JAVA8新特性讲解

鲁班学院-周瑜

曾参与大型电商平台、互联网金融产品等多家互联网公司的开发，曾就职于大众点评，任项目经理等职位，参与并主导千万级并发电商网站与系统架构搭建

Lambda表达式

在JDK8之前，一个方法能接受的参数都是变量，例如： `object.method(Object o)`

那么，如果需要传入一个**动作**呢？比如回调。

那么你可能会想到匿名内部类。

例如：

匿名内部类是需要依赖接口的，所以需要先定义个接口

```
@FunctionalInterface
public interface PersonCallback {
    void callback(Person person);
}
```

Person类：

```
public class Person {
    private int id;
    private String name;
    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
    // 创建一个Person后，进行回调
    public static void create(Integer id, String name, PersonCallback personCallback)
{
        Person person = new Person(id, name);
        personCallback.callback(person);
    }
}
```

```
    public static void main(String[] args) {
        Person.create(1, "周瑜", new PersonCallback() {
            public void callback(Person person) {
                System.out.println("去注册...");
            }
        });
        Person.create(2, "老师", new PersonCallback() {
            public void callback(Person person) {
                System.out.println("去登陆...");
            }
        });
    }
```

上面的PersonCallback其实就是一种动作，但是我们真正关心的只有callback方法里的内容而已，我们用Lambda表示，可以将上面的代码就可以优化成：

```
Person.create(1, "周瑜", (Person person) -> {System.out.println("去注册...");})
```

有没有发现特别简单了...，但是我们发现Person.create这个方法其实接收的对象依然是PersonCallback这个接口，但是现在传的是一个Lambda表达式，那么**难道Lambda表达式也实现了这个接口？**。问题也放这，我们先看一下Lambda表达式的接口。

Lambda允许把函数作为一个方法的参数，一个lambda由用**逗号分隔的参数列表、->符号、函数体**三部分表示。对于上面的表达式

> 1. **(Person person)**为Lambda表达式的入参，**{System.out.println("去注册...");}**为函数体
> 2. 重点是这个表达式是没有名字的。

我们知道，当我们实现一个接口的时候，肯定要实现接口里面的方法，那么现在一个Lambda表达式应该也要遵循这一个基本准则，那么一个Lambda表达式它实现了接口里的什么方法呢？

答案是：一个Lambda表达式实现了接口里的有且仅有的唯一一个抽象方法。那么对于这种接口就叫做**函数式接口**。

Lambda表达式其实完成了**实现接口并且实现接口里的方法**这一功能，也可以认为Lambda表达式代表一种**动作**，我们可以直接把这种特殊的**动作**进行传递。

当然，对于上面的Lambda表达式你可以简化：

```
Person.create(1, "周瑜", person -> System.out.println("去注册..."));
```

这归功于Java8的类型推导机制。因为现在接口里只有一个方法，那么现在这个Lambda表达式肯定是对应实现了这个方法，既然是唯一的对应关系，那么入参肯定是Person类，所以可以简写，并且方法体只有唯一的一条语句，所以也可以简写，以达到表达式简洁的效果。

函数式接口

函数式接口是新增的一种接口定义。

用**@FunctionalInterface**修饰的接口叫做**函数式接口**，或者，**函数式接口就是一个只具有一个抽象方法的普通接口**，**@FunctionalInterface**可以起到校验的作用。

下面的接口只有一个抽象方法能编译正确：

```
@FunctionalInterface
public interface TestFunctionalInterface {
    void test1();
}
```

下面的接口有多个抽象方法会编译错误：

```
@FunctionalInterface
public interface TestFunctionalInterface {

    void test1();

    void test2();

}
```

在JDK7中其实就已经有一些函数式接口了，比如**Runnable**、**Callable**、**FileFilter**等等。

在JDK8中也增加了很多函数式接口，比如**java.util.function**包。

比如这四个常用的接口：

接口	描述
Supplier	无参数，返回一个结果
Function	接受一个输入参数，返回一个结果
Consumer	接受一个输入参数，无返回结果
Predicate	接受一个输入参数，返回一个布尔值结果。

> 那么Java8中给我们加了这么多函数式接口有什么作用？

上文我们分析到，一个Lambda表达式其实也可以理解为一个函数式接口的实现者，但是作为表达式，它的写法其实是多种多样的，比如

- () -> {return 0;}，没有传入参数，有返回值
- (int i) -> {return 0;}，传入一个参数，有返回值
- (int i) -> {System.out.println(i)}，传入一个int类型的参数，但是没有返回值
- (int i, int j) -> {System.out.println(i)}，传入两个int类型的参数，但是没有返回值
- (int i, int j) -> {return i+j;}，传入两个int类型的参数，返回一个int值
- (int i, int j) -> {return i>j;}，传入两个int类型的参数，返回一个boolean值

等等，还有许多许多种情况。那么这**每种表达式的写法其实都应该是某个函数式接口的实现类，需要特定函数式接口进行对应**，比如上面的四种情况就分别对应 `Supplier<T>` ， `Function<T,R>` ， `Consumer<T>` ， `BiConsumer<T, U>` ， `BiFunction<T, U, R>` ， `BiPredicate<T, U>` 。

答案已经明显了，**Java8中提供给我们这么多函数式接口就是为了让我们写Lambda表达式更加方便**，当然遇到特殊情况，你还是需要定义你自己的函数式接口然后才能写对应的Lambda表达式。

> 总的来说，如果没有函数式接口，就不能写Lambda表达式.

接口的默认方法与静态方法

在JDK7中，如果想对接口Collection新增一个方法，那么你需要修改它所有的实现类源码（这是非常恐怖的），在那么Java8之前是怎么设计来解决这个问题的呢，用的是抽象类，比如：

现在有一个接口PersonInterface接口，里面有1个抽象方法:

```
public interface PersonInterface {

    void getName();

}
```

有三个实现类：

```
public class YellowPerson implements PersonInterface {

    @Override
    public void getName() {
        System.out.println("yellow");
    }
}

public class WhitePerson implements PersonInterface {

    @Override
    public void getName() {
        System.out.println("white");
    }
}

public class BlackPerson implements PersonInterface {

    @Override
    public void getName() {
        System.out.println("black");
    }
}
```

现在我需要在PersonInterface接口中新增一个方法，那么势必它的三个实现类都需要做相应改动才能编译通过，这里我就不进行演示了，那么我们在最开始设计的时候，其实可以增加一个抽象类PersonAbstract，三个实现类改为继承这个抽象类，按照这种设计方法，对PersonInterface接口中新增一个方法是，其实只需要改动PersonAbstract类去实现新增的方法就好了，其他实现类不需要改动了：

```
public interface PersonInterface {
    void getName();
    void walk();
}
public abstract class PersonAbstract implements PersonInterface {
    @Override
    public void walk() {
        System.out.println("walk");
    }
}
public class BlackPerson extends PersonAbstract {
    @Override
    public void getName() {
        System.out.println("black");
    }
}
public class WhitePerson extends PersonAbstract {
    @Override
    public void getName() {
        System.out.println("white");
    }
}
public class YellowPerson extends PersonAbstract {
    @Override
    public void getName() {
        System.out.println("yellow");
    }
}
```

那么在Java8中支持直接在接口中添加已经实现了的方法，一种是Default方法（默认方法），一种Static方法（静态方法）。

接口的默认方法

在接口中用**default**修饰的方法称为**默认方法**。

接口中的默认方法一定要有默认实现（方法体），接口实现者可以继承它，也可以覆盖它。

```
    default void testDefault(){
        System.out.println("default");
    };
```

接口的静态方法

在接口中用**static**修饰的方法称为**静态方法**。

```
    static void testStatic(){
        System.out.println("static");
    };
```

调用方式：

```
    TestInterface.testStatic();
```

因为有了默认方法和静态方法，所以你不用去修改它的实现类了，可以进行直接调用。

方法引用

有个函数式接口Consumer，里面有个抽象方法accept能够接收一个参数但是没有返回值，这个时候我想实现accept方法，让它的功能为打印接收到的那个参数，那么我可以使用Lambda表达式这么做：

```
Consumer<String> consumer = s -> System.out.println(s);
consumer.accept("周瑜老师");
```

但是其实我想要的这个功能PrintStream类（也就是System.out的类型）的println方法已经实现了，这一步还可以再简单点，如：

```
Consumer<String> consumer = System.out::println;
consumer.accept("周瑜老师");
```

这就是方法引用，方法引用方法的参数列表必须与函数式接口的抽象方法的参数列表保持一致，返回值不作要求。

使用方法

- 引用方法
- 引用构造方法
- 引用数组

引用方法

- 实例对象::实例方法名
- 类名::静态方法名
- 类名::实例方法名

实例对象::实例方法名

```
//Consumer<String> consumer = s -> System.out.println(s);
Consumer<String> consumer = System.out::println;
consumer.accept("周瑜老师");
```

System.out代表的就是PrintStream类型的一个实例，println是这个实例的一个方法。

类名::静态方法名

```
//Function<Long, Long> f = x -> Math.abs(x);
Function<Long, Long> f = Math::abs;
Long result = f.apply(-3L);
```

Math是一个类而abs为该类的静态方法。Function中的唯一抽象方法apply方法参数列表与abs方法的参数列表相同，都是接收一个Long类型参数。

类名::实例方法名

若Lambda表达式的参数列表的第一个参数，是实例方法的调用者，第二个参数(或无参)是实例方法的参数时，就可以使用这种方法

```
//BiPredicate<String, String> b = (x,y) -> x.equals(y);
BiPredicate<String, String> b = String::equals;
b.test("a", "b");
```

String是一个类而equals为该类的定义的实例方法。BiPredicate中的唯一抽象方法test方法参数列表与equals方法的参数列表相同，都是接收两个String类型参数。

引用构造器

在引用构造器的时候，构造器参数列表要与接口中抽象方法的参数列表一致,格式为 类名::new。如：

```
//Function<Integer, StringBuffer> fun = n -> new StringBuffer(n);
Function<Integer, StringBuffer> fun = StringBuffer::new;
StringBuffer buffer = fun.apply(10);
```

Function接口的apply方法接收一个参数，并且有返回值。在这里接收的参数是Integer类型，与StringBuffer类的一个构造方法StringBuffer(int capacity)对应，而返回值就是StringBuffer类型。上面这段代码的功能就是创建一个Function实例，并把它apply方法实现为创建一个指定初始大小的StringBuffer对象。

引用数组

引用数组和引用构造器很像，格式为 类型[]::new，其中类型可以为基本类型也可以是类。如:

```
// Function<Integer, int[]> fun = n -> new int[n];
Function<Integer, int[]> fun = int[]::new;
int[] arr = fun.apply(10);

Function<Integer, Integer[]> fun2 = Integer[]::new;
Integer[] arr2 = fun2.apply(10);
```

Optional

空指针异常是导致Java应用程序失败的最常见原因，以前，为了解决空指针异常，Google公司著名的Guava项目引入了Optional类，Guava通过使用检查空值的方式来防止代码污染，它鼓励程序员写更干净的代码。受到Google Guava的启发，Optional类已经成为Java 8类库的一部分。

Optional实际上是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

创建Optional对象的几个方法：

1. Optional.of(T value)， 返回一个Optional对象，value不能为空，否则会出空指针异常
2. Optional.ofNullable(T value)， 返回一个Optional对象，value可以为空
3. Optional.empty()，代表空

其他API:

1. optional.isPresent()，是否存在值（不为空）
2. optional.ifPresent(Consumer<? super T> consumer), 如果存在值则执行consumer
3. optional.get()，获取value
4. optional.orElse(T other)，如果没值则返回other
5. optional.orElseGet(Supplier<? extends T> other)，如果没值则执行other并返回
6. optional.orElseThrow(Supplier<? extends X> exceptionSupplier)，如果没值则执行exceptionSupplier，并抛出异常

那么，我们之前对于防止空指针会这么写：

```
public class Order {
    String name;
    public String getOrderName(Order order ) {
        if (order == null) {
            return null;
        }
        return order.name;
    }
}
```

现在用Optional，会改成：

```
public class Order {

    String name;

    public String getOrderName(Order order ) {
//        if (order == null) {
//            return null;
//        }
//
//        return order.name;

        Optional<Order> orderOptional = Optional.ofNullable(order);
        if (!orderOptional.isPresent()) {
            return null;
        }
        return orderOptional.get().name;
    }
}
```

那么如果只是改成这样，实质上并没有什么分别，事实上isPresent() 与 obj != null 无任何分别，并且在使用get()之前最好都使用isPresent() ，比如下面的代码在IDEA中会有警告:'Optional.get()' without 'isPresent()' check。

```
    public void createOrder(String userName) {
        Optional optional = Optional.ofNullable(userName);
        optional.get();
    }
```

另外把 Optional 类型用作属性或是方法参数在 IntelliJ IDEA 中更是强力不推荐的。

对于上面的代码我们利用IDEA的提示可以优化成一行（666！）：

```
public class Order {

    String name;

    public String getOrderName(Order order ) {
//        if (order == null) {
//            return null;
//        }
//
//        return order.name;

        return Optional.ofNullable(order).map(order1 -> order1.name).orElse(null);
    }
}
```

这个优化过程中map()起了很大作用。

高级API：

1. optional.map(Function<? super T, ? extends U> mapper)，映射，映射规则由function指定，返回映射值的Optional，所以可以继续使用Optional的API。
2. optional.flatMap(Function<? super T, Optional< U > > mapper)，同map类似，区别在于map中获取的返回值自动被Optional包装，flatMap中返回值保持不变,但入参必须是Optional类型。
3. optional.filter(Predicate<? super T> predicate)，过滤，按predicate指定的规则进行过滤，不符合规则则返回empty，也可以继续使用Optional的API。

Optional总结

使用 Optional 时尽量不直接调用 Optional.get() 方法, Optional.isPresent() 更应该被视为一个私有方法, 应依赖于其他像 Optional.orElse(), Optional.orElseGet(), Optional.map() 等这样的方法.