# day19---JDK8新特性

## JDK8接口新特性

1. 从JDK8版本开始, 接口中不仅可以定义出抽象方法, 也可以定义出非抽象方法; JDK8版本中可以在接口中添加默认方法(default), 也可以添加静态方法(static), 默认和静态都是具有方法体; 只有抽象方法没有方法体

### 接口中默认方法

JDK中或者很多项目中, 存在很多老接口(存在已经很久, 使用很多的接口), 举例 : Collection 单列集合顶层父接口, 如果需要优化和改造呢? 如果接口中只能定义出抽象方法, 如果父接口中添加抽象方法, 那么所有实现类都会受到影响; 项目也会有影响,需要改动的场景就会很多; 因此在JDK8版本下, 可以在接口中设计default默认修饰的方法, 因为父接口中的默认方法可以直接被实现类使用, 不强制重写

default 返回值类型 方法名(参数列表){

// 方法体;

}

1. default方法的使用:

a : 父接口中的默认方法, 可以直接被实现类使用, 不需要强制实现类重写

b : 实现类中可以重写父接口中的默认方法

1) 类中不能写default 默认修饰, 不写才表示默认修饰

2) 实现类重写默认方法时, 修饰符需要是public修饰

c : 类与接口之间可以多实现(一个类可以同时实现多个接口), 如果多个父接口中定义出相同default默认方法,实现类使用就会冲突; 解决方案 : 要求实现类一定要重写冲突的默认方法

如果实现类重写冲突方法功能时, 发现现在想调用某一个父接口中默认方法实现过程 : 语法 父接口名.super.方法名(实际参数);

d : 一个类继承一个父类的同时, 还可以实现很多接口; “父优先” , 不管继承关系,还是 方法的重写,以及调用, 一律父优先; 如果父类中没有的, 再到父接口中寻找使用

代码

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public class InterfaceJDK8Impl extends Fu implements MyInter,MyMyInter2{  @Override  public void eat() {  System.out.println("MyInter接口的实现类重写抽象方法eat");  }    // 实现类中可以重写父接口中的默认方法  // 1) 类中不能写default 默认修饰的, 不写才表示默认修饰  // 2) 实现类重写默认方法时, 修饰符需要是public修饰  @Override  public void sleep() {  System.out.println("重写的父接口中的默认方法sleep");  }  @Override  public void fun() {  // 如果实现类重写冲突方法功能时, 发现现在想调用某一个父接口中默认方法实现过程 : 语法 父接口名.super.方法名(实际参数);  MyInter.super.fun();  MyMyInter2.super.fun();  System.out.println("父接口中fun冲突, 实现类只能重写, 否则代码会报错");  }  }  interface MyInter{  public abstract void eat();    default void fun() {  System.out.println("父接口MyInter中默认方法fun");  }    // 在接口中定义出一个使用default修饰的默认方法  default void sleep() {  System.out.println("父接口MyInter中默认方法sleep");  }  }  interface MyMyInter2{  // MyMyInter2接口的默认方法fun与接口MyInter中默认方法一致  default void fun() {  System.out.println("父接口MyInter2中冲突的默认方法fun");  }  } |

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public class Fu {  public void fun() {  System.out.println("父类的fun");  }  } |

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public class TestInterface {  public static void main(String[] args) {  // 1. 测试实现类可以直接调用父接口中的默认功能  InterfaceJDK8Impl impl = new InterfaceJDK8Impl();  impl.eat();  impl.sleep();  impl.fun();  }  } |

### 接口中的静态方法

1. 问题1 : 父类中静态方法可以被子类继承使用; 子类不能重写从父类继承来静态方法, 原因 : 静态属于类; 子类中定义出的静态方法就属于子类,不是重写的父类
2. JDK8版本中, 可以在接口中定义使用static修饰的静态方法

static 返回值类型 方法名字(参数列表){

}

1. 父接口中静态方法功能不能被实现类使用, 只能通过接口名.直接调用

实现类不能继承父接口中的静态方法原因 :

类与接口之间可以多实现; 如果有个实现类同时实现了多个接口, 而多个接口中有相同静态方法, 发生继承冲突性; 解决问题就是重写静态方法, 但是静态方法不能重写; 矛盾,因此父接口中静态方法不让实现类使用

代码

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public interface MyInterStaticMethod {  static void function() {  System.out.println("MyInterStaticMethod接口中的静态方法");  }  } |

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public class InterStaticImpl implements MyInterStaticMethod {  // InterStaticImpl实现类没有继承到父接口中的静态方法  } |

|  |
| --- |
| package com.ujiuye.interfaceJDK8;  public class TestInterface {  public static void main(String[] args) {  // 2. 测试实现类与父接口中静态方法之间的使用  //InterStaticImpl实现类没有继承到父接口中的静态方法  // InterStaticImpl.function();  MyInterStaticMethod.function();  }  } |

## Lambda表达式

1. Lambda表达式本质 :

就是一个接口的实现类对象

1. Lambda表达式使用前提 :

只能够实现函数式接口(接口中只有一个抽象方法,这个接口称为函数式接口)

使用注释 : @FunctionalInterface 验证接口是否为一个函数式接口

1. Lambda表达式优势:

比匿名内部类对象实现更加简洁; 效率更高

1. Lambda表达式使用 :

(参数列表)->{方法体/Lambda体};

1. 解释Lambda表达式:
2. -> : 称为箭头运算符, 也称为Lambda运算符, 作用就是将需要实现唯一的抽象方法分成两部分
3. 参数列表 : 唯一抽象方法的参数列表
4. 方法体 : 唯一抽象方法重写实现过程
5. Lambda表达式使用注意事项:

a : 参数列表

1. 参数列表只需要写变量名, 不需要写变量数据类型
2. 如果参数列表只有一个参数, 那么小括号可以省略

b : 方法体: 与普通方法一样实现, 如果有返回值类型需要return

1. 如果方法体只有一句代码, 大括号可以省略
2. 如果方法有返回值类型, 且只有一句语句代码, 这一句就作为返回值类型, 那么return关键字和大括号可以同时省略

代码

|  |
| --- |
| package com.ujiuye.lambda;  public class LambdaDemo {  public static void main(String[] args) {  // 1. 使用匿名内部类对象实现InterfaceDemo接口  new InterfaceDemo() {  @Override  public void fun() {  System.out.println("匿名内部类对象实现接口");  }  }.fun();    // 2. Lambda表达式实现InterfaceDemo函数式接口, 语法结构  // (参数列表)->{方法体/Lambda体};  // 接口多态性  InterfaceDemo id = ()->{  System.out.println("Lambda表达式实现接口");  };  id.fun();    // 3. Lambda表达式实现方法只有1个参数情况  InterfaceDemo2 id2 = (x)->{  for(int i = 0; i <= x; i++) {  System.out.println(i);  }  };  id2.print(5);    System.out.println("------------");    //4. Lambda表达式实现方法只有1个参数, 小括号可以省略  InterfaceDemo2 id3 = x->{  for(int i = 0; i <= x; i++) {  System.out.println(i);  }  };  id3.print(3);    // Lambda表达式方法体只有一句, 大括号可以省略  InterfaceDemo3 id4 = (a,b)->System.out.println(a==b);  id4.equal(3.14, 3.14);// true    // 5. Lambda表达式实现的方法有返回值类型, 与普通方法一样, 需要return  InterfaceDemo4 id5 = (x,y)->{  return x + y;  };    System.out.println(id5.getSum(5, 6));// 11.0    // 6. 如果Lambda体只有一句代码,且这一句正好是返回值结果, return和大括号可以同时省略  InterfaceDemo4 id6 = (x,y)->x + y;  System.out.println(id6.getSum(6.6, 7.2));// 13.8  }  }  @FunctionalInterface  interface InterfaceDemo{  public abstract void fun();  }  @FunctionalInterface  interface InterfaceDemo2{  public abstract void print(int n);  }  @FunctionalInterface  interface InterfaceDemo3{  public abstract void equal(double x, double y);  }  @FunctionalInterface  interface InterfaceDemo4{  public abstract double getSum(double x, double y);  } |

## 函数式接口

### 函数式接口概述

1. 说明函数式接口:
2. 一个接口中只定义了一个抽象方法, 这个接口就称为函数式接口
3. 函数式接口, 使用注解 : @FunctionalInterface 进行验证
4. Lambda表达式只能使用于函数式接口, 作为函数式接口的实现类对象
5. 为什么要有函数式接口?
6. 函数 : 在java中就用于表示方法, 因为方法本身并不属于某一种数据类型, 是一种功能实现, 作为函数理解
7. 当在Java代码中定义出一个方法功能时, 需要给方法设计参数列表, 数据类型 变量名, 数据类型 变量名..., 有时候需要的参数不仅仅是数据, 也需要对于数据的处理方式,处理理念(对于数据的处理方式应该是一个方法功能, 就是一个函数); 因此可能会需要将方法作为参数传递, 而又因为方法本身不是数据类型不能传递, 因此将方法封装在一个接口中, 这个接口就是为了封装一个方法而存在的; 以后通过将接口作为参数传递, 实际上就是在传递接口中所封装的方法功能.

举例 : 定义出一个方法功能, 实现对于两个整数的任意操作; 方法实际对于两个整数处理逻辑有很多种情况, 实际处理方式根据客户的需求决定

客户1 : 两个数和

客户2 : 两个数差

客户3 : 两数乘积

... 可能有无数客户对于此方法有不同的需求实现

public int dealTwoNumber(int a, int b,需要提供对于a和b两数的处理方式){

}

### JDK提供的常用4种函数式接口

JDK在1.8版本中,提供了4种内置函数式接口, 为了在不同的场景下可以直接使用而不需要开发人员自己定义

1. 消费型接口:

Consumer<T>

void accept(T t) : 对于T类型数据的处理方式

1. 供给型接口:

Supplier<T>

T get(): 获取到T类型数据

1. 函数型接口:

Function<T,R>

R apply(T) : 通过T类型参数, 获取到R类型数据结果

1. 断言型接口 :

Predicate<T>

boolean test(T) : 提供一个T类型参数,判断出T类型参数是否符合规则和要求

### 消费型接口

消费型接口:

Consumer<T> : 来自于java.util.function包

void accept(T t) : 如果方法中, 已经有一个参数是T类型, 但是方法中需要处理使用这个T类型参数, 可是使用T类型参数的方式可能有很多种, 具体是哪一种, 不能确定, 那么就可以将Consumer接口作为方法参数传递, 相当于传递accept方法, accept功能就是处理T类型参数

需求 : 定义出一个方法功能, 客户消费指定金额, 这些金额都如何消费的

客户1 : 花了500元, 买了一把大宝剑

客户2 : 花了400元, 买了一双球鞋

客户3 : 花了888元, 买了一套护肤品

... 还有很多很多的客户, 对于指定金额有不同的消费

代码

|  |
| --- |
| package com.ujiuye.functioninterface;  import java.util.function.Consumer;  public class Demo01\_Consumer {  public static void main(String[] args) {  //客户1 : 花了500元, 买了一把大宝剑  Consumer<Double> con = x->System.out.println("花了"+ x + "元,买了一把大宝剑");  testConsumer(500,con);    //客户2 : 花了400元, 买了一双球鞋  Consumer<Double> con1 = x->{  if(x <= 500) {  System.out.println("非常节省,花了" + x + "元买了一双限量版球鞋");  }else {  System.out.println("先不买,攒钱");  }  };  testConsumer(400,con1);  }    /\*需求 : 定义出一个方法功能, 客户消费指定金额, 这些金额都如何消费的  客户1 : 花了500元, 买了一把大宝剑  客户2 : 花了400元, 买了一双球鞋  客户3 : 花了888元, 买了一套护肤品  ... 还有很多很多的客户, 对于指定金额有不同的消费    \*  \* 分析 :  \* 1) 需要知道客户消费金额, double money  \* 2) 需要知道客户对于money怎么消费, 于是可以将Cunsumer接口作为方法参数传递,相当于传递accept(Double money)  \*/    public static void testConsumer(double money, Consumer<Double> con) {  con.accept(money);  }  } |

### 供给型接口

供给型接口:

Supplier<T> : 来自于java.util.function包

T get(): 获取到T类型数据, 而不知道具体该如何获取, 那么可以将Supplier作为方法参数传递, 相当于传递唯一方法功能get, get()不需要参数列表, 只获取到T类型结果

需求 : 定义出一个方法功能, 方法能给客户返回一个ArrayList<Integer>容器, 返回的容器中包含几个数据, 集合中存储的数据有什么规律, 根据客户的实际要求决定

客户1 : 存储5个数据, 数据是30-80之间的随机数

客户2 : 存储8个数据, 数据是1-100之间的随机偶数

...

每一个客户都需要得到符合条件的集合容器

代码

|  |
| --- |
| package com.ujiuye.functioninterface;  import java.util.ArrayList;  import java.util.Random;  import java.util.function.Supplier;  public class Demo02\_Supplier {  public static void main(String[] args) {  //客户1 : 存储5个数据, 数据是30-80之间的随机数  // (0-50) + 30  Supplier<Integer> sup = ()->new Random().nextInt(51) + 30;  ArrayList<Integer> list1 = testSupplier(5,sup);  System.out.println(list1);    //客户2 : 存储8个数据, 数据是1-100之间的随机偶数  Supplier<Integer> sup1 = ()->{  Random ran = new Random();  int number = ran.nextInt(100) + 1;  while(true) {  if(number % 2 == 0) {  break;  }else {  number = ran.nextInt(100) + 1;  }  }  return number;  };  System.out.println(testSupplier(8,sup1));  }  /\*需求 : 定义出一个方法功能, 方法能给客户返回一个ArrayList<Integer>容器,  \* 返回的容器中包含几个数据, 集合中存储的数据有什么规律, 根据客户的实际要求决定  客户1 : 存储5个数据, 数据是30-80之间的随机数  客户2 : 存储8个数据, 数据是1-100之间的随机偶数  ...  每一个客户都需要得到符合条件的集合容器    \*  \* 分析 :  \* 1) 第一个参数 : 需要集合中存储的数据个数 int count  \* 2) 第二个参数 : 需要给出集合中存储的数据获取规律, 可以将Supplier作为方法的参数传递,  \* 相当于传递T get()  \*/  public static ArrayList<Integer> testSupplier(int count, Supplier<Integer> sup){  ArrayList<Integer> list = new ArrayList<>();  // 向list集合中添加元素数据  for(int i = 1; i <= count ; i ++) {  // 就需要Integer类型数据, 问题就是Integer类型数据的获取方式没法确定  list.add(sup.get());  }  return list;  }  } |

### 函数型接口

函数型接口:

Function<T,R> : 来自于java.util.function包

R apply(T) : 如果方法中已经有数据T类型, 想通过T类型参数, 获取到R类型数据结果, 具体怎么通过T计算出R类型, 方式有很多, 不能具体确定; 那么可以将Function接口作为方法参数传递, 相当于传递apply(T)

需求 : 定义出一个方法功能, 根据int类型x的值, 计算出另外一个int类型y的值, y获取方式根据客户的要求决定

客户1 : y值为x的2倍

客户2 : y值为x + 1

客户3 : y值为x-1

...

代码

|  |
| --- |
| package com.ujiuye.functioninterface;  import java.util.function.Function;  public class Demo03\_Function {  public static void main(String[] args) {  //客户1 : y值为x的2倍  Function<Integer,Integer> fun1 = x->x\*2;  System.out.println(testFunction(6,fun1));// 12    //客户2 : y值为x + 1  Function<Integer,Integer> fun2 = x->x+1;  System.out.println(testFunction(-9,fun2));// -8    //客户3 : y值为x-1  Function<Integer,Integer> fun3 = x->x-1;  System.out.println(testFunction(88,fun3));  }  /\*需求 : 定义出一个方法功能, 根据int类型x的值, 计算出另外一个int类型y的值,  y获取方式根据客户的要求决定  客户1 : y值为x的2倍  客户2 : y值为x + 1  客户3 : y值为x-1  ...    \*  \* 分析:  \* 1) 第一个参数 : 需要提供int类型x int x  \* 2) 第二个参数 : 需要提供 计算y值方式(规律), 通过x得到y, 可以将Function接口作为方法参数传递,  \* 实际传递 int apply(int x)  \*/    public static int testFunction(int x, Function<Integer,Integer> fun) {  return fun.apply(x);  }  } |

### 断言型接口

断言型接口

Predicate<T> : 来自于java.util.function包

boolean test(T) : 如果方法中有T类型数据,需要判断出T类型参数是否符合规则和要求,返回boolean类型结果; 可以将Predicate断言型接口作为方法参数传递, 相当于传递test方法功能

需求 : 万能的数据筛选功能; 客户提供一个ArrayList<Integer>容器, 根据客户的要求, 将容器中符合条件的数据筛选出来, 放置到一个新的ArrayList<Integer>容器给客户, 筛选规则由客户决定

客户1 : 筛选出集合中所有小于100的偶数

客户2 : 筛选出集合中所有的奇数

客户3 : 筛选出集合中所有大于50的数

...

代码

|  |
| --- |
| package com.ujiuye.functioninterface;  import java.util.ArrayList;  import java.util.function.Predicate;  public class Demo04\_Predicate {  public static void main(String[] args) {  ArrayList<Integer> list = new ArrayList<>();  list.add(12);  list.add(105);  list.add(99);  list.add(86);  list.add(7);  list.add(120);  // 客户1 : 筛选出集合中所有小于100的偶数  Predicate<Integer> pre = x->x < 100 && x % 2 == 0;  System.out.println(testPredicate(list,pre));    // 客户2 : 筛选出集合中所有的奇数  Predicate<Integer> pre1 = x->x % 2 != 0;  System.out.println(testPredicate(list,pre1));    // 客户3 : 筛选出集合中所有大于50的数  Predicate<Integer> pre2 = x->x > 50;  System.out.println(testPredicate(list,pre2));  }    /\*需求 : 万能的数据筛选功能; 客户提供一个ArrayList<Integer>容器,  根据客户的要求, 将容器中符合条件的数据筛选出来,  放置到一个新的ArrayList<Integer>容器给客户, 筛选规则由客户决定  客户1 : 筛选出集合中所有小于100的偶数  客户2 : 筛选出集合中所有的奇数  客户3 : 筛选出集合中所有大于50的数  ...    \* 分析 :  \* 1) 第一个参数 : 需要客户提供原始待筛选容器 ArrayList<Integer>  \* 2) 第二个参数 : 需要验证元素是否符合条件的规律(思想),将Predicate作为方法参数,  \* 为了传递test方法功能, boolean test(Integer)  \*/  public static ArrayList<Integer> testPredicate(ArrayList<Integer> list,Predicate<Integer> pre){  ArrayList<Integer> newList = new ArrayList<>();  // 遍历待筛选的集合, 获取到集合中的每一个元素  for(int index = 0; index < list.size(); index++) {  int ele = list.get(index);  if(pre.test(ele)) {  newList.add(ele);  }  }  return newList;  }  } |

### 方法引用

有一个函数式接口, 可以使用Lambda表达式作为这个函数式接口的实现类对象; 如果Lambda表达式的方法体(Lambda体)实现过程已经被某类型中某个方法实现过了, 就不需要在Lambda表达式中再写一遍相同的代码, 可以使用方法引用代替Lambda表达式

函数式接口名 变量名 = Lambda表达式;

方法引用:

函数式接口名 变量名 = 类对象 :: 方法名; // 对象引用, 使用在非静态方法

函数式接口名 变量名 = 类型名 :: 方法名;// 类型引用, 使用在静态方法

代码

|  |
| --- |
| package com.ujiuye.functioninterface;  public class Demo05\_方法引用 {  public static void main(String[] args) {  // Lambda表达式实现:  InterDemo id = x->System.out.println(x);  id.print(5);// 5    // System系统类中静态成员常量out, 调用返回PrintStream(打印流)对象, 对象名.调用println方法功能做标准数据输出  // 方法引用实现InterDemo接口  InterDemo id1 = System.out :: println;  id1.print(99);// 99    // 方法引用实现InterDemo2接口  InterDemo2 id2 = x->System.out.println(Integer.parseInt(x) + 1);  // 对象引用  InterDemo2 id3 = new NumberPrint() :: print;  id3.print("125");// 126    // 类型引用  InterDemo2 id4 = NumberPrint2 :: print;  id4.print("88");// 93  }  }  class NumberPrint{  public void print(String x) {  System.out.println(Integer.parseInt(x) + 1);  }  }  class NumberPrint2{  public static void print(String x) {  System.out.println(Integer.parseInt(x) + 5);  }  }  @FunctionalInterface  interface InterDemo{  public abstract void print(int x);  }  @FunctionalInterface  interface InterDemo2{  public abstract void print(String x);  } |

## StreamAPI

1. Collection接口中 : 自JDK8版本, 添加了几个默认(default)方法, 其中有一个方法功能stream() : 获取到当前集合的Stream<T>流资源
2. Stream : 接口,来自于 java.util.stream , 表示一个集合的流资源, 作用就是可以对于当前的集合进行操作, 例如 : 过滤, 遍历, 筛选...
3. Stream类型中常用的方法功能 : 会让集合的操作更加便捷
4. filter(Predicate<T> pre) : 功能就是筛选集合中的数据, 根据参数给出的pre表示的规则, 验证都有哪些集合元素符合筛选条件的, 返回值类型Stream<T>, 返回的是接口的实现类对象, 可以继续调用Stream中方法功能
5. forEach(Consumer<T> con) : 功能就是遍历集合中的每一个元素, 循环遍历过程中, 每一次获取到一个集合数据之后, 想要对这个集合数据进行什么操作, 可以通过参数con设计对于集合数据的使用过程, 返回值类型void
6. count() : 获取到当前流资源中正在操作的元素的个数,返回值类型long
7. limit(long size) : 将流资源中的前size个元素获取到, size之后的元素不再操作,返回值类型Stream<T>,可以继续调用Stream中方法功能
8. skip(long n) : 丢弃流资源中前n个元素, 剩下的元素继续进行操作,返回值类型Stream<T>,可以继续调用Stream中方法功能

需求 : 定义出一个集合ArrayList<String>, 中存储的都是一些人物的姓名, 筛选出所有姓张的, 名字中有3个字的名称数据

代码

|  |
| --- |
| package com.ujiuye.stream;  import java.util.ArrayList;  public class StreamDemo {  // 需求 : 定义出一个集合ArrayList<String>,  // 中存储的都是一些人物的姓名, 筛选出所有姓张的, 名字中有3个字的名称数据  public static void main(String[] args) {  ArrayList<String> listName = new ArrayList<>();  listName.add("张三丰");  listName.add("张无忌");  listName.add("赵敏");  listName.add("周芷若");  listName.add("张五");  listName.add("张成功");    // 1. 通过stream方法获取到可以操作集合数据的Stream流资源  // predicate : 唯一抽象方法 boolean test(T)  listName.stream().filter(x->x.startsWith("张") && x.length() == 3)  .forEach(System.out :: println);    // 2. 姓张的,名字长度为3的姓名有几个  long number = listName.stream().filter(x->x.startsWith("张") && x.length() == 3).count();  System.out.println(number);// 3    // 3. 排除前两个姓名之外, 剩下的姓张的,名字长度为3的姓名  listName.stream().skip(2).filter(x->x.startsWith("张") && x.length() == 3)  .forEach(System.out :: println);      System.out.println("-------");    // 4. 前3个元素中,姓张的,名字长度为3的姓名  listName.stream().limit(3).filter(x->x.startsWith("张") && x.length() == 3)  .forEach(System.out :: println);  }  } |