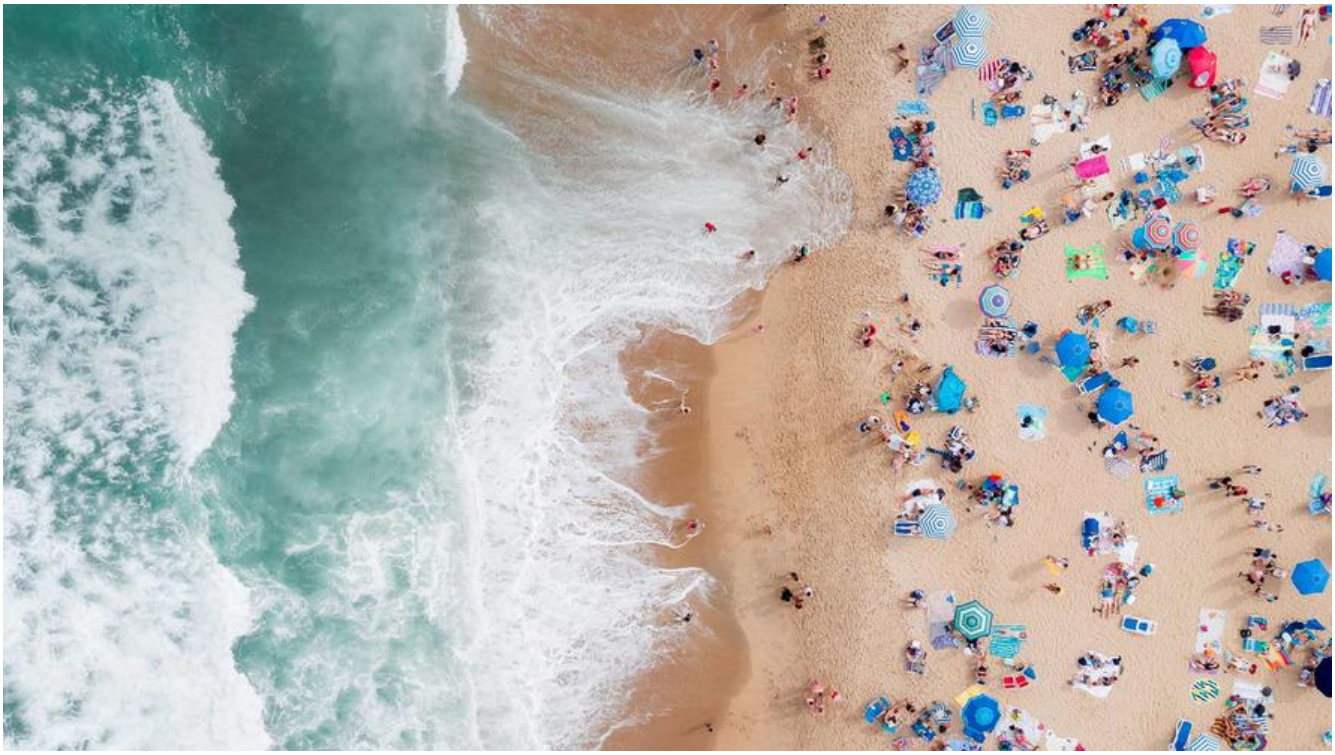
链滴

Java 8 新特性 之 函数式编程

作者：wubo8196
原文链接：https://ld246.com/article/1575968096608
来源网站：链滴



一、函数式接口

1.1 概念

函数式接口在Java中是指：**有且仅有一个抽象方法的接口。**

函数式接口，即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambd才能顺利地进行推导。

备注：“语法糖”是指使用更加方便，但是原理不变的代码语法。例如在遍历集合时使用的for-each法，其实底层的实现原理仍然是迭代器，这便是“语法糖”。从应用层面来讲，Java中的Lambda可被当做是匿名内部 类的“语法糖”，但是二者在原理上是不同的。

1.2 格式

只要确保接口中有且仅有一个抽象方法即可：

```
private interface 接口名称 {
    public abstract 返回值类型 方法名称(可选参数信息);
    // 其他非抽象方法内容
}
```

由于接口当中抽象方法的 public abstract 是可以省略的，所以定义一个函数式接口很简单：

```
public interface MyFunctionalInterface {
```

```
    void myMethod();
}
```

1.3 @FunctionalInterface注解

与@Override注解的作用类似，Java 8中专门为函数式接口引入了一个新的注解@FunctionalInterface该注解可用于一个接口的定义上：

```
@FunctionalInterface

public interface MyFunctionalInterface {

void myMethod();

}
```

一旦使用该注解来定义接口，编译器将会强制检查该接口是否确实有且仅有一个抽象方法，否则将会错。需要注 意的是，即使不使用该注解，只要满足函数式接口的定义，这仍然是一个函数式接口，使起来都一样。

1.4 自定义函数式接口

对于刚刚定义好的MyFunctionalInterface函数式接口，典型使用场景就是作为方法的参数：

```
public class Demo09FunctionalInterface {
    // 使用自定义的函数式接口作为方法参数
    private static void doSomething(MyFunctionalInterface inter) {
        inter.myMethod();
        // 调用自定义的函数式接口方法
    }
    public static void main(String[] args) {
        // 调用使用函数式接口的方法
        doSomething(() -> System.out.println("Lambda执行啦！"));
    }
}
```

二、函数式编程

在兼顾面向对象特性的基础上，Java语言通过Lambda表达式与方法引用等，为开发者打开了函数式程的大门。

2.1 Lambda的延迟执行

有些场景的代码执行后，结果不一定会被使用，从而造成性能浪费。而Lambda表达式是延迟执行的

这正好可以 作为解决方案，提升性能。

性能浪费的日志案例

```
public class Demo01Logger {
    private static void log(int level, String msg) {
        if (level == 1) {
            System.out.println(msg);
        }
    }
    public static void main(String[] args) {
        String msgA = "Hello";
        String msgB = "World";
        String msgC = "Java";
        log(1, msgA + msgB + msgC);
    }
}
```

这段代码存在问题：无论级别是否满足要求，作为 log 方法的第二个参数，三个字符串一定会首先被接并传入方 法内，然后才会进行级别判断。如果级别不符合要求，那么字符串的拼接操作就白做了，在性能浪费。

备注：SLF4J是应用非常广泛的日志框架，它在记录日志时为了解决这种性能浪费的问题，并不推荐先进行 字符串的拼接，而是将字符串的若干部分作为可变参数传入方法中，仅在日志级别满足要求的况下才会进 行字符串拼接。例如： LOGGER.debug("变量{}的取值为{}。", "os", "macOS") ，其中大括号 {} 为占位 符。如果满足日志级别要求，则会将“os”和“macOS”两个字符串依次拼接到大号的位置；否则不会进行字 符串拼接。这也是一种可行解决方案，但Lambda可以做到更好。

体验Lambda的更优写法

使用Lambda必然需要一个函数式接口：

```
@FunctionalInterface
public interface MessageBuilder {

    String buildMessage();
}
```

然后对log方法进行改造：

```
public class Demo02LoggerLambda {
    private static void log(int level, MessageBuilder builder) {
        if (level == 1) {
            System.out.println(builder.buildMessage());
        }
    }
    public static void main(String[] args) {
        String msgA = "Hello";
        String msgB = "World";
        String msgC = "Java";
        log(1, () -> msgA + msgB + msgC );
    }
}
```

这样一来，只有当级别满足要求的时候，才会进行三个字符串的拼接；否则三个字符串将不会进行拼
。

证明Lambda的延迟

下面的代码可以通过结果进行验证：

```
public class Demo03LoggerDelay {
    private static void log(int level, MessageBuilder builder) {
        if (level == 1) {
            System.out.println(builder.buildMessage());
        }
    }
    public static void main(String[] args) {
        String msgA = "Hello";
        String msgB = "World";
        String msgC = "Java";
        log(2, () -> {
            System.out.println("Lambda执行！");
            return msgA + msgB + msgC;
        });
    }
}
```

从结果中可以看出，在不符合级别要求的情况下，Lambda将不会执行。从而达到节省性能的效果。

扩展：实际上使用内部类也可以达到同样的效果，只是将代码操作延迟到了另外一个对象当中通过调方法 来完成。而是否调用其所在方法是在条件判断之后才执行的。

2.2 使用Lambda作为参数和返回值

如果抛开实现原理不说，Java中的Lambda表达式可以被当作是匿名内部类的替代品。如果方法的参是一个函数 式接口类型，那么就可以使用Lambda表达式进行替代。使用Lambda表达式作为方法参，其实就是使用函数式 接口作为方法参数。

例如java.lang.Runnable接口就是一个函数式接口，假设有一个startThread方法使用该接口作为参，那么就可以使用Lambda进行传参。这种情况其实和Thread类的构造方法参数为Runnable没有本区别。

```
public class Demo04Runnable {
    private static void startThread(Runnable task){
        new Thread(task).start();
    }
    public static void main(String[] args){
        startThread(() -> System.out.println("线程任务执行！"));
    }
}
```

类似地，如果一个方法的返回值类型是一个函数式接口，那么就可以直接返回一个Lambda表达式。需要通过一 个方法来获取一个java.util.Comparator接口类型的对象作为排序器时,就可以调该方法获
。

```
import java.util.Arrays;
import java.util.Comparator;
public class Demo06Comparator {
    private static Comparator<String> newComparator() {
        return (a, b) -> b.length() - a.length();
    }
    public static void main(String[] args){
        String[] array = { "abc", "ab", "abcd" };
        System.out.println(Arrays.toString(array));
        Arrays.sort(array, newComparator());
        System.out.println(Arrays.toString(array));
    }
}
```

其中直接return一个Lambda表达式即可。

三、常用函数式接口

JDK提供了大量常用的函数式接口以丰富Lambda的典型使用场景，它们主要在 java.util.function 包被提供。 下面是最简单的几个接口及使用示例。

3.1 Supplier接口

---

java.util.function.Supplier<T>接口仅包含一个无参的方法： T get() 。用来获取一个泛型参数指定型的对 象数据。由于这是一个函数式接口，这也就意味着对应的Lambda表达式需要“**对外提供**”一符合泛型类型的对象数据。

```
import java.util.function.Supplier;
public class Demo08Supplier {
    private static String getString(Supplier<String> function) {
        return function.get();
    }
    public static void main(String[] args) {
        String msgA = "Hello";
        String msgB = "World";
        System.out.println(getString(() -> msgA + msgB));
    }
}
```

3.2例：求数组元素最大值

使用Supplier接口作为方法参数类型，通过Lambda表达式求出int数组中的最大值。提示：接口的泛请使用java.lang.Integer类。

```
public class Demo02Test {
    //定一个方法,方法的参数传递Supplier,泛型使用Integer
```

```
    public static int getMax(Supplier<Integer> sup){
        return sup.get();
    }
    public static void main(String[] args) {
        int arr[] = {2,3,4,52,333,23};
        //调用getMax方法,参数传递Lambda
        int maxNum = getMax(()->{
            //计算数组的最大值
            int max = arr[0];
            for(int i : arr){
                if(i>max){
                    max = i;
                }
            }
            return max;
        });
        System.out.println(maxNum);
    }
}
```

3.3 Consumer接口

---

java.util.function.Consumer<T>接口则正好与Supplier接口相反，它不是生产一个数据，而是**消费**个数据，其数据类型由泛型决定。

抽象方法：accept

Cousumer接口包含抽象方法void accept(T t)，意为消费一个指定泛型的数据，基本使用如：

```
import java.util.function.Consumer;
public class Demo9Consumer{
    private static void consumeString(Consumer<String> function){
        function.accept("Hello");
    }
    public statis void main(String[] args){
        consumeString(s -> System.out.println)
    }
}
```

当然，更好的写法是使用方法引用。

默认方法：andThen

如果一个方法的参数和返回值全都是Consumer类型，那么就可以实现效果：消费数据的时候，首先一个操作，

然后再做一个操作，实现组合。而这个方法就是Consumer接口中的default方法andThen 。下面是JK的源代码：

```
default Consumer<T> andThen(Consumer<? super T> after) {
    Objects.requireNonNull(after);
```

```
    return (T t) -> { accept(t); after.accept(t); };
}
```

备注：java.util.Objects的requireNonNull静态方法将会在参数为null时主动抛出NullPointerException异常。这省去了重复编写if语句和抛出空指针异常的麻烦。

要想实现组合，需要两个或多个Lambda表达式即可，而andThen的语义正是“一步接一步”操作。如两个步骤组合的情况：

```
import java.util.function.Consumer;
public class Demo10ConsumerAndThen {
    private static void consumeString(Consumer<String> one, Consumer<String> two) {
        one.andThen(two).accept("Hello");
    }
    public static void main(String[] args) {
        consumeString(
            s -> System.out.println(s.toUpperCase()),
            s -> System.out.println(s.toLowerCase())
        );
    }
}
```

运行结果将会首先打印完全大写的HELLO，然后打印完全小写的hello。当然，通过链式写法可以实更多步骤的

组合。

3.4 例：格式化打印信息

下面的字符串数组当中存有多条信息，请按照格式“姓名：XX。性别：XX。”的格式将信息打印出。要求将打印姓名的动作作为第一个Consumer接口的Lambda实例，将打印性别的动作作为第二个Consumer接口的Lambda实例，将两个Consumer 接口按照顺序“拼接”到一起。

```
public static void main(String[] args) {
    String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };
}
```

代码：

```
import java.util.function.Consumer;
public class DemoConsumer {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男" };
        printInfo(s -> System.out.print("姓名：" + s.split(",")[0]),
        s -> System.out.println("。性别：" + s.split(",")[1] + "。"),
        array);
    }
    private static void printInfo(Consumer<String> one, Consumer<String> two, String[] array)

        for (String info : array) {
        one.andThen(two).accept(info); // 姓名：迪丽热巴。性别：女。
        }
    }
}
```

3.5 Predicate接口

有时候我们需要对某种类型的数据进行判断，从而得到一个boolean值结果。这时可以使用java.util.function.Predicate<T>接口。

抽象方法：test

Predicate接口中包含一个抽象方法： boolean test(T t) 。用于条件判断的场景：

```
import java.util.function.Predicate;
public class Demo15PredicateTest {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }
    public static void main(String[] args) {
        method(s -> s.length() > 5);
    }
}
```

条件判断的标准是传入的Lambda表达式逻辑，只要字符串长度大于5则认为很长。

默认方法：and

既然是条件判断，就会存在与、或、非三种常见的逻辑关系。其中将两个Predicate条件使用“与”辑连接起来实现“**并且**”的效果时，可以使用default方法and 。其JDK源码为：

```
default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
}
```

如果要判断一个字符串既要包含大写“H”，又要包含大写“W”，那么：

```
import java.util.function.Predicate;
public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.and(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }
    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：or

与and的"与"类似，默认方法or实现逻辑与中的"或"，JDK源码为：

```
default Predicate<T> or(Predicate<? super T> other) {
```

```
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
}
```

如果希望实现逻辑“字符串包含大写H或者包含大写W”，那么代码只需要将“and”修改为“or”称即可，其他都不变：

```
import java.util.function.Predicate;
public class Demo16PredicateAnd {
    private static void method(Predicate<String> one, Predicate<String> two) {
        boolean isValid = one.or(two).test("Helloworld");
        System.out.println("字符串符合要求吗：" + isValid);
    }
    public static void main(String[] args) {
        method(s -> s.contains("H"), s -> s.contains("W"));
    }
}
```

默认方法：negate

“与”、“或”已经了解了，剩下的“非”（取反）也会简单。默认方法negate 的JDK源代码为：

```
default Predicate<T> negate() {
    return (t) -> !test(t);
}
```

从实现中很容易看出，它是执行了test方法之后，对结果boolean值进行“!”取反而已。一定要在test法调用之前调用negate方法，正如and和or方法一样：

```
import java.util.function.Predicate;
public class Demo17PredicateNegate {
    private static void method(Predicate<String> predicate) {
        boolean veryLong = predicate.negate().test("HelloWorld");
        System.out.println("字符串很长吗：" + veryLong);
    }
    public static void main(String[] args) {
        method(s -> s.length() < 5);
    }
}
```

3.6 例：集合信息筛选

数组当中有多条“姓名+性别”的信息如下，请通过Predicate 接口的拼装将符合要求的字符串筛选集合

ArrayList 中，需要同时满足两个条件：

1. 必须为女生；
2. 姓名为4个字。

```
public class DemoPredicate {
```

```
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
    }
}
```

代码：

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
public class DemoPredicate {
    public static void main(String[] args) {
        String[] array = { "迪丽热巴,女", "古力娜扎,女", "马尔扎哈,男", "赵丽颖,女" };
        List<String> list = filter(array,
        s -> "女".equals(s.split(",")[1]),
        s -> s.split(",")[0].length() == 4);
        System.out.println(list);
    }
    private static List<String> filter(String[] array, Predicate<String> one,
        Predicate<String> two) {
        List<String> list = new ArrayList<>();
        for (String info : array) {
            if (one.and(two).test(info)) {
                list.add(info);
            }
        }
        return list;
    }
}
```

3.7 Function接口

---

java.util.function.Function<T,R>接口用来根据一个类型的数据得到另一个类型的数据，前者称为前条件，

后者称为后置条件。

抽象方法：apply

Function接口中最主要的抽象方法为：R apply(T t)，根据类型T的参数获取类型R的结果。

使用的场景例如：将String类型转换为Integer类型。

```
import java.util.function.Function;
public class Demo11FunctionApply {
    private static void method(Function<String, Integer> function) {
        int num = function.apply("10");
        System.out.println(num + 20);
    }
    public static void main(String[] args) {
        method(s -> Integer.parseInt(s));
    }
```

```
}
```

当然，最好是通过方法引用的写法。

默认方法：andThen

Function接口中有一个默认的andThen方法，用来进行组合操作。JDK源代码如：

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
}
```

该方法同样用于“先做什么，再做什么”的场景，和Consumer中的andThen差不多：

```
import java.util.function.Function;
public class Demo12FunctionAndThen {
    private static void method(Function<String, Integer> one, Function<Integer, Integer> two)

        int num = one.andThen(two).apply("10");
        System.out.println(num + 20);
    }
    public static void main(String[] args) {
        method(str->Integer.parseInt(str)+10, i-> i*= 10);
    }
}
```

第一个操作是将字符串解析成为int数字，第二个操作是乘以10。两个操作通过andThen按照前后顺组合到了一起。

请注意，Function的前置条件泛型和后置条件泛型可以相同。

3.8 例：自定义函数模型拼接

使用Function进行函数模型的拼接，按照顺序需要执行的多个函数操作为：

String str = "赵丽颖,20";

1. 将字符串截取数字年龄部分，得到字符串；
2. 将上一步的字符串转换成为int类型的数字；
3. 将上一步的int数字累加100，得到结果int数字。

代码：

```
import java.util.function.Function;
public class DemoFunction {
    public static void main(String[] args) {
        String str = "赵丽颖,20";
        int age = getAgeNum(str, s -> s.split(",")[1],
        s ->Integer.parseInt(s),
        n -> n += 100);
        System.out.println(age);
    }
```

```
    private static int getAgeNum(String str, Function<String, String> one,
        Function<String, Integer> two,
        Function<Integer, Integer> three) {
        return one.andThen(two).andThen(three).apply(str);
    }
}
```

原文链接：Java 8 新特性 之 函数式编程