# 编程的宗派

总是有人喜欢争论这类问题，到底是“函数式编程”（FP）好，还是“面向对象编程”（OOP）好。既然出了两个帮派，就有人积极地做它们的帮众，互相唾骂和鄙视。然后呢又出了一个“好好先生帮”，这个帮的人喜欢说，管它什么范式呢，能解决问题的工具就是好工具！我个人其实不属于这三帮人中的任何一个。

## 面向对象编程（Object-Oriented Programming）

如果你看透了表面现象就会发现，其实“面向对象编程”本身没有引入很多新东西。所谓“面向对象语言”，就是经典的“过程式语言”（比如 Pascal），加上一点抽象能力。所谓“类”和“对象”，基本是过程式语言里面的记录（record，或者叫结构，structure），它本质其实是一个从名字到数据的“映射表”（map）。

你可以用名字从这个表里面提取相应的数据。比如 `point.x`，就是用名字 `x` 从记录 `point` 里面提取相应的数据。这比起数组来是一件更方便的事情，因为你不需要记住存放数据的下标。即使你插入了新的数据成员，仍然可以用原来的名字来访问已有的数据，而不用担心下标错位的问题。

所谓“对象思想”（区别于“面向对象”），实际上就是对这种数据访问方式的进一步抽象。一个经典的例子就是平面点的数据结构。如果你把一个点存储为：

```
struct Point {
  double x;
  double y;
}
```

那么你用 `point.x` 和 `point.y` 可以直接访问它的 X 和 Y 坐标。你也可以把它存储为“极坐标”方式：

```
struct Point {
  double r;
  double angle;
}
```

这样你可以用 `point.r` 和 `point.angle` 访问它的模和角度。现在问题来了，如果你的代码开头把 Point 定义为第一种 XY 的方式，使用 `point.x`, `point.y` 访问 X 和 Y 坐标，后来你又决定改变 Point 的存储方式，用极坐标，你却不想修改已有的含有 `point.x` 和 `point.y` 的代码，怎么办呢？

这就是“对象思想”的价值，它让你可以通过“间接”（indirection，或者叫做“抽象”）来 改变 `point.x`和`point.y` 的语义，从而让使用者的代码完全不用修改。虽然你的实际数据结构里面可能没有 `x` 和 `y` 这两个成员，但由于 `.x` 和 `.y` 可以被重新定义，所以你可以通过改变 `.x` 和 `.y` 的定义来“模拟”它们。在你使用 `point.x` 和 `point.y` 的时候，系统内部其实在运行两片代码（所谓 getter），它们的作用是从 `r` 和 `angle` 计算出 `x` 和 `y` 的值。这样你的代码就感觉 `x` 和 `y` 是实际存在的成员一样，而其实它们是被临时算出来的。

在 Python 之类的语言里面，你可以通过定义 “property” 来直接改变 `point.x` 和 `point.y` 的语义。在 Java 里稍微麻烦一些，你需要使用 `point.getX()` 和 `point.getY()` 这样的写法。然而它们最后的目的其实都是一样的——它们为数据访问提供了一层“间接”（抽象）。

这种抽象有时候是个好主意，它甚至可以跟量子力学的所谓“不可观测性”扯上关系。你觉得这个原子里面有 10 个电子？也许它们只是像 `point.x` 给你的幻觉一样，也许宇宙里根本就没有电子这种东西，也许你每次看到所谓的电子，它都是临时生成出来逗你玩的呢？然而，对象思想的价值也就到此为止了。你见过的所谓“面向对象思想”，几乎无一例外可以从这个想法推广出来。面向对象语言的绝大部分特性，其实是过程式语言早就提供的。因此我觉得，其实没有语言可以叫做“面向对象语言”。就像一个人为一个公司贡献了一点点代码，并不足以让公司以他的名字命名一样。

“对象思想”作为数据访问的方式，是有一定好处的。然而“面向对象”（多了“面向”两个字），就是把这种本来良好的思想东拉西扯，牵强附会，发挥过了头。很多面向对象语言号称“所有东西都是对象”（Everything is an Object），把所有函数都放进所谓对象里面，叫做“方法”（method），把普通的函数叫做“静态方法”（static method）。实际上呢，就像我之前的例子，只有极少需要抽象的时候，你需要使用内嵌于对象之内，跟数据紧密结合的“方法”。其他的时候，你其实只是想表达数据之间的变换操作，这些完全可以用普通的函数表达，而且这样做更加简单和直接。

这种把所有函数放进方法的做法是本末倒置的，因为函数并不属于对象。绝大部分函数是独立于对象的，它们不能被叫做“方法”。强制把所有函数放进它们本来不属于的对象里面，把它们全都作为“方法”，导致了面向对象代码逻辑过度复杂。很简单的想法，非得绕好多道弯子才能表达清楚。很多时候这就像把自己的头塞进屁股里面。

这就是为什么我喜欢开玩笑说，面向对象编程就像“地平说”（Flat Earth Theory）。当然你可以说地球是一个平面。对于局部的，小规模的现象，它没有问题。然而对于通用的，大规模的情况，它却不是自然，简单和直接的。直到今天，你仍然可以无止境的寻找证据，扭曲各种物理定律，自圆其说地平说的幻觉，然而这会让你的理论非常复杂，经常需要缝缝补补还难以理解。

面向对象语言不仅有自身的根本性错误，而且由于面向对象语言的设计者们常常是半路出家，没有受到过严格的语言理论和设计训练却又自命不凡，所以经常搞出另外一些奇葩的东西。比如在 JavaScript 里面，每个函数同时又可以作为构造函数（constructor），所以每个函数里面都隐含了一个 this 变量，你嵌套多层对象和函数的时候就发现没法访问外层的 this，非得“bind”一下。Python 的变量定义和赋值不分，所以你需要访问全局变量的时候得用 global 关键字，后来又发现如果要访问“中间层”的变量，没有办法了，所以又加了个 nonlocal 关键字。Ruby 先后出现过四种类似 lambda 的东西，每个都有自己的怪癖…… 有些人问我为什么有些语言设计成那个样子，我只能说，很多语言设计者其实根本不知道自己在干什么。

软件领域就是喜欢制造宗派。“面向对象”当年就是乘火打劫，扯着各种幌子，成为了一种宗派，给很多人洗了脑。到底什么样的语言才算是“面向对象语言”？这样基本的问题至今没有确切的答案，足以说明所谓面向对象，基本都是扯淡。每当你指出某个 OO 语言 X 的弊端，就会有人跟你说，其实 X 不是“地道的” OO 语言，你应该去看看另外一个 OO 语言 Y。等你发现 Y 也有差不多的问题，有人又会让你去看 Z…… 直到最后，他们告诉你，只有 Smalltalk 才是地道的 O O语言。这不是很搞笑吗，说一个根本没人用的语言才是地道的 OO 语言，这就像在说只有死人的话才是对的。这就像是一群政客在踢皮球，推卸责任。

等你真正看看 Smalltalk 才发现，其实面向对象语言的根本毛病就是由它而来的， Smalltalk 并不是很好的语言。很多人至今不知道自己所用的“面向对象语言”里面的很多优点，都是从过程式语言继承来的。每当发生函数式与面向对象式语言的口水战，都会有面向对象的帮众拿出这些过程式语言早就有的优点来进行反驳：“你说面向对象不好，看它能做这个……” 拿别人的优点撑起自己的门面，却看不到事物实质的优点，这样的辩论纯粹是鸡同鸭讲。

## 函数式编程（Functional Programming）

函数式语言一直以来比较低调，直到最近由于并发计算编程瓶颈的出现，以及 Haskell，Scala 之类语言社区的大力鼓吹，它忽然变成了一种宗派。有人盲目的相信函数式编程能够奇迹般的解决并发计算的难题，而看不到实质存在的，独立于语言的问题。被函数式语言洗脑的帮众，喜欢否定其它语言的一切，看低其它程序员。特别是有些初学编程的人，俨然把函数式编程当成了一天瘦二十斤的减肥神药，以为自己从函数式语言入手，就可以对经验超过他十年以上的老程序员说三道四，仿佛别人不用函数式语言就什么都不懂一样。

#### 函数式编程的优点

函数式编程当然提供了它自己的价值。函数式编程相对于面向对象最大的价值，莫过于对于函数的正确理解。在函数式语言里面，函数是“一类公民”（first-class）。它们可以像 1, 2, “hello”，true，对象…… 之类的“值”一样，在任意位置诞生，通过变量，参数和数据结构传递到其它地方，可以在任何位置被调用。这些是很多过程式语言和面向对象语言做不到的事情。很多所谓“面向对象设计模式”（design pattern），都是因为面向对象语言没有 first-class function，所以导致了每个函数必须被包在一个对象里面才能传递到其它地方。

函数式编程的另一个贡献，是它们的类型系统。函数式语言对于类型的思维，往往非常的严密。函数式语言的类型系统，往往比面向对象语言来得严密和简单很多，它们可以帮助你对程序进行严密的逻辑推理。然而类型系统一是把双刃剑，如果你对它看得太重，它反而会带来不必要的复杂性和过度工程。这个我在下面讲讲。

### 各种“白象”（white elephant）

所谓白象，“white elephant”，是指被人奉为神圣，价格昂贵，却没有实际用处的东西。函数式语言里面有很好的东西，然而它们里面有很多多余的特性，这些特性跟白象的性质类似。

函数式语言的“拥护者”们，往往认为这个世界本来应该是“纯”（pure）的，不应该有任何“副作用”。他们把一切的“赋值操作”看成低级弱智的作法。他们很在乎所谓尾递归，类型推导，fold，currying，maybe type 等等。他们以自己能写出使用这些特性的代码为豪。可是殊不知，那些东西其实除了能自我安慰，制造高人一等的幻觉，并不一定能带来真正优秀可靠的代码。

#### 纯函数

半壶水都喜欢响叮当。很多喜欢自吹为“函数式程序员”的人，往往并不真的理解函数式语言的本质。他们一旦看到过程式语言的写法就嗤之以鼻。比如以下这个 C 函数：

```
int f(int x) {
    int y = 0;
    int z = 0;
    y = 2 * x;
    z = y + 1;
    return z / 3;
}
```

很多函数式程序员可能看到那几个赋值操作就皱起眉头，然而他们看不到的是，这是一个真正意义上的“纯函数”，它在本质上跟 Haskell 之类语言的函数是一样的，也许还更加优雅一些。

盲目鄙视赋值操作的人，也不理解“数据流”的概念。其实不管是对局部变量赋值还是把它们作为参数传递，其实本质

上都像是把一个东西放进一个管道，或者把一个电信号放在一根导线上，只不过这个管道或者导线，在不同的语言范式里放置的方向和样式有一点不同而已！

#### 对数据结构的忽视

函数式语言的帮众没有看清楚的另一个重要的，致命的东西，是数据结构的根本性和重要性。数据结构的有些问题是“物理”和“本质”地存在的，不是换个语言或者换个风格就可以奇迹般消失掉的。函数式语言的拥护者们喜欢盲目的相信和使用列表（list），而没有看清楚它的本质以及它所带来的时间复杂度。列表带来的问题，不仅仅是编程的复杂性。不管你怎么聪明的使用它，很多性能问题是根本没法解决的，因为列表的拓扑结构根本就不适合用来干有些事情！

从数据结构的角度看，Lisp 所谓的list就是一个单向链表。你必须从上一个节点才能访问下一个，而这每一次“间接寻址”，都是需要时间的。在这种数据结构下，很简单的像 length 或者 append 之类函数，时间复杂度都是 O(n)！为了绕过这数据结构的不足，所谓的“Lisp风格”告诉你，不要反复 append，因为那样复杂度是 $O(n^2)$。如果需要反复把元素加到列表末尾，那么应该先反复 cons，然后再 reverse一下。

很可惜的是，当你同时有递归调用，就会发现 cons + reverse 的做法颠来倒去的，非常容易出错。有时候列表是正的，有时候是反的，有时候一部分是反的…… 这种方式用一次还可以，多几层递归之后，自己都把自己搞糊涂了。好不容易做对了，下次修改可能又会出错。然而就是有人喜欢显示自己聪明，喜欢自虐，迎着这类人为制造的“困难”勇往直前。

富有讽刺意味的是，半壶水的 Lisp 程序员都喜欢用 list，真正的 Lisp 大师级人物，却知道什么时候应该使用记录（结构）或者数组。在 Indiana 大学，我曾经上过一门 Scheme（一种现代 Lisp 方言）编译器的课程，授课的老师是 R. Kent Dybvig，他是世界上最先进的 Scheme 编译器 Chez Scheme 的作者。我们的课程编译器的数据结构（包括 AST）都是用 list 表示的。到了期末的时候，Kent 对我们说：“你们的编译器已经可以生成跟我的 Chez Scheme 媲美的代码，然而 Chez Scheme 不止生成高效的目标代码，它的编译速度是你们的 700 倍以上。它可以在 5 秒钟之内编译它自己。” 然后他透露了一点 Chez Scheme 速度快的原因。其中一个原因，就是因为 Chez Scheme 的内部数据结构不是 list。在编译一开头的时候，Chez Scheme 就已经把输入代码转换成了数组一样的，固定长度的结构。后来在工业界的经验教训也告诉了我，数组比起链表，确实在某些时候有大幅度的性能提升。在什么时候该用链表，什么时候该用数组，是一门艺术。

#### 副作用的根本价值

对数据结构的忽视，跟纯函数式语言盲目排斥副作用的“教义”有很大关系。过度的使用副作用当然是有害的，然而副作用这种东西，其实是根本的，有用的。对于这一点，我喜欢跟人这样讲：在计算机和电子线路最开头发明的时候，所有的线路都是“纯”的，因为逻辑门和导线没有任何记忆数据的能力。后来有人发明了触发器（flip-flop），才有了所谓“副作用”。是副作用让我们可以存储中间数据，从而不需要把所有数据都通过不同的导线传输到需要的地方。没有副作用的语言，就像一个没有无线电，没有光的世界，所有的数据都必须通过实在的导线传递，这许多纷繁的电缆，必须被正确的连接和组织，才能达到需要的效果。我们为什么喜欢 WiFi，4G 网，Bluetooth，这也就是为什么一个语言不应该是“纯”的。

副作用也是某些重要的数据结构的重要组成元素。其中一个例子是哈希表。纯函数语言的拥护者喜欢盲目的排斥哈希表的价值，说自己可以用纯的树结构来达到一样的效果。然而事实却是，这些纯的数据结构是不可能达到有副作用的数据结构的性能的。所谓纯函数数据结构，因为在每一次“修改”时都需要保留旧的结构，所以往往需要大量的拷贝数据，然后依赖垃圾回收（GC）去消灭这些旧的数据。要知道，内存的分配和释放都是需要时间和能量的。盲目的依赖 GC，导致了纯函数数据结构内存分配和释放过于频繁，无法达到有副作用数据结构的性能。要知道，副作用是电子线路和物理支持的高级功能。盲目的相信和使用纯函数写法，其实是在浪费已有的物理支持的操作。

#### fold以及其他

大量使用 fold 和 currying 的代码，写起来貌似很酷，读起来却不必要的痛苦。很多人根本不明白 fold 的本质，却老喜欢用它，因为他们觉得那是函数式编程的“精华”，可以显示自己的聪明。然而他们没有看到的是，fold 包含的精髓，只不过是在列表（list）上做递归的“通用模板”，这个模板需要你填进去三个参数，就可以生成一个新的递归函数调用。所以每一个 fold 的调用，本质上都包含了一个在列表上的递归函数定义。

Fold 的问题在于，它定义了一个递归函数，却没有给它一个一目了然的名字。使用 fold 的结果是，每次看到一个 fold 调用，你都需要重新读懂它的定义，琢磨它到底是干什么的。而且 fold 调用只显示了递归模板需要的部分，而把递归的主体隐藏在了 fold 本身的“框架”里。比起直接写出整个递归定义，这种遮遮掩掩的做法，其实是更难理解的。比如，当你看到这句 Haskell 代码：

```
foldr (+) 0 [1,2,3]
```

你知道它是做什么的吗？也许你一秒钟之后就凭经验琢磨出，它是在对 `[1,2,3]` 里的数字进行求和，本质上相当于 `sum [1,2,3]`。虽然只花了一秒钟，可你仍然需要琢磨。如果 fold 里面带有更复杂的函数，而不是 `+`，那么你可能一分钟都琢磨不透。写起来倒没有费很大力气，可为什么我每次读这段代码，都需要看到 `+` 和 `0` 这两个跟自己的意图毫无关系的东西？万一有人不小心写错了，那里其实不是 `+` 和 `0` 怎么办？为什么我需要搞清楚 `+`, `0`, `[1,2,3]` 的相对位置以及它们的含义？

这样的写法其实还不如老老实实写一个递归函数，给它一个有意义名字（比如 `sum`），这样以后看到这个名字被调用，比如 `sum [1,2,3]`，你想都不用想就知道它要干什么。定义 `sum` 这样的名字虽然稍微增加了写代码时的工作，却给读代码的时候带来了方便。为了写的时候简洁或者很酷而用 fold，其实增加了读代码时的脑力开销。要知道代码被读的次数，要比被写的次数多很多，所以使用 fold 往往是得不偿失的。然而，被函数式编程洗脑的人，却看不到这一点。他们太在乎显示给别人看，我也会用 fold！

与 fold 类似的白象，还有 currying，Hindley-Milner 类型推导等特性。看似很酷，但等你仔细推敲才发现，它们带来的麻烦，比它们解决的问题其实还要多。有些特性声称解决的问题，其实根本就不存在。现在我把一些函数式语言的特性，以及它们包含的陷阱简要列举一下：

1. fold。fold 等“递归模板”，相当于把递归函数定义插入到调用的地方，而不给它们名字。这样导致每次读代码都需要理解几乎整个递归函数的定义。

2. currying。貌似很酷，可是被部分调用的参数只能从左到右，依次进行。如何安排参数的顺序成了问题。大部分时候还不如直接制造一个新的 lambda，在内部调用旧的函数，这样可以任意的安排参数顺序。

3. Hindley-Milner 类型推导（HM）。为了避免写参数和返回值的类型，结果给程序员写代码增加了很多的限制。为了让类型推导引擎开心，导致了很多完全合法合理优雅的代码无法写出来。其实还不如直接要程序员写出参数和返回值的类型，这工作量真的不多，而且可以准确的帮助阅读者理解参数的范围。HM 类型推导的根本问题其实在于它使用 unification 算法。Unification 其实只能表示数学里的“等价关系”（equivalence relation），而程序语言最重要的关系，subtyping，并不是一个等价关系，因为它不具有对称性（symmetry）。

4. 代数数据类型（algebraic data type）。所谓“代数数据类型”，其实并不如普通的类型系统（比如 Java 的）通用。很多代数数据类型系统具有所谓 sum type，这种类型其实带来过多的类型嵌套，不如通用的 union type。盲目崇拜代数数据类型的人，往往是因为盲目的相信“数学是优美的语言”。而其实事实是，数学是一种历史遗留的，毛病很多的语言。数学的语言根本没有经过系统的，全球协作的设计。往往是数学家在黑板上随便写个符号，说这个表示某概念，然后就定下来了。

5. Tuple。有代数数据类型的的语言里面经常有一种构造叫做 tuple，比如 Haskel l里面可以写 `(1, "hello")`，表示一个类型为 `(Int, String)` 的结构。这种构造经常被人看得过于高尚，以至于用在超越它能力的地方。其实 tuple 的本质就是一个没有名字的结构（类似 C 的 structure）。临时使用 tuple 貌似很方便，因为不需要定义一个结构类型。然而因为 tuple 没有名字，里面的成员没法用名字访问，一旦多加几个成员就发现很麻烦了。Tuple 往往只能通过模式匹配来获得里面的域，一旦你增加了新的域进去，所有对这个 tuple 的模式匹配代码都需要修改。所以 tuple 一般只能用在大小不超过 2 的情况下，而且必须确信以后不会增加新的域进去。

6. 惰性求值（lazy evaluation）。貌似数学上很优雅，但其实有严重的逻辑漏洞。因为 bottom（死循环）成为了任何类型的一个元素，所以取每一个值，都可能导致死循环。同时导致代码性能难以预测，因为求值太懒，所以可能临时抱佛脚做太多工作，而平时浪费 CPU 的时间。由于到需要的时候才求值，所以在有多个处理器的时候无法有效地利用它们的计算能力。

7. 尾递归。大部分尾递归都相当于循环语句，然而却不像循环语句一样，能够一目了然看明白它们的意图。你需要仔细看每个分支的返回位置，判断它是不是递归，然后才能判断这代码其实是个循环。而循环语句从关键字（for，while）就知道这是一个循环，不需要去看里面是什么结构。所以等价于循环的尾递归，最好还是写成专门的循环语句。当然，尾递归在另一些情况下是有用的，这些情况不等价于循环，而是“树递归”。在这种情况下使用循环，经常需要复杂的 break 或者 continue 条件，导致循环不易理解。所以循环和尾递归都是有必要的，不要总想着用一个代替另外一个，要分情况选择合适的方式。

## 好好先生

很多人避免“函数式 vs 面向对象”的辩论，于是他们成为了“好好先生”。这种人没有原则的认为，任何能够解决当前问题的工具就是好工具。也就是这种人，喜欢使用 shell script，喜欢折腾各种 Unix 工具，因为显然，它们能解决他“手头的问题”。

然而这种思潮是有害的，它的害处其实更胜于投靠函数式或者面向对象。没有原则的好好先生们忙着“解决问题”，却不能清晰地看到这些问题为什么存在。他们所谓的问题，往往是由于现有工具的设计失误。由于他们的“随和”，他们从来不去思考，如何从根源上消灭这些问题。他们在一堆历史遗留的垃圾上缝缝补补，妄图使用设计恶劣的工具建造可靠地软件系统。当然，这代价是非常大的。不但劳神费力，而且也许根本不能解决问题。

所以每当有人让我谈谈“函数式 vs 面向对象”，我都避免说“各有各的好处”，因为那样的话我会很容易被当成这种毫无原则的好好先生。

## 符号必须简单的对世界建模

从上面你已经看出，我既不是一个铁杆“函数式程序员”，也不是一个铁杆“面向对象程序员”，我也不是一个爱说“各有各的好处”的好好先生。我是一个有原则的批判性思维者。我不但看透了各种语言的本质，而且看透了它们之间的统一关系。我编程的时候看到的不是表面的语言和程序，而是一个类似电路的东西。我看到数据的流动和交换，我看到效率的瓶颈，而这些都是跟具体的语言和范式无关的。

在我的心目中其实只有一个概念，它叫做“编程”（programming），它不带有任何附加的限定词（比如“函数式”或者“面向对象”）。我研究的领域称叫做“Programming Languages”，它研究的内容不局限于某一个语言，也不局限于某一类语言，而是所有的语言。在我的眼里，所有的语言都不过是各个特性的组合。所以最近出现的所谓“新语言”，其实不大可能再有什么真正意义上的创新。我不喜欢说“发明一个程序语言”，不喜欢使用“发明”这个词，因为不管你怎么设计一个语言，所有的特性几乎都早已存在于现有的语言里面了。我更喜欢使用“设计”这个词，因为虽然一个语言没有任何新的特性，它却有可能在细节上更加优雅。

编程最重要的事情，其实是让写出来的符号，能够简单地对实际或者想象出来的“世界”进行建模。一个程序员最重要的能力，是直觉地看见符号和现实物体之间的对应关系。不管看起来多么酷的语言或者范式，如果必须绕着弯子才能表达程序员心目中的模型，那么它就不是一个很好的语言或者范式。有些东西本来就是有随时间变化的“状态”的，如果你偏要用“纯函数式”语言去描述它，当然你就进入了那些 monad 之类的死胡同。最后你不但没能高效的表达这种副作用，而且让代码变得比过程式语言还要难以理解。如果你进入另一个极端，一定要用对象来表达本来很纯的数学函数，那么你一样会把简单的问题搞复杂。Java 的所谓 design pattern，很多就是制造这种问题的，而没有解决任何问题。

关于建模的另外一个问题是，你心里想的模型，并不一定是最好的，也不一定非得设计成那个样子。有些人心里没有一个清晰简单的模型，觉得某些语言“好用”，就因为它们能够对他那种扭曲纷繁的模型进行建模。所以你就跟这种人说不清楚，为什么这个语言不好，因为显然这个语言对他是有用的！如何简化模型，已经超越了语言的范畴，在这里我就不细讲了。