链滴

java【function】

作者：haxLook

原文链接：https://ld246.com/article/1619516660041

来源网站：链滴

函数式编程/lambda表达式

关于为什么需要学习函数式编程，什么是函数式编程，之前在学习javase的时候，学到关于lambda达式对一些知识，但是时间消磨了我，最近在做项目分析别人代码的时候，看到一大堆的function的法使用，使得我引起了重视。

函数式编程：面向结果，对过程不在乎，例如实现的细节之类的。

一般的我们自己代码（命令式编程）：面向过程编程。

代码演示

```
@Test
public void whyisfunction() {
    /**
     * 命令式编程
     */
    int[] nums = {12,10,13,14,78};
    //过程
    Integer max =Integer.MIN_VALUE;

    for (int i :nums) {

        if(i> max) {
            max = i;
        }
    }
    System.out.println(String.format("【命令式】%s", max));

    /**
     * 函数式编程
     */

    int asInt = IntStream.of(nums).max().getAsInt();

    System.out.println(String.format("【函数式】%s", asInt));
}
```

优势在于函数式编程可以快速是一个多线程，并行等，效率更高

我们以创建一个线程为实例的方式，进行演示

```
@Test
    public void runableTest() {
    /**
     * 命令式写法
     */
    new Thread(new Runnable() {

        @Override
        public void run() {
            System.out.println("【命令式】启动一个线程实例");
```

```
        }
    }).start();;

    /**
     * 配置lambda表示写法
     */

    new Thread(()->System.out.println("【函数式】启动一个线程实例")).start();
}
```

可能现在还不能很好的看出来，我们换一种方式写出来

```
@Test
public void runableTest2() {
    /**
     * 命令式写法
     */
    Runnable runnable = new Runnable() {

        @Override
        public void run() {
            System.out.println("【命令式】启动一个线程实例");

        }
    };

    new Thread(runnable).start();

    /**
     * 配置lambda表示写法
     */
    Runnable runnable2 = ()->System.out.println("【函数式】启动一个线程实例");
    Runnable runnable3 = ()->System.out.println("【函数式】启动一个线程实例");

    System.out.println(runnable2 == runnable3); //输出为false

    new Thread(runnable2).start();
}
```

从上面实例就可以看出来，对于lambda表示而言，他并不关注你实现的是哪个方法，它实现的就是个接口里面的一个不带参数的方法，返回的就是指定接口实例。说到这里，小伙伴应该会想到为什么前在Thread的时候，它可以使用lambda，因为Thread这个接口符合function函数规则【定义为接口并且只有一个抽象方法】，所以当我们使用（）->System.out.println("【函数式】启动一个线程实例),就能够找到返回的实例找到实现方法，如果有多个话，就可能出现不知道找谁了。

lambda的案例使用

```
interface IMoneyout{
  String format2(int i);
}
class IMyMoney {
    private  int money =99999;

    public IMyMoney(int money) {
        this.money = money;
```

```
    }

    public void  getresult(IMoneyout iMoneyout){

        System.out.println(iMoneyout.format2(money));
    }

}

public class function02 {
    @Test
    public void test01() {

        IMyMoney im=new IMyMoney(99999);
        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        im.getresult(hax->{
            arrayList.add(hax);
            return arrayList.toString();
        });

        im.getresult(hax->new DecimalFormat("#,###").format(hax));

    }
}
```

输出接口为

[99999],

99,999

使用function函数接口操作,Function<Integer, String>

```
class IMyMoney {

    private  int money =99999;

    public IMyMoney(int money) {
        this.money = money;
    }

    public void  getresult(Function<Integer, String> iMoneyout){

        System.out.println(iMoneyout.apply(money));
    }

}

public class function02 {

    @Test
    public void test01() {

        IMyMoney im=new IMyMoney(99999);
```

```
        ArrayList<String> arrayList = new ArrayList<String>();

        /**
         * 使用函数接口，可以进行链式操作
         */

        Function<Integer, String> iMoneyout = hax->new DecimalFormat("#,###").format(hax);

        im.getresult(iMoneyout.andThen(s->{
            arrayList.add("人民币"+s);
            return arrayList.toString();
        }));
    }

}
```

基本的函数接口，使用代码的形式为大家解释，代码中都存在注解。

```
public class function03 {

    class person{

        private String name;
        private char sex;
        private Integer age;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
        public char getSex() {
            return sex;
        }
        public void setSex(char sex) {
            this.sex = sex;
        }
        public Integer getAge() {
            return age;
        }
        public void setAge(Integer age) {
            this.age = age;
        }
        public person(String name, char sex, Integer age) {
            super();
            this.name = name;
            this.sex = sex;
            this.age = age;
        }
        public person(String name) {
            super();
            this.name = name;
        }
```

```
        public person() {
            super();
        }
        @Override
        public String toString() {
            return "person [name=" + name + ", sex=" + sex + ", age=" + age + "]";
        }
    }

    @Test
    public void test01() {

        //断言函数
        Predicate<Integer> predicate = i->i>=10;
        System.out.println(predicate.test(10));

        //消费者->消费一个数据
        Consumer<String> consumer =  i ->System.out.println(i);
        consumer.accept("消费者函数");

        //function函数->输入输出
        Function<Date, String> function = hax-> new SimpleDateFormat("yyyy-MM-ss:HH:mm:s
").format(hax);
        System.out.println(function.apply(new Date()));

        //一元函数->输入和输出是一样的类型
        UnaryOperator<function03.person> uOperator = hax -> {

            if(hax!=null && hax.getAge()<18 && "女".equals(hax.getSex()) ) {

                hax.setAge(18);
            }
            return hax;

        };

        System.out.println(uOperator.apply(new function03.person("ff",'女',19)).toString());

        /**
         * 二个输入，一个输出函数
         */
        function03.person person = new function03.person();

        BiFunction<function03.person, String, function03.person> biFunction = (hax,haxstring)->
new function03.person(haxstring);

        System.out.println(biFunction.apply(person, "yy"));;

        /**
         * 提供者
         */
        Supplier<String> supplier = () ->  new function03.person("supplier").toString();

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

```
        /**
         * 二元函数->参数2个,传入的参数相同，返回的参数也相同
         */

        BinaryOperator<String> binaryOperator = (hax1,hax2)->new StringBuilder(hax1).append
hax2).toString();

        System.out.println(binaryOperator.apply("二元", "函数"));

        /**
         * 对于基本类型来说，function提供了一些基本的类型实现，不需要在写泛型
         */
        //
        IntPredicate predicate2 = i ->i >=10;
        System.out.println(predicate2.test(10));

    //string类型的没有
        IntConsumer consumer2 = i ->System.out.println(i);
        consumer2.accept(10);

    }

}
```

返回结果为

true

消费者函数

2021-04-53:10:13:53

person [name=ff, sex=女, age=19]

person [name=yy, sex= , age=null]

person [name=supplier, sex=, age=null]

二元函数

true

10

总而言之使用函数式的编程方式，发现我们不在去关注你需要实现的是什么方法，关注点只在于，你要什么类型参数，返回值就可以，我们在lambda表达式中写入自己想实现的操作，可以用法等一系的操作进行实现，然后将结果拿到进行返回操作，方便弄好实现，并且我们可以通过自己自定义的函接口，对制定的一些方法进行标准化。