# java8新特性

## Lambda表达式

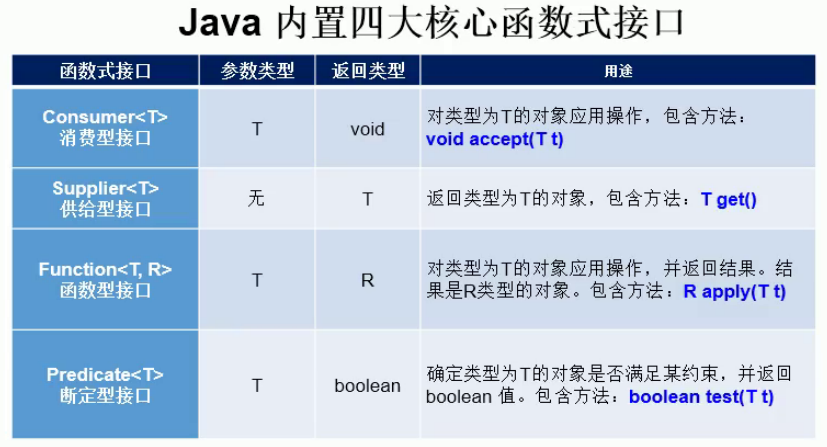
### 示例

|  |
| --- |
| **import** java.util.Comparator;  **public class** LambdaDemoA {  **public static void** main(String[] args) {  *// 1. Runnable* Runnable runnable = () -> System.***out***.println(**"我爱北京~"**);  runnable.run();   *// 2. Comparator-Lambda* Comparator<Integer> comparator = (a,b)->Integer.*compare*(a,b);  **int** val = comparator.compare(12,43);  System.***out***.println(val);   *// 3. Comparator-方法引用* Comparator<Integer> comparatorFun = Integer::*compare*;  **int** valFun = comparatorFun.compare(12,43);  System.***out***.println(valFun);  } } |

### Lambda的使用

|  |
| --- |
| **import** java.util.function.Consumer;  */\*\*  \* Lambda表达式的使用  \* 1、 示例：(a,b)->Integer.compare(a,b);  \* 2、 格式  \* -> : lambda操作符  \* ->左边 : lambda形参列表（接口中的抽象方法的形参列表）  \* ->右边 ： lambda体（重写的抽象方法的方法体）  \* 3、 Lambda表达式的使用（6种情况）  \* 1、 无参无返回值  \* Runnable runnable = () -> System.out.println("我爱北京~");  \* 2、 需要一个参数无返回值  \* Consumer<String> consumer = (String s) -> {System.out.println(s);}  \* 3、 数据类型可以省略，因为可由编译器推断得出，称为“类型判断”  \* Consumer<String> consumer = (s) -> System.out.println(s);  \* 4、 Lambda若只需要一个参数时，参数的小括号可以省略  \* Consumer<String> consumer = s -> System.out.println(s);  \* 5、 Lambda需要两个或两个以上参数，多条执行语句并且可以有返回值  \* Comparator<Integer> comparator = (a,b)->Integer.compare(a,b);  \* 6、 Lambda体只有一条语句时，return与大括号若有，都可以省略  \* Consumer<String> consumer = (String s) -> System.out.println(s);  \* 4、 Lambda表达式的本质：作为接口的实例  \*/* **public class** LambdaDemoB {  **public static void** main(String[] args) {  Consumer<String> consumer = (String s) -> System.***out***.println(s);  consumer.accept(**"谎言和誓言~"**);  } } |

## 函数式（Functional）接口：java.util.function



|  |
| --- |
| **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.List; **import** java.util.function.Consumer; **import** java.util.function.Function; **import** java.util.function.Predicate; **import** java.util.function.Supplier;  */\*\*  \* Java内置四大核心函数式接口  \* 1、 消费型：Consumer  \* 2、 供给型：Supplier  \* 3、 函数型：Function  \* 4、 断定型：Predicate  \*/* **public class** LambdaDemoC {  **public static void** main(String[] args) {  *testConsumer*(200, d -> {  System.***out***.println(**"消费型---"** + d);  });   Object objSupplier = *testSupplier*(() -> {  **return "你好"**;  });  System.***out***.println(**"供给型---"** + objSupplier);   Integer objFunction = *testFunction*(50, integer -> integer.intValue());  System.***out***.println(**"函数型---"** + objFunction);   List<String> lists = Arrays.*asList*(**"1500"**, **"2500"**, **"3000"**, **"4250"**);  List<String> strings = *testPredicate*(lists, s -> s.contains(**"3"**));  **for** (String s : strings) {  System.***out***.println(**"断定型---"** + s);  }   }   **public static void** testConsumer(**double** d, Consumer consumer) {  consumer.accept(d);  }   **public static** Object testSupplier(Supplier supplier) {  **if** (supplier.get().equals(**"中国"**)) {  **return "中国"**;  } **else** {  **return "中国"**;  }  }   **public static** Integer testFunction(Integer integer, Function<Integer, Integer> function) {  **if** (function.apply(integer) > 100) {  **return** integer \* 2;  } **else** {  **return** integer;  }  }   **public static** List<String> testPredicate(List<String> lists, Predicate<String> predicate) {  ArrayList<String> filterList = **new** ArrayList<>();  **for** (String s : lists) {  **if** (predicate.test(s)) {  filterList.add(s);  }  }  **return** filterList;  } } |

## 方法引用与构造器引用

### 方法引用

|  |
| --- |
| **import** java.io.PrintStream; **import** java.util.Comparator; **import** java.util.function.Consumer;  */\*\*  \* 方法引用  \* 1、 使用情景：  \* 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  \* 2、 方法引用：  \* 本质上就是Lambda表达式，而Lambda表达式作为函数式接口的实例，所以方法引用也是函数式接口的实例。  \* 3、 格式：  \* 类（或对象） :: 方法名  \* 4、 具体分为如下三种情况：  \* 1、 对象 :: 非静态方法  \* 2、 类 :: 静态方法  \*  \* 3、 类 :: 非静态方法  \* 5、 方法引用的使用条件：要求接口中的抽象方法的形参列表和返回值类型与方法引用的方法的形参列表与返回值类型相同（主要针对1和2）  \*/* **public class** LambdaDenoD {  **public static void** main(String[] args) {  *// 对象 :: 非静态方法* PrintStream out = System.***out***;  Consumer<String> consumer = out::println;  consumer.accept(**"你好啊"**);   *// 类 :: 静态方法* Comparator<Integer> comparator = Integer::compareTo;  System.***out***.println(comparator.compare(21, 12));   *// 类 :: 非静态方法* Comparator<String> stringComparator = String::compareTo;  System.***out***.println(stringComparator.compare(**"abc"**, **"abm"**));  } } |

### 构造器引用

|  |
| --- |
| **import** java.util.function.Function; **import** java.util.function.Supplier;  */\*\*  \* 构造器引用  \* 1、 和方法引用类似  \* 2、 函数式接口的抽象方法的形参列表和构造器的形参列表一致，抽象方法的返回值类型即为构造器所属的类的类型  \* 数组引用  \* 将数组看做一个特殊的类，用法与构造器引用一致  \*/* **public class** LamdbaDemoE {  **public static void** main(String[] args) {  *// 构造器引用* Supplier<String> supplier = String::**new**;  System.***out***.println(supplier.get());   *// 数组引用* Function<Integer, String[]> function = String[]::**new**;  String[] apply = function.apply(12);  System.***out***.println(apply.**length**);  } } |

## Stream API：java.util.stream

### Stream与Collection

Stream与Collection集合的区别：Collection是一种静态的内存数据结构，而Stream是有关计算的。前者主要面向内存，存储在内存中，后者主要面向CPU，通过CPU实现运算。

### **注意事项**

* Stream不会存储元素
* Stream不会改变源对象，会返回一个持有结果的新Stream
* String操作是延迟执行的，意味着会等到需要结果的时候再执行

### **操作三步曲**

* 创建Stream：一个数据源，获取一个流
* 中间操作：一个中间操作链，对数据源的数据进行处理
* 终止操作（终端操作）：一旦执行终止操作，就执行中间操作链，并产生结果，之后不会再被使用。

### 使用

#### 创建Stream

|  |
| --- |
| **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.List; **import** java.util.stream.Stream;  */\*\*  \* Stream API  \* Stream关注的是对数据的运算，与CPU打交道，集合关注的是数据的存储，与内存打交道  \*/* **public class** StreamAPIDemoA {  **public static void** main(String[] args) {  *// 创建Stream方式一：通过集合* List<String> lists = **new** ArrayList<>();  *// 返回顺序流* Stream<String> stream = lists.stream();  *// 返回并行流* Stream<String> stringStream = lists.parallelStream();   *// 创建Stream方式二：通过数组* String[] strings = **new** String[10];  Stream<String> stream1 = Arrays.*stream*(strings);   *// 创建Stream方式三：通过Stream的of()* Stream<Integer> integerStream = Stream.*of*(1, 2, 3, 4);   *// 创建Stream方式四：通过无限流（不常用）  // 迭代* Stream.*iterate*(0, t -> t + 2).limit(10).forEach(System.***out***::println);  *// 生成* Stream.*generate*(Math::*random*).limit(10).forEach(System.***out***::println);;  } } |

#### 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”。

##### 筛选与切片

|  |
| --- |
| **import** java.util.ArrayList; **import** java.util.List; **import** java.util.stream.Stream;  */\*\*  \* 筛选与切片  \*/* **public class** StreamAPIDemoB {  **public static void** main(String[] args) {  List<String> lists = **new** ArrayList<>();  **for** (**int** i = 0; i < 10; i++) {  lists.add(Double.*toString*(Math.*random*()));  }  Stream<String> stream = lists.stream();  *// filter(Predicate p)：接收Lambda，从流中排除某些元素。* stream.filter(s -> Double.*valueOf*(s) > 0.5).forEach(System.***out***::println);  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  *// limit(n)：截断流，使其元素不超过给定数量。* lists.stream().limit(2).forEach(System.***out***::println);  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  *// skip(n)：跳过元素， 返回一个扔掉了前n个元素的流。* lists.stream().skip(3).forEach(System.***out***::println);  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  *// distinct()：筛选，通过流所生成元素的hashCode()和equals()去除重复元素* lists.stream().distinct().forEach(System.***out***::println);  } } |

##### 映射

|  |
| --- |
| **import** java.util.Arrays; **import** java.util.List;  */\*\*  \* 映射  \*/* **public class** StreamAPIDemoC {  **public static void** main(String[] args) {  *// map(Function f)：接收一个函数作为参数，将元素转成其他形式或提取信息，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* List<String> lists = Arrays.*asList*(**"aa"**, **"bb"**, **"cc"**);  lists.stream().map(str -> str.toUpperCase()).forEach(System.***out***::println);  *// 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，然后再把流连接成一个流  // 说明：map(Function f)与flatMap(Function f)的区别类似于add()：将集合元素作为一个整体加入 和 addAll()：将集合中元素逐一加入* } } |

##### 排序

|  |
| --- |
| **import** java.util.Arrays; **import** java.util.List;  */\*\*  \* 排序  \*/* **public class** StreamAPIDemoD {  **public static void** main(String[] args) {  *// sorted()* List<String> lists = Arrays.*asList*(**"bb"**, **"aa"**, **"cc"**);  lists.stream().sorted().forEach(System.***out***::println);  *// sorted(Comparator comparator)* } } |

#### 3、 终止操作

##### 1、 匹配与查找

|  |
| --- |
| **import** java.util.Arrays; **import** java.util.List;  */\*\*  \* 匹配与查找  \*/* **public class** StreamAPIDemoE {  **public static void** main(String[] args) {  *// allMathch(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)：检查是否没有匹配元素  // findFirst 返回第一个元素  // findAny 返回当前流中的任意元素  // count 返回流中元素的总个数* List<String> lists = Arrays.*asList*(**"bb"**, **"aa"**, **"cc"**);  **long** count = lists.stream().count();  System.***out***.println(count);  *// 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* } } |

##### 2、 归约

|  |
| --- |
| **import** java.util.Arrays; **import** java.util.List;  */\*\*  \* 归约  \* 说明：map和reduce的连接通常称为map-reduce模式  \*/* **public class** StreamAPIDemoF {  **public static void** main(String[] args) {  *// 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* List<Integer> lists = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7);  Integer reduce = lists.stream().reduce(0, Integer::*sum*);  System.***out***.println(reduce);  *// 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>* } } |

##### 3、 收集

|  |
| --- |
| **import** java.util.Arrays; **import** java.util.Iterator; **import** java.util.List; **import** java.util.stream.Collectors;  */\*\*  \* 收集  \*/* **public class** StreamAPIDemoG {  **public static void** main(String[] args) {  *// 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* List<Integer> lists = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7);  List<Integer> collect = lists.stream().filter(s -> s > 5).collect(Collectors.*toList*());  Iterator<Integer> iterator = collect.iterator();  **while** (iterator.hasNext()) {  System.***out***.println(iterator.next());  }  } } |

## Optional类：避免空指针异常



|  |
| --- |
| **import** java.util.Optional;  */\*\*  \* Optional  \*/* **public class** OptionalDemoA {  **public static void** main(String[] args) {  String s1 = **new** String();  s1 = **null**;  Optional<String> s = Optional.*ofNullable*(s1);  s.orElse(**new** String(**"hello"**));  System.***out***.println(s.toString());  } } |