数，小括号可以不写

x->System.out.println(x);

2.1.4 有两个以上的参数，有返回值，并且Lambda体中有多条语句

```
@Test
public void test3(){
    Comparator<Integer> com=(x,y)->{
        System.out.println("函数式接口");
        return Integer.compare(x, y);
    };
}
```

2.1.5 若Lambda体中只有一条语句，大括号和 return 都可以省略不写

```java
@Test
public void test4(){
    Comparator<Integer> com=(x,y)->Integer.compare(x, y);
}
```

2.1.6 省略参数

Lambda表达式的参数列表的数据类型可以省略不写，因为JVM编译器通过上下文推断出，数据类型，即“类型推断”。(Integer x,Integer y)->Integer.compare(x,y);

2.2 函数式接口

只包含一个抽象方法的接口，称为 函数式接口。

可以通过 Lambda 表达式来创建该接口的对象。（若 Lambda表达式抛出一个受检异常，那么该异常需要在目标接口的抽象方法上进行声明）。

可以在任意函数式接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口，同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。

作为参数传递 Lambda 表达式：为了将 Lambda 表达式作为参数传递，接收 Lambda 表达式的参数类型必须是与该 Lambda 表达式兼容的函数式接口的类型。

2.2.1 快速入门

定义函数式接口

```java
@FunctionalInterface
public interface MyFun {
    public Integer getValue(Integer num);
}
```

使用函数式接口

```java
//需求：对一个数进行运算
@Test
public void  test6(){
    Integer num=operation(100, (x)->x*x);
    System.out.println(num);

    System.out.println(operation(200, (y)->y+200));
}

public Integer operation(Integer num,MyFun mf){
    return mf.getValue(num);
}
```

使用函数式接口

```java
// 定制排序比较两个Employee(先按年龄比，年龄相同按姓名比)
List<Employee> employees=Arrays.asList(
            new Employee("张三",18,9496.2),
            new Employee("李四",52,2396.2),
            new Employee("王五",56,996.2),
            new Employee("赵六",8,94.2)
    );

@Test
public void test1(){
    Collections.sort(employees, (e1,e2)->{
        if(e1.getAge()==e2.getAge()){
            return e1.getName().compareTo(e2.getName());
        }else{
            return Integer.compare(e1.getAge(), e2.getAge());
        }
    });

    for (Employee employee : employees) {
        System.out.println(employee);
    }
}
```

2.3 4大核心内置函数式接口

2.3.1 Supplier接口

`java.util.function.Supplier<T>`接口仅包含一个无参的方法：`T get()`。 用来获取一个泛型参数指定类型的对象数据。由于这是一个函数式接口，这也就意味着对应的Lambda表达式需要“**对外提供**”一个符合泛型类型的对象数据。

```java
import java.util.function.Supplier;

public class Demo08Supplier {
    private static String getString(Supplier<String> function) {
        return function.get();
    }

    public static void main(String[] args) {
        String msgA = "Hello";
        String msgB = "World";
        System.out.println(getString(() -> msgA + msgB));
    }
}
```

题目：求数组元素最大值

使用`Supplier`接口作为方法参数类型，通过Lambda表达式求出int数组中的最大值。提示：接口的泛型请使用`java.lang.Integer`类。

解答

```
public class Demo02Test {
    //定一个方法,方法的参数传递Supplier,泛型使用Integer
    public static int getMax(Supplier<Integer> sup){
        return sup.get();
    }

    public static void main(String[] args) {
        int arr[] = {2,3,4,52,333,23};

        //调用getMax方法,参数传递Lambda
        int maxNum = getMax(()->{
           //计算数组的最大值
           int max = arr[0];
           for(int i : arr){
               if(i>max){
                   max = i;
               }
           }
           return max;
        });
        System.out.println(maxNum);
    }
}
```

2.3.2 Consumer接口

`java.util.function.Consumer<T>`接口则正好与Supplier接口相反，它不是生产一个数据，而是**消费**一个数据，其数据类型由泛型决定。

抽象方法：accept

`Consumer`接口中包含抽象方法`void accept(T t)`，意为消费一个指定泛型的数据。基本使用如：

```
import java.util.function.Consumer;

public class Demo09Consumer {
    private static void consumeString(Consumer<String> function) {
        function.accept("Hello");
    }

    public static void main(String[] args) {
        consumeString(s -> System.out.println(s));
    }
}
```

当然，更好的写法是使用方法引用。

默认方法：andThen

如果一个方法的参数和返回值全都是`Consumer`类型，那么就可以实现效果：消费数据的时候，首先做一个操作，然后再做一个操作，实现组合。而这个方法就是`Consumer`接口中的default方法`andThen`。下面是JDK的源代码：

```
default Consumer<T> andThen(Consumer<? super T> after) {
    Objects.requireNonNull(after);
    return (T t) -> { accept(t); after.accept(t); };
}
```

> 备注：`java.util.Objects`的`requireNonNull`静态方法将会在参数为null时主动抛出`NullPointerException`异常。这省去了重复编写if语句和抛出空指针异常的麻烦。

要想实现组合，需要两个或多个Lambda表达式即可，而`andThen`的语义正是“一步接一步”操作。例如两个步骤组合的情况：

```
import java.util.function.Consumer;

public class Demo10ConsumerAndThen {
    private static void consumeString(Consumer<String> one, Consumer<String> two) {
        one.andThen(two).accept("Hello");
    }

    public static void main(String[] args) {
        consumeString(
            s -> System.out.println(s.toUpperCase()),
            s -> System.out.println(s.toLowerCase()));
    }
}
```

运行结果将会首先打印完全大写的HELLO，然后打印完全小写的hello。当然，通过链式写法可以实现更多步骤的组合。

题目：格式化打印信息

下面的字符串数组当中存有多条信息，请按照格式“`姓名：XX。性别：XX。`”的格式将信息打印出来。要求将打印姓名的动作作为第一个`Consumer`接口的Lambda实例，将打印性别的动作作为第二个`Consumer`接口的Lambda实例，将两个`Consumer`接口按照顺序“拼接”到一起。

```
public static void main(String[] args) {
    String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };
}
```

解答

```
import java.util.function.Consumer;

public class DemoConsumer {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };
        printInfo(s -> System.out.print("姓名：" + s.split(",")[0]),
```

```
                    s -> System.out.println("。性别：" + s.split(",")[1] + "。"),
                    array);
    }

    private static void printInfo(Consumer<String> one, Consumer<String> two,
String[] array) {
        for (String info : array) {
            one.andThen(two).accept(info); // 姓名：迪丽热巴。性别：女。
        }
    }
}
```

2.3.3 Predicate接口

有时候我们需要对某种类型的数据进行判断，从而得到一个boolean值结果。这时可以使用`java.util.function.Predicate<T>`接口。

抽象方法：test

`Predicate`接口中包含一个抽象方法：`boolean test(T t)`。用于条件判断的场景：

```
import java.util.function.Predicate;

public class Demo15PredicateTest {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }

    public static void main(String[] args) {
        method(s -> s.length() > 5);
    }
}
```

条件判断的标准是传入的Lambda表达式逻辑，只要字符串长度大于5则认为很长。

默认方法：and

既然是条件判断，就会存在与、或、非三种常见的逻辑关系。其中将两个`Predicate`条件使用“与”逻辑连接起来实现“**并且**”的效果时，可以使用default方法`and`。其JDK源码为：

```
default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
}
```

如果要判断一个字符串既要包含大写“H”，又要包含大写“W”，那么：

```
import java.util.function.Predicate;

public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.and(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }

    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：or

与 `and` 的“与”类似，默认方法 `or` 实现逻辑关系中的“**或**”。JDK源码为：

```
default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}
```

如果希望实现逻辑“字符串包含大写H或者包含大写W”，那么代码只需要将“and”修改为“or”名称即可，其他都不变：

```
import java.util.function.Predicate;

public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.or(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }

    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：negate

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。默认方法 `negate` 的JDK源代码为：

```
default Predicate<T> negate() {
    return (t) -> !test(t);
}
```

从实现中很容易看出，它是执行了test方法之后，对结果boolean值进行“!”取反而已。一定要在 `test` 方法调用之前调用 `negate` 方法，正如 `and` 和 `or` 方法一样：

```java
import java.util.function.Predicate;

public class Demo17PredicateNegate {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.negate().test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }

    public static void main(String[] args) {
        method(s -> s.length() < 5);
    }
}
```

题目：集合信息筛选

数组当中有多条“姓名+性别”的信息如下，请通过`Predicate`接口的拼装将符合要求的字符串筛选到集合`ArrayList`中，需要同时满足两个条件：

1. 必须为女生；
2. 姓名为4个字。

```java
public class DemoPredicate {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
    }
}
```

解答

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class DemoPredicate {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
        List<String> list = filter(array,
                                   s -> "女".equals(s.split(",")[1]),
                                   s -> s.split(",")[0].length() == 4);
        System.out.println(list);
    }

    private static List<String> filter(String[] array, Predicate<String> one,
                                       Predicate<String> two) {
        List<String> list = new ArrayList<>();
        for (String info : array) {
            if (one.and(two).test(info)) {
                list.add(info);
            }
        }
        return list;
    }
```

```
24 }
```

2.3.4 Function接口

`java.util.function.Function<T,R>`接口用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件。

抽象方法：apply

`Function`接口中最主要的抽象方法为：`R apply(T t)`，根据类型T的参数获取类型R的结果。

使用的场景例如：将`String`类型转换为`Integer`类型。

```
1  import java.util.function.Function;
2
3  public class Demo11FunctionApply {
4      private static void method(Function<String, Integer> function) {
5          int num = function.apply("10");
6          System.out.println(num + 20);
7      }
8
9      public static void main(String[] args) {
10         method(s -> Integer.parseInt(s));
11     }
12 }
```

当然，最好是通过方法引用的写法。

默认方法：andThen

`Function`接口中有一个默认的`andThen`方法，用来进行组合操作。JDK源代码如：

```
1 default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
2     Objects.requireNonNull(after);
3     return (T t) -> after.apply(apply(t));
4 }
```

该方法同样用于“先做什么，再做什么”的场景，和`Consumer`中的`andThen`差不多：

```java
import java.util.function.Function;

public class Demo12FunctionAndThen {
    private static void method(Function<String, Integer> one, Function<Integer, Integer> two) {
        int num = one.andThen(two).apply("10");
        System.out.println(num + 20);
    }

    public static void main(String[] args) {
        method(str->Integer.parseInt(str)+10, i -> i *= 10);
    }
}
```

第一个操作是将字符串解析成为int数字，第二个操作是乘以10。两个操作通过`andThen`按照前后顺序组合到了一起。

> 请注意，Function的前置条件泛型和后置条件泛型可以相同。

题目：自定义函数模型拼接

请使用`Function`进行函数模型的拼接，按照顺序需要执行的多个函数操作为：

String str = "赵丽颖,20";

1. 将字符串截取数字年龄部分，得到字符串；
2. 将上一步的字符串转换成为int类型的数字；
3. 将上一步的int数字累加100，得到结果int数字。

解答

```java
import java.util.function.Function;

public class DemoFunction {
    public static void main(String[] args) {
        String str = "赵丽颖,20";
        int age = getAgeNum(str, s -> s.split(",")[1],
                            s ->Integer.parseInt(s),
                            n -> n += 100);
        System.out.println(age);
    }

    private static int getAgeNum(String str, Function<String, String> one,
                                 Function<String, Integer> two,
                                 Function<Integer, Integer> three) {
        return one.andThen(two).andThen(three).apply(str);
    }
}
```

2.4 其他函数式接口

函数式接口	参数类型	返回类型	用途
BiFunction<T, U, R>	T, U	R	对类型为 T, U 参数应用操作，返回 R 类型的结果。包含方法为 R apply(T t, U u);
UnaryOperator<T> (Function子接口)	T	T	对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为 T apply(T t);
BinaryOperator<T> (BiFunction 子接口)	T, T	T	对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为 T apply(T t1, T t2);
BiConsumer<T, U>	T, U	void	对类型为T, U 参数应用操作。包含方法为 void accept(T t, U u)
ToIntFunction<T> ToLongFunction<T> ToDoubleFunction<T>	T	int long double	分别计算int、long、double、值的函数
IntFunction<R> LongFunction<R> DoubleFunction<R>	int long double	R	参数分别为int、long、double 类型的函数

http://blog.csdn.net/zxm1306192988

2.5 方法引用

双冒号 :: 为引用运算符，而它所在的表达式被称为方法引用。如果Lambda要表达的函数方案已经存在于某个方 法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者

2.5.1 对象::实例方法名

```
1  //对象::实例方法名
2  @Test
3  public void test1(){
4      PrintStream ps1=;
5      Consumer<String> con=(x)->ps1.println(x);//生成了一个实现了Consumer接口的类的对象
6
7      PrintStream ps=System.out;
8      Consumer<String> con1=ps::println;//相当于上面，引用了ps对象的println()方法
9
10     Consumer<String> con2=System.out::println;
11     con2.accept("abcdef");
12 }
13
14 @Test
15 public void test2(){
16     final Employee emp=new Employee();
17     Supplier<String> sup=()->emp.getName();//代替匿名内部类
18     String str=sup.get();
19     System.out.println(str);
20
```

```
    Supplier<Integer> sup2=emp::getAge;
    Integer num=sup2.get();
    System.out.println(num);
}
```

2.5.2 类::静态方法名

```
//类::静态方法名
@Test
public void test3(){
    Comparator<Integer> com=(x,y)->Integer.compare(x,y);
    Comparator<Integer> com1=Integer::compare;
}
```

2.5.3 类::实例方法名

```
//类::实例方法名
@Test
public void test4(){
    BiPredicate<String,String> bp=(x,y)->x.equals(y);
    BiPredicate<String, String> bp2=String::equals;
}
```

2.5.4 构造器引用

格式：ClassName::new

> 注意：需要调用的构造器的参数列表要与函数式接口中抽象方法的参数列表保持一致！

```
//构造器引用
@Test
public void test5(){
    Supplier<Employee> sup=()->new Employee();

    //构造器引用方式
    Supplier<Employee> sup2=Employee::new;//使用无参构造器
    Employee emp=sup2.get();
    System.out.println(emp);

    Function<Integer,Employee> fun2=(x)->new Employee(x);
    Employee emp2=fun2.apply(101);
    System.out.println(emp2);

    BiFunction<String,Integer,Employee> bf=Employee::new;
}
```

2.5.5 数组引用

格式： Type::new

```java
//数组引用
@Test
public void test6(){
    Function<Integer,String[]> fun=(x)->new String[x];
    String[] strs=fun.apply(10);
    System.out.println(strs.length);

    Function<Integer,String[]> fun2=String[]::new;
    String[] str2=fun2.apply(20);
    System.out.println(str2.length);
}
```

3. Stream API

3.1 Stream的优雅

传统集合的多步遍历代码

```java
public class Demo02NormalFilter {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        List<String> zhangList = new ArrayList<>();
        for (String name : list) {
            if (name.startsWith("张")) {
                zhangList.add(name);
            }
        }

        List<String> shortList = new ArrayList<>();
        for (String name : zhangList) {
            if (name.length() == 3) {
                shortList.add(name);
            }
        }
        for (String name : shortList) {
            System.out.println(name);
        }
    }
}
```

stream流的写法

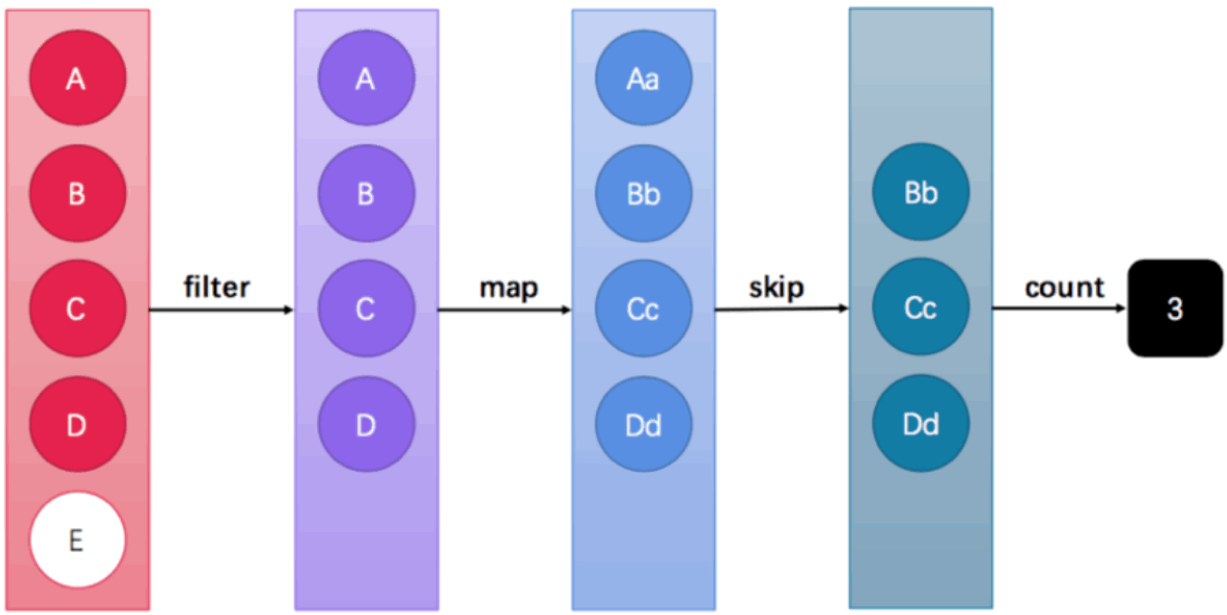
```
```

```java
public class Demo02NormalFilter {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("张无忌");
        list.add("周芷若");
        list.add("赵敏");
        list.add("张强");
        list.add("张三丰");

        list.stream()
            .filter(s -> s.startsWith("张"))
            .filter(s -> s.length() == 3)
            .forEach(System.out::println);
    }
}
```

直接阅读代码的字面意思即可完美展示无关逻辑方式的语义：获取流、过滤姓张、过滤长度为3、逐一打印。代码中并没有体现使用线性循环或是其他任何算法进行遍历，我们真正要做的事情内容被更好地体现在代码中

3.2 流式思想



这张图中展示了过滤、映射、跳过、计数等多步操作，这是一种集合元素的处理方案，而方案就是一种“函数模型”。图中的每一个方框都是一个“流”，调用指定的方法，可以从一个流模型转换为另一个流模型。而最右侧的数字3是最终结果。

这里的 `filter` 、 `map` 、 `skip` 都是在对函数模型进行操作，集合元素并没有真正被处理。只有当终结方法 `count` 执行的时候，整个模型才会按照指定策略执行操作。而这得益于Lambda的延迟执行特性。

> 备注：“Stream流”其实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（或其地址值）。

Stream操作还有两个基础的特征：

- 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， 比如延迟执行(laziness)和短路(short-circuiting)

- 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。

当使用一个流的时候，通常包括三个基本步骤：**获取一个数据源（source）→ 数据转换→执行操作**获取想要的结果，每次转换原有 Stream 对象不改变，返回一个新的 Stream 对象（可以有多次转换），这就允许对其操作可以像链条一样排列，变成一个管道。

3.3 获取流

1.可以通过Collection 系列集合提供的stream()或parallelStream()方法

- default Stream< E> stream() : 返回一个顺序流
- default Stream< E> parallelStream() : 返回一个并行流

2.通过 Arrays 中的静态方法stream()获取数组流

static < T> Stream< T> stream(T[] array): 返回一个流

重载形式，能够处理对应基本类型的数组：

- public static IntStream stream(int[] array)
- public static LongStream stream(long[] array)
- public static DoubleStream stream(double[] array)

3.通过Stream 类中的静态方法of()，通过显示值创建一个流。它可以接收任意数量的参数

- public static< T> Stream< T> of(T... values) : 返回一个流

4.创建无限流

可以使用静态方法 Stream.iterate() 和Stream.generate(), 创建无限流

```
    //创建Stream
    @Test
    public void test1(){
        //1.可以通过Collection 系列集合提供的stream()或parallelStream()
        List<String> list = new ArrayList<>();
        Stream<String> stream1 = list.stream();

        //2.通过 Arrays 中的静态方法stream()获取数组流
        Employee[] emps = new Employee[10];
        Stream<Employee> stream2 = Arrays.stream(emps);

        //3.通过Stream 类中的静态方法of()
        Stream<String> stream3 = Stream.of("aa","bb","cc");

        //4.创建无限流
        //迭代
        Stream<Integer> stream4 = Stream.iterate(0, (x) -> x+2);
        stream4.limit(10).forEach(System.out::println);

        //生成
        Stream.generate(() -> Math.random())
              .limit(5)
              .forEach(System.out::println);
```

```
    }
```

3.4 处理方法

3.4.1 筛选与切片

方　法	描　述
filter(Predicate p)	接收 Lambda ， 从流中排除某些元素。
distinct()	筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
limit(long maxSize)	截断流，使其元素不超过给定数量。
skip(long n)	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补

```
    List<Employee> employees=Arrays.asList(
            new Employee("张三",18,9999.99),
            new Employee("李四",58,5555.55),
            new Employee("王五",26,3333.33),
            new Employee("赵六",36,6666.66),
            new Employee("田七",12,8888.88),
            new Employee("田七",12,8888.88)
            );

    /*
     * 中间操作
     * 筛选与切片
     *  filter--接收Lambda，从流中排除某些元素。
     *  limit--截断流，使其元素不超过给定数量。
     *  skip(n)--跳过元素，返回一个扔掉了前n个元素的流。若流中元素不足n个，则返回一个空流。与 limit(n) 互补
     *  distinct--筛选，通过流所生成元素的 hashCode() 和 equals() 去掉重复元素
     */

    //内部迭代：迭代操作由 Stream API 完成
    @Test
    public void test1(){
        //中间操作：不会执行任何操作
        Stream<Employee> stream=employees.stream()
                                .filter((e) -> e.getAge()>35 );
        //终止操作：一次性执行全部内容，即 惰性求值
        stream.forEach(System.out::println);
    }
    //外部迭代
    @Test
    public void test2(){
        Iterator<Employee> it=employees.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
```

```
        }
    }

    @Test
    public void test3(){
        //发现“短路”只输出了两次，说明只要找到 2 个 符合条件的就不再继续迭代
        employees.stream()
                .filter((e)->{
                    System.out.println("短路! ");
                    return e.getSalary()>5000;
                })
                .limit(2)
                .forEach(System.out::println);
    }

    @Test
    public void test4(){
        employees.stream()
                .filter((e)->e.getSalary()>5000)
                .skip(2)//跳过前两个
                .distinct()//去重，注意：需要Employee重写hashCode 和 equals 方法
                .forEach(System.out::println);
    }

    @Test
    public void test5() {
        //第一支队伍
        ArrayList<String> one = new ArrayList<>();
        one.add("迪丽热巴");
        one.add("宋远桥");
        one.add("苏星河");
        one.add("石破天");
        one.add("石中玉");
        one.add("老子");
        one.add("庄子");
        one.add("洪七公");

        //第二支队伍
        ArrayList<String> two = new ArrayList<>();
        two.add("古力娜扎");
        two.add("张无忌");
        two.add("赵丽颖");
        two.add("张三丰");
        two.add("尼古拉斯赵四");
        two.add("张天爱");
        two.add("张二狗");

        Stream<String> stream1 = one.stream()
                .filter(s -> s.length() == 3)
                .limit(3);

        Stream<String> stream2 = two.stream()
                .filter(s -> s.startsWith("张"))
```

```
                .skip(2);

        Stream.concat(stream1, stream2)
                .map(s -> new Person(s))
                .forEach(System.out::println);

    }
```

3.4.2 映射

方　法	描　述
map(Function f)	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。
mapToDouble(ToDoubleFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
mapToInt(ToIntFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
mapToLong(ToLongFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 LongStream。
flatMap(Function f)	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

http://blog.csdn.net/qq_36582588

```
    /*
     * 映射
     * map--接收Lambda，将元素转换成其他形式或提取信息。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元素。
     * flatMap--扁平化map.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
     */
    @Test
    public void test5(){
        List<String> list=Arrays.asList("aaa","bbb","ccc","ddd");
        list.stream()
            .map((str)->str.toUpperCase())
            .forEach(System.out::println);

        System.out.println("-------------------------");

        employees.stream()
                 .map(Employee::getName)
                 .forEach(System.out::println);

        System.out.println("-------------------------");

        Stream<Stream<Character>> stream=list.stream()
                                             .map(TestStreamAPI2::filterChatacter);
        stream.forEach((sm)->{
            sm.forEach(System.out::println);
        });

        System.out.println("-------------------------");
```

```
28
29          Stream<Character> sm=list.stream()
30                                  .flatMap(TestStreamAPI2::filterChatacter);
31          sm.forEach(System.out::println);
32      }
33
34      public static Stream<Character> filterChatacter(String str){
35          List<Character> list=new ArrayList<>();
36          for (Character ch : str.toCharArray()) {
37              list.add(ch);
38          }
39          return list.stream();
40      }
41
42      @Test
43      public void test6(){
44          //map和flatMap的关系  类似于 add(Object)和addAll(Collection coll)
45          List<String> list=Arrays.asList("aaa","bbb","ccc","ddd");
46          List list2=new ArrayList<>();
47          list2.add(11);
48          list2.add(22);
49          list2.addAll(list);
50          System.out.println(list2);
51      }
```

3.4.2 排序

方　法	描　述
sorted()	产生一个新流，其中按自然顺序排序
sorted(Comparator comp)	产生一个新流，其中按比较器顺序排序

```
1       /*
2        * 排序
3        * sorted()-自然排序 (按照对象类实现Comparable接口的compareTo()方法 排序)
4        * sorted(Comparator com)-定制排序 (Comparator)
5        */
6       @Test
7       public void test7(){
8           List<String> list=Arrays.asList("ccc","bbb","aaa");
9           list.stream()
10              .sorted()
11              .forEach(System.out::println);
12
13          System.out.println("------------------------");
14
15          employees.stream()
16                   .sorted((e1,e2)->{
17                       if(e1.getAge().equals(e2.getAge())){
18                           return e1.getName().compareTo(e2.getName());
19                       }else{
20                           return e1.getAge().compareTo(e2.getAge());
```

```
                    }
                }).forEach(System.out::println);
    }
```

3.5 终止操作

3.5.1 查找与匹配

方　法	描　述
allMatch(Predicate p)	检查是否匹配所有元素
anyMatch（Predicate p）	检查是否至少匹配一个元素
noneMatch(Predicate p)	检查是否没有匹配所有元素
findFirst()	返回第一个元素
findAny()	返回当前流中的任意元素

http://blog.csdn.net/zxm1306192988

count()	返回流中元素总数
max(Comparator c)	返回流中最大值
min(Comparator c)	返回流中最小值
forEach(Consumer c)	内部迭代(使用 Collection 接口需要用户去做迭代，称为外部迭代。相反，Stream API 使用内部迭代——它帮你把迭代做了)

```
public static void main(String[] args) {
        List<Person> personList = Arrays.asList(
                new Person("zx呵呵", 1),
                new Person("cx呵呵", 23),
                new Person("ax嘻嘻", 1));

        System.out.println("集合中全部含有'呵呵': " + personList.stream().allMatch(s -> s.getName().contains("呵呵")));
        System.out.println("集合中没有'呵呵': " + personList.stream().noneMatch(s -> s.getName().contains("呵呵")));
        System.out.println("集合中至少有一个'嘻嘻': " + personList.stream().anyMatch(s -> s.getName().contains("呵呵")));
```

```
        System.out.println("集合中的第一个元素: " +
personList.stream().findFirst().get());
        System.out.println("返回集合中任意一个元素: " + personList.stream().findAny());
        System.out.println("返回集合中年龄最大的元素: " +
personList.stream().max(Comparator.comparingInt(Person::getAge)));
        System.out.println("返回集合中年龄最小的元素: " +
personList.stream().min(Comparator.comparingInt(Person::getAge)));
        System.out.println("返回集合中元素的总数: " + personList.stream().count());
    }
```

3.5.2 reduce(规约)

```
    /*
     * 归约: 可以将流中元素反复结合起来，得到一个值
     * reduce(T identity,BinaryOperator b) / reduce(BinaryOperator b)
     */
    @Test
    public void test3(){
        List<Integer> list=Arrays.asList(1,2,3,4,5,6,7,8,9,10);
        //reduce(T identity,BinaryOperator b)
        Integer sum=list.stream()
                        .reduce(0, (x,y)->x+y);//0为起始值
        System.out.println(sum);

        System.out.println("--------------------------");

        // reduce(BinaryOperator b)//没有起始值，map返回可能为空，所以返回Optional类型
        // map-reduce模式
        Optional<Double> op=employees.stream()
                                     .map(Employee::getSalary)
                                     .reduce(Double::sum);
        System.out.println(op.get());
    }
```

3.6 收集

方　法	描　述
collect(Collector c)	将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法

Collector接口中方法的实现决定了如何对流执行收集操作（如收集到List、Set、Map）。但是Collectors实用类提供了很多静态方法，可以方便地创建常见收集器实例，具体方法与实例如下表：

方法	返回类型	作用
toList	List<T>	把流中元素收集到List
List<Employee> emps= list.stream().collect(Collectors.toList());		
toSet	Set<T>	把流中元素收集到Set
Set<Employee> emps= list.stream().collect(Collectors.toSet());		
toCollection	Collection<T>	把流中元素收集到创建的集合
Collection<Employee>emps=list.stream().collect(Collectors.toCollection(ArrayList::new));		
counting	Long	计算流中元素的个数
long count = list.stream().collect(Collectors.counting());		
summingInt	Integer	对流中元素的整数属性求和
inttotal=list.stream().collect(Collectors.summingInt(Employee::getSalary));		
averagingInt	Double	计算流中元素Integer属性的平均值
doubleavg= list.stream().collect(Collectors.averagingInt(Employee::getSalary));		
summarizingInt	IntSummaryStatistics	收集流中Integer属性的统计值。如：平均值
IntSummaryStatisticsiss= list.stream().collect(Collectors.summarizingInt(Employee::getSalary));		

https://blog.csdn.net/qq_43092988

```java
public class CollectDemo {

    public static void main(String[] args) {
        List<Person> personList = Arrays.asList(
                new Person("zx呵呵", 1),
                new Person("cx呵呵", 23),
                new Person("ax嘿嘿", 13),
                new Person("bx哈哈", 33),
                new Person("dx啪啪", 28),
                new Person("ax嘻嘻", 1));

        List<Integer> ageList
            = personList.stream()
                        .map(Person::getAge)
                        .collect(Collectors.toList());
        ageList.forEach(s -> System.out.print(s + ","));
        System.out.println();

        Set<Integer> ageSet
            = personList.stream()
                .map(Person::getAge).collect(Collectors.toSet());
        ageSet.forEach(s -> System.out.print(s + ","));
        System.out.println();

        ArrayList<Person> collect
            = personList.stream()
                .collect(Collectors.toCollection(ArrayList::new));
        System.out.println(collect.size());

```

```
        System.out.println("流中有" + personList.stream()
                           .map(Person::getAge)
                           .distinct()
                           .collect(Collectors.counting()) + "个不重复的元素");

        System.out.println("计算流中年龄属性的总和：" + personList.stream()
                           .collect(Collectors.summingDouble(Person::getAge)));

        System.out.println("计算流中年龄属性的各种值：" + personList.stream()
                           .collect(Collectors.summarizingDouble(Person::getAge)));
        System.out.println("计算流中年龄属性的平均值：" + personList.stream()
                           .collect(Collectors.summarizingDouble(Person::getAge))
                           .getAverage());
    }
}
```

joining	String	连接流中每个字符串
String str= list.stream().map(Employee::getName).collect(Collectors.joining());		
maxBy	Optional<T>	根据比较器选择最大值
Optional<Emp>max= list.stream().collect(Collectors.maxBy(comparingInt(Employee::getSalary)));		
minBy	Optional<T>	根据比较器选择最小值
Optional<Emp> min = list.stream().collect(Collectors.minBy(comparingInt(Employee::getSalary)));		
reducing	归约产生的类型	从一个作为累加器的初始值开始，利用BinaryOperator与流中元素逐个结合，从而归约成单个值
inttotal=list.stream().collect(Collectors.reducing(0, Employee::getSalar, Integer::sum));		
collectingAndThen	转换函数返回的类型	包裹另一个收集器，对其结果转换函数
inthow= list.stream().collect(Collectors.collectingAndThen(Collectors.toList(), List::size));		
groupingBy	Map<K, List<T>>	根据某属性值对流分组，属性为K，结果为V
Map<Emp.Status, List<Emp>> map= list.stream() .collect(Collectors.groupingBy(Employee::getStatus));		
partitioningBy	Map<Boolean, List<T>>	根据true或false进行分区
Map<Boolean,List<Emp>>vd= list.stream().collect(Collectors.partitioningBy(Employee::getManage));		

https://blog.csdn.net/zxm1306192888

```
    /*
     * 收集
     * collect-将流转换为其他形式，接收一个Collector接口的实现，用于给Stream中元素做汇总的方法。
     */
    @Test
    public void test4(){
        List<Employee> employees=Arrays.asList(
            new Employee("张三",18,9999.99),
            new Employee("李四",58,5555.55),
            new Employee("王五",26,3333.33),
```

```
            new Employee("赵六",36,6666.66),
            new Employee("田七",12,8888.88),
            new Employee("田七",12,8888.88)
            );

        //总和
        Long count=employees.stream()
                            .collect(Collectors.counting());
        System.out.println(count);

        //平均值
        Double avg=employees.stream()

.collect(Collectors.averagingDouble(Employee::getSalary));
        System.out.println(avg);

        //总和
        Double sum=employees.stream()

.collect(Collectors.summingDouble(Employee::getSalary));
        System.out.println(sum);

        //最大值
        Optional<Employee> max=employees.stream()
                                        .collect(Collectors.maxBy((e1,e2)->Double.compare(e1.getSalary(), e2.getSalary())));
        System.out.println(max.get());

        //最小值
        Optional<Double> min=employees.stream()
                                      .map(Employee::getSalary)
                                      .collect(Collectors.minBy(Double::compare));
        System.out.println(min.get());

        System.out.println("----------------------------");

        //分组
        Map<Status,List<Employee>> map=employees.stream()

.collect(Collectors.groupingBy(Employee::getStatus));
        System.out.println(map);//{FREE=[Employee [name=张三, age=18, salary=9999.99, Status=FREE], Employee [name=赵六, age=36, salary=6666.66, Status=FREE]], VOCATION=[Employee [name=王五, age=26, salary=3333.33, Status=VOCATION]], BUSY=[Employee [name=李四, age=58, salary=5555.55, Status=BUSY], Employee [name=田七, age=12, salary=8888.88, Status=BUSY]]}

        //多级分组
        Map<Status,Map<String,List<Employee>>> map2=employees.stream()
            .collect(
                Collectors.groupingBy( Employee::getStatus,
                        Collectors.groupingBy((e)->{
                                if(e.getAge()<=35){
                                    return "青年";
```

```
                                }else if(e.getAge()<=50){
                                    return "中年";
                                }else{
                                    return "老年";
                                }
                            }
                        )
                    )
                );
        System.out.println(map2);//{FREE={青年=[Employee [name=张三, age=18, salary=9999.99, Status=FREE]], 中年=[Employee [name=赵六, age=36, salary=6666.66, Status=FREE]]}, VOCATION={青年=[Employee [name=王五, age=26, salary=3333.33, Status=VOCATION]]}, BUSY={青年=[Employee [name=田七, age=12, salary=8888.88, Status=BUSY]], 老年=[Employee [name=李四, age=58, salary=5555.55, Status=BUSY]]}}

        //分区
        Map<Boolean,List<Employee>> map3=employees.stream()

.collect(Collectors.partitioningBy((e)->e.getSalary()>8000));
        System.out.println(map3);//{false=[Employee [name=李四, age=58, salary=5555.55, Status=BUSY], Employee [name=王五, age=26, salary=3333.33, Status=VOCATION], Employee [name=赵六, age=36, salary=6666.66, Status=FREE]], true=[Employee [name=张三, age=18, salary=9999.99, Status=FREE], Employee [name=田七, age=12, salary=8888.88, Status=BUSY]]}

        System.out.println("----------------------------------");

        DoubleSummaryStatistics dss=employees.stream()

.collect(Collectors.summarizingDouble(Employee::getSalary));
        System.out.println(dss.getSum());
        System.out.println(dss.getAverage());
        System.out.println(dss.getMax());

        System.out.println("----------------------------------");
        String strr=employees.stream()
                             .map(Employee::getName)
                             .collect(Collectors.joining(","));
        System.out.println(strr);//张三李四王五赵六田七
    }
```

3.7 并行流

并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

Java 8 中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream API 可以声明性地通过 parallel（）与 sequential（） 在并行流与顺序流之间进行切换。

```java
    @Test
    public void test2(){
        Instant start=Instant.now();

        Long sum=LongStream.rangeClosed(0L, 10000000000L)
                           .parallel()
                           .reduce(0,Long::sum);
        System.out.println(sum);

        Instant end=Instant.now();
        //消耗时间2418ms
        System.out.println("消耗时间"+Duration.between(start, end).toMillis()+"ms");
    }
```

3.8 Stream练习

```java
public class TestTransaction {
    List<Transaction> transaction=null;

    @Before
    public void before(){
        Trader raoul=new Trader("Raoul","Cambridge");
        Trader mario=new Trader("Mario","Milan");
        Trader alan=new Trader("Alan","Cambridge");
        Trader brian=new Trader("Brian","Cambridge");

        transaction=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)
        );
    }

    //1.找出2011年发生的所有交易，并按交易额排序(从低到高)
    @Test
    public void test1(){
        transaction.stream()
                   .filter((e)->e.getYear()==2011)
                   .sorted((e1,e2)->Integer.compare(e1.getValue(), e2.getValue()))
                   .forEach(System.out::println);
    }

    //2.交易员都在哪些不同的城市工作过?
    @Test
    public void test2(){
        transaction.stream()
                   .map((e)->e.getTrader().getCity())
                   .distinct()//去重
```

```
                .forEach(System.out::println);
    }

    //3.查找所有来自剑桥的交易员，并按姓名排序
    @Test
    public void test3(){
        transaction.stream()
                .filter((e)->e.getTrader().getCity().equals("Cambridge"))
                .map(Transaction::getTrader)
                .sorted((e1,e2)->e1.getName().compareTo(e2.getName()))
                .distinct()
                .forEach(System.out::println);
    }

    //4.返回所有交易员的姓名字符串，按字母顺序排序
    @Test
    public void test4(){
        transaction.stream()
                .map(Transaction::getTrader)
                .map(Trader::getName)
                .distinct()
                .sorted()
                .forEach(System.out::println);

        System.out.println("-------------------------");

        String str=transaction.stream()
                              .map((e)->e.getTrader().getName())
                              .distinct()
                              .sorted()
                              .reduce("", String::concat);
        System.out.println(str);//AlanBrianMarioRaoul

        System.out.println("-------------------------");

         transaction.stream()
                    .map((t)->t.getTrader().getName())
                        //返回的每个String合成一个流
                    .flatMap(TestTransaction::filterCharacter)
                    .sorted((s1,s2)->s1.compareToIgnoreCase(s2))
                        //aaaaaAaBiiilllMMnnoooorRRrruu
                    .forEach(System.out::print);
    }
    public static Stream<String> filterCharacter(String str){
        List<String> list=new ArrayList<>();
        for(Character ch:str.toCharArray()){
            list.add(ch.toString());
        }
        return list.stream();
    }

    //5.有没有交易员是在米兰工作的?
    @Test
```

```
    public void test5(){
        boolean b1=transaction.stream()
                              .anyMatch((t)>t.getTrader()
                                        .getCity().equals("Milan"));
        System.out.println(b1);
    }

    //6.打印生活在剑桥的交易员的所有交易额
    @Test
    public void test6(){
        Optional<Integer> sum=transaction.stream()
                                         .filter((e)->e.getTrader().getCity().equals("Cambridge"))
                                         .map(Transaction::getValue)
                                         .reduce(Integer::sum);
        System.out.println(sum.get());
    }

    //7.所有交易中，最高的交易额是多少
    @Test
    public void test7(){
        Optional<Integer> max=transaction.stream()
                                         .map((t)->t.getValue())
                                         .max(Integer::compare);
        System.out.println(max.get());
    }

    //8.找到交易额最小的交易
    @Test
    public void test8(){
        Optional<Transaction> op=transaction.stream()
.min((t1,t2) ->
                                    Integer.compare(t1.getValue(),t2.getValue()));
        System.out.println(op.get());
    }
}
```

4. Optional类

User --》 Address --》 Location

User

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {

    private String name;

    private Integer age;

    private Address address;
}
```

Address

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Address {

    private String street;

    private Location location;

}
```

Location

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Location {
    private Double longitude;

    private Double latitude;
}
```

4.1 1.8之前的非空判断

```
 /**
     * 1.8 之前 方法一
     */
    public static void saveUser1(User user) {
        if (null != user) {
            if (null != user.getAddress()) {
                if (null != user.getAddress().getLocation()) {
                    //保存user
                }
            }
        }
    }
```

```
13
14      /**
15       * 1.8 之前 方法二
16       */
17      public static void saveUser2(User user) {
18          if (null == user) {
19              return;
20          }
21
22          if (null == user.getAddress()) {
23              return;
24          }
25
26          if (null == user.getAddress().getLocation()) {
27              return;
28          }
29
30          //保存user
31      }
```

4.2 Optional

Optional< T>类(java.util.Optional) 是一个容器类，代表一个值存在或不存在。 原来用null表示一个值不存在，现在 Optional可以更好的表达这个概念。并且可以避免空指针异常。

4.2.1 常用api：

> Optional.of(T t) : 创建一个 Optional 实例 Optional.empty() : 创建一个空的 Optional 实例 Optional.ofNullable(T t):若 t 不为 null,创建 Optional 实例,否则创建空实例 isPresent() : 判断是否包含值 orElse(T t) : 如果调用对象包含值，返回该值，否则返回t orElseGet(Supplier s) :如果调用对象包含值，返回该值，否则返回 s 获取的值 map(Function f): 如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty() flatMap(Function mapper):与 map 类似，要求返回值必须是Optional

4.2.2 1.8后的非空处理

实体类

User

```
1  @Data
2  @AllArgsConstructor
3  @NoArgsConstructor
4  public class User {
5
6      private String name;
7
8      private Integer age;
9
10     private Optional<Address> address;
11 }
```

Address

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Address {

    private String street;

    private Optional<Location> location;
}
```

Location

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Location {
    private Double longitude;

    private Double latitude;
}
```

测试1

```
// 创建一个空的Optional对象
Optional<Address> optionalAddress1 = Optional.empty();
//  判断是否包含值
if (!optionalAddress1.isPresent()) {
    //如果调用对象包含值，返回该值，否则返回指定值
    System.out.println(optionalAddress1.orElse(new Address("北京路",
                    Optional.ofNullable(new Location()))));
}
```

测试2

```
// 创建一个非空的Optional对象，如果为空则抛出空指针异常
Optional<Address> optionalAddress2 = Optional.of(new Address("香港路",
  Optional.ofNullable(new Location())));
if (optionalAddress2.isPresent()) {
    System.out.println(optionalAddress2.orElse(new Address()));
}
```

测试3

```
// orElseGet和orElse没有太大的区别，但是orElseGet可以写lambda表达式
Optional<Address> optionalAddress3 = Optional.empty();
int i = 100;
//如果调用对象包含值，返回该值，否则返回 s 获取的值
if (!optionalAddress3.isPresent()) {
    System.out.println(optionalAddress3.orElseGet(()->{
        if (i % 2 == 0) {
            return new Address("台北路", Optional.ofNullable(new Location()));
        }
        return new Address();
    }));
}
```

测试4

```
 // 创建一个可接受Null值的Optional
 Optional<Address> optionalAddress4 = Optional.ofNullable(new Address());
 if (optionalAddress4.isPresent()) {
    System.out.println(saveUser1(null));
 }

 public static Double saveUser1(User user) {
//        Optional<Optional<Address>> address
//                = Optional.ofNullable(user).map(User::getAddress);

//        Optional<Address> optionalAddress = Optional.ofNullable(user)
//                .flatMap(User::getAddress);

        return Optional.ofNullable(user)
                    .flatMap(User::getAddress)
                    .flatMap(Address::getLocation)
                    .map(Location::getLatitude)
                    .orElse(0D);

    }
```

5. 接口中的默认方法与静态方法

Java8中允许接口中包含具有具体实现的方法，该方法称为“默认方法”，默认方法使用 default 关键字修饰。

接口默认方法的“类优先”原则： 若一个接口中定义了一个默认方法，而另一个父类或接口中又定义了一个同名的方法时

- 选择父类中的方法。如果一个父类提供了具体的实现，那么接口中具有相同名称和参数的默认方法会被忽略。
- 接口冲突。如果一个父接口提供一个默认方法，而另一个接口也提供了一个具有相同名称和参数列表的方法（不管方法是否是默认方法），那么必须覆盖该方法来解决冲突。

```
public interface MyFunc {
```

```
    default String getName() {
        return "interface";
    }

}

public class Parent {

    public String getName() {
        return "super class";
    }
}

public class Sub extends Parent implements MyFunc{
}

public class Main {

    public static void main(String[] args) {
        System.out.println(new Sub().getName());    //super class
    }
}
```

```
public interface MyFunc {

    default String getName() {
        return "MyFunc呵呵";
    }


    static String getStr(){
        return "hello";
    }
}

public interface MyHappy {
    default String getName() {
        return "MyHappy呵呵";
    }
}

public class MyInterface implements MyFunc, MyHappy {

    @Override
    public String getName() {
        return MyFunc.super.getName();
    }
}

public class Main {

    public static void main(String[] args) {
```

```
30          System.out.println(MyFunc.getStr());     //MyFunc呵呵
31      }
32  }
```

6. 新的时间API

以前的时间API（包括SimpleDateFormate类）是线程不安全的，多线程情况下要加锁

1.8后新增了一些时间方面的API

- 用于让人读的时间日期：LocalDate、LocalTime、LocalDateTime
- 用于让机器读的时间日期：

6.1 给人读的时间日期类

LocalDate、LocalTime、LocalDateTime 类的实例是不可变的对象，分别表示使用 ISO-8601日历系统的日期、时间、日期和时间。它们提供了简单的日期或时间，并不包含当前的时间信息。也不包含与时区相关的信息。

方法	描述	示例
now()	静态方法，根据当前时间创建对象	LocalDate localDate = LocalDate.now(); LocalTime localTime = LocalTime.now(); LocalDateTime localDateTime = LocalDateTime.now();
of()	静态方法，根据指定日期/时间创建对象	LocalDate localDate = LocalDate.of(2016, 10, 26); LocalTime localTime = LocalTime.of(02, 22, 56); LocalDateTime localDateTime = LocalDateTime.of(2016, 10, 26, 12, 10, 55);
plusDays, plusWeeks, plusMonths, plusYears	向当前 LocalDate 对象添加几天、几周、几个月、几年	
minusDays, minusWeeks, minusMonths, minusYears	从当前 LocalDate 对象减去几天、几周、几个月、几年	
plus, minus	添加或减少一个 Duration 或 Period	
withDayOfMonth, withDayOfYear, withMonth, withYear	将月份天数、年份天数、月份、年份修改为指定的值并返回新的 LocalDate 对象	
getDayOfMonth	获得月份天数(1-31)	
getDayOfYear	获得年份天数(1-366)	
getDayOfWeek	获得星期几(返回一个 DayOfWeek 枚举值)	
getMonth	获得月份, 返回一个 Month 枚举值	
getMonthValue	获得月份(1-12)	
getYear	获得年份	
until	获得两个日期之间的 Period 对象，或者指定 ChronoUnits 的数字	
isBefore, isAfter	比较两个 LocalDate	
isLeapYear	判断是否是闰年	

http://blog.csdn.net/zxm1306192988

```
1      @Test
2      public void contextLoads() {
3          LocalDateTime now = LocalDateTime.now();
4          System.out.println("当前时间: " + now);  //  当前时间: 2019-03-21T02:54:01.788
5
6          LocalDateTime date = LocalDateTime.of(2020, 2, 18, 10, 21, 19);
7          //  指定一个日期时间:2020-02-18T10:21:19
```

```
        System.out.println("指定一个日期时间:" + date);
        System.out.println("年: " + date.getYear());        //  年: 2020
        System.out.println("月: " + date.getMonthValue()); //  月: 2
        System.out.println("星期: " + date.getDayOfWeek()); //  星期: TUESDAY
        System.out.println("时-分-秒: " + date.getHour() + "-" + date.getMinute() + "-" + date.getSecond());    //时-分-秒: 10-21-19
         // 加2年: 2022-02-18T10:21:19
        System.out.println("加2年: " + date.plusYears(2));
        //  减2个月: 2019-12-18T10:21:19
        System.out.println("减2个月: " + date.minusMonths(2));
    }
```

LocalDate、LocalTime、LocalDateTime 用法一样，只是作用范围不同。分别是日期、时间、日期时间。

6.2 给机器读的时间日期类

6.2.1 Instant和Duration

```
@Test
public void contextLoads2() throws InterruptedException {
    Instant start = Instant.now();
    System.out.println("开始时间: " + start);//开始时间: 2019-03-20T19:02:18.116Z
    Thread.sleep(200);
    Instant end = Instant.now();

    //耗时: 201
    System.out.println("耗时: " + Duration.between(start, end).toMillis());
}
```

```
    @Test
    public void contextLoads2() throws InterruptedException {
        LocalTime lt1=LocalTime.now();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        LocalTime lt2=LocalTime.now();
        System.out.println(Duration.between(lt1, lt2).toMillis());  //101
    }
```

```
@Test
   public void contextLoads2() throws InterruptedException {
       LocalDate start = LocalDate.of(2019, 3, 11);
       LocalDate end = LocalDate.now();
       Period p = Period.between(start, end);
       System.out.println("年:" + p.getYears() + "月:" + p.getMonths() + "日:" + p.getDays());
    }
```

6.2 时间校验器 TemporalAdjuster

```
@Test
    public void contextLoads2() throws InterruptedException {
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);        //2019-03-21T03:21:26.818

        //  天份修改为10号
        System.out.println(now.withDayOfMonth(10)); //2019-03-10T03:21:26.818

        //  查看下一个星期一的日期
        System.out.println(now.with(TemporalAdjusters.next(DayOfWeek.MONDAY)));

        //自定义：下一个工作日
        now.with((s) -> {
            LocalDateTime dateTime = (LocalDateTime) s;
            DayOfWeek nowWeek = dateTime.getDayOfWeek();
            if (nowWeek.equals(DayOfWeek.FRIDAY)) {
                return dateTime.plusDays(3);
            } else if (nowWeek.equals(DayOfWeek.SATURDAY)) {
                return dateTime.plusDays(2);
            } else {
                return dateTime.plusDays(1);
            }
        });

        System.out.println(now);
    }
```

6.3 格式化

```
    @Test
    public void DateTimeFormatterTest() {
        //自定义格式化格式
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss");
        System.out.println(dtf);

        LocalDateTime now = LocalDateTime.now();
        String format = now.format(dtf);
        System.out.println("格式化时间:" + format);
    }
```

与传统日期处理的转换

类	To 遗留类	From 遗留类
java.time.Instant java.util.Date	Date.from(instant)	date.toInstant()
java.time.Instant java.sql.Timestamp	Timestamp.from(instant)	timestamp.toInstant()
java.time.ZonedDateTime java.util.GregorianCalendar	GregorianCalendar.from(zonedDateTime)	cal.toZonedDateTime()
java.time.LocalDate java.sql.Time	Date.valueOf(localDate)	date.toLocalDate()
java.time.LocalTime java.sql.Time	Date.valueOf(localDate)	date.toLocalTime()
java.time.LocalDateTime java.sql.Timestamp	Timestamp.valueOf(localDateTime)	timestamp.toLocalDateTime()
java.time.ZoneId java.util.TimeZone	Timezone.getTimeZone(id)	timeZone.toZoneId()
java.time.format.DateTimeFormatter java.text.DateFormat	formatter.toFormat()	无

http://blog.csdn.net/zxm1306192988

6.4 时区

```
    @Test
    public void ZonedTest() {
        //获取指定时区的日期时间类型
        LocalDateTime ldt=LocalDateTime.now(ZoneId.of("Asia/Shanghai"));
        System.out.println(ldt);//2017-07-20T14:01:23.417

        LocalDateTime ldt2=LocalDateTime.now(ZoneId.of("Asia/Shanghai"));
        System.out.println(ldt2);
        ZonedDateTime zdt=ldt2.atZone(ZoneId.of("Asia/Shanghai"));//获取带时区的时间类型
        //2017-07-20T14:01:23.420+08:00[Asia/Shanghai]//与UTC时间有8个小时的时差
        System.out.println(zdt);
    }
```