函数式编程

软件系统的整体设计说明

组织大型程序，会受到我们对于被模拟系统的认识的支配，一般有两种特点鲜明的组织策略，它们源于对系统结构的两种非常不同的世界观;

1. 第一种组织策略式将注意里集中在对象上，将一个大型系统看成一大批的对象，它们的行为可能随着时间的进展而不断变化，
2. 另一种组织策略将注意力集中在流过系统的信息流上，非常像电子工程师观察一个信号处理系统；

基于对象的途径和流的途径，都对程序设计提出了具有意义的语言要求，对于对象而言，我们必须关注计算对象可以怎样变化而又同时保持其标识，这将抛弃函数式编程的代换模型，转向更机械式的，理论上也更不容易把握的计算的环境模型；在处理对象、变化和标识时，各种困难的基本根源在于我们需要在这一计算模型中与时间搏斗；如果允许程序的并发执行的可能性，事情就会变的困难许多，流方式特别能够用于松懈在我们的模型中对时间的模拟与计算机求值过程中的各种事情发生的顺序，可以通过一种延时求值的技术做到这一点，将流编程之前，首先要介绍一下函数式编程；
代换模型和环境模型的区别

函数式编程简介

函数式编程作为一种编程范式，在科学领域，是一种编写计算机程序数据结构和元素的方式，它把计算过程当做是数学函数的求值，而避免更改状态和可变数据; 一般来讲就是数学上怎么用,你编程就可以这么编写;因为函数式编程没有赋值的负作用; 相同的输入只能得到相同的输出;并且函数可以用作变量的命名,可以提供给过程作为参数,可以作为返回结果,可以包含在数据接口中,";
当函数作为参数传递的时候,一般可以叫做回调函数
函数式编程语言里面没有for/next循环，因为这逻辑意味着有状态的改变相替代的是，这种循环逻辑在函数式编程语言里是通过递归、把函数当成参数传递的方式实现的;

Lambda表达式

Java Lambda表达式的一个重要用法是简化某些匿名内部类（Anonymous Classes）的写法。实际上Lambda表达式并不仅仅是匿名内部类的语法糖，JVM内部是通过invokedynamic指令来实现Lambda表达式的。 下面是匿名内部类和Lambda式的转化

函数式接口

能够使用lambda的依据是必须有相应的函数式接口(函数接口,是指内部只有一个抽象方法的接口),这一点跟java是强类型语言吻合,也就是你不能在任何地方任性的写lambda的写lambda表达式,实际上lambda的类型就是对应函数接口的类型; **这里可以了解一下类型签名的概念**;
为了使用的方便java8给我提供基本的函数式接口;其中有4类是最基本的;

1. Consumer: 消费型接口(void accept(T t)) --参数没有返回值

```
/**
 * 消费型接口Consumer<T>
 */
@Test
public void Test1() {
    cosum(500, (x) -> (system.out::println)); //函数作为参数
}
//函数的声明
public void cosum(Integer money, Consumer<Integer> c)  {
    c.accept(money);
}
```

2. Supplier 供给型接口（T get（））-- 没有输入，只有返回值

```
/**
 * 供给型接口
 */
 @Test
 public void test() {
     Random ran = new Random();
     List<Integer> list = supplier(10, () -> ran.nextInt(10);

     for (Integer i : list) {
         System.out.println(i);
     }
 }
 /**
  * 随机产生sum个数量得集合
  */
  public List<Integer> supplier(int sum, Supplier<Integer> sup) {
      List<Integer> list = new ArrayList<Integer>();
      for (int i = 0; i < sum; i++) {
          list.add(sup.get());
      }
      return list;
  }
```

3. Function<T,R>: 函数型接口(R apply (T t)) --输入一个类型得参数, 输出一个类型得参数

```
/**
 * 函数型接口: Function<R, T>
 */
 @Test
 public void test() {
     String s = strOperar("asdf", x-> x.substring(0,2));
     System.out.println(s);
     String s1 = strOperar("asdf", x -> x.trim());
     System.out.println(s1);
```

```
}

/**
 * 字符串操作
 */
public String strOperar(String str, Function<String, string> fun) {
    return fun.apply(str);
}
```

4. Predicate：断言型接口（boolean test（T t）） :也是我们常说的谓词表达式 输入一个参数，输出一个boolean类型得返回值

```
@Test
public void test() {
    List<Integer> list = new Arraylist<>();
    list.add(102);
    list.add(172);
    list.add(13);
    list.add(82);
    List<Integer> list = filterInt(list, (x) -> (x > 100));
    for (Integer i: list) {
        System.out.println(i);
    }
}

/**
 * 过滤集合
 */
public List<Integer> filterInt(List<Integer> list, Predicate<Integer> pre) {
    List<Integer> l = new ArrayList<>();
    for (Integer i: list) {
        if (pre.test(i)) {
            l.add(integer);
        }
    }
    return l;
}
```

lambda 遍历Collection集合

因为java8 为Iterable接口新增了一个ForEach(Consumer action)默认方法，该方法所需要得参数类型是一个函数式接口，而Iterable接口是Collection接口的父接口，因此Collection集合也可以直接调用该方法;
当程序调用的Iterable的`forEach(Consumer action)`遍历集合元素时，程序会依次将集合元素传给`Consumer`的`accept(T t)`方法，(该接口中唯一的抽象方法),正因为Consumer是函数式接口，因此可以用Lambda表达式来遍历元素

```
@Test
public void test() {
    Collection c = new HashSet();
    c.add("name1");
    c.add("name2");
    c.add("name3");
    c.forEach(item -> System.out :: println);

}
```

Lambda表达式遍历Iterator迭代器

java8为了Iterator引入了一个`forEachRemaining(Consumer action)` 默认方法，该方法所需的Consumer参数同样也是函数式接口，当程序调用Iterator的`forEachRemaining(Consumer action)`遍历集合元素时，程序会依次将集合元素传给`Consumer`的`accept(T t)`方法，（该接口中唯一的抽象方法)

```
    @Test
    public void test() {
        Collection c = new HashSet();
        c.add("name1");
        c.add("name2");
        c.add("name3");
        Iterator it = c.iterator();

        it.forEachRemaining(item -> System.out::println);
    }
```

Predicate操作Collection集合

Java 8 起为 Collection 集合新增了一个 removeIf(Predicate filter) 方法，该方法将会批量删除符合 filter 条件的所有元素。该方法需要一个 Predicate 对象作为参数，Predicate 也是函数式接口，因此可使用 Lambda 表达式作为参数。

```
    @Test
    public void test() {
        Collection c = new HashSet();
        c.add(new String("name1"));
        c.add(new String("name2"));
        c.add(new String("name3"));
        c.removeIf(ele -> ((String)ele).length() < 12);
    }
```

高阶函数

高阶函数（Higher-order Function）只是一个消费或产生函数的函数。

```
// functional/ProduceFunction.java

import java.util.function.*;

interface
FuncSS extends Function<String, String> {} // [1]

public class ProduceFunction {
  static FuncSS produce() {
    return s -> s.toLowerCase(); // [2]
  }
  public static void main(String[] args) {
    FuncSS f = produce();
    System.out.println(f.apply("YELLING"));
  }
}
```

这里的produce()就是高阶函数

闭包和回调函数

说到闭包,需要知道约束变量和自由变量,

1. 约束变量指的是函数内部的变量和用作函数参数的变量
2. 不是约束变量都可以称作自由变量 闭包指的是一个持有自由变量的函数和该自由变量所组成的环境就是闭包;

```
// functional/Closure1.java
// 这里函数makeFun调用了外部变量i就形成了闭包
import java.util.function.*;

public class Closure1 {
  int i;
  IntSupplier makeFun(int x) {
    return () -> x + i++;
  }
}
```

下面的编译会出错;从 Lambda 表达式引用的局部变量必须是 final 或者是等同 final 效果的。

```
// functional/Closure3.java

// {WillNotCompile}
import java.util.function.*;
```

```
public class Closure3 {
  IntSupplier makeFun(int x) {
    int i = 0;
    // x++ 和 i++ 都会报错:
    return () -> x++ + i++;
  }
}
```

函数组合

函数组合（Function Composition）意为"多个函数组合成新函数"。它通常是函数式编程的基本组成部分。 一些 java.util.function 接口中包含支持函数组合的方法;

组合方法	描述
andThen(argument)	根据参数执行原始操作
compose(argument)	根据参数执行原始操作
and(argument)	短路逻辑与原始谓词和参数谓词
or(argument)	短路逻辑或原始谓词和参数谓词
negate()	该谓词的逻辑否谓词

```
// functional/FunctionComposition.java

import java.util.function.*;

public class FunctionComposition {
  static Function<String, String>
    f1 = s -> {
      System.out.println(s);
      return s.replace('A', '_');
    },
    f2 = s -> s.substring(3),
    f3 = s -> s.toLowerCase(),
    f4 = f1.compose(f2).andThen(f3);
  public static void main(String[] args) {
    System.out.println(
      f4.apply("GO AFTER ALL AMBULANCES"));
  }
}

\\AFTER ALL AMBULANCES
\\_fter _ll _mbul_nces
```

下例是 Predicate 的逻辑运算演示.代码示例：

```
// functional/PredicateComposition.java

import java.util.function.*;
import java.util.stream.*;

public class PredicateComposition {
  static Predicate<String>
    p1 = s -> s.contains("bar"),
    p2 = s -> s.length() < 5,
    p3 = s -> s.contains("foo"),
    p4 = p1.negate().and(p2).or(p3);
  public static void main(String[] args) {
    Stream.of("bar", "foobar", "foobaz", "fongopuckey")
      .filter(p4)
      .forEach(System.out::println);
  }
}
\\foobar,foobaz
```

科里化和部分求值

科里化就是将一个多参数的函数，转换为一系列单参数函数

```
import java.util.function.*;

public class Curry3Args {
   public static void main(String[] args) {
      Function<String,
        Function<String,
          Function<String, String>>> sum =
            a -> b -> c -> a + b + c;
      Function<String,
        Function<String, String>> hi =
          sum.apply("Hi ");
      Function<String, String> ho =
        hi.apply("Ho ");
      System.out.println(ho.apply("Hup"));
   }
}
\\Hi Ho Hup
```