# 第17章 Lambda表达式与StreamAPI

Java 8 于2014年3月14号发布，可以看成是自Java 5 以来最具革命性的版本。

## 17.1 Lambda表达式

Lambda表达式不是Java最早使用的，狠多语言就支持Lambda表达式，例如：C++，C#，Python，Scala等。如果有Python或者Javascript的语言基础，对理解Lambda表达式有很大帮助，可以这么说lambda表达式其实就是实现SAM接口的语法糖，使得Java也算是支持函数式编程的语言。Lambda**写的好**可以极大的减少代码冗余，同时可读性也好过冗长的匿名内部类。

### 17.1.1 Lambda表达式引入

#### 1、示例1：Runnable实现线程

|  |
| --- |
| @Test  **public** **void** test1(){  **new** Thread(**new** Runnable(){  **public** **void** run(){  System.***out***.println("do something..");  }  }).start();  } |
| @Test  **public** **void** test2(){  **new** Thread(() -> System.***out***.println("do something..")).start();  } |

#### 2、示例2：foreach遍历

|  |
| --- |
| @Test  **public** **void** test3(){  List<String> list = Arrays.*asList*("hello","java","atguigu","lambda");  **for** (String string : list) {  System.***out***.println(string);  }  } |
| @Test  **public** **void** test4(){  List<String> list = Arrays.*asList*("hello","java","atguigu","lambda");  list.forEach(System.***out***::println);  } |

#### 3、示例3：FileFilter文件过滤

|  |
| --- |
| @Test  **public** **void** test5(){  //1.文件目录  File fileDir=**new** File("D:/resource");  //2.创建筛选规则，帅选出所有.java文件  FileFilter filter=**new** FileFilter() {  @Override  **public** **boolean** accept(File file) {  **if**(!file.isDirectory()&&file.getName().endsWith(".java")){  **return** **true**;  }  **return** **false**;  }  };  //3.得到筛选文件  File[] files=fileDir.listFiles(filter);  **for** (File file : files) {  System.***out***.println(file);  }  } |
| @Test  **public** **void** test6(){  //1.文件目录  File fileDir=**new** File("D:/resource");  //2.得到筛选文件，帅选出所有.java文件  File[] files=fileDir.listFiles((file) -> !file.isDirectory() && file.getName().endsWith(".java"));  //3、遍历查看结果  Arrays.*asList*(files).forEach(System.***out***::println);  } |

### 17.1.2 函数式接口概念

Lambda表达式是用来实现SAM接口的，所谓SAM接口就是Single Abstract Method，即该接口中只有一个抽象方法需要实现，当然该接口可以包含其他非抽象方法。

其实只要满足“SAM”特征的接口都可以称为函数式接口，但是如果要更明确一点，最好在声明接口时，加上@FunctionalInterface。

JDK1.8之前，核心类库中就已经存在很多SAM接口了，例如：

（1）java.lang.Runnable

（2）java.util.concurrent.Callable

（3）java.util.Comparator

（4）java.lang.Comparable

（5）java.lang.Iterable

（6）java.io.FileFilter

（7）java.lang.reflect.InvocationHandler

......等

但是在JDK1.8，只有（1）（2）（3）（6）加了@FunctionalInterface，那些没有加@FunctionalInterface的SAM接口，现在使用Lambda表达式实现，但是存在将来增加抽象方法变成非SAM接口的风险，因此建议只对加了@FunctionalInterface的接口使用Lambda表达式实现。

JDK1.8在java.util.function包增加了很多函数式接口，不过他们可以归纳为四类：消费型接口、供给型接口、功能型接口、判断型接口，一共43个，基本上可以满足开发中函数式接口的基本使用需求，如你在开发中需要设计函数式接口，请先从以下接口中选择是否有满足需求的，如果有几不需要重新设计了。

#### 1、消费型接口

这类接口的抽象方法特点：有形参，但是返回值类型是void

|  |  |  |
| --- | --- | --- |
| 接口名 | 抽象方法 | 描述 |
| Consumer<T> | void accept(T t) | 接收一个对象用于完成功能 |
| BiConsumer<T,U> | void accept(T t, U u) | 接收一个对象用于完成功能 |
| DoubleConsumer | void accept(double value) | 接收一个double值 |
| IntConsumer | void accept(int value) | 接收一个int值 |
| LongConsumer | void accept(long value) | 接收一个long值 |
| ObjDoubleConsumer<T> | void accept(T t, double value) | 接收一个对象和一个double值 |
| ObjIntConsumer<T> | void accept(T t, int value) | 接收一个对象和一个int值 |
| ObjLongConsumer<T> | void accept(T t, long value) | 接收一个对象和一个long值 |

#### 2、供给型接口

这类接口的抽象方法特点：无参，但是无返回值

|  |  |  |
| --- | --- | --- |
| 接口名 | 抽象方法 | 描述 |
| Supplier<T> | T get() | 返回一个对象 |
| BooleanSupplier | boolean getAsBoolean() | 返回一个boolean值 |
| DoubleSupplier | double getAsDouble() | 返回一个double值 |
| IntSupplier | int getAsInt() | 返回一个int值 |
| LongSupplier | long getAsLong() | 返回一个long值 |

#### 3、判断型接口

这里接口的抽象方法特点：有参，但是返回值类型是boolean结果。

|  |  |  |
| --- | --- | --- |
| 接口名 | 抽象方法 | 描述 |
| Predicate<T> | boolean test(T t) | 接收一个对象 |
| BiPredicate<T,U> | boolean test(T t, U u) | 接收两个对象 |
| DoublePredicate | boolean test(double value) | 接收一个double值 |
| IntPredicate | boolean test(int value) | 接收一个int值 |
| LongPredicate | boolean test(long value) | 接收一个long值 |

#### 4、功能型接口

这类接口的抽象方法特点：既有参数又有返回值

|  |  |  |
| --- | --- | --- |
| 接口名 | 抽象方法 | 描述 |
| Function<T,R> | R apply(T t) | 接收一个T类型对象，返回一个R类型对象结果 |
| UnaryOperator<T> | T apply(T t) | 接收一个T类型对象，返回一个T类型对象结果 |
| DoubleFunction<R> | R apply(double value) | 接收一个double值，返回一个R类型对象 |
| IntFunction<R> | R apply(int value) | 接收一个int值，返回一个R类型对象 |
| LongFunction<R> | R apply(long value) | 接收一个long值，返回一个R类型对象 |
| ToDoubleFunction<T> | double applyAsDouble  (T value) | 接收一个T类型对象，返回一个double |
| ToIntFunction<T> | int applyAsInt(T value) | 接收一个T类型对象，返回一个int |
| ToLongFunction<T> | long applyAsLong  (T value) | 接收一个T类型对象，返回一个long |
| DoubleToIntFunction | int applyAsInt  (double value) | 接收一个double值，返回一个int结果 |
| DoubleToLongFunction | long applyAsLong  (double value) | 接收一个double值，返回一个long结果 |
| IntToDoubleFunction | double applyAsDouble  (int value) | 接收一个int值，返回一个double结果 |
| IntToLongFunction | long applyAsLong  (int value) | 接收一个int值，返回一个long结果 |
| LongToDoubleFunction | double applyAsDouble  (long value) | 接收一个long值，返回一个double结果 |
| LongToIntFunction | int applyAsInt  (long value) | 接收一个long值，返回一个int结果 |
| DoubleUnaryOperator | double applyAsDouble  (double operand) | 接收一个double值，返回一个double |
| IntUnaryOperator | int applyAsInt  (int operand) | 接收一个int值，返回一个int结果 |
| LongUnaryOperator | long applyAsLong  (long operand) | 接收一个long值，返回一个long结果 |
|  |  |  |
| BiFunction<T,U,R> | R apply(T t, U u) | 接收一个T类型和一个U类型对象，返回一个R类型对象结果 |
| BinaryOperator<T> | T apply(T t, T u) | 接收两个T类型对象，返回一个T类型对象结果 |
| ToDoubleBiFunction<T,U> | double applyAsDouble  (T t, U u) | 接收一个T类型和一个U类型对象，返回一个double |
| ToIntBiFunction<T,U> | int applyAsInt(T t, U u) | 接收一个T类型和一个U类型对象，返回一个int |
| ToLongBiFunction<T,U> | long applyAsLong  (T t, U u) | 接收一个T类型和一个U类型对象，返回一个long |
| DoubleBinaryOperator | double applyAsDouble  (double left, double right) | 接收两个double值，返回一个double结果 |
| IntBinaryOperator | int applyAsInt  (int left, int right) | 接收两个int值，返回一个int结果 |
| LongBinaryOperator | long applyAsLong(long left, long right) | 接收两个long值，返回一个long结果 |

### 17.1.3 Lambda表达式

Lambda表达式是用来实现SAM接口的，它相当于一个匿名函数，语法格式如下：

|  |
| --- |
| (Type1 param1, Type2 param2, ..., TypeN paramN) -> {  statment1;  statment2;  //.............  return statmentM;  } |

简单的说就是：

|  |
| --- |
| (形参列表) -> {Lambda体} |

这个操作符为 “->” ， 该操作符被称为 Lambda 操作符或箭头操作符。它将 Lambda 分为两个部分：

（1）左侧：指定了 Lambda 表达式需要的参数列表，它其实就是函数式接口的抽象方法的形参列表

（2）右侧：指定了 Lambda 体，即 Lambda 表达式要执行的功能，它其实就是实现函数式接口的抽象方法的方法体。

例如：Lambda表达式只写了Runnable接口的抽象方法public void run()的()空参列表以及对run()方法的实现代码。

|  |
| --- |
| @Test  **public** **void** test1(){  **new** Thread(**new** Runnable(){  **public** **void** run(){  System.***out***.println("do something..");  }  }).start();  } |
| @Test  **public** **void** test2(){  **new** Thread(() -> System.***out***.println("do something..")).start();  } |

格式要求：

1、关于(形参列表)

1. 如果没有形参，那么()不可以省略；
2. 如果有形参，并且形参只有一个，并且形参类型已知或可推断，那么可以省略()和数据类型，只写形参名；
3. 如果形参不止一个，那么()不可以省略，但是如果形参类型已知或可推断，那么可以数据类型。

2、关系{Lambda体}

（1）如果函数式接口的抽象方法有返回值，即返回值类型不是void，那么Lambda体必须要有“return 返回值;”语句；

（2）如果{Lambda体}只有一个语句，那么{}可以省略，如果{}省略了，那么语句后面的;也要省略，如果{Lambda体}只有一个return 返回值;语句，那么{return;}都可以省略，Lambda体只写返回值即可。

#### 1、形式一：无参无返回值

代码示例：Runnable函数式接口

|  |
| --- |
| **package** com.atguigu.lambda.learn;  **import** org.junit.Test;  **public** **class** TestLambda1 {  @Test  **public** **void** test1(){  //不使用Lambda表达式  **new** Thread(**new** Runnable(){  **public** **void** run(){  System.***out***.println("do something..");  }  }).start();  }    @Test  **public** **void** test2(){  //使用Lambda表达式  **new** Thread(() -> {System.***out***.println("do something..");}).start();  }    @Test  **public** **void** test3(){  //省略了可以省略的部分  **new** Thread(() -> System.***out***.println("do something..")).start();  }  } |

#### 2、形式二：有参无返回值

代码示例：Consumer<T>接口

在JDK1.8中Collection集合接口的父接口Iterable接口中增加了一个默认方法：

default void forEach(Consumer<? super T> action) ，遍历Collection集合的每个元素，执行“xxx消费型”操作。

在JDK1.8中Map集合接口中增加了一个默认方法：

default void forEach(BiConsumer<? super K,? super V> action)遍历Map集合的每对映射关系，执行“xxx消费型”操作。

|  |
| --- |
| **package** com.atguigu.lambda.learn;  **import** java.util.Arrays;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map.Entry;  **import** java.util.Set;  **import** org.junit.Test;  **public** **class** TestLambda2 {  @Test  **public** **void** test1(){  //不使用Lambda表达式  List<String> list = Arrays.*asList*("hello","java","lambda","atguigu");  **for** (String string : list) {  System.***out***.println(string);  }    HashMap<Integer,String> map = **new** HashMap<>();  map.put(1, "hello");  map.put(2, "java");  map.put(3, "lambda");  map.put(4, "atguigu");  Set<Entry<Integer, String>> entrySet = map.entrySet();  **for** (Entry<Integer, String> entry : entrySet) {  System.***out***.println(entry.getKey() + "->" + entry.getValue());  }  }    @Test  **public** **void** test2(){  //使用Lambda表达式  List<String> list = Arrays.*asList*("hello","java","lambda","atguigu");  list.forEach((String s) -> {System.***out***.println(s);});    HashMap<Integer,String> map = **new** HashMap<>();  map.put(1, "hello");  map.put(2, "java");  map.put(3, "lambda");  map.put(4, "atguigu");  map.forEach((Integer k,String v) -> {System.***out***.println(k+"->"+v);});  }    @Test  **public** **void** test3(){  //省略了可以省略的部分  List<String> list = Arrays.*asList*("hello","java","lambda","atguigu");  list.forEach(s -> System.***out***.println(s));    HashMap<Integer,String> map = **new** HashMap<>();  map.put(1, "hello");  map.put(2, "java");  map.put(3, "lambda");  map.put(4, "atguigu");  map.forEach((k,v) -> System.***out***.println(k+"->"+v));  }  } |

#### 3、形式三：无参有返回值

代码示例：Supplier<T>接口

在JDK1.8中增加了StreamAPI，Stream是一个数据流，一个不同于集合的数据流。关于Stream的详细介绍请看$17.3 Stream API。

|  |
| --- |
| **package** com.atguigu.lambda.learn;  **import** java.util.stream.Stream;  **import** org.junit.Test;  **public** **class** TestLambda3 {  @Test  **public** **void** test1(){  //使用Lambda表达式  Stream<Double> stream = Stream.*generate*(() -> {**return** Math.*random*();});  stream.forEach((Double num) -> {System.***out***.println(num);});  }    @Test  **public** **void** test2(){  //省略了可以省略的部分  Stream<Double> stream = Stream.*generate*(() -> Math.*random*());  stream.forEach(num -> System.***out***.println(num));  }  } |

#### 4、形式四：有参有返回值

代码示例：Funtion<T,R>接口

在JDK1.8时Map接口增加了很多方法，其中一个是：

default void replaceAll(BiFunction<? super K,? super V,? extends V> function) 按照function指定的操作替换map中的value。

|  |
| --- |
| **package** com.atguigu.lambda.learn;  **import** java.util.HashMap;  **import** org.junit.Test;  **public** **class** TestLambda4 {  @Test  **public** **void** test1(){  HashMap<String,Employee> map = **new** HashMap<>();  map.put("张三", **new** Employee("张三", 8000));  map.put("李四", **new** Employee("李四", 9000));  map.put("王五", **new** Employee("王五", 12000));  map.put("赵六", **new** Employee("赵六", 11000));    //把薪资低于10000的工资修改为10000  map.replaceAll((String k,Employee v) -> {  **if**(v.getSalary()<10000){  v.setSalary(10000);  }  **return** v;  });  map.forEach((String k, Employee v) -> {System.***out***.println(k+"->"+v);});  }    @Test  **public** **void** test2(){  HashMap<String,Employee> map = **new** HashMap<>();  map.put("张三", **new** Employee("张三", 8000));  map.put("李四", **new** Employee("李四", 9000));  map.put("王五", **new** Employee("王五", 12000));  map.put("赵六", **new** Employee("赵六", 11000));    //把薪资低于10000的工资修改为10000  //省略了可省略部分  map.replaceAll((k,v) -> {**if**(v.getSalary()<10000)v.setSalary(10000);**return** v;});  map.forEach((k, v) -> System.***out***.println(k+"->"+v));  }  } |
| **package** com.atguigu.lambda.bean;  **public** **class** Employee {  **private** String name;  **private** **double** salary;  **public** Employee(String name, **double** salary) {  **super**();  **this**.name = name;  **this**.salary = salary;  }  **public** Employee() {  **super**();  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **double** getSalary() {  **return** salary;  }  **public** **void** setSalary(**double** salary) {  **this**.salary = salary;  }  @Override  **public** String toString() {  **return** "Employee [name=" + name + ", salary=" + salary + "]";  }  } |

代码示例：Predicate<T>接口

JDK1.8时，Collecton<E>接口增加了一下方法，其中一个如下：

default boolean removeIf(Predicate<? super E> filter) 用于删除集合中满足filter指定的条件判断的。

|  |
| --- |
| @Test  **public** **void** test3(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三", 8000));  list.add(**new** Employee("李四", 9000));  list.add(**new** Employee("王五", 12000));  list.add(**new** Employee("赵六", 11000));    //删除那些薪资低于10000的  list.removeIf((Employee e) -> {**return** e.getSalary()>10000;});  list.forEach((Employee e) -> {System.***out***.println(e);});  }    @Test  **public** **void** test4(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三", 8000));  list.add(**new** Employee("李四", 9000));  list.add(**new** Employee("王五", 12000));  list.add(**new** Employee("赵六", 11000));    //删除那些薪资低于10000的  //省略了可以省略的部分  list.removeIf(e -> e.getSalary()>10000);  list.forEach(e -> System.***out***.println(e));  } |

### 17.1.4 方法引用和构造器引用

当Lambda体的实现是通过调用一个现有的方法来完成功能时，那么可以考虑再次简化代码，使用方法引用和构造器引用。

此时要求函数式接口的抽象方法的形参列表与返回值类型与该方法的形参列表与返回值类型要对应。

方法引用的语法格式：

|  |
| --- |
| 类或对象：：方法名 |

构造器引用的语法格式：

|  |
| --- |
| 类名或数组类型：：new |

#### 1、对象::实例方法名

例如Consumer<T>的抽象方法void accept(T t)，它的Lambda体是通过调用System.out.println(x)来完成，它们都是有参无返回值。

示例代码：

|  |
| --- |
| @Test  **public** **void** test1(){  List<String> list = Arrays.*asList*("hello","java","lambda","atguigu");  //使用Lambda表达式  list.forEach(e -> System.***out***.println(e));  }    @Test  **public** **void** test2(){  List<String> list = Arrays.*asList*("hello","java","lambda","atguigu");  //使用方法引用  list.forEach(System.***out***::println);  } |

例如：Comparator<T>接口的抽象方法int compare(T t1,T t2)，它的Lambda体是通过调用Collator文本校对器对象的compare(t1,t2)方法来完成，他们的形参列表和返回值类型完成可以对上。

示例代码：

|  |
| --- |
| @Test  **public** **void** test3(){  //使用Lambda表达式  TreeSet<String> set = **new** TreeSet<>((t1,t2) -> Collator.*getInstance*().compare(t1, t2));  set.add("张三");  set.add("李四");  set.add("王五");  set.add("赵六");  set.forEach(e -> System.***out***.println(e));  }    @Test  **public** **void** test4(){  //使用方法引用  TreeSet<String> set = **new** TreeSet<>(Collator.*getInstance*()::compare);  set.add("张三");  set.add("李四");  set.add("王五");  set.add("赵六");  set.forEach(System.***out***::println);  } |

#### 2、类::静态方法名

例如：Supplier<T>的抽象方法T get()，它的lambda体是通过调用Math.random()来完成的，它们都是无参有返回值。

示例代码：

|  |
| --- |
| @Test  **public** **void** test5(){  //使用Lambda表达式  Stream<Double> stream = Stream.*generate*(() -> Math.*random*());  stream.forEach(num -> System.***out***.println(num));  }  @Test  **public** **void** test6(){  //使用方法引用  Stream<Double> stream = Stream.*generate*(Math::*random*);  stream.forEach(System.***out***::println);  } |

#### 3、类::实例方法名

例如：Comparator<T>接口的抽象方法int compare(T t1,T t2)，它的Lambda体是通过调用String对象的compareToIgnoreCase(String o)来完成的，而且这里调用compareToIgnoreCase方法是用compare()方法的第一个参数，而compare()方法剩下的参数正好是给compareToIgnoreCase的实参，它俩的返回值类型都是int。

示例代码：

|  |
| --- |
| @Test  **public** **void** test7(){  String[] arr = {"Hello","hello","abc","world","ABC"};  //使用generate  Arrays.*sort*(arr, (s1,s2)-> s1.compareToIgnoreCase(s2));  System.***out***.println(Arrays.*toString*(arr));  }    @Test  **public** **void** test8(){  String[] arr = {"Hello","hello","abc","world","ABC"};  //使用方法引用  Arrays.*sort*(arr, String::compareToIgnoreCase);  System.***out***.println(Arrays.*toString*(arr));  } |

#### 4、类名::new

例如：在JDK1.8中增加了一个工具类Optional<T>，它是一个容器，可以用来包装一个对象，如果所包装的对象不为null，那么通过get()方法可以获取到这个对象，如果所包装的对象是null，调用get()方法会报异常，所以很多时候需要通过调用orElseGet(Supplier<? extends T> other) 来获取，即如果该对象存在，就返回该对象，否则返回由Supplier接口实现所提供的对象。关于Optional类的详细介绍请看$17.2 Optinal类

示例代码：

|  |
| --- |
| @Test  **public** **void** test9(){  HashMap<String,Employee> map = **new** HashMap<>();  map.put("张三", **new** Employee("张三", 8000));  map.put("李四", **new** Employee("李四", 9000));    Optional<Employee> opt = Optional.*ofNullable*(map.get("王五"));  //使用Lambda表达式  Employee emp = opt.orElseGet(() -> **new** Employee());  System.***out***.println(emp);  }    @Test  **public** **void** test10(){  HashMap<String,Employee> map = **new** HashMap<>();  map.put("张三", **new** Employee("张三", 8000));  map.put("李四", **new** Employee("李四", 9000));    Optional<Employee> opt = Optional.*ofNullable*(map.get("王五"));  //使用构造器引用  Employee emp = opt.orElseGet(Employee::**new**);  System.***out***.println(emp);  } |

#### 5、数组类型::new

例如：我声明了一个工具方法createArray，用于创建一个长度为2的幂次方的数组。

|  |
| --- |
| **public** <R> R[] createArray(Function<Integer,R[]> fun,**int** length){  **int** n = length - 1;  n |= n >>> 1;  n |= n >>> 2;  n |= n >>> 4;  n |= n >>> 8;  n |= n >>> 16;  length = n < 0 ? 1 : n + 1;  **return** fun.apply(length);  }    @Test  **public** **void** test11(){  //使用Lambda表达式  Object[] arr1 = createArray(len -> **new** Object[len], 5);  System.***out***.println(arr1.length);  //使用构造器引用  Object[] arr2 = createArray(Object[]::**new**, 5);  System.***out***.println(arr2.length);  } |

## 17.2 Optional类

到目前为止，臭名昭著的空指针异常是导致Java应用程序失败的最常见原因。以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。

Optional实际上是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

创建Optional类对象的方法：

* Optional.of(T t) : 创建一个 Optional 实例，t必须非空；
* Optional.empty() : 创建一个空的 Optional 实例
* Optional.ofNullable(T t)：t可以为null

判断Optional容器中是否包含对象：

* boolean isPresent() : 判断是否包含对象
* void ifPresent(Consumer<? super T> consumer) ：如果有值，就执行Consumer接口的实现代码，并且该值会作为参数传给它。

获取Optional容器的对象：

* T get(): 如果调用对象包含值，返回该值，否则抛异常
* T orElse(T other) ：如果有值则将其返回，否则返回指定的other对象。
* T orElseGet(Supplier<? extends T> other) ：如果有值则将其返回，否则返回由Supplier接口实现提供的对象。
* T orElseThrow(Supplier<? extends X> exceptionSupplier) ：如果有值则将其返回，否则抛出由Supplier接口实现提供的异常。

示例代码：

|  |
| --- |
| @Test  **public** **void** test1(){  Boy b = **new** Boy("张三");  Optional<Girl> opt = Optional.*ofNullable*(b.getGrilFriend());  //如果女朋友存在就打印女朋友的信息  opt.ifPresent(System.***out***::println);  }    @Test  **public** **void** test2(){  Boy b = **new** Boy("张三");  Optional<Girl> opt = Optional.*ofNullable*(b.getGrilFriend());  //如果有女朋友就返回他的女朋友，否则只能欣赏“嫦娥”了  Girl girl = opt.orElse(**new** Girl("嫦娥"));  System.***out***.println("他的女朋友是：" + girl.getName());  } |

对Optional中的对象进行xx操作：

* Optional<T> filter(Predicate<? super T> predicate) ：如果有值，就判断该值是否满足predicate的条件，否则返回空的Optional。
* Optional<U> map(Function<? super T,? extends U> mapper) ：如果有值，对该值执行Function接口的实现代码，并返回一个Optional的结果，否则报空指针异常。
* Optional<U> flatMap(Function<? super T,Optional<U>> mapper) ：如果有值，对该值执行Function接口的实现代码，并返回一个Optional的结果，否则报空指针异常。该方法和map方法的区别是，map方法的Function接口的实现不负责把结果包装为Optional对象，该操作由map方法自己完成，而flatMap方法的Function接口的实现直接把结果包装为Optional对象了，flatMap方法不需要再次包装了。

|  |
| --- |
| @Test  **public** **void** test3(){  Optional<Employee> opt = Optional.*of*(**new** Employee("张三", 8888));  //判断opt中员工对象是否满足条件，如果满足就保留，否则返回空  Optional<Employee> emp = opt.filter(e -> e.getSalary()>10000);  System.***out***.println(emp);  }    @Test  **public** **void** test4(){  Optional<Employee> opt = Optional.*of*(**new** Employee("张三", 8888));  //如果opt中员工对象不为空，就涨薪10%  Optional<Employee> emp = opt.map(e -> {e.setSalary(e.getSalary()%1.1);**return** e;});  System.***out***.println(emp);  } |

## 17.3 Stream API

Java8中有两大最为重要的改变。第一个是 Lambda 表达式；另外一个则是 Stream API。

Stream API ( java.util.stream) 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为Stream API可以极大提高Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream 是 Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。 使用Stream API 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用 Stream API 来并行执行操作。简言之，Stream API 提供了一种高效且易于使用的处理数据的方式。

Stream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，负责存储数据，Stream流讲的是计算，负责处理数据！”

注意：

①Stream 自己不会存储元素。

②Stream 不会改变源对象。每次处理都会返回一个持有结果的新Stream。

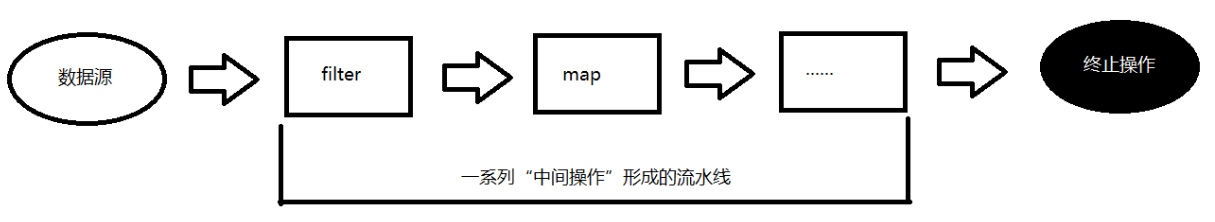
③Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

Stream 的操作三个步骤：

1- 创建 Stream：通过一个数据源（如：集合、数组），获取一个流

2- 中间操作：中间操作是个操作链，对数据源的数据进行n次处理，但是在终结操作前，并不会真正执行。

3- 终止操作：一旦执行终止操作，就执行中间操作链，最终产生结果并结束Stream。



### 17.3.1 创建Stream流

#### 1、创建 Stream方式一：通过集合

Java8 中的 Collection 接口被扩展，提供了两个获取流的方法：

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

#### 2、创建 Stream方式二：通过数组

Java8 中的 Arrays 的静态方法 stream() 可以获取数组流：

* 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)：返回一个整型数据流
* public static LongStream stream(long[] array)：返回一个长整型数据流
* public static DoubleStream stream(double[] array)：返回一个浮点型数据流

#### 3、创建 Stream方式三：通过Stream的of()

可以调用Stream类静态方法 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) : 返回一个顺序流

#### 4、创建 Stream方式四：创建无限流

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

* public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f):返回一个无限流
* public static<T> Stream<T> generate(Supplier<T> s) ：返回一个无限流

示例代码：

|  |
| --- |
| **package** com.atguigu.stream;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.stream.Stream;  **import** org.junit.Test;  **public** **class** TestCreateStream {  @Test  **public** **void** test1() {  List<String> list = Arrays.*asList*("hello", "world", "java", "lambda", "stream");  // 创建stream  Stream<String> stream = list.stream();  // 终结遍历stream流中的数据  stream.forEach(System.***out***::println);  }  @Test  **public** **void** test2() {  String[] array = { "hello", "world", "java", "lambda", "stream" };  // 创建stream  Stream<String> stream = Arrays.*stream*(array);  // 终结遍历stream流中的数据  stream.forEach(System.***out***::println);  }  @Test  **public** **void** test3() {  // 创建stream  Stream<String> stream = Stream.*of*("hello", "world", "java", "lambda", "stream");  // 终结遍历stream流中的数据  stream.forEach(System.***out***::println);  }  @Test  **public** **void** test4() {  // 创建stream，由奇数组成  Stream<Integer> stream = Stream.*iterate*(1, num->num+2);  // 终结遍历stream流中的数据  stream.forEach(System.***out***::println);  }  } |

### 17.3.2 中间处理数据操作

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”。

|  |  |
| --- | --- |
| **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 |
| **peek(Consumer action)** | 接收Lambda，对流中的每个数据执行Lambda体操作 |
| **sorted()** | 产生一个新流，其中按自然顺序排序 |
| **sorted(Comparator com)** | 产生一个新流，其中按比较器顺序排序 |
| **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 | |

1、示例代码：找出流所有偶数，并且要求不重复

|  |
| --- |
| @Test  **public** **void** test1(){  //取出所有偶数，并且要求不重复  Stream.*of*(1,1,3,3,4,4,6,6)  .filter(num -> num%2==0)  .distinct()  .forEach(System.***out***::println);  } |

2、示例代码：随机产生10个100以内整数，遍历显示，并找出最大值

|  |
| --- |
| @Test  **public** **void** test2(){  //随机产生10个100以内整数，遍历显示，并找出最大值  Random random = **new** Random();  Optional<Integer> max = Stream.*generate*(() -> random.nextInt(100))  .limit(10)  .peek(System.***out***::println)  .max(Integer::compareTo);  System.***out***.println("max = " + max.get());  } |

3、示例代码：按照薪资排序

|  |
| --- |
| @Test  **public** **void** test3(){  //按照薪资排序  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    list.stream()  .sorted((e1,e2)->e1.getSalary()==e2.getSalary()?0:(**int**)Math.*ceil*(e1.getSalary()-e2.getSalary()))  .forEach(System.***out***::println);  } |

4、示例代码：给每个员工涨薪10%

|  |
| --- |
| @Test  **public** **void** test4(){  //给每个员工涨薪10%  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    list.stream()  .map(e->{e.setSalary(e.getSalary()\*1.1);**return** e;})  .forEach(System.***out***::println);  } |

5、示例代码：求每个数的平方根

|  |
| --- |
| @Test  **public** **void** test5(){  //求每个数的平方根  Stream.*of*(4,9,16,25,49)  .mapToDouble(num -> Math.*sqrt*(num))  .forEach(System.***out***::println);  } |

6、示例代码：把首字母变成大写

|  |
| --- |
| @Test  **public** **void** test6(){  //把首字母变成大写  Stream.*of*("hello","world","java","stream").  map(s -> Character.*toUpperCase*(s.charAt(0)) + s.substring(1))  .forEach(System.***out***::println);  } |

7、示例代码：把每个单词拆成一个个字母

|  |
| --- |
| @Test  **public** **void** test7(){  //把每个单词拆成一个个字母  Stream.*of*("hello","world","java","stream")  .flatMap(t -> Stream.*of*(t.split("|")))  .forEach(System.***out***::println);  } |

### 17.3.3 终结操作

终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至是 void。流进行了终止操作后，不能再次使用。

|  |  |
| --- | --- |
| **方法** | **描述** |
| **boolean allMatch(Predicate p)** | 检查是否匹配所有元素 |
| **boolean anyMatch**(**Predicate p**) | 检查是否至少匹配一个元素 |
| **boolean noneMatch(Predicate  p)** | 检查是否没有匹配所有元素 |
| **Optional<T> findFirst()** | 返回第一个元素 |
| **Optional<T> findAny()** | 返回当前流中的任意元素 |
| **long count()** | 返回流中元素总数 |
| **Optional<T> max(Comparator c)** | 返回流中最大值 |
| **Optional<T> min(Comparator c)** | 返回流中最小值 |
| **void forEach(Consumer c)** | 迭代 |
| **T reduce(T iden, BinaryOperator b)** | 可以将流中元素反复结合起来，得到一个值。返回 T |
| **U reduce(BinaryOperator b)** | 可以将流中元素反复结合起来，得到一个值。返回 Optional<T> |
| **R 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的操作(如收集到 List、Set、Map)。另外， 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。

1、代码示例：是否所有员工的薪资都低于10000

|  |
| --- |
| @Test  **public** **void** test1(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    **boolean** result = list.stream().allMatch(e -> e.getSalary()<10000);  System.***out***.println("所有员工的薪资都低于10000：" + result);  } |

2、示例代码：是否有员工薪资低于10000

|  |
| --- |
| @Test  **public** **void** test2(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    **boolean** result = list.stream().anyMatch(e -> e.getSalary()<10000);  System.***out***.println("是否有员工的薪资都低于10000：" + result);  } |

3、代码示例：薪资最高的员工

|  |
| --- |
| @Test  **public** **void** test3(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    Optional<Employee> max = list.stream()  .max((e1,e2)->e1.getSalary()==e2.getSalary()?0:(**int**)Math.*ceil*(e1.getSalary()-e2.getSalary()));  System.***out***.println("薪资最高的员工是：" + max.get());  } |

4、代码示例：求累加和

|  |
| --- |
| @Test  **public** **void** test4(){  Optional<Integer> sum = Stream.*of*(1,2,3,4,5,6,7,8,9,10)  .reduce((t1,t2) -> t1+t2);  System.***out***.println("和：" + sum.get());  } |

5、代码示例：找出薪资高于10000的员工放到新的List集合中

|  |
| --- |
| @Test  **public** **void** test5(){  ArrayList<Employee> list = **new** ArrayList<>();  list.add(**new** Employee("张三",8888));  list.add(**new** Employee("李四",9999));  list.add(**new** Employee("王五",10000));  list.add(**new** Employee("赵六",11111));  list.add(**new** Employee("钱七",7777));    List<Employee> collect = list.stream().  filter(e->e.getSalary()>10000)  .collect(Collectors.*toList*());    collect.forEach(System.***out***::println);  } |

6、示例代码：按照男女分组

|  |
| --- |
| @Test  **public** **void** test6(){  ArrayList<Student> list = **new** ArrayList<>();  list.add(**new** Student("张三",'男'));  list.add(**new** Student("李四",'女'));  list.add(**new** Student("王五",'男'));  list.add(**new** Student("赵六",'女'));  list.add(**new** Student("钱七",'男'));    Map<Character, List<Student>> collect = list.stream()  .collect(Collectors.*groupingBy*(s -> s.getGender()));    collect.forEach((k,v) -> System.***out***.println(k+"->" + v));  } |