函数式接口@FunctionalInterface

# 一、函数式接口定义

在java8中，满足下面任意一个条件的接口都是函数式接口：

1、被@FunctionalInterface注释的接口，满足@FunctionalInterface注释的约束。

2、没有被@FunctionalInterface注释的接口，但是满足@FunctionalInterface注释的约束

 @FunctionalInterface注释的约束：

1、接口有且只能有个一个抽象方法，只有方法定义，没有方法体

2、在接口中覆写Object类中的public方法，不算是函数式接口的方法。

java8针对原生类型int，long，double都提供了相应的函数式接口。如：DoubleConsumer， DoubleFunction，IntConsumer等等，使用方式都相同，见java.util.function包。

下面三个接口都是函数式接口：

1. *//接口一*
2. @FunctionalInterface
3. public interface FunctionInterfaceTest {
4. String getInfo(String input);
6. @Override
7. String toString(); *//Object中的方法*
9. @Override
10. boolean equals(Object obj); *//Object中的方法*
11. }
12. *//接口二*
13. @FunctionalInterface
14. public interface FunctionInterfaceTest {
15. String getInfo(String input);
16. }
18. *//接口三*
19. public interface FunctionInterfaceTest {
20. String getInfo(String input);
21. }

# 二、函数式接口创建

***@FunctionalInterface***

**public interface ProduceService {**

**String info(String param);**

**}**

## **2.1 lambda表达式**

**Function<String, String> function = item -> item + "\_A";**

**Consumer<String> consumer = item -> {**

**System.out.println(item);**

**};**

**Predicate<String> predicate = item -> item.length() > 2;**

**Supplier<String> supplier = () -> new String("无参,返回T");**

**System.out.println(supplier.get());**

**输出：无参,返回T**

## **2.2 方法引用**

**public static String getInstance(String item) {**

**return item + "世界！";**

**}**

**ProduceService produceInstance = ProduceMain::getInstance;**

**System.out.println(produceInstance.info("getInstance"));**

**输出：getInstance世界！**

## **2.3 构造方法引用**

**ProduceService produceNew = String::new;**

**System.out.println(produceNew.info("getInstance"));**

**输出：getInstance**

## **2.4 常用函数式接口**

|  |  |
| --- | --- |
| 接口名 | 说明 |
| Function<T,R> | 接收一个T类型的参数，返回R类型的结果 |
| Predicate<T> | 接收一个T类型的参数，返回Boolean类型的结果 |
| Consumer<T> | 接收一个T类型的参数，不返回值 |
| Supplier<T> | 不接受参数，返回T类型的结果 |

## **2.5 多参数运算**

在实际使用中，我们往往会输入多个参数，而不是一个参数。针对于多个参数的计算，最终都可以拆分两个参数的运算，然后将两个参数的运算结合起来。

如：1+2+3+4 = 10，可以拆分为1+2 = 3,   3+3=6； 6+4 = 10； 三个步骤完成；

|  |  |
| --- | --- |
| 接口名 | 说明 |
| BiFunction<T, U, R> | 接收T类型和U类型的两个参数，返回R类型的结果 |
| BiPredicate<T, U> | 接收T类型和U类型的两个参数，返回Boolean类型的结果 |
| BiConsumer<T , U> | 接收T类型和U类型的两个参数，不返回值 |

# 三、Function接口的andThen和compose

andThen和compose实际上是一种链式组合。

## **3.1 andThen方法**

andThen方法与compose方法很类似。不同之处在于，andThen是先执行自身的apply方法，将apply的返回值作为after接口的输入值。相对于compose方法，只是方向的不同。

**源码：**

default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {

Objects.requireNonNull(after);

return (T t) -> after.apply(apply(t));

}

## **3.2 compose方法**

Compose方法接收Function类型的参数，返回一个值。在执行compose方法中的apply方法之前，它先执行了before接口的apply方法(也是compose方法的输入参数)。然后将before方法执行的返回值作为compose中apply方法的输入参数。

**源码：**

 default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {

Objects.requireNonNull(before);

return (V v) -> apply(before.apply(v));

}

# 四、Consumer接口的andThen方法

将输入参数分别赋给andThen内部的accept方法和after内部的accept方法。

after的计算在andThen之后，起到了后置连接的作用。

在这里没有compose方法，因为后置连接反过来就是前置连接，所以不需要一个多余的compose方法了。只需要在传递时，交换两个consumer的顺序即可。

**源码：**

default Consumer<T> andThen(Consumer<? super T> after) {

Objects.requireNonNull(after);

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

}

**BiConsumer源码：**

default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {

Objects.requireNonNull(after);

return (l, r) -> {

accept(l, r);

after.accept(l, r);

};

}

# 五、BiFunction andThen方法

BiFunction只有andThen方法，这是有bi类型接口的特征决定的。 bi类型的接口需要接收两个参数，然而java中是没有返回两个参数的情况的, 所以只有andThen方法，且其参数是function类型的，接收的是一个参数，返回一个值。

biFunction.andThen(function).apply(t, u);