Java8—Lambda 表达式详解_侯彦庆

Lambda简介

Lambda 表达式是 JDK8 的一个新特性，可以取代大部分的匿名内部类，写出更优雅的 Java 代码，尤其在集合的遍历和其他集合操作中，可以极大地优化代码结构。

JDK 也提供了大量的内置函数式接口供我们使用，使得 Lambda 表达式的运用更加方便、高效。

对接口的要求

虽然使用 Lambda 表达式可以对某些接口进行简单的实现，但并不是所有的接口都可以使用 Lambda 表达式来实现。**Lambda 规定接口中只能有一个需要被实现的方法，不是规定接口中只能有一个方法**

> jdk 8 中有另一个新特性：default， 被 default 修饰的方法会有默认实现，不是必须被实现的方法，所以不影响 Lambda 表达式的使用。

@FunctionalInterface

修饰函数式接口的，要求接口中的抽象方法只有一个。 这个注解往往会和 lambda 表达式一起出现。

Lambda 基础语法

我们这里给出六个接口，后文的全部操作都利用这六个接口来进行阐述。

```
/**多参数无返回*/
@FunctionalInterfacepublic
interface NoReturnMultiParam {
    void method(int a, int b);
}
/**无参无返回值*/
@FunctionalInterface
public interface NoReturnNoParam {
    void method();
}
/**一个参数无返回*/
@FunctionalInterface
public interface NoReturnOneParam {
    void method(int a);
}
/**多个参数有返回值*/
@FunctionalInterface
public interface ReturnMultiParam {
    int method(int a, int b);
}
/*** 无参有返回*/
@FunctionalInterface
public interface ReturnNoParam {
    int method();
}
/**一个参数有返回值*/
@FunctionalInterface
```

```
public interface ReturnOneParam {
    int method(int a);
}
```

语法形式为 () -> {}，其中 () 用来描述参数列表，{} 用来描述方法体，-> 为 lambda运算符 ， 读作(goes to)。

```
public class Test1 {
    public static void main(String[] args) {
        //无参无返回
        NoReturnNoParam noReturnNoParam = () -> {
            System.out.println("NoReturnNoParam");
        };
        noReturnNoParam.method();
        //一个参数无返回
        NoReturnOneParam noReturnOneParam = (int a) -> {
            System.out.println("NoReturnOneParam param:" + a);
        };
        noReturnOneParam.method(6);
        //多个参数无返回
        NoReturnMultiParam noReturnMultiParam = (int a, int b) -> {
            System.out.println("NoReturnMultiParam param:" + "{" + a +"," + + b
+"}");
        };
        noReturnMultiParam.method(6, 8);
        //无参有返回值
        ReturnNoParam returnNoParam = () -> {
            System.out.print("ReturnNoParam");
            return 1;
        };
        int res = returnNoParam.method();
        System.out.println("return:" + res);
        //一个参数有返回值
        ReturnOneParam returnOneParam = (int a) -> {
            System.out.println("ReturnOneParam param:" + a);
            return 1;
        };
        int res2 = returnOneParam.method(6);
        System.out.println("return:" + res2);
        //多个参数有返回值
        ReturnMultiParam returnMultiParam = (int a, int b) -> {
            System.out.println("ReturnMultiParam param:" + "{" + a + "," + b
+"}");
            return 1;
        };
        int res3 = returnMultiParam.method(6, 8);
        System.out.println("return:" + res3);
    }
}
```

Lambda 表达式常用示例

- **lambda 表达式引用方法**

有时候我们不是必须要自己重写某个匿名内部类的方法，我们可以可以利用 lambda表达式的接口快速指向一个已经被实现的方法。

语法

方法归属者::方法名 静态方法的归属者为类名，普通方法归属者为对象

```
public class Exe1 {
    public static void main(String[] args) {
        ReturnOneParam lambda1 = a -> doubleNum(a);
        System.out.println(lambda1.method(3));
        //lambda2 引用了已经实现的 doubleNum 方法
        ReturnOneParam lambda2 = Exe1::doubleNum;
        System.out.println(lambda2.method(3));
        Exe1 exe = new Exe1();
        //lambda4 引用了已经实现的 addTwo 方法
        ReturnOneParam lambda4 = exe::addTwo;
        System.out.println(lambda4.method(2));
    }
    /**
    * 要求
    * 1.参数数量和类型要与接口中定义的一致
    * 2.返回值类型要与接口中定义的一致
    */
    public static int doubleNum(int a) {
        return a * 2;
    }
    public int addTwo(int a) {
        return a + 2;
    }
}
```

- **构造方法的引用**

一般我们需要声明接口，该接口作为对象的生成器，通过 类名::new 的方式来实例化对象，然后调用方法返回对象。

```
interface ItemCreatorBlankConstruct {
    Item getItem();
}
interface ItemCreatorParamContruct {
    Item getItem(int id, String name, double price);
}
public class Exe2 {
    public static void main(String[] args) {
        ItemCreatorBlankConstruct creator = () -> new Item();
        Item item = creator.getItem();
        ItemCreatorBlankConstruct creator2 = Item::new;
        Item item2 = creator2.getItem();
        ItemCreatorParamContruct creator3 = Item::new;
        Item item3 = creator3.getItem(112, "鼠标", 135.99);
    }
}
```

- **lambda 表达式创建线程**

我们以往都是通过创建 Thread 对象，然后通过匿名内部类重写 run() 方法，一提到匿名内部类我们就应该想到可以使用 lambda 表达式来简化线程的创建过程。

```
Thread t = new Thread(() -> {
    for (int i = 0; i < 10; i++) {
        System.out.println(2 + ":" + i);
    }
});
t.start();
```

- **遍历集合**

我们可以调用集合的 `public void forEach(Consumer action)` 方法，通过 lambda 表达式的方式遍历集合中的元素。以下是 Consumer 接口的方法以及遍历集合的操作。Consumer 接口是 jdk 为我们提供的一个函数式接口。

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    //....
}
```

```
ArrayList<Integer> list = new ArrayList<>();
Collections.addAll(list, 1,2,3,4,5);
//lambda表达式 方法引用
list.forEach(System.out::println);
list.forEach(element -> {
    if (element % 2 == 0) {
        System.out.println(element);
    }
});
```

- **删除集合中的某个元素**

我们通过`public boolean removeIf(Predicate filter)`方法来删除集合中的某个元素，Predicate 也是 jdk 为我们提供的一个函数式接口，可以简化程序的编写。

```
ArrayList<Item> items = new ArrayList<>();
items.add(new Item(11, "小牙刷", 12.05 ));
items.add(new Item(5, "日本马桶盖", 999.05 ));
items.add(new Item(7, "格力空调", 888.88 ));
items.add(new Item(17, "肥皂", 2.00 ));
items.add(new Item(9, "冰箱", 4200.00 ));
items.removeIf(ele -> ele.getId() == 7);
//通过 foreach 遍历，查看是否已经删除
items.forEach(System.out::println);
```

- **集合内元素的排序**

在以前我们若要为集合内的元素排序，就必须调用 sort 方法，传入比较器匿名内部类重写 compare 方法，我们现在可以使用 lambda 表达式来简化代码。

```
ArrayList<Item> list = new ArrayList<>();
list.add(new Item(13, "背心", 7.80));
list.add(new Item(11, "半袖", 37.80));
```

```
list.add(new Item(14, "风衣", 139.80));
list.add(new Item(12, "秋裤", 55.33));
/*
list.sort(new Comparator<Item>() {
    @Override
    public int compare(Item o1, Item o2) {
        return o1.getId()  - o2.getId();
    }
});
*/
list.sort((o1, o2) -> o1.getId() - o2.getId());
System.out.println(list);
```

Lambda 表达式中的闭包问题

这个问题我们在匿名内部类中也会存在，如果我们把注释放开会报错，告诉我 num 值是 final 不能被改变。这里我们虽然没有标识 num 类型为 final，但是在编译期间虚拟机会帮我们加上 final 修饰关键字。

```
import java.util.function.Consumer;
public class Main {
    public static void main(String[] args) {
        int num = 10;
        Consumer<String> consumer = ele -> {
            System.out.println(num);
        };
        //num = num + 2;
        consumer.accept("hello");
    }
}
```