# Lambda表达式

<https://jiankunking.blog.csdn.net/article/details/79825928>

## 为什么要有lambda

<https://blog.csdn.net/weixin_43687181/article/details/116053262>

1. 需要给方法传递一段代码作为参数
2. 在C和C++中可以使用指针，一两个\*号&号能解决的问题。 但是在Java中并没有指针这个东东。
3. 而且Java的函数接收的参数都是对象（这个时候可以认为基本类型也是对象），可从来没听说过有函数对象这个说法啊。

例如：

|  |
| --- |
| static void executeFunc(方法？？,String word) {  // todo 用传入的方法打印word变量  } |

根据传入的方法不同，打印的格式也会不同。

### 匿名内部类写法

创建

|  |
| --- |
| 1. package main; 2. interface Wrapper { 3. void myPrint(String w); 4. } 5. class Solution { 6. static void executeFunc(Wrapper w, String word) { 7. w.myPrint(word); 8. } 9. } |

调用

|  |
| --- |
| 1. public static void main(String[] args) { 2. executeFunc(new Wrapper() { 3. @Override 4. public void myPrint(String w) { 5. *// 个性化拓展，例如在打印之前记录时间什么的* 6. System.out.println(w); 7. } 8. }, "Hello Lambda!"); 9. } |

如果需要用executeFunc的次数多起来的时候，显然就会造成很多“不太必要”的代码了。这里说的不太必要是因为，我们必须满足编译器的需求来进行规范的语法编写，但实际上我们关心的逻辑仅仅是里面那一小段打印的语句：

// 个性化拓展，例如在打印之前记录时间什么的

System.out.println(w);

### Lambda写法

|  |
| --- |
| 1. // lambda写法1 2. executeFunc(w -> { 3. // 个性化拓展，例如在打印之前记录时间什么的 4. System.out.println(w); 5. }, "Lambda写法1 "); 7. // lambda写法2 8. Wrapper wrapper = (w) ->{ 9. System.out.println(w); 10. }; 11. wrapper.myPrint("Lambda写法2 "); |

我们看到，一大串的new操作，@Override重写被替换成一个传入参数，一对大括号，代码量大大减少。(实际上可以通过方法引用来进一步化简，但是在这里暂不作讨论了)

### 两种写法对比

#### 匿名内部类

|  |
| --- |
| 1. package main; 2. interface Wrapper { 3. void myPrint(String w); 4. } 5. class Solution { 6. static void executeFunc(Wrapper w, String word) { 7. w.myPrint(word); 8. } 9. public static void main(String[] args) { 10. *// 匿名内部类写法* 11. executeFunc(new Wrapper() { 12. @Override 13. public void myPrint(String w) { 14. *// 个性化拓展，例如在打印之前记录时间什么的* 15. System.out.println(w); 16. } 17. }, "Hello Lambda!"); 19. *// lambda写法* 20. executeFunc(w -> { 21. *// 个性化拓展，例如在打印之前记录时间什么的* 22. System.out.println(w); 23. }, "Hello Lambda!"); 25. } 26. } |

利用IDEA的ShowByteCode功能进行查看。字节码（ByteCode）是JVM运行时读取执行的，一切的语法特性，都会在这里暴露无遗/

JVM里是不管你内部内部类的，它只认类和类实例化的对象。

接口是不能实例化的，也就是说，是不能通过直接new的方式新建一个纯接口对象，而是要编写一个类来实现接口，进而实例化这个类。

匿名内部类通过编译器新建了一个类，叫Solution$1

在Solution$1内部实现了抽象方法。

文本

描述已自动生成

本来内部类的名字应该是类似Solution$MyInnerClass这样的命名的，$符号之前其所在的父类Solution，MyInnerClass是内部类的名字。但是由于我们创建的是实现了Wrapper接口的匿名内部类，匿名没有名字，编译器就想，干脆就叫1吧，没错，就简简单单用个1来命名了。

#### Lambda

1. 在类编译时，会生成一个私有静态方法+一个内部类；
2. 在内部类中实现了函数式接口，在实现接口的方法中，会调用编译器生成的静态方法；
3. 在使用lambda表达式的地方，通过传递内部类实例，来调用函数式接口方法。

文本

描述已自动生成

编译器把我们写的lambda表达式转换成了一个静态的 私有的 函数，

函数的方法体就是lambda表达式的代码， 通过调用这个函数来解决传递一段代码的问题。

# 什么是lambda表达式?

Lambda表达式脱胎于原来的匿名类给方法传递函数功能

Lambda表达式的核心思想，就是将一段函数作为参数进行传递

Java8之前: 每次为了实现不同的功能,需要写大量的代码

Java8: 我们精简代码, 只需要写核心的代码即可

|  |
| --- |
| 1. // Java 8之前： 2. **new** Thread(**new** Runnable() { 3. @Override 4. **public** **void** run() { 5. System.out.println("Before Java8, too much code for too little to do"); 6. } 7. }).start(); 9. //Java 8方式： 10. **new** Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start(); |

# Lambda 表达式的基础语法

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

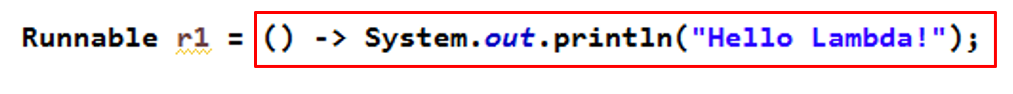
箭头操作符将 Lambda 表达式拆分成两部分：

左侧：Lambda 表达式的参数列表

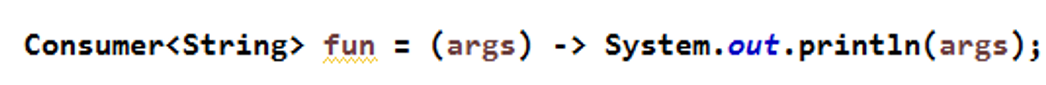
右侧：Lambda 表达式中所需执行的功能， 即 Lambda 体

## 语法格式

* 语法格式一：无参，无返回值，Lambda 体只需一条语句



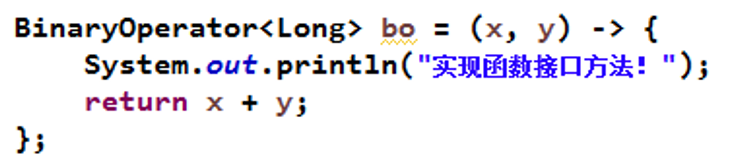
* 语法格式二：Lambda 需要一个参数



* 语法格式三：Lambda 只需要一个参数时，参数的小括号可以省略



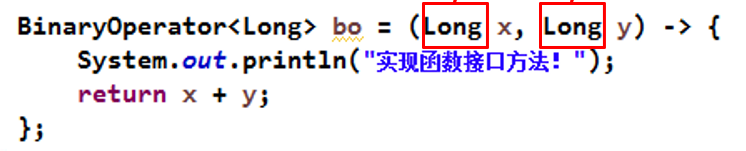
* 语法格式四：有两个以上的参数，有返回值，并且 Lambda 体中有多条语句



* 语法格式五：若 Lambda 体中只有一条语句， return 和 大括号都可以省略不写



* 语法格式六：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”



上联：左右遇一括号省

下联：左侧推断类型省

横批：能省则省

## 使用lambda表达式的几点必要条件

1. 方法的参数必须为接口才能使用Lambda
2. 并且作为参数的接口中有且仅有一个抽象方法（@ FunctionalInterface）

## 作为参数传递Lambda 表达式

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。

文本

描述已自动生成

Lambda 表达式作为参数传递：

解析: 首先lambda表达式实现了getValue()方法,

然后在operation方法中, 调用lambda定义的方法(其中参数是operation的num, 而且其类型必须和MyFun接口的类型一致)

文本

描述已自动生成

# 什么是函数式接口

@FunctionalInterface被该注解修饰的接口，只能有一个抽象方法。

Lambda 表达式需要“函数式接口”的支持

函数式接口：接口中只有一个抽象方法的接口，称为函数式接口。

可以使用注解 @FunctionalInterface 修饰

@FunctionalInterface 可以检查是否是函数式接口

# Lambda表达式原理

1. 在类编译时，会生成一个私有静态方法+一个内部类；
2. 在内部类中实现了函数式接口，在实现接口的方法中，会调用编译器生成的静态方法；
3. 在使用lambda表达式的地方，通过传递内部类实例，来调用函数式接口方法。

文本

描述已自动生成

编译器把我们写的lambda表达式转换成了一个静态的 私有的 函数，

函数的方法体就是lambda表达式的代码， 通过调用这个函数来解决传递一段代码的问题。

## Lambda表达式与匿名内部类的区别

1. 所需类型不同

匿名内部类：可以是接口，也可以是抽象类，还可以是具体类。

Lambda表达式：只能是接口

1. 抽象方法的数量不一样

匿名内部类，中接口的抽象方法的数量是随意的。

Lambda表达式中只能有一个抽象方法(函数式接口)。

1. 实现原理不同

匿名内部类：编译之后会产生一个单独的.class字节码文件

Lambda表达式：编译之后不会产生一个单独的.class字节码文件。对应的字节码会在运行的时候动态生成。

1. 调用默认方法不同

匿名内部类实现的抽象方法的方法体允许调用接口中定义的默认方法；

但 Lambda 表达式的代码块不允许调用接口中定义的默认方法。

# 接口增强

<https://blog.csdn.net/aitangyong/article/details/54134385>

在JDK8之前，接口内部只能有静态常量和抽象方法。

1. Interface 接口名{
2. 静态常量；
3. 抽象方法；
4. }

在JDK8，接口中可以有默认方法和静态方法

1. Interface 接口名{
2. 静态常量；
3. 抽象方法；
4. 默认方法;
5. 静态方法；
6. }
7. public interface JDK8Interface {
9. *// static修饰符定义静态方法*
10. static void staticMethod() {
11. System.out.println("接口中的静态方法");
12. }
14. *// default修饰符定义默认方法*
15. default void defaultMethod() {
16. System.out.println("接口中的默认方法");
17. }
18. }

## 默认方法

1. 对于默认接口创建自己的默认方法，可以覆盖重写接口的默认方法：
2. 使用 super 来调用指定接口的默认方法：

## 静态方法

1. 静态方法是不可以被重写的。
2. 只能通过 类名.静态方法名(); 调用

## 静态方法与默认方法的区别

1. 默认方法通过实例调用, 静态方法通过接口名调用
2. 默认方法可以被继承, 被重写
3. 静态方法不能被继承, 不能被重写

# 函数式接口

我们知道使用Lambda表达式的前提是需要有函数式接口，而Lambda表达式使用时不关心接口名、抽象方法名。只关心抽象方法的参数列表和返回值类型。因此为了让我们使用Lambda表达式更加的方便，在JDK中提供了大量常用的函数式接口

## 常用的函数式接口

在JDK中帮我们提供的有函数式接口，主要是在 java.util.function 包中。

## Supplier 生产数据

无参有返回值的接口，对于的Lambda表达式需要提供一个返回数据的类型。

文本

描述已自动生成

使用

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## Consumer 消费数据

Consumer接口是用来消费数据的，使用的时候需要指定一个泛型来定义参数类型。

文本

描述已自动生成

使用

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

默认方法:andThen如果一个方法的参数和返回值全部是Consumer类型，那么就可以实现效果，消费一个数据的时候，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方法

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

## Function

有参有返回值的接口，Function接口是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。有参数有返回值。

# 方法引用与构造器引用

## 方法引用

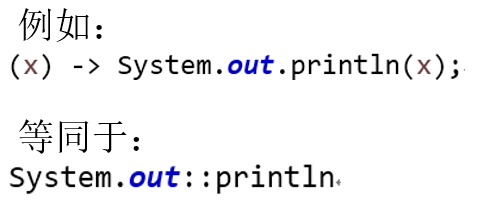
当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！

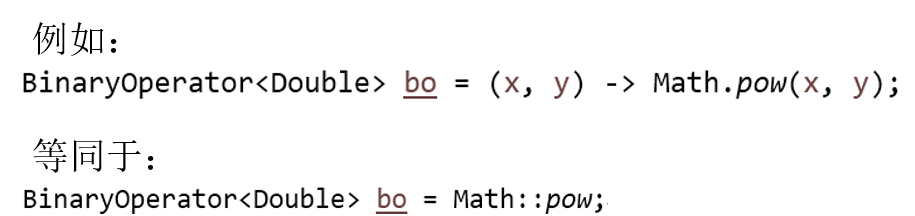
（实现抽象方法的参数列表，必须与方法引用方法的参数列表保持一致！）

方法引用：使用操作符 “::” 将方法名和对象或类的名字分隔开来。

如下三种主要使用情况：

### 对象::实例方法





### 类::静态方法

|  |
| --- |
| 1. //lambda 2. Comparator<Integer> com = (x, y) -> Integer.compare(x, y); 4. //等效于 5. Comparator<Integer> com2 = Integer::compare; |

### 类::实例方法

|  |
| --- |
| 1. //lambda方法 2. Function<Employee, String> fun = (e) -> e.show(); 4. //类名::实例方法 5. Function<Employee, String> fun2 = Employee::show; |

## 注意：

①方法引用所引用的方法的参数列表与返回值类型，需要与函数式接口中抽象方法的参数列表和返回值类型保持一致！

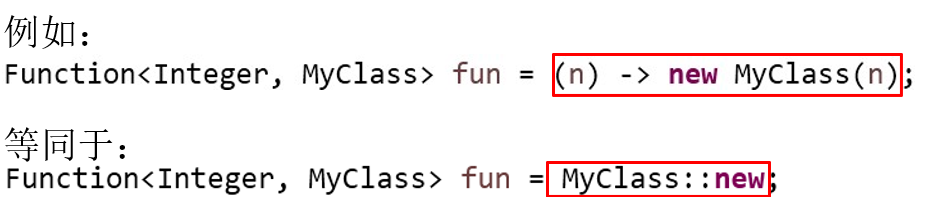
②若Lambda 的参数列表的第一个参数，是实例方法的调用者，第二个参数(或无参)是实例方法的参数时，格式： ClassName::MethodName

# 构造器引用

格式： ClassName::new

与函数式接口相结合，自动与函数式接口中方法兼容。

可以把构造器引用赋值给定义的方法，与构造器参数 列表要与接口中抽象方法的参数列表一致！

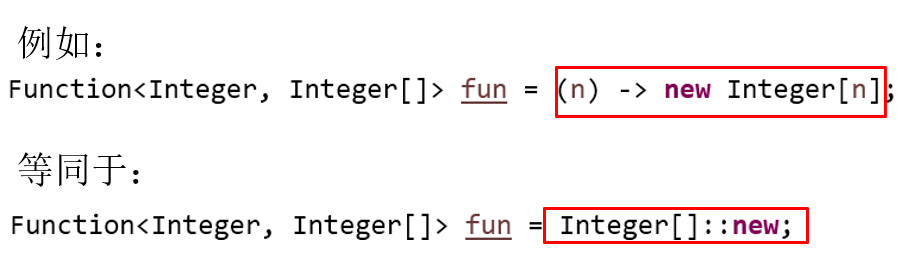


构造函数多个参数

|  |
| --- |
| 1. Function<String, Employee> fun = Employee::**new**; 3. BiFunction<String, Integer, Employee> fun2 = Employee::**new**; |

# 数组引用

格式： type[] :: new



# 强大的Stream API

## 什么是Stream

流(Stream) 到底是什么呢？

是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。

“集合讲的是数据，流讲的是计算！”

## 注意：

①Stream 自己不会存储元素。

②Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。

③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。(在多线程的时候可能会产生问题)

# Stream的三个步骤

1. 创建 Stream

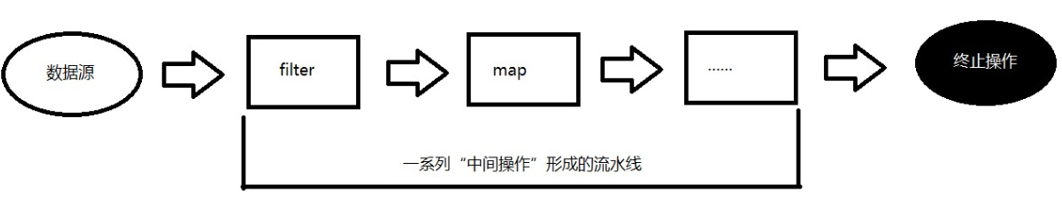
一个数据源（如：集合、数组），获取一个流

1. 中间操作

一个中间操作链，对数据源的数据进行处理

1. 终止操作(终端操作)

一个终止操作，执行中间操作链，并产生结果



## 创建Stream

### 使用collection系列的stream() 与 parallelStream()

|  |
| --- |
| 1. //1. Collection 提供了两个方法  stream() 与 parallelStream() 2. List<String> list = **new** ArrayList<>(); 3. Stream<String> stream = list.stream(); //获取一个顺序流 4. Stream<String> parallelStream = list.parallelStream(); //获取一个并行流 |

### 通过 Arrays 中的 stream() 获取一个数组流

|  |
| --- |
| 1. //2. 通过 Arrays 中的 stream() 获取一个数组流 2. Integer[] nums = **new** Integer[10]; 3. Stream<Integer> stream1 = Arrays.stream(nums); |

### 通过Stream中的静态方法of()

|  |
| --- |
| 1. //3. 通过 Stream 类中静态方法 of() 2. Stream<Integer> stream2 = Stream.of(1,2,3,4,5,6); |

### 创建无限流

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

1. 迭代

public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)

1. 生成

public static<T> Stream<T> generate(Supplier<T> s)

|  |
| --- |
| 1. //4. 创建无限流 2. //迭代 3. Stream<Integer> stream3 = Stream.iterate(0, (x) -> x + 2).limit(10); 4. stream3.forEach(System.out::println); |

### 生成随机数

|  |
| --- |
| 1. //生成 2. Stream<Double> stream4 = Stream.generate(Math::random).limit(2); 3. stream4.forEach(System.out::println); |

# Stream 的中间操作

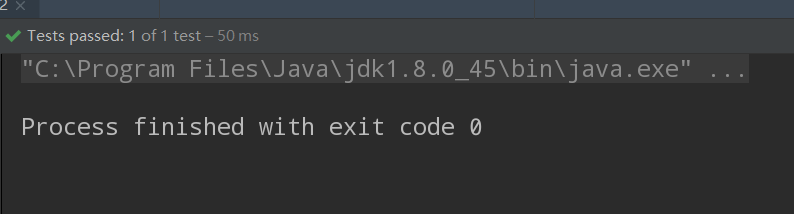
多个中间操作可以连接起来形成一个流水线，除非流水 线上触发终止操作，否则中间操作不会执行任何的处理！

而在终止操作时一次性全部处理，称为“惰性求值”。

## 没有终止操作(或者终止操作不执行)

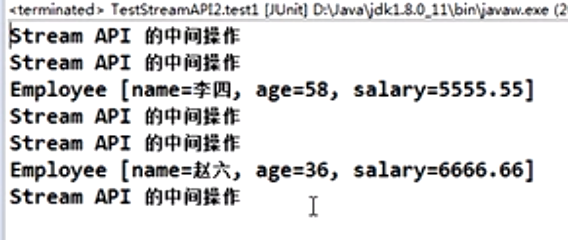
如果没有终止操作(或者终止操作不执行),中间操作不会执行任何处理

|  |
| --- |
| 1. //所有的中间操作不会做任何的处理 2. Stream<Employee> stream = emps.stream() 3. .filter((e) -> { 4. System.out.println("测试中间操作"); 5. **return** e.getAge() <= 35; 6. }); |



## 当终止操作执行时

|  |
| --- |
| 1. //所有的中间操作不会做任何的处理 2. Stream<Employee> stream = emps.stream() 3. .filter((e) -> { 4. System.out.println("测试中间操作"); 5. **return** e.getAge() <= 35; 6. }); 8. //只有当做终止操作时，所有的中间操作会一次性的全部执行，称为“惰性求值” 9. stream.forEach(System.out::println); |

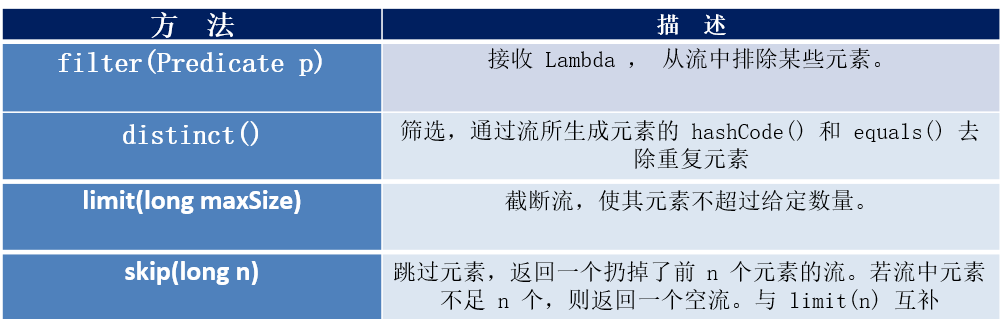


# 内部迭代

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

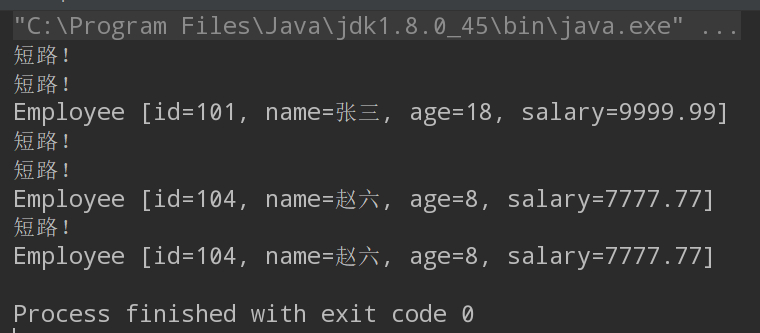
|  |
| --- |
| 1. //内部迭代 2. stream.forEach(System.out::println); 4. /\*=====================\*/ 6. //外部迭代 7. Iterator<Employee> it = emps.iterator(); 8. **while**(it.hasNext()){ 9. System.out.println(it.next()); 10. } |

# 筛选与切片



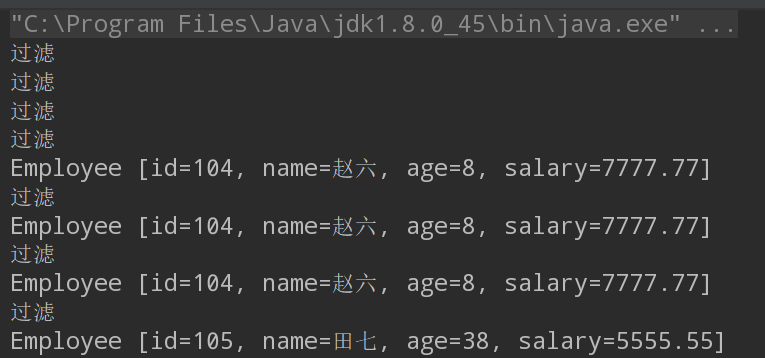
## Filter和limit的使用

|  |
| --- |
| 1. //emps有七个对象, 其中满足要求的有4个 2. //其中 短路!出现了5次 3. //输出Employee对象的有三次 4. //当Employee出现了三次之后, 剩下的所有元素就不会进行遍历了 5. emps.stream() 6. .filter((e) -> { 7. System.out.println("短路！"); // &&  || 8. **return** e.getSalary() >= 7000; 9. }).limit(3) 10. .forEach(System.out::println); |



## Skip的使用

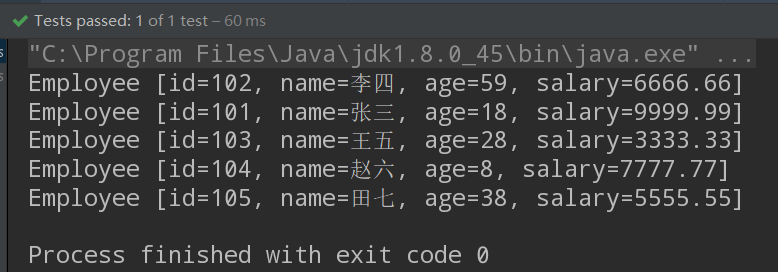
|  |
| --- |
| 1. //emps有七个对象, 其中满足要求的有6个 2. //其中 过滤 出现了7次 3. //输出Employee对象的有四次 4. //其中有两个Employee跳过了两次 5. emps.stream().filter((e) -> { 6. System.out.println("过滤"); 7. **return** e.getSalary() >= 5000; 8. }).skip(2).forEach(System.out::println); |



## Distinct去除重复元素

需要重写equals()和hash()

|  |
| --- |
| 1. emps.stream() 2. .distinct() 3. .forEach(System.out::println); |



# 映射

## Map

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的元素。

|  |
| --- |
| 1. List<String> strList = Arrays.asList("aaa", "bbb", "ccc", "ddd", "eee"); 3. Stream<String> stream = strList.stream() 4. .map(String::toUpperCase); 5. stream.forEach(System.out::println); |

## Flatmap

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

## 实例

注意两个返回值的 类型不同

调用map, 返回的是Stream<Stream<Character>>

如果使用的是map方法，返回的是[ ...['y', 'o', 'u', 'r'], ['n', 'a', 'm', 'e']]

调用flatMap,返回是Stream<Character>

如果使用的是flatMap方法，返回的是['y', 'o', 'u', 'r', 'n', 'a', 'm', 'e']

|  |
| --- |
| 1. **public** **static** Stream<Character> characterStream(String s) { 2. List<Character> result = **new** ArrayList<>(); 3. **for** (**char** c : s.toCharArray()) { 4. result.add(c); 5. } 6. **return** result.stream(); 7. } 8. @Test 9. **public** **void** test1() { 10. List<String> words = **new** ArrayList<String>(); 11. words.add("your"); 12. words.add("name"); 13. Stream<Stream<Character>> result = words.stream().map(w -> characterStream(w)); 15. System.out.println("==============================="); 16. Stream<Character> letters = words.stream().flatMap(w -> characterStream(w)); 17. } |

# 排序

sorted()——自然排序

sorted(Comparator com)——定制排序