# 83-开源实战三（下）：借GoogleGuava学习三大编程范式中的函数式编程

现在主流的编程范式主要有三种，面向过程、面向对象和函数式编程。在理论部分，我们已经详细讲过前两种了。今天，我们再借机会讲讲剩下的一种，函数式编程。

函数式编程并非一个很新的东西，早在50多年前就已经出现了。近几年，函数式编程越来越被人关注，出现了很多新的函数式编程语言，比如Clojure、Scala、Erlang等。一些非函数式编程语言也加入了很多特性、语法、类库来支持函数式编程，比如Java、Python、Ruby、JavaScript等。除此之外，Google Guava也有对函数式编程的增强功能。

函数式编程因其编程的特殊性，仅在科学计算、数据处理、统计分析等领域，才能更好地发挥它的优势，所以，我个人觉得，它并不能完全替代更加通用的面向对象编程范式。但是，作为一种补充，它也有很大存在、发展和学习的意义。所以，我觉得有必要在专栏里带你一块学习一下。

话不多说，让我们正式开始今天的学习吧！

## 到底什么是函数式编程？

函数式编程的英文翻译是Functional Programming。 那到底什么是函数式编程呢？

在前面的章节中，我们讲到，面向过程、面向对象编程并没有严格的官方定义。在当时的讲解中，我也只是给出了我自己总结的定义。而且，当时给出的定义也只是对两个范式主要特性的总结，并不是很严格。实际上，函数式编程也是如此，也没有一个严格的官方定义。所以，接下来，我就从特性上来告诉你，什么是函数式编程。

严格上来讲，函数式编程中的“函数”，并不是指我们编程语言中的“函数”概念，而是指数学“函数”或者“表达式”（比如，y=f(x)）。不过，在编程实现的时候，对于数学“函数”或“表达式”，我们一般习惯性地将它们设计成函数。所以，如果不深究的话，函数式编程中的“函数”也可以理解为编程语言中的“函数”。

每个编程范式都有自己独特的地方，这就是它们会被抽象出来作为一种范式的原因。面向对象编程最大的特点是：以类、对象作为组织代码的单元以及它的四大特性。面向过程编程最大的特点是：以函数作为组织代码的单元，数据与方法相分离。那函数式编程最独特的地方又在哪里呢？

实际上，函数式编程最独特的地方在于它的编程思想。函数式编程认为，程序可以用一系列数学函数或表达式的组合来表示。函数式编程是程序面向数学的更底层的抽象，将计算过程描述为表达式。不过，这样说你肯定会有疑问，真的可以把任何程序都表示成一组数学表达式吗？

理论上讲是可以的。但是，并不是所有的程序都适合这么做。函数式编程有它自己适合的应用场景，比如开篇提到的科学计算、数据处理、统计分析等。在这些领域，程序往往比较容易用数学表达式来表示，比起非函数式编程，实现同样的功能，函数式编程可以用很少的代码就能搞定。但是，对于强业务相关的大型业务系统开发来说，费劲吧啦地将它抽象成数学表达式，硬要用函数式编程来实现，显然是自讨苦吃。相反，在这种应用场景下，面向对象编程更加合适，写出来的代码更加可读、可维护。

刚刚讲的是函数式编程的编程思想，如果我们再具体到编程实现，函数式编程跟面向过程编程一样，也是以函数作为组织代码的单元。不过，它跟面向过程编程的区别在于，它的函数是无状态的。何为无状态？简单点讲就是，函数内部涉及的变量都是局部变量，不会像面向对象编程那样，共享类成员变量，也不会像面向

过程编程那样，共享全局变量。函数的执行结果只与入参有关，跟其他任何外部变量无关。同样的入参，不管怎么执行，得到的结果都是一样的。这实际上就是数学函数或数学表达式的基本要求。我举个例子来简单解释一下。

```
// 有状态函数：执行结果依赖b的值是多少，即便入参相同，多次执行函数，函数的返回值有可能不同，因为b值有可能不同。
int b;
int increase(int a) {
  return a + b;
}

// 无状态函数：执行结果不依赖任何外部变量值，只要入参相同，不管执行多少次，函数的返回值就相同
int increase(int a, int b) {
  return a + b;
}
```

这里稍微总结一下，不同的编程范式之间并不是截然不同的，总是有一些相同的编程规则。比如，不管是面向过程、面向对象还是函数式编程，它们都有变量、函数的概念，最顶层都要有main函数执行入口，来组装编程单元（类、函数等）。只不过，面向对象的编程单元是类或对象，面向过程的编程单元是函数，函数式编程的编程单元是无状态函数。

## Java对函数式编程的支持

我们前面讲到，实现面向对象编程不一定非得使用面向对象编程语言，同理，实现函数式编程也不一定非得使用函数式编程语言。现在，很多面向对象编程语言，也提供了相应的语法、类库来支持函数式编程。

接下来，我们就看下Java这种面向对象编程语言，对函数式编程的支持，借机加深一下你对函数式编程的理解。我们先来看下面这样一段非常典型的Java函数式编程的代码。

```
public class FPDemo {
  public static void main(String[] args) {
    Optional<Integer> result = Stream.of("f", "ba", "hello")
            .map(s -> s.length())
            .filter(l -> l <= 3)
            .max((o1, o2) -> o1-o2);
    System.out.println(result.get()); // 输出2
  }
}
```

这段代码的作用是从一组字符串数组中，过滤出长度小于等于3的字符串，并且求得这其中的最大长度。

如果你不了解Java函数式编程的语法，看了上面的代码或许会有些懵，主要的原因是，Java为函数式编程引入了三个新的语法概念：Stream类、Lambda表达式和函数接口（Functional Inteface）。Stream类用来支持通过“.”级联多个函数操作的代码编写方式；引入Lambda表达式的作用是简化代码编写；函数接口的作用是让我们可以把函数包裹成函数接口，来实现把函数当做参数一样来使用（Java不像C一样支持函数指针，可以把函数直接当参数来使用）。

**首先，我们来看下Stream类。**

假设我们要计算这样一个表达式：(3-1)*2+5。如果按照普通的函数调用的方式写出来，就是下面这个样子：

```
add(multiply(subtract(3,1),2),5);
```

不过，这样编写代码看起来会比较难理解，我们换个更易读的写法，如下所示：

```
subtract(3,1).multiply(2).add(5);
```

我们知道，在Java中，“.”表示调用某个对象的方法。为了支持上面这种级联调用方式，我们让每个函数都返回一个通用的类型：Stream类对象。在Stream类上的操作有两种：中间操作和终止操作。中间操作返回的仍然是Stream类对象，而终止操作返回的是确定的值结果。

我们再来看之前的例子。我对代码做了注释解释，如下所示。其中，map、filter是中间操作，返回Stream类对象，可以继续级联其他操作；max是终止操作，返回的不是Stream类对象，无法再继续往下级联处理了。

```
public class FPDemo {
  public static void main(String[] args) {
    Optional<Integer> result = Stream.of("f", "ba", "hello") // of返回Stream<String>对象
            .map(s -> s.length()) // map返回Stream<Integer>对象
            .filter(l -> l <= 3) // filter返回Stream<Integer>对象
            .max((o1, o2) -> o1-o2); // max终止操作：返回Optional<Integer>
    System.out.println(result.get()); // 输出2
  }
}
```

**其次，我们再来看下Lambda表达式。**

我们前面讲到，Java引入Lambda表达式的主要作用是简化代码编写。实际上，我们也可以不用Lambda表达式来书写例子中的代码。我们拿其中的map函数来举例说明一下。

下面有三段代码，第一段代码展示了map函数的定义，实际上，map函数接收的参数是一个Function接口，也就是待会儿要讲到的函数接口。第二段代码展示了map函数的使用方式。第三段代码是针对第二段代码用Lambda表达式简化之后的写法。实际上，Lambda表达式在Java中只是一个语法糖而已，底层是基于函数接口来实现的，也就是第二段代码展示的写法。

```
// Stream中map函数的定义：
public interface Stream<T> extends BaseStream<T, Stream<T>> {
  <R> Stream<R> map(Function<? super T, ? extends R> mapper);
  //...省略其他函数...
}
```

```
// Stream中map的使用方法：
Stream.of("fo", "bar", "hello").map(new Function<String, Integer>() {
  @Override
  public Integer apply(String s) {
    return s.length();
  }
});

// 用Lambda表达式简化后的写法：
Stream.of("fo", "bar", "hello").map(s -> s.length());
```

Lambda表达式语法不是我们学习的重点。我这里只稍微介绍一下。如果感兴趣，你可以自行深入研究。

Lambda表达式包括三部分：输入、函数体、输出。表示出来的话就是下面这个样子：

```
(a, b) -> { 语句1;  语句2; ...; return 输出; } //a,b是输入参数
```

实际上，Lambda表达式的写法非常灵活。我们刚刚给出的是标准写法，还有很多简化写法。比如，如果输入参数只有一个，可以省略()，直接写成a->{…}；如果没有入参，可以直接将输入和箭头都省略掉，只保留函数体；如果函数体只有一个语句，那可以将{}省略掉；如果函数没有返回值，return语句就可以不用写了。

如果我们把之前例子中的Lambda表达式，全部替换为函数接口的实现方式，就是下面这样子的。代码是不是多了很多？

```
Optional<Integer> result = Stream.of("f", "ba", "hello")
        .map(s -> s.length())
        .filter(l -> l <= 3)
        .max((o1, o2) -> o1-o2);

// 还原为函数接口的实现方式
Optional<Integer> result2 = Stream.of("fo", "bar", "hello")
        .map(new Function<String, Integer>() {
          @Override
          public Integer apply(String s) {
            return s.length();
          }
        })
        .filter(new Predicate<Integer>() {
          @Override
          public boolean test(Integer l) {
            return l <= 3;
          }
        })
        .max(new Comparator<Integer>() {
          @Override
          public int compare(Integer o1, Integer o2) {
            return o1 - o2;
          }
        });
```

**最后，我们来看下函数接口。**

实际上，上面一段代码中的Function、Predicate、Comparator都是函数接口。我们知道，C语言支持函数指针，它可以把函数直接当变量来使用。但是，Java没有函数指针这样的语法。所以，它通过函数接口，将函数包裹在接口中，当作变量来使用。

实际上，函数接口就是接口。不过，它也有自己特别的地方，那就是要求只包含一个未实现的方法。因为只有这样，Lambda表达式才能明确知道匹配的是哪个接口。如果有两个未实现的方法，并且接口入参、返回值都一样，那Java在翻译Lambda表达式的时候，就不知道表达式对应哪个方法了。

我把Java提供的Function、Predicate这两个函数接口的源码，摘抄过来贴到了下面，你可以对照着它们，理解我刚刚对函数接口的讲解。

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);  // 只有这一个未实现的方法

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

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

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t); // 只有这一个未实现的方法

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

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

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

    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
```

```
}
```

以上讲的就是Java对函数式编程的语法支持，我想，最开始给到的那个函数式编程的例子，现在你应该能轻松看懂了吧？

## Guava对函数式编程的增强

如果你是Google Guava的设计者，对于Java函数式编程，Google Guava还能做些什么呢？

颠覆式创新是很难的。不过我们可以进行一些补充，一方面，可以增加Stream类上的操作（类似map、filter、max这样的终止操作和中间操作），另一方面，也可以增加更多的函数接口（类似Function、Predicate这样的函数接口）。实际上，我们还可以设计一些类似Stream类的新的支持级联操作的类。这样，使用Java配合Guava进行函数式编程会更加方便。

但是，跟我们预期的相反，Google Guava并没有提供太多函数式编程的支持，仅仅封装了几个遍历集合操作的接口，代码如下所示：

```
Iterables.transform(Iterable, Function);
Iterators.transform(Iterator, Function);
Collections.transfrom(Collection, Function);
Lists.transform(List, Function);
Maps.transformValues(Map, Function);
Multimaps.transformValues(Mltimap, Function);
...
Iterables.filter(Iterable, Predicate);
Iterators.filter(Iterator, Predicate);
Collections2.filter(Collection, Predicate);
...
```

从Google Guava的GitHub Wiki中，我们发现，Google对于函数式编程的使用还是很谨慎的，认为过度地使用函数式编程，会导致代码可读性变差，强调不要滥用。这跟我前面对函数式编程的观点是一致的。所以，在函数式编程方面，Google Guava并没有提供太多的支持。

之所以对遍历集合操作做了优化，主要是因为函数式编程一个重要的应用场景就是遍历集合。如果不使用函数式编程，我们只能for循环，一个一个的处理集合中的数据。使用函数式编程，可以大大简化遍历集合操作的代码编写，一行代码就能搞定，而且在可读性方面也没有太大损失。

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天，我们讲了一下三大编程范式中的最后一个，函数式编程。尽管越来越多的编程语言开始支持函数式编程，但我个人觉得，它只能是其他编程范式的补充，用在一些特殊的领域发挥它的特殊作用，没法完全替代面向对象、面向过程编程范式。

关于什么是函数式编程，实际上不是很好理解。函数式编程中的“函数”，并不是指我们编程语言中的“函

数”概念，而是数学中的“函数”或者“表达式”概念。函数式编程认为，程序可以用一系列数学函数或表达式的组合来表示。

具体到编程实现，函数式编程以无状态函数作为组织代码的单元。函数的执行结果只与入参有关，跟其他任何外部变量无关。同样的入参，不管怎么执行，得到的结果都是一样。

具体到Java语言，它提供了三个语法机制来支持函数式编程。它们分别是Stream类、Lambda表达式和函数接口。Google Guava对函数式编程的一个重要应用场景，遍历集合，做了优化，但并没有太多的支持，并且我们强调，不要为了节省代码行数，滥用函数式编程，导致代码可读性变差。

## 课堂讨论

你可以说一说函数式编程的优点和缺点，以及你对函数式编程的看法。你觉得它能否替代面向对象编程，成为最主流的编程范式?

欢迎留言和我分享你的想法，如果有收获，也欢迎你把这篇文章分享给你的朋友。

## 精选留言：

- javaadu 2020-05-13 00:38:38
  我对函数式编程的看法有几点
  1. 在集合操作方面非常强大，集合遍历、过滤、转换、分组等等，我现在在工作中经常用
  2. 函数式编程的语法对于设计模式来说是一种具体的实现方式，可能代码行数会比较少，但是思路是一样的，所以最重要的还是前面一直强调的设计原则
  3. 函数式编程最大的两个特点：函数是一等公民、函数没有副作用、强调对象的不变性，对于我们在面向对象编程时处理并发问题有指导意义 [10赞]

- 麦兜爸爸 2020-05-13 11:36:25
  争哥你好，冒昧打扰下，我在极客时间买了你的两门课，算法课马上就学完了，看到你给一个35岁的运维建议、规划方向，很有感触，我现在也很迷茫希望听听争哥的建议。我马上32岁了，研究生非计算机科班，毕业后一直从事iOS Native开发，刚开始工作的前几年觉得任何一个方向都能发展好，但现在越来越觉得iOS这个方向太鸡肋了，平常开发中接触不到稍微高深点的计算机知识，不像服务端离计算机本质更近点，我个人还是对计算机技术感兴趣的，想从事更有技术含量的工作，现在觉得有3条路可走，1、向大前端方向发展，这条路对我来说可能更好走，毕竟自己的经验就是客户端，但个人对前端UI、计算机图形图像不感兴趣；2、找机会内部转java服务端，这个方向我比较感兴趣，数据库、网络我之前都系统的学习过，spring boot也做过一些demo，但担心的是java从业人员太多，竞争太高，且内部也不知道有没有机会转；3、转机器学习，也是内部转，目前公司说后面可能有这方面的需求，算法是我的薄弱项，所以在抓紧时间学争哥的课程😁，这个方向肯定门槛比较高，但在我们这个二线城市几乎没这方面的工作岗位。所以实在纠结该怎么选，另外2年前拿到过阿里iOS P6的offer，目前在一家外企，非常想听听争哥的建议，打扰了 [9赞]

- 辣么大 2020-05-13 09:57:47
  视角不同：
  FP：数据围绕操作
  OOP：操作围绕数据 [4赞]

- Heaven 2020-05-13 11:41:21
  优点:降低代码编写,提高编写效率,更加抽象.如果编写的好,复用性也很不错(因为无状态)
  缺点:入门门槛不低,对于一些业务复杂的逻辑,有心而无力 [2赞]

- 小晏子 2020-05-13 09:38:04
  我觉得函数式编程并不能代替面向对象语言，并不是适合除了数学计算分析等大部分的场景，从系统设计的角度来讲，使用面向对象设计还是更亦理解的方式。
  函数式编程的优点：
  1. 代码量少，比如文中的例子就是最直接的展示。
  2. 因为都是“无状态函数”，固定输入产生固定输出，那么单元测试和调试都很简单
  3. 同样是因为无状态，所以适合并发编程，不用担心兵法安全问题。
  缺点：
  1. 滥用函数式编程会导致代码难以理解，比如一大型项目有大量高阶函数混着变量，开发人员随意把函数当作参数和返回值，项目会变得很难维护。
  2.函数式编程会导致大量递归，算法效率太低。
  [2赞]

- 迷羊 2020-05-13 09:29:10
  Java8的函数式编程太香了，点点点很爽。 [2赞]

- 三木子 2020-05-14 08:31:00
  最爽莫过于集合遍历。简单集合遍历 一行就可以搞定。太多for看这难受。 [1赞]

- 落尘kira 2020-05-13 20:07:56
  Java的函数式编程有一定的学习成本，而且由于强调不可变性，导致必须要求外部参数为final，这种情况下就老老实实的for循环；另外就是语法糖真香，相比原生的Stream，Flux更香 [1赞]

- bboy孙晨杰 2020-05-13 15:05:03
  复杂的业务逻辑我一般不会用函数式编程，可读性差，也不方便debug。。。发这条评论的主要目的是庆祝自己这几个月落下的进度终于补上了，哈哈 [1赞]

- Jxin 2020-05-13 13:25:04
  a.优缺点：
  优
  1.代码量少（可读性相对就高，开发成本相对就低）
  2.无状态，纯函数（幂等）。（可测试性就好，对并发编程友好，对迁移serverless友好）

  1.每个函数返回的都是一个新对象。（额外的资源成本）

  2.设计难度高。（设计一个恰到好处的领域对象难，设计一个符合“函数编程思想”的表达式也难）。难就意味着成本，意味着不好推广普及。

  缺
  3.相较于面向对象对业务流程的抽象。函数表达式更像是对业务流程做重定义。 相对更不易于理解。

  b.能取代面向对象吗？

  不能。与文中相驳的点是，我认为函数式编程可读性其实更好（代码量少），可维护性也更好（可测试性）。但是函数式编程的代码和具体的业务流程间的映射关系，更难理解。这会导致要设计出一个完美满足业务流程的代码会比较难，需要有更多的转换和考量。而面向对象在构建这种业务模型上，只是对原业务流程做抽象，相对更好理解。其传承能力，以及跨部门达成共识的能力都远优于函数式编程。

  我看好函数式编程，在无状态的计算领域，和一些高并发场景，它能发挥出很优益的价值。只是取代面向

对象这种就有点过了。目前来看各有其应用场景，按需选择是挺好的方式，不必执着于谁替换谁。毕竟从结果来看，就连流程式编程，也不是面向对象能完美替代的。各有应用场景，关键在权衡。 [1赞]

- 守拙 2020-05-13 10:00:49
函数编程在Android开发领域已经是家常便饭了.
无论是RxJava还是LiveData都应用了函数式编程思想.

在MVVM架构中, 应用函数编程可以做到层之间的解耦彻底, 链式调用很好的体现编程优雅性.

函数式编程缺点是学习成本较高. 从面向对象思想向面向函数思想的转变需要付出一定的学习精力.
如果团队开发水平参差不齐还是慎用, 可能导致你的同事无法维护你写的代码.

函数式编程另一优点是纯函数思想与不可变(Immutable)思想隔绝了恼人的局部变量, 全局变量等对流程的影响.

另最近我在codewar上刷题的时候, 发现函数编程相比传统面向对象对数据的处理确实更加简洁优雅, 相信经常刷题的同学会有相同的感受. [1赞]

- jaryoung 2020-05-13 09:24:49
所谓的面向过程，面向对象，函数式的编程范式，我们都是应该根据场景进行选择的。例如，如果是大量的异步编程个人觉得使用函数式编程范式相对比较合理。面向对象的话，对于一些业务非常复杂的系统来说更加合适，面向过程本人没有做过相应经验，就不胡扯了。 [1赞]

- mooneal 2020-05-13 08:22:11
函数式编程，相对于面向对象以及面向过程，最大的优点就是无状态了，就像数学表达式，给定输入一定有一个唯一的输出映射。所以，函数式编程又可以看作是对一类数据到另一类数据的映射。 [1赞]

- 小喵喵 2020-05-13 07:53:16
函数式编程是无状态的，它和接口的幂等性设计有什么区别呢？是不是接口的幂等性设计可以用函数式编程来实现呢？ [1赞]

- 墨雨 2020-05-15 08:57:26
做个笔记。
函数式编程在工作中基本很少用。
1.jdk 开发环境没用 1.8 。(硬伤)
2.使用后可读性较差。(同事读不懂，有学习成本)
后期考虑用它的点:
1.遍历集合，集合过滤及处理(保证可读性)
2.记得是stream对大批量数据处理性能会好些(好像是这样,不对请指正)

- ， 2020-05-13 11:27:34
函数式编程在做数据的筛选,过滤,转换,搜索,存放上,因为流式api的原因,可以提高可读性和可维护性
因为在处理时不会改变输入值,因此无副作用,无状态,在做并发编程时具备一定优势
同时由于自身特性,做并行处理时具备一定优势
不过以上均是建立有良好的函数式编程规范的情况下,假设一段代码里,有stream,有foreach迭代,有方法引用,有lambda表达式,函数还特别长,那么可读性,可维护性势必受到影响
假设一段代码里同时有迭代和stream,那么可能会有数据的改变,有状态有副作用,影响并发操作
假设一段stream代码里有limit,有使用Stream,iterate等除了arraylist,hashmap的非标准容器,使容器不能被很好的分割为多个任意大小的子范围,那么并行操作也会受到影响
同时相当多的业务操作,不能用stream来表示,这样看来函数式编程有一定优势,但也有自己的局限性