day12_函数式接口

一.函数式接口

函数式接口，即适用于函数式编程场景的接口。Java 中函数式编程的体现就是 Lambda 表达式，然而只有当接口中的抽象方法有且仅有一个时，Lambda 表达式才能顺利地进行推导。所以函数式接口在 Java 中是指：有且仅有一个抽象方法的接口。

```
修饰符 interface 接口名称 {

    // 仅有的一个抽象方法
    public abstract 返回值类型 方法名称(参数列表);

    // 其他非抽象方法内容

}
```

1.@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，用来检查一个接口是否是函数式接口。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，否则编译会报错。

```
@FunctionalInterface
public interface DemoFunctionalInterface {

    // 仅有的一个抽象方法
    public abstract void method();

    // 其他非抽象方法内容

}
```

2.函数式接口的使用

#.注意："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 表达式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。

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、方法引用等，为开发者打开了函数式编程的大门。

1.Lambda 表达式的延迟执行

有些场景的代码在执行后，结果不一定会被使用，从而造成性能浪费。如下性能浪费的日志案例：

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

        // 定义三个日志信息字符串
        String msg1 = "Hello";
        String msg2 = "World";
        String msg3 = "Java";

        // 调用showLog方法，传递日志级别和拼接后的日志信息
        showLog( level: 2,  message: msg1 + msg2 + msg3);

    }

    // 定义一个根据日志级别，显示日志信息的方法：若级别是1，则输出日志信息
    private static void showLog(int level, String message) {
        if (level == 1) {
            System.out.println(message);
        }
    }
}
```

这段代码存在的问题：无论日志级别是否为1，三个字符串一定会首先被拼接并传入方法内，然后才会进行级别判断。如果级别不是1，那么字符串的拼接操作就白做了，存在性能浪费。

日志可以帮助我们快速的定位问题，记录程序运行过程中的情况，以便项目的监控和优化。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，并不推荐首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情况下才会进行字符串拼接，这也是一种可行解决方案。

但 Lambda 表达式可以做到更好，由于 Lambda 表达式是延迟执行的，这正好可以作为解决方案，提升程序性能。使用 Lambda 表达式优化日志案例的代码如下：

```java
// 要想使用Lambda表达式，必须有一个函数式接口
@FunctionalInterface
public interface MessageBuilder {

    // 定义一个拼接消息的抽象方法，返回被拼接的消息
    public abstract String buildMessage();

}

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

        // 定义三个日志信息字符串
        String msg1 = "Hello";
        String msg2 = "World";
        String msg3 = "Java";

        // 调用showLog方法，方法的参数是一个接口，所以可以传递接口的匿名内部类
        showLog( level: 2, new MessageBuilder() {
            @Override
            public String buildMessage() {
                return msg1 + msg2 + msg3;
            }
        });

        // 调用showLog方法，参数MessageBuilder是一个函数式接口，所以可以传递Lambda表达式
        showLog( level: 2, () -> { return msg1 + msg2 + msg3; });
    }

    // 定义一个显示日志的方法，方法的参数传递日志的等级和MessageBuilder接口
    private static void showLog(int level, MessageBuilder mb) {
        if (level == 1) {
            System.out.println(mb.buildMessage());
        }
    }
}
```

此时匿名内部类的重写方法体也不会被执行

此时Lambda表达式不会被执行

我们调用 showLog 方法传递参数时有：level = 2，MessageBuilder mb = ( ) -> { return msg1 +msg2 +msg3; }。由于日志等级为2，所以此时 Lambda 表达式仅仅作为一段代码被传递给方法的参数 mb(Lambda 表达式就相当于是接口的一个实现类对象)，表达式内部的方法体并没有被执行，所以不会进行字符串的拼接。当日志的等级为1时，mb 调用 buildMessage 方法，此时才会执行表达式内部的方法体，进行字符串的拼接。(传递接口的匿名内部类的方式同理)

所以Lambda 表达式只有当"接口的抽象方法以多态的形式被调用的时候"才会被执行，在传递过程中不执行，这就是 Lambda 表达式的延迟执行。

2.使用 Lambda 表达式作为方法的参数和返回值

三.常用函数式接口

JDK 中提供了大量常用的函数式接口以丰富 Lambda 表达式的典型使用场景，它们主要在 java.util.function 包中被提供。下面是几个简单的函数式接口及使用示例。

1.Supplier 接口

java.util.function.Supplier<T> 接口仅包含一个无参的抽象方法：T get( )，用来获取一个泛型参数指定类型的对象数据。基本用法如下：

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}

public class SupplierImpl implements Supplier<String> {
    // 制定数据的生产规则：返回一个字符串常量
    @Override
    public String get() {
        return "Don't be afraid to fail!";
    }
}

public class DemoMain {
    public static void main(String[] args) {
        SupplierImpl s = new SupplierImpl();
        System.out.println(s.get()); // "Don't be afraid to fail!"
    }
}
```

Supplier<T> 接口被称为生产型接口，即在实现类中指定接口的泛型是什么类型，那么在实现类中重写 get 方法时，就可以随便 return 一个指定泛型类型的数据。那么该 get 方法的作用就是可以返回一个指定泛型类型的数据，即生产一个指定泛型类型的数据。

(要体会"接口是用来制定规范的"这种思想：接口中的抽象方法通过返回值、参数来制定该方法的规则，即该方法是用来做什么的，那么其实现类在重写抽象方法时也要遵守这种规则，即实现类重写后的抽象方法也是用来做这种事的。注意体会这种思想在 Supplier 接口中的应用)

(1).Supplier 接口的 Lambda 用法

(2).Supplier 接口的练习

2.Consumer 接口

java.util.function.Consumer<T> 接口与 Supplier 接口正好相反，它不是生产一个数据，而是消费一个数据。接口中仅有的一个抽象方法 void accept(T var1)，用来消费一个指定泛型类型的数据。accept 方法的基本用法如下：

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T var1);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (t) -> {
            this.accept(t);
            after.accept(t);
        };
    }
}

public class ConsumerImpl implements Consumer<String> {
    // 制定数据的消费规则：将字符串转换成大写形式输出
    @Override
    public void accept(String s) {
        System.out.println(s.toUpperCase());
    }
}

public class DemoMain {
    public static void main(String[] args) {
        ConsumerImpl cImpl = new ConsumerImpl();
        cImpl.accept( s: "Stay brave and optimistic!"); // STAY BRAVE AND OPTIMISTIC!
    }
}
```

Consumer<T> 接口被称为消费型接口，即在实现类中指定接口的泛型是什么类型，那么在实现类中重写 accept 方法时，就可以给方法传递一个指定泛型类型的数据，然后可以在方法内部自定义数据的使用方式(输出、计算等)。那么该 accept 方法的作用就是对一个指定泛型类型的数据进行使用，即消费一个指定泛型类型的数据。

(1).Consumer 接口的 Lambda 用法

(2).Consumer 接口中的默认方法 andThen

(3).Consumer 接口的练习

3.Predicate 接口

java.util.function.Predicate<T> 接口仅包含一个抽象方法：boolean test(T t)，用来对一个指定泛型类型的数据进行条件判断，从而得到一个 boolean 值结果。基本用法如下：

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T var1);

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

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

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

```java
public class PredicateImpl implements Predicate<String> {
    // 制定数据的条件判断规则：字符串的长度是否大于5
    @Override
    public boolean test(String s) {
        return s.length() > 5;
    }
}

public class DemoMain {
    public static void main(String[] args) {
        PredicateImpl pImpl = new PredicateImpl();
        boolean b = pImpl.test( s: "abcdef");
        System.out.println("字符串的长度是否大于5：" + b);
    }
}
```

Predicate<T> 接口被称为判断型接口，即在实现类中指定接口的泛型是什么类型，那么在实现类中重写 test 方法时，就可以给方法传递一个指定泛型类型的数据，然后可以在方法内部自定义数据的判断条件，并返回一个 boolean 值结果。那么该 test 方法的作用就是对一个指定泛型类型的数据进行条件判断，即判断一个指定泛型类型的数据。

(1).Predicate 接口的 Lambda 用法

(2).Predicate 接口中的默认方法 and、or、negate

(3).Predicate 接口的练习

4.Function 接口

java.util.function.Function<T, R> 接口仅包含一个无参的抽象方法：R apply(T t)，用来根据一个指定泛型类型的数据得到另一个指定泛型类型的数据。基本用法如下：

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T var1);

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

public class FunctionImpl implements Function<String, Integer> {
    // 制定数据的转换规则：将String类型数据转换成Integer类型
    @Override
    public Integer apply(String s) {
        return Integer.parseInt(s);
    }
}

public class DemoMain {
    public static void main(String[] args) {
        FunctionImpl fImpl = new FunctionImpl();
        int i = fImpl.apply( s: "123456"); // 自动拆箱
        System.out.println(i); // 123456
    }
}
```

Function<T, R> 接口被称为转换型接口，前者称为前置条件，后者称为后置条件。即在实现类中指定前置条件是什么类型，那么在实现类中重写 apply 方法时，就可以给方法传递一个指定前置条件类型的数据，然后可以在方法内部自定义数据的转换方式，即将数据由前置条件类型转换成后置条件类型。那么该 apply 方法的作用就是将一个指定泛型类型的数据转换成另一种指定泛型类型的数据并返回，即转换一个指定泛型类型的数据。

(1).Function 接口的 Lambda 用法

(2).Function 接口中的默认方法 andThen

(3).Function 接口的练习