# **Jdk8**

# Lambda表达式

在java中，无法将函数作为参数传递给一个方法，也无法声明返回一个函数的方法

在javascript中，函数参数是一个函数，返回值是另一个函数的情况是非常常见的；javascript是一种非常典型的函数式语言

例子1

|  |
| --- |
| 片段1  package com.gardenia.jdk8;  import javax.swing.\*; import java.awt.event.ActionEvent; import java.awt.event.ActionListener;  public class SwingTest {   public static void main(String[] args) {   JFrame jframe = new JFrame("My JFrame");  JButton jButton = new JButton("My JButton");  // jButton.addActionListener(new ActionListener() { // public void actionPerformed(ActionEvent e) { // System.out.println("Button Pressed!"); // } // });   jButton.addActionListener(event -> {  System.*out*.println("Button Pressed!");  System.*out*.println("Button Pressed!");  System.*out*.println("Button Pressed!");  });   jframe.add(jButton);  jframe.pack();  jframe.setVisible(true);  jframe.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);    }  } |

lambda表达式基本结构：

(param1,param2,param3){

}

函数式接口：

1. 若果一个接口只有一个抽象方法，那么该接口就是一个函数式接口
2. 若果我们在某个接口上声明了FunctionalInterface注解，那么编译器就会按照函数式接口的定义来要求该接口
3. 如果某个接口只有一个抽象方法，的那我们并没有给该接口声明FunctionalInterface注解，那么编译器依旧会将该接口看做是函数式接口。

例子2

|  |
| --- |
| 片段1  package com.gardenia.jdk8;  @FunctionalInterface public interface MyInterface {   void test();   // 加上这个会报错  //String myString();   // 而这个不会报错  // 重写了Object类的方法，不会向抽象接口的方法数量加1  String toString(); }  片段2  package com.gardenia.jdk8;  @FunctionalInterface interface MyInterface2 {   void test();   String toString(); }  public class Test2 {   public void myTest(MyInterface2 myInterface2){  System.*out*.println(1);  myInterface2.test();  System.*out*.println(2);  }    public static void main(String[] args) {  Test2 test2 = new Test2();  // 传统方式  test2.myTest(new MyInterface2() {  @Override  public void test() {  System.*out*.println("myTest");  }  });  // lambda表达式  test2.myTest(()->{  System.*out*.println("mytest");  });   //  MyInterface2 myInterface2 = ()->{  System.*out*.println("hello");  };    } } |

Lambda表达式为Java添加了确实的函数式编程特性，使我们能将函数当做一等公民看待

在将函数作为一等公民的语言中，Lambda表达式的类型是函数。但在Java中，Lambda表达式是对象，他们必须依附于一类特别的对象类型--函数式接口

内部迭代和外部迭代

内部迭代：lambda表达式迭代

Java Lambda表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符，返回值声明和名字

lambda表达式作用

传递行为，而不仅仅是值

提升抽象层次

API重用性更好

更加灵活

Java中的Lambda表达式基本语法

(argument)->(body)

比如说

(arg1,arg2)->{body}

(type1 arg1,type2 age2...)->{body}

(int a,int b)->{return a + b;}

()->System.out.println(“hello world”);

(String s)->{System.out.println(s);}

()->42

()->{return 3.1415}

一个Lambda表达式有0个或多个参数

参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与（a）效果相同

所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a,b)或(int a,int b)或(String a,int b,fload c)

空圆括号嗲表参数集为空，例如：（）->42

当只有一个参数，且类型可以推到出来时，圆括号可以省略

Lambda表达式的主体可包含零条或多条语句

如果Lambda表达式的主体只有一条语句，或括号{}可省略。匿名函数的返回类型与该主体表达式一致

如果Lambda表达式的主体包含一条以上语句，则表达式必须包含在或括号{}中，匿名函数的返回类型与代码块的返回类型一致，若没有返回则为空

|  |
| --- |
| 片段1  package com.gardenia.jdk8;  import java.util.Arrays; import java.util.Collections; import java.util.List;  public class StringComparator {   public static void main(String[] args) {  List<String> names = Arrays.*asList*("zhangsan", "lisi", "wangwu", "zhaoliu");  // Collections.sort(names, new Comparator<String>() { // @Override // public int compare(String o1, String o2) { // return o2.compareTo(o1); // } // }); // System.out.println(names);   // o2.compareTo(o1) experssion 表达式方式  Collections.*sort*(names, (o1, o2) -> o2.compareTo(o1));  // { return o2.compareTo(o1);} statement 语句块  Collections.*sort*(names, (o1, o2) -> {  return o2.compareTo(o1);  });   System.*out*.println(names);   }  } |

# 函数式接口

## 2.1 Function

Compose

Identify

|  |
| --- |
| 片段1  package com.gardenia.jdk8;  import java.util.function.Function;  public class FunctionTest2 {   public static void main(String[] args) {  FunctionTest2 test = new FunctionTest2();  // 12  System.*out*.println(test.compute(2,value->value\*3,value->value\*value));  // 36  System.*out*.println(test.compute2(2,value->value\*3,value->value\*value));   }   public int compute(int a, Function<Integer,Integer> function1,Function<Integer,Integer> function2){  return function1.compose(function2).apply(a);  }    public int compute2(int a, Function<Integer,Integer> function1,Function<Integer,Integer> function2){  return function1.andThen(function2).apply(a);  } }  片段2  package com.gardenia.jdk8;  import java.util.Arrays; import java.util.List; import java.util.function.BiFunction; import java.util.stream.Collectors;  public class PersonTest {  public static void main(String[] args) {   Person person1 = new Person("zhangsan", 20);  Person person2 = new Person("lisi", 30);  Person person3 = new Person("wangwu", 30);   List<Person> personList = Arrays.*asList*(person1, person2, person3);   PersonTest test = new PersonTest();  List<Person> personResult = test.getPersonsByUsername("zhangsan", personList);  personResult.forEach(person -> System.*out*.println(person.getUsername()));  System.*out*.println("-------------------");  List<Person> personsByAge2 = test.getPersonsByAge2(20, personList, (age, persons) -> {  return personList.stream().filter(person -> person.getAge() > age).collect(Collectors.*toList*());  });  personsByAge2.forEach(person -> System.*out*.println(person.getUsername()));    }   public List<Person> getPersonsByUsername(String username, List<Person> persons) {  return persons.stream().filter(person -> person.getUsername().equals(username)).collect(Collectors.*toList*());  }   public List<Person> getPersonsByAge(int age, List<Person> persons) {  BiFunction<Integer, List<Person>, List<Person>> biFunction = (ageOfPerson, personList) -> {  return personList.stream().filter(person -> person.getAge() > ageOfPerson).collect(Collectors.*toList*());  };  return biFunction.apply(age, persons);  }    public List<Person> getPersonsByAge2(int age, List<Person> persons, BiFunction<Integer, List<Person>, List<Person>> biFunction) {  return biFunction.apply(age, persons);  }  } |

## 2.2 Predicate

## 2.3 Suppier

# 方法引用与构造器

|  |
| --- |
| 片段1  package com.gardenia.jdk8.methodreference;  import java.util.Arrays; import java.util.Collections; import java.util.List; import java.util.function.Function; import java.util.function.Supplier;  */\*\*  \* 要保证引用的方法的入参和lambda表达式的入参是一致的  \* 方法引用的4中类型：  \* 1.类名::静态方法名  \* 2.引用名（对象名）::实例方法名  \* 3.类名::实例方法  \* 4.构造方法引用:类名::new  \*  \*/* public class MethodReferenceTest {    public String getString(Supplier<String> supplier) {  return supplier.get() + "test";  }   public String getString2(String str, Function<String, String> function) {  return function.apply(str);  }    public static void main(String[] args) {   Student student1 = new Student("zs", 30);  Student student2 = new Student("ls", 20);  Student student3 = new Student("w5", 50);  Student student4 = new Student("zl", 40);    List<Student> students = Arrays.*asList*(student1, student2, student3, student4);  // lambda 表达式方式 // students.sort((studentParam1, studentParam2) -> Student.compareStudentByAge(studentParam1, studentParam2)); // students.forEach(student -> System.out.println(student.getName() + " " + student.getAge())); // System.out.println("-----------------------");  // 方法引用方式,方法引用和lambda表达式是一样的效果，且没有做功能上的增强，只是语法糖 // students.sort(Student::compareStudentByAge); // students.forEach(student -> System.out.println(student.getName() + " " + student.getAge()));   System.*out*.println("-----------------------");  // 不是静态方法，lambda  StudentCompartor studentCompartor = new StudentCompartor(); // students.sort((studentParam1, studentParam2) -> studentCompartor.compareStudentByName(studentParam1, studentParam2)); // students.forEach(student -> System.out.println(student.getName() + " " + student.getAge()));  // students.sort(studentCompartor::compareStudentByName); // students.forEach(student -> System.out.println(student.getName() + " " + student.getAge()));    // sort 接收的第一个参数调用的compareByAge方法 // students.sort(Student::compareByAge); // students.forEach(student -> System.out.println(student.getName() + " " + student.getAge()));   // List<String> cities = Arrays.asList("qingdao","chongqing","tianjin","beijing"); // Collections.sort(cities,(city1,city2)->city1.compareToIgnoreCase(city2)); // cities.forEach(System.out::println); // Collections.sort(cities,String::compareToIgnoreCase);    MethodReferenceTest methodReferenceTest = new MethodReferenceTest();  System.*out*.println(methodReferenceTest.getString(String::new));   System.*out*.println(methodReferenceTest.getString2("hello", String::new));   }  }  Student  package com.gardenia.jdk8.methodreference;  public class Student {   private String name;   private int age;   public Student() {  }   public Student(String name, int age) {  this.name = name;  this.age = age;  }   public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public int getAge() {  return age;  }   public void setAge(int age) {  this.age = age;  }   public static int compareStudentByName(Student student1, Student student2) {  return student1.name.compareTo(student2.name);  }   public static int compareStudentByAge(Student student1, Student student2) {  return student1.age - student2.age;  }   public int compareByAge(Student student) {  return this.getAge() - student.getAge();  }   public int compareByName(Student student) {  return this.getName().compareToIgnoreCase(student.getName());  }  }  StudentCompartor  package com.gardenia.jdk8.methodreference;  public class StudentCompartor {    public int compareStudentByName(Student student1, Student student2) {  return student1.getName().compareTo(student2.getName());  }   public int compareStudentByAge(Student student1, Student student2) {  return student1.getAge() - student2.getAge();  }  } |

# Stream API

## 4.1 概述

Collection提供了新的stream()方法

流不存储值，通过管道的方式获取值

本质是函数式的，对流的操作会生成一个结果，不过并不会修改底层的数据源，集合可以作为流的底层数据源

延迟查找，很多流操作(过滤、映射、排序等)都可以延迟实现

|  |
| --- |
| 片段1  package com.gardenia.jdk8.stream;   import java.util.ArrayList; import java.util.Arrays; import java.util.LinkedList; import java.util.List; import java.util.stream.Collectors; import java.util.stream.Stream;  public class StreamTest4 {   public static void main(String[] args) {   // Stream<String> stream = Stream.of("hello", "world", "helloworld"); // // String[] stringArray = stream.toArray(length -> new String[length]); // // Arrays.asList(stringArray).forEach(System.out::println);    Stream<String> stream = Stream.*of*("hello", "world", "helloworld"); // List<String> list = stream.collect(Collectors.toList()); // list.forEach(System.out::println);   // 不懂  //stream.collect(() -> new ArrayList<>(), (theList, item) -> theList.add(item), (theList1, theList2) -> theList1.addAll(theList2));  // 不懂  // stream.collect(LinkedList::new, LinkedList::add, LinkedList::addAll);   }   } |

集合关注的是数据与数据存储本身

流关注的则是对数据的计算。

流与迭代器类似的：

中间操作都会返回一个Stream对象。比如说返回Stream<Student>,Stream<Integer>

终止操作则不会返回Stream类型，可能不反回值，也肯能返回其他类型的单个值

## 4.2 Collectors

1. collect:收集器
2. Collector作为collect方法的参数
3. Collector是一个接口，它是一个可变的汇聚操作，将输入元素累积到一个可变的结果容器中；他会在所有元素都处理完毕后，将累积的结果装换为一个最终的表示；它支持串行与并行两种方式
4. Collectors本身提供了关于Collector的常见汇聚实现，本身是个工厂

# 5.接口中默认方法与静态方法

# 6.新时间日期API

# 7.其他新特性