JDK8 新特性

1 Java 的发展

1991 年, SUN 公司中 James Goslin 带领团队启动”Green”项目.开发了一种称为”Oak”语言

1995 年, SUN 公司把 Oak 语言改名为 Java.

2004 年 9 月，发布 J2SE1.5. 为了表示该版本的重要性,把 J2SE1.5 更名为 JavaSE 5.0

2014 年，发布了 Java8 正式版

2018 年 9 月发布了 Java11.

Oracle 支持 Java8 到 2025 年，支持 Java11 到 2026 年.

2 Java8 语言增强

2.1 Lambda 表达式

Lambda 是数学中的一个函数. Java 中使用方法来代替函数,方法总是作为类或对象的一部分存在的.

可以把 Lambda 看作是一个匿名方法，拥有更简洁的语法

2.1.1 Lambda 的语法

语法:

(参数列表) -> {语句;}

Lambda 表达式由参数列表和一个 Lambda 体组成，通过箭头连接

说明:

1) 当只有一个参数时，参数列表的小括弧可以省略

```
x -> { System.out.println(x); }
```

2) 参数列表中参数的数据类型可以省略

```
( x,y ) -> {x.compareTo(y);}
```

3) 如果 Lambda 体只有一条语句,大括弧也可以省略

```
x -> { return x + 2 ; }
```

4) 如果 Lambda 体中只有一条 return 语句, return 关键字可以省略

```
(x,y) -> x+y
```

以下表达式是有效的 Lambda 表达式

```
(String s ) -> s.length()
(Student stu) -> stu.getAge() > 18
(int x,    int y ) -> {
    System.out.print("result");
    System.out.println( x + y );
}
```

() -> {}

() -> “hehe”

(String s) -> { “wkcto” } //不合法

(String s) -> { return “wkcto” ; }

(String s) -> “wkcto”

Lambda 使用案例:

布尔表达式，判断参数接收的 list 集合是否为空

(List<String> list) -> list.isEmpty()

创建对象,并返回

() -> new Student()

消费(使用)一个对象，把参数接收学生对象的姓名打印出来

(Student stu) -> { System.out.println(stu.name) ; }

从一个对象中选择，返回参数对象的成绩

(Student stu) -> stu.getScore()

组合两个值

(int a, int b) -> a*b

比较两个对象

(Student stu1, Student stu2) -> stu1.getScore() - stu2.getScore()

2.1.2 函数式接口

在 JDK8 中，引用了函数式接口. 就是只定义一个抽象方法的接口. 如:

Comparator 接口 , Runnable 接口

```
@FunctionalInterface          //注解,声明接口为函数式接口
public  interface  Adder{
    int add(int x, int y);
}

public  interface  ByteAdder extends Adder{
    byte add( byte b1, byte b2);
}
```

当前 ByteAdder 接口不是函数式接口，从 Adder 接口中继承了一个抽象方法,在本接口中又定义了一个抽象方法

```
public  inteface  Nothing{
}
```

当前 Nothing 接口也不函数式接口,因为没有抽象方法

函数式接口就是为 Lambda 表达式准备的,或者说 Lambda 表达式必须实现一个函数式接口

java.util.function 包中定义了一些基本的函数式接口,如 Predicate, Consumer, Function,Supplier 等

1 Predicate

Predicate<T>接口中定义一个抽象方法 test(T),接收一个 T 类型的对象参数,返回一个布尔值

当需要一个涉及类型 T 的布尔表达式时,可以使用这个接口

```
public class Test01 {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("lisi", "zhangsan", "wangwu", "chenqi",
"feifei");

        List<String>  result = filter(list, x -> x.length() > 6);
        System.out.println(result);
    }
    //定义方法, 方法可以把 List 列表中符合条件的元素存储到一个新的 List 列表中返回
    public  static <T> List<T> filter(List<T> list, Predicate<T> predicate){
        List<T>  result = new ArrayList<>();
        //遍历 list 参数列表,把符合 predicate 条件的元素存储到 result 中
        for( T  t : list){
            if( predicate.test(t)){
                result.add(t);
            }
        }
        return result;
    }
```

```
}
```

2 Consumer 接口

Consumer<T>接口定义了一个 accept(T)抽象方法,可以接收一个 T 类型的对象,没有返回值. 如果需要访问类型 T 的对象,对该对象做一些操作,就可以使用这个接口. 在 Collection 集合和 Map 集合中都有 forEach(Consumer)方法

```
public class Test02Consumer {
    public static void main(String[] args) {
        List<String > list =Arrays.asList("lisi", "gg", "jj", "tuantuan", "daimeir",
"timo","XDD");
        list.forEach(s -> System.out.println(s) );

        Map<String,Integer> map = new HashMap<>();
        map.put("lisi", 22);
        map.put("feifei", 28);
        map.put("zhangxiaosan", 20);
        map.put("chenqi", 30);
        map.forEach((k,v)-> System.out.println(k + "->" + v) );

    }
}
```

3 Function 接口

Function<T,R>接口中定义了 accept(T)方法,接收一个 T 类型的参数

对象,返回一个 R 类型的数据. 如果需要定义一个 Lambda,将一个输入对象的信息加工后映射到输出,就可以使用该接口

```
public class Test03Function {
    public static void main(String[] args) {
        List<String > list = Arrays.asList("lisi", "gg", "jj", "tuantuan",
"daimeir", "timo","XDD");

        //把 list 集合中存储字符串的长度映射出来
        List<Integer>  result = map(list, x->x.length() );
        System.out.println(result);

    }
    //把一个 List 列表映射到另外一个 List 列表中
    public static <T,R>List<R>  map(List<T> list, Function<T,R> function){
        List<R> result = new ArrayList<>();         //存储映射后的数据
        for( T t : list){
            result.add( function.apply(t) );        //把 apply 对 t 对象的操作结果保存
到 result 集合中
        }
        return  result;
    }
}
```

4 原始类型的处理

泛型只能绑定引用类型,不能使用基本类型

Java8 中函数式接口为基本类型也提供了对应的接口,可以避免在进行输入输出原始数据时频繁进行装箱/拆箱操作.

一般来说,在针对专门的基本类型数据的函数式接口名称前面加上了对应的原始类型前缀,如: IntPredicate, IntConsumer,

IntFunction 等.

IntPredicate evenNumbers = (int x) -> x % 2 == 0 ;

evenNumbers.test(10); //返回 true

evenNumbers.test(11); //返回 false

2.1.3 捕获 Lambda

Lambda 表达式可以使用外层作用域中定义的变量,如成员变量,局部变量,称为捕获 Lambda.

```
package com.wkcto.lambda;

import java.util.function.IntUnaryOperator;

/**
 * author: 动力节点老崔
 * 2019/3/13
 */
public class Test04LambdaVariable {
    int xx = 123;       //实例变量
    static  int yy = 456;       //静态变量

    public static void main(String[] args) {

        //在 Lambda 表达式中使用成员变量,当前 main 方法是静态方法,只能使用静态变量
        IntUnaryOperator operator = i ->{
            return  i + yy;         //把参数接收的数据与静态变量的值相加并返回
        };
        System.out.println( operator.applyAsInt(10));

        //在 Lambda 表达式中使用局部变量,局部变量必须是 final 修饰,或者是事实上的 final
        int zz = 789;       //局部变量
        final  int ff = 147;    //final 修饰的局部变量
        IntUnaryOperator operator2 = i->{
            return  i + ff;    //把参数接收的数据与 ff 的值相加,然后返回
```

```
        };
        System.out.println(operator2.applyAsInt(10));

        operator2 = i ->{
            return  i + zz; //zz 虽然没有使用final 修饰,如果它是事实的final,后面没有修改zz 值的代码
        };
        System.out.println( operator2.applyAsInt(10));
//        zz = 258;     //如果再对zz 重新赋值,则上面的lambda 表达式语法错误

    }
}
```

2.2 方法引用

方法引用可以让你重复使用现有的方法定义，并像 Lambda 一样传递它们. 如:

list.forEach(x -> System.out.println(x));

使用方法引用可以这样:

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

方法引用可以看作是仅仅调用特定方法的 Lambda 表达式的一种快捷写法.

需要使用方法引用时，目标引用放在分隔符::前面，方法名放在::的后面，注意,只需要方法名不需要小括弧

(Student stu) -> stu.getScore()　改为方法引用: Student::getScore

() ->Thread.currentThread().dumpStack() 改为方法引用:

Thread.currentThread()::dumpStack

(Str, i) -> Str.substring(i)　改为方法引用: String::substring

a -> System.out.println(a)　改为方法引用: System.out::prinltn

1 如何构建方法引用

方法引用主要有三类:

1)指向静态方法的方法引用

Lambda表达式	(args) -> ClassName.staticMethod(args)
方法引用	ClassName::staticMethod

Integer[] data　={65,23,87,2,34,99};

Arrays.sort(　data,　Integer::compare);

2) 指向任意类型的实例方法的引用

Lambda 表达式 (args0, args1) -> args0.instanceMethod(args1)

args0 是 Classname 类型的一个对象

方法引用:　Classname :: instanceMethod

3) 指向现有对象的实例方法的引用

Lambda 表达式: (args) -> obj.instanceMethod(args)

方法引用:　obj::instanceMethod

```
public class Test05MethodReference {
    public static void main(String[] args) {
```

```
        Integer[] data  ={65,23,87,2,34,99};
        Arrays.sort(  data,  Integer::compare );        //引用静态方法
        System.out.println(Arrays.toString(data));

        List<String> list = Arrays.asList("WKcto", "Abc", "XXx");
        list.sort(String::compareTo);                   //引用实例方法
        System.out.println(list);

        list.forEach(System.out::println);      //对 System.out 这个实例方法的引用
    }
}
```

2 构造方法引用

对于一个现有的构造方法,可以使用类名和关键字 new 来创建一个构造方法的引用: Classname::new

```
package com.wkcto.lambda;

import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

/**
 * author: 动力节点老崔
 * 2019/3/13
 */
public class Test06ConstructorReference {
    public static void main(String[] args) {
        //1)引用无参构造方法
        Supplier<Person> supplier = Person::new;
        Person p1 = supplier.get();
        System.out.println(p1);

        //2)引用有一个参数的构造方法
        Function<String,Person> function = Person::new;
```

```
        Person p2 = function.apply("feifei");
        System.out.println( p2 );

        //3)引用有两个参数的构造方法
        BiFunction<String,Integer,Person> biFunction = Person::new;
        Person p3 = biFunction.apply("feifei", 28);
        System.out.println(p3);

        //4)如果引用有三个参数及三个以上参数的构造方法,需要自定义匹配的函数式接口
        TriFunction<String, Integer, String, Person> triFunction = Person::new;
        Person p4 = triFunction.apply("laodu", 35, "男");
        System.out.println(p4 );
    }
}

@FunctionalInterface
interface TriFunction<T,U,V,R>{
    R apply(T t, U u, V v);
}

class  Person{
    String name;
    int age;
    String gender;

    public Person(String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this.name = name;
    }

    public Person() {
    }
```

```
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", gender='" + gender + '\'' +
                '}';
    }
}
```

3 Lambda 练习

```
package com.wkcto.lambda;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

/**
 * author: 动力节点老崔
 * 2019/3/13
 */
public class Test07Exercise {
    public static void main(String[] args) {
        //定义List集合存储Student
        List<Student> list = new ArrayList<>();
        list.add( new Student("lisi", 80));
        list.add( new Student("zhangsan", 30));
        list.add( new Student("wangwu", 90));
        list.add( new Student("feifei", 60));
        list.add( new Student("mingge", 100));
        System.out.println(list);

        //使用匿名内部类排序
        list.sort(new Comparator<Student>() {
            @Override
            public int compare(Student o1, Student o2) {
```

```
                return o1.name.compareTo(o2.name);
            }
        });
        System.out.println(list);

        //使用 Lambda 表达式
        list.sort((p1,p2) -> p2.name.compareTo(p1.name));
        System.out.println(list);

        //Comparator 接口中有一个 comparing 静态方法返回 Comparator 比较器
        list.sort(Comparator.comparing((stu)->stu.name));
        System.out.println(list);

        //方法引用,Student 类中 getScore 方法返回成绩
        list.sort(Comparator.comparing(Student::getScore));
        System.out.println(list);
    }
}

class  Student{
    String name;
    int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", score=" + score +
                '}';
    }

    public int getScore() {
        return score;
    }
}
```