lambda 表达式之美

作者：sirwsl
原文链接：https://ld246.com/article/1601888353386
来源网站：链滴

背景

2020年9月27日，我开始成为了一个社畜，同级的同学都在考研、保研、考公务员、考事业单位等等各种考试准备着，这一路有很多很多的老师都劝过我考研，可是我还是决定做一个码农。然而我进入九机科技。然后去了之后在熟悉项目的过程中我发现一个有趣的问题，就是在项目中使用了较多的labda和Stream流操作，所以准备学习一下。前几天因为睡觉、逛街各种事情耽误了，但一想好久没有新博客了，所以还是准备写一下。

简介

lambda表达式是在java8中新增加的新特性，lambda表达式其实就是一个匿名函数，有助于帮助开者对一个接口更好的实现，也可以理解为　lambda表达式就是用来实现接口中的抽象方法。

基础语法

因为lambda是一个匿名函数，而对于一个方法需要关注他的名称、参数类型、个数，返回值。但是于它是匿名函数，所以对于lambda表达式来说就不需要关注他的名称，对于放回值类型也可以省略

- **语法**

```
( ) -> { }
-------------------------------------------------
( ) : 用来描述参数列表
{ } : 用来描述方法体
-> : lambda运算符， goes to
```

- **lambda语法精简**

1：在接口中参数的类型已经限制，所以可以省略（PS：要么全部省略，要么全写）

2：如果参数只有一个，则（）可以省略

原文链接：lambda 表达式之美

3：如果方法只有一句代码{}可以省略

4：如果方法体中唯一语句是return，则省略{}的同时省略return

lambda语法实现与语法精简

```
import org.junit.Test;

public class Demo02 {

    //无精简的普通语法讲解
    @Test
    public void main1(){
        //无参无返回
        NoReturnNoParam lambda1 = ()->{
            System.err.println("无参无放回的lambda简单应用");
        };

        //一个参数无返回
        NoReturnSingleParam lambda2 = (int a)->{
            System.err.println("一个参数无返回，参数："+a);
        };

        //多个参数无返回
        NoResturnSomeParam lambda3 = (int a, int b, int c)->{
            System.err.println("多个参数无返回，参数:"+a+" "+b+" "+c);
        };

        //无参数有返回
        ReturnNoParam lambda4 = ()->{return 100;};


        //一个参数有返回
        ReturnSingleParam lambda5 = (int a)->{ return a+100;};

        //多个参数有返回
        ReturnSomeParam lambda6 = (int a, int b, int c)->{
          return a + b+ c;
        };

        lambda1.test();
        lambda2.test(666);
        lambda3.test(666,888,999);
        System.err.println(lambda4.test());
        System.err.println(lambda5.test(66));
        System.err.println(lambda6.test(1,2,3));

    }


    //精简过的语法
    @Test
    public void mian2(){

        //无参无返回
```

```
        NoReturnNoParam lambda1 = ()->System.err.println("无参无放回的lambda简单应用");


        //一个参数无返回
        NoReturnSingleParam lambda2 = a->System.err.println("一个参数无返回，参数：  "+a);


        //多个参数无返回
        NoResturnSomeParam lambda3 = (a,b,c)->System.err.println("多个参数无返回，参数:"+a+
 "+b+" "+c);

        //无参数有返回
        ReturnNoParam lambda4 = ()->100;


        //一个参数有返回
        ReturnSingleParam lambda5 = a->a+100;

        //多个参数有返回
        ReturnSomeParam lambda6 = (a,b,c)->a + b+ c;

        lambda1.test();
        lambda2.test(66);
        lambda3.test(66,88,99);
        System.err.println(lambda4.test());
        System.err.println(lambda5.test(6));
        System.err.println(lambda6.test(1,2,3));

    }

}

//无参数，无返回
@FunctionalInterface
interface NoReturnNoParam{
    void test();
}

//无返回单参数
@FunctionalInterface
interface NoReturnSingleParam{
    void test(int a);
}

//无返回多参数
@FunctionalInterface
interface  NoResturnSomeParam{
    void test(int a, int b,int c);
}

//有返回值无参数
@FunctionalInterface
interface ReturnNoParam{
    int test();
```

```
}

//有返回值，单参数
@FunctionalInterface
interface ReturnSingleParam{
    int test(int a);
}

//有返回多参数
@FunctionalInterface
interface ReturnSomeParam{
    int test(int a, int b, int c);
}
```

- **lambda语法进阶**
-
    - **方法引用**:

当在不同的地方采用的实现是一样的，则使用方法引用

-
    - **语法**： ::

eg: 方法的隶属者::方法名

方法是静态则引用则采用类名

(Class::Method)

方法的非静态应用采用对象

(Object::Method)

-
    - **PS**：参数类型与返回值以及个数要一致

lambda方法引用代码实现

```
public class Demo03 {
    public static void main(String[] args){

        //引用IsMax方法
        TestInterface test = Demo03::isMax;
        System.err.println(test.testBoolean(1,2));
    }

    static boolean isMax(int a, int b){
        return a > b ? true:false;
    }
}
interface TestInterface{
    boolean testBoolean(int a,int b);
}
```

2. **构造方法的引用**

在接口方法中返回了某个类的对象

构造方法引用lambda表达式实现

```
public class Demo04 {
    public static void main(String[] args){
        CreatePerson person = () -> new Person();

        //构造方法的引用
        CreatePerson person1 = Person::new;
        person1.getPerson();
        CreatePerson2 person2 = Person::new;
        person2.getPerson(18,"sirwsl");

    }

}

interface CreatePerson{
    Person getPerson();
}
interface CreatePerson2{
    Person getPerson(int age,String name);
}

class Person{
    int age;
    String name;

    public Person() {
        System.err.println("person无参构造方法已经被执行");
    }

    public Person(int age, String name) {
        this.age = age;
        this.name = name;
        System.err.println("person带参数构造方法已经被执行");
    }

}
```

接口

优： 使用lambda可以使得一个接口进行非常简洁的实现

PS： lambda表达式可以对某些接口进行简单的实现，但是并不是可以对所有接口进行实现。（接口要实现的抽象方法只能是一个,不能被default修饰）

PS： @FunctionalInterface 修饰函数式接口，接口中抽象方法只能有一个

lambda对接口的实现

```
/**
 * lambda对接口的实现
 * **/
```

```
public class Demo01 {
    public static void main(String[] args){
        //使用接口实现
        Comper comper = new MyComper();
        System.err.println(comper.compare(1,2));


        //采用匿名内部类
        Comper comper1 = new Comper() {
            @Override
            public boolean compare(int a, int b) {
                return a == b ? true:false;
            }
        };
        System.err.println(comper1.compare(1,2));


        //lambda表达式实现接口
        Comper comper2 = (a,b) -> a < b ? true:false;
        System.err.println(comper2.compare(1,2));
    }
}

class MyComper implements Comper{
    @Override
    public boolean compare(int a, int b) {
        return a > b ? true:false;
    }
}
@FunctionalInterface
interface Comper{
    boolean compare(int a, int b);
    //int test();
}
```

lambda综合案例

说了这么多大概也了解了，现在我们来看几个综合案例分别是集合排序中sort方法实现、lambda实现omparator接口、forEach方法实现、removeIf方法实现

综合案例lambda代码实现

```
import org.junit.Test;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.TreeSet;

public class Demo05 {
    //ArrayList中sort实现排序
    @Test
    public void main1 (){
        //在一个ArrayList集合中存许多People对象，需要按照age进行排序
```

```
        ArrayList<People> list = new ArrayList<>();
            list.add(new People(13,"zhangsan"));
            list.add(new People(21,"lisi"));
            list.add(new People(43,"sirwsl"));
            list.add(new People(54,"test"));
            list.add(new People(12,"wangming"));
            list.add(new People(12,"miss li"));

        //lambda排序
        list.sort((o1, o2) -> o1.age-o2.age);
        //System.err.println(list.toString());
        list.forEach(System.out::println);
        list.forEach(people->{
            if (people.age>50) System.out.println("age>50 :"+people);
        });
    }

    @Test
    public void main2(){
        //用lambda表达式实现Comparator接口，并实例化一个TreeSet对象
        TreeSet<People> set = new TreeSet<>(((o1, o2) -> {
            if (o1.age<=o2.age){
                return -1;
            }else {
                return 1;
            }
        }));
            set.add(new People(21,"lisi"));
            set.add(new People(43,"sirwsl"));
            set.add(new People(54,"test"));
            set.add(new People(12,"wangming"));
            set.add(new People(12,"miss li"));
            set.add(new People(13,"zhangsan"));

        //System.out.println(set);
        set.forEach(System.out::println);

        //删除元素,非lambda实现
        /*Iterator<People> peo = set.iterator();
        while (peo.hasNext()){
            if (peo.next().age>20) peo.remove();
        }
        */

        //删除元素，lambda实现
        set.removeIf(people -> people.age>20);
        System.out.println("移除age>20结果：");
        set.forEach(System.out::println);
    }
}



class People{
```

```
    int age;
    String name;

    public People() {
    }

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

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

lambda线程实例化

废话不多说，直接看代码，因为好像没什么可以说的

线程实例化代码实现

```
public class Demo06 {
    public static void main(String[] args){

        Thread thread = new Thread(()->{
            for (int i = 0; i < 100;i++) System.err.println(i);
        });

        thread.start();
    }
}
```

function函数式接口

归纳总结如下，部分没列出来，如果需要，按需枚举就可以

Function函数式接口		
函数	参数	返回值
Predicate<T>	T	boolean
IntPredicate	int	boolean
LongPredicate	long	boolean
DoublePredicate	double	boolean
Consumer<T>	T	void
IntConsumer	int	void
LongConsumer	long	void
DoublePredicate	double	void
Function<T,R>	T	R
IntFunction<R>	int	R
DoubleFunction<R>	double	R
LongFunction<R>	long	R
IntToLongFunction	int	long
IntToDoubleFunction	int	double
DoubleToIntFunction	double	int
DoubleToLongFunction	double	long
LongToDoubleFunction	long	double
LongToIntFunction	long	int
Supplier<T>	void	T
UnaryOperator<T>	T	T
BinaryOperator<T>	（T,T）	T
BiFunction<T,U,R>	(T,U)	R

闭包问题

在lambda表达式中使用闭包提升变量生命周期，使得被引用的变量在方法执行结束之后不被销毁

在lambda表达式中，使用变量需要保证使用的是常量，如过不是常量，在编译阶段会编译为常量，而不能被修改

闭包问题lambda代码实现

```
import org.junit.Test;

import java.util.function.Consumer;
import java.util.function.Supplier;

public class Demo07 {
    //使用闭包延迟变量的生命周期
    @Test
    public void main1(){
        int n = getNumber().get();
        System.err.println(n);
    }
    private Supplier<Integer> getNumber(){
        int num = 100;

        return ()->{
          return num;
        };
    }


    @Test
    public void main2(){
        int a = 10;
        Consumer<Integer> c = t ->{
            System.err.println(a);
        };

        //取消a++注释则程序报错
        //a++;
        c.accept(a);

    }

}
```

样例打包源码

□□heart 点此处下载样例文件，lambda.zip□□heart

□□heart 点此处下载样例文件，lambda.zip□□heart

□□heart 点此处下载样例文件，lambda.zip□□heart