day12【函数式接口、方法引用】

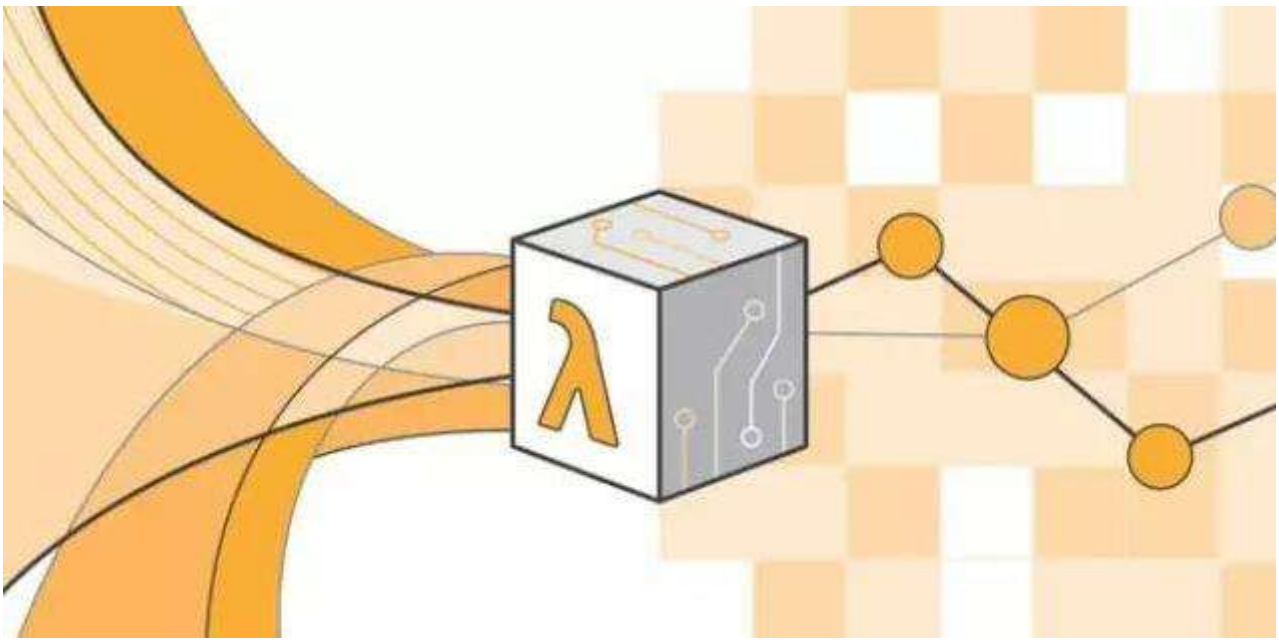
主要内容

- 函数式接口
- 方法引用

教学目标

- [ ] 能够定义函数式接口
- [ ] 能够使用@FunctionalInterface注解
- [ ] 能够自定义无参数、无返回值的函数式接口并使用
- [ ] 能够自定义有参数、有返回值的函数式接口
- [ ] 能够使用输出语句的方法引用
- [ ] 能够理解方法引用与Lambda的关系
- [ ] 能够通过4种方式使用方法引用
- [ ] 能够使用类和数组的构造器引用

第一章 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，所以函数式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Lambda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语法，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可以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：

```
修饰符 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，否则将会报错。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，使用起来都一样。

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！"));
    }
}
```

1.5 练习：自定义函数式接口（无参无返回）

题目

请定义一个函数式接口`Eatable`，内含抽象`eat`方法，没有参数或返回值。使用该接口作为方法的参数，并进而通过Lambda来使用它。

解答

函数式接口的定义：

```
@FunctionalInterface
public interface Eatable {
    void eat();
}
```

应用场景代码：

```
public class DemoLambdaEatable {
    private static void keepAlive(Eatable human) {
        human.eat();
    }

    public static void main(String[] args) {
        keepAlive(() -> System.out.println("吃饭饭！"));
    }
}
```

1.6 练习：自定义函数式接口（有参有返回）

题目

请定义一个函数式接口`Sumable`，内含抽象`sum`方法，可以将两个int数字相加返回int结果。使用该接口作为方法的参数，并进而通过Lambda来使用它。

解答

函数式接口的定义：

```
@FunctionalInterface
public interface Sumable {
    int sum(int a, int b);
}
```

应用场景代码：

```
public class DemoLambdaSumable {
    private static void showSum(int x, int y, Sumable sumCalculator) {
        System.out.println(sumCalculator.sum(x, y));
    }

    public static void main(String[] args) {
        showSum(10, 20, (m, n) -> m + n);
    }
}
```

第二章 方法引用

在使用Lambda表达式的时候，我们实际上传递进去的代码就是一种解决方案：拿什么参数做什么操作。那么考虑一种情况：如果我们在Lambda中所指定的操作方案，已经有地方存在相同方案，那是否还有必要再写重复逻辑？

2.1 冗余的Lambda场景

来看一个简单的函数式接口以应用Lambda表达式：

```
@FunctionalInterface
public interface Printable {
    void print(String str);
}
```

在`Printable`接口当中唯一的抽象方法`print`接收一个字符串参数，目的就是为了打印显示它。那么通过Lambda来使用它的代码很简单：

```
public class Demo01PrintSimple {
    private static void printString(Printable data) {
        data.print("Hello, World!");
    }

    public static void main(String[] args) {
        printString(s -> System.out.println(s));
    }
}
```

其中`printString`方法只管调用`Printable`接口的`print`方法，而并不管`print`方法的具体实现逻辑会将字符串打印到什么地方去。而`main`方法通过Lambda表达式指定了函数式接口`Printable`的具体操作方案为：**拿到String（类型可推导，所以可省略）数据后，在控制台中输出它。**

2.2 问题分析

这段代码的问题在于，对字符串进行控制台打印输出的操作方案，明明已经有了现成的实现，那就是`System.out`对象中的`println(String)`方法。既然Lambda希望做的事情就是调用`println(String)`方法，那何必自己手动调用呢？

2.3 用方法引用改进代码

能否省去Lambda的语法格式（尽管它已经相当简洁）呢？只要“引用”过去就好了：

```
public class Demo02PrintRef {
    private static void printString(Printable data) {
        data.print("Hello, World!");
    }

    public static void main(String[] args) {
        printString(System.out::println);
    }
}
```

请注意其中的双冒号`::`写法，这被称为“**方法引用**”，而双冒号是一种新的语法。

2.4 方法引用符

双冒号`::`为引用运算符，而它所在的表达式被称为**方法引用**。如果Lambda要表达的函数方案已经存在于某个方法的实现中，那么则可以通过双冒号来引用该方法作为Lambda的替代者。

语义分析

例如上例中，`System.out`对象中有一个重载的`println(String)`方法恰好就是我们所需要的。那么对于`printString`方法的函数式接口参数，对比下面两种写法，完全等效：

- Lambda表达式写法：`s -> System.out.println(s);`

- 方法引用写法：`System.out::println`

第一种语义是指：拿到参数之后经Lambda之手，继而传递给`System.out.println`方法去处理。

第二种等效写法的语义是指：直接让`System.out`中的`println`方法来取代Lambda。两种写法的执行效果完全一样，而第二种方法引用的写法复用了已有方案，更加简洁。

推导与省略

如果使用Lambda，那么根据"**可推导就是可省略**"的原则，无需指定参数类型，也无需指定的重载形式——它们都将被自动推导。而如果使用方法引用，也是同样可以根据上下文进行推导。

函数式接口是Lambda的基础，而方法引用是Lambda的孪生兄弟。

下面这段代码将会调用`println`方法的不同重载形式，将函数式接口改为int类型的参数：

```
@FunctionalInterface
public interface PrintableInteger {
    void print(int str);
}
```

由于上下文变了之后可以自动推导出唯一对应的匹配重载，所以方法引用没有任何变化：

```
public class Demo03PrintOverload {
    private static void printInteger(PrintableInteger data) {
        data.print(1024);
    }

    public static void main(String[] args) {
        printInteger(System.out::println);
    }
}
```

这次方法引用将会自动匹配到`println(int)`的重载形式。

2.5 通过对象名引用成员方法

这是最常见的一种用法，与上例相同。如果一个类中已经存在了一个成员方法：

```
public class MethodRefObject {
    public void printUpperCase(String str) {
        System.out.println(str.toUpperCase());
    }
}
```

函数式接口仍然定义为：

```
@FunctionalInterface
public interface Printable {
    void print(String str);
}
```

那么当需要使用这个`printUpperCase`成员方法来替代`Printable`接口的Lambda的时候，已经具有了`MethodRefObject`类的对象实例，则可以通过对象名引用成员方法，代码为：

```
public class Demo04MethodRef {
    private static void printString(Printable lambda) {
        lambda.print("Hello");
    }

    public static void main(String[] args) {
        MethodRefObject obj = new MethodRefObject();
        printString(obj::printUpperCase);
    }
}
```

2.6 练习：对象名引用成员方法

题目

假设有一个助理类`Assistant`，其中含有成员方法`dealFile`如下：

```
public class Assistant {
    public void dealFile(String file) {
        System.out.println("帮忙处理文件：" + file);
    }
}
```

请自定义一个函数式接口`WorkHelper`，其中的抽象方法`help`的预期行为与`dealFile`方法一致，并定义一个方法使用该函数式接口作为参数。通过方法引用的形式，将助理对象中的`help`方法作为Lambda的实现。

解答

函数式接口可以定义为：

```
@FunctionalInterface
public interface WorkHelper {
    void help(String file);
}
```

通过对象名引用成员方法的使用场景代码为：

```
public class DemoAssistant {
    private static void work(WorkHelper helper) {
        helper.help("机密文件");
    }

    public static void main(String[] args) {
        Assistant assistant = new Assistant();
        work(assistant::dealFile);
    }
}
```

2.7 通过类名称引用静态方法

由于在`java.lang.Math`类中已经存在了静态方法`abs`，所以当我们需要通过Lambda来调用该方法时，有两种写法。首先是函数式接口：

```
@FunctionalInterface
public interface Calcable {
    int calc(int num);
}
```

第一种写法是使用Lambda表达式：

```
public class Demo05Lambda {
    private static void method(int num, Calcable lambda) {
        System.out.println(lambda.calc(num));
    }

    public static void main(String[] args) {
        method(-10, n -> Math.abs(n));
    }
}
```

但是使用方法引用的更好写法是：

```
public class Demo06MethodRef {
    private static void method(int num, Calcable lambda) {
        System.out.println(lambda.calc(num));
    }

    public static void main(String[] args) {
        method(-10, Math::abs);
    }
}
```

在这个例子中，下面两种写法是等效的：

- Lambda表达式：`n -> Math.abs(n)`
- 方法引用：`Math::abs`

2.8 练习：类名称引用静态方法

题目

假设有一个`StringUtils`字符串工具类，其中含有静态方法`isBlank`如下：

```
public final class StringUtils {
    public static boolean isBlank(String str) {
        return str == null || "".equals(str.trim());
    }
}
```

请自定义一个函数式接口`StringChecker`，其中的抽象方法`checkBlank`的预期行为与`isBlank`一致，并定义一个方法使用该函数式接口作为参数。通过方法引用的形式，将`StringUtils`工具类中的`isBlank`方法作为Lambda的实现。

解答

函数式接口的定义可以为：

```
@FunctionalInterface
public interface StringChecker {
    boolean checkString(String str);
}
```

应用场景代码为：

```
public class DemoStringChecker {
    private static void methodCheck(StringChecker checker) {
        System.out.println(checker.checkString("   "));
    }

    public static void main(String[] args) {
        methodCheck(StringUtils::isBlank);
    }
}
```

2.9 通过super引用成员方法

如果存在继承关系，当Lambda中需要出现super调用时，也可以使用方法引用进行替代。首先是函数式接口：

```
@FunctionalInterface
public interface Greetable {
    void greet();
}
```

然后是父类`Human`的内容：

```
public class Human {
    public void sayHello() {
        System.out.println("Hello!");
    }
}
```

最后是子类`Man`的内容，其中使用了Lambda的写法：

```
public class Man extends Human {
    @Override
    public void sayHello() {
        method(() -> super.sayHello());
    }

    private void method(Greetable lambda) {
        lambda.greet();
        System.out.println("I'm a man!");
    }
}
```

但是如果使用方法引用来调用父类中的`sayHello`方法会更好，例如另一个子类`Woman`：

```
public class Woman extends Human {
    @Override
    public void sayHello() {
        method(super::sayHello);
    }

    private void method(Greetable lambda) {
        lambda.greet();
        System.out.println("I'm a woman!");
    }
}
```

在这个例子中，下面两种写法是等效的：

- Lambda表达式：`() -> super.sayHello()`
- 方法引用：`super::sayHello`

2.10 通过this引用成员方法

this代表当前对象，如果需要引用的方法就是当前类中的成员方法，那么可以使用“this::成员方法”的格式来使用方法引用。首先是简单的函数式接口：

```
@FunctionalInterface
public interface Richable {
    void buy();
}
```

下面是一个丈夫`Husband`类：

```
public class Husband {
    private void marry(Richable lambda) {
        lambda.buy();
    }

    public void beHappy() {
        marry(() -> System.out.println("买套房子"));
    }
}
```

开心方法`beHappy`调用了结婚方法`marry`，后者的参数为函数式接口`Richable`，所以需要一个Lambda表达式。但是如果这个Lambda表达式的内容已经在本类当中存在了，则可以对`Husband`丈夫类进行修改：

```
public class Husband {
    private void buyHouse() {
        System.out.println("买套房子");
    }

    private void marry(Richable lambda) {
        lambda.buy();
    }

    public void beHappy() {
        marry(() -> this.buyHouse());
    }
}
```

如果希望取消掉Lambda表达式，用方法引用进行替换，则更好的写法为：

```
public class Husband {
    private void buyHouse() {
        System.out.println("买套房子");
    }

    private void marry(Richable lambda) {
        lambda.buy();
    }

    public void beHappy() {
        marry(this::buyHouse);
    }
}
```

在这个例子中，下面两种写法是等效的：

- Lambda表达式：`() -> this.buyHouse()`
- 方法引用：`this::buyHouse`

2.11 类的构造器引用

由于构造器的名称与类名完全一样，并不固定。所以构造器引用使用`类名称::new`的格式表示。首先是一个简单的`Person`类：

```
public class Person {
    private String name;

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

    public String getName() {
        return name;
    }
}
```

然后是用来创建`Person`对象的函数式接口：

```
public interface PersonBuilder {
    Person buildPerson(String name);
}
```

要使用这个函数式接口，可以通过Lambda表达式：

```
public class Demo09Lambda {
    public static void printName(String name, PersonBuilder builder) {
        System.out.println(builder.buildPerson(name).getName());
    }

    public static void main(String[] args) {
        printName("赵丽颖", name -> new Person(name));
    }
}
```

但是通过构造器引用，有更好的写法：

```
public class Demo10ConstructorRef {
    public static void printName(String name, PersonBuilder builder) {
        System.out.println(builder.buildPerson(name).getName());
    }

    public static void main(String[] args) {
        printName("赵丽颖", Person::new);
    }
}
```

在这个例子中，下面两种写法是等效的：

- Lambda表达式：`name -> new Person(name)`
- 方法引用：`Person::new`

2.12 数组的构造器引用

数组也是`Object`的子类对象，所以同样具有构造器，只是语法稍有不同。如果对应到Lambda的使用场景中时，需要一个函数式接口：

```
@FunctionalInterface
public interface ArrayBuilder {
    int[] buildArray(int length);
}
```

在应用该接口的时候，可以通过Lambda表达式：

```
public class Demo11ArrayInitRef {
    private static int[] initArray(int length, ArrayBuilder builder) {
        return builder.buildArray(length);
    }

    public static void main(String[] args) {
        int[] array = initArray(10, length -> new int[length]);
    }
}
```

但是更好的写法是使用数组的构造器引用：

```
public class Demo12ArrayInitRef {
    private static int[] initArray(int length, ArrayBuilder builder) {
        return builder.buildArray(length);
    }

    public static void main(String[] args) {
        int[] array = initArray(10, int[]::new);
    }
}
```

在这个例子中，下面两种写法是等效的：

- Lambda表达式：`length -> new int[length]`
- 方法引用：`int[]::new`

> 备注：数组的构造器引用，可以和Java 8的Stream API结合，在一定程度上“解决”集合中`toArray`方法的泛型擦除问题。