思考函数式编程

函数式语言的编程是近来的热点，本期因此选登了两篇这方面的文章，可以促使我们对这方面的技术有一理解。

■ 文 / 蔡学镛

FP的历史典故

什么是面向对象编程（OOP）？只要你写过几年程序，如果你没太混的话，一定说得出封装、继承、多态这三个术语。什么是函数式编程（Functional Programming，FP）？即使你写了很多年的程序，应该也是答不出来，大家对它相当陌生，很少有人能正确地叙述出函数式编程是什么，有什么好处。某些人或许只能空泛地回答："我听说函数式编程…"，但口气却仍不甚坚定。

函数式编程长期以来没有出现在主流的商业软件世界。欠缺主流语言的支持，函数式编程只能偏安一隅，躲在学术界。而学术界，喔！这你也是知道的，总是有办法把简单的东西，讲解得相当精确（而抽象），让人敬畏（而摸不着边际）。欠缺书籍文章浅显且正确的介绍，这就是为何函数式编程会依然躲在象牙塔中的原因。

一直以来，真正让FP无法被接受的原因可能是"执行效率"。传统上，函数式编程语言的效率确实比命令式（imperative）编程语言来得差，这在商业系统上是不能忍受的。命令式语言让我们用贴近冯纽曼架构（van Neumann Architecture）机器的方式写程序，比较低阶，所以效率会比较高。而函数式编程语言却是使用较高阶的数学抽象，所以效率比较低。

但是这个原因却有了变化。过去这十多年，连Java这种龟速语言都能席卷世界，显然我们对于"跨平台"和"反微软"的重视已经超越"执行效率"；而在Java流行十年后的今天，我们有了新的衡量标准："简单""快速开发"比其它因素都更重要，因为现在软件的复杂度已经到了我们无法忍受的地步了，而IT产业的竞争也比以往激烈许多。

想要"简单"、"快速开发"，就要用比较高阶的抽象，因此函数式编程比命令式编程更适合现在的开发环境。这些年来硬件的进步，让我们对于函数式编程的效率不再是大问题；甚至由于编译技术的进步，函数式编程语言的执行速度，现在也已经不再是吴下阿蒙。

> 以往的纯函数式语言会被某些人认为"不食人间烟火"，而不纯的函数式语言，则被认为比较实际、实用，但是最近大家的看法似乎有了改变。

妙的是，还不只这样，局势似乎180度反转成为对FP有利的局面：多CPU、多核心、超线程（HT）的硬件架构普及，以及分布式运算的流行，这根本就是专为滋养FP繁殖而打造的环境。我不是算命仙，不敢铁口直断FP会从明天或明年就开始大流行，但是我确实注意到趋势呈现出对FP有利的局面，所以我们必须开始注意FP。

但究竟什么是FP？留待下一节的文章继续说明，本节文章先来一则FP的故事。

如果你修过大学一年级的"微积分"（calculus），你会发现数学是一种谜题：我们先定义好一些基本且兼容（不冲突）的原理（principle），再制订好一些用来操作这些原理的规则（rule）。规则可以彼此互相套用，产生更复杂的规则。数学家称这种方法为"正规系统"（formal system）或者"算术"（calculus）。数学家认为，宇宙的一切现象都可以用数学来描述。

1930年代，普林斯敦（Princeton）大学的四个人（大师们登场，同学请立正站好）Alonzo Church、Alan Turing、John von Neumann、Kurt Gödel都对formal system做研究，他们对于实体世界兴趣不大，他们探讨的问题都是抽象的数学证明题。

他们虽然各自做研究，但他们的问题确有共同点：想要回答关于计算（computation）的问题。如果我们具有一部机器，它的运算威力无限，那么我们能用它来解决哪些问题？这些问题能否自动解决？哪些问题不能解决？为什么不能解决？如果两部机器具有不同的设计，可否具有相同的运算威力？……我一直认为数学家是科学化的哲学家，这不是没有原因的。

Alonzo Church开发出一套formal

system，名为lambda calculus。这个系统本质上是一个编程语言，为一部“想象中的机器”所设计的语言。lambda calculus的函数可以接受函数当作输入（自变量）和输出（传出值）。这样的函数用希腊字母λ当作识别，所以这个语言才名为lambda calculus。利用lambda calculus，Alonzo能够回答上述的许多问题，提出最终的答案。

Alan Turing也在做类似的研究，开发出不同的系统，就是大名鼎鼎的Turing machine，他得到的结论和Alonzo Church类似。后来证实Turing machine和lambda calculus的威力一样强大。

在信息科学的研究领域，资金充裕的美国军方一直都是很重要的推手之一。二次世界大战时，IBM为了帮美国军方解决弹道计算的问题，建立了知名的Mark I计算机。1949年EDVAC计算机诞生，这是第一部采用von Neumann架构的计算机，也是Turning Machine的真实版本。Turing machine领先lambda calculus做出实体机器。

1958年，对lambda calculus相当感兴趣的MIT教授John McCarthy（毕业于普林斯敦，他是人工智能的先驱）设计出Lisp语言，Lisp实践了lambda calculus，让lambda calculus可以在von Neumann计算机上执行！大家开始注意到Lisp的威力。1973年，MIT的人工智能实验室开发出所谓的Lisp machine硬件，等于是将lambda calculus的机器实践出来了！

FP的优势

只要遵守FP的原则，管他用什么语言，都可以进行FP。你可以用非函数式的语言（例如Java）进行FP；正如同你可以用非面向对象的语言（例如C）进行OOP一样。但是只有想不开的人才会这么做，毕竟事倍功半。

LISP是第一个函数式语言，越来越多函数式语言随之出现。真实世界的函数式语言无法像Lambda Calculus那样，毕竟Lambda Calculus是让虚幻不存在的机器执行的，没有受到真实世界的限制。所以函数式语言虽然都是源自于Lambda Calculus，但是却都和Lambda Calculus之间存在差异。由于FP只是一些构想，各种语言实践这些构想的作法，彼此之间也可能有不小的差异。

尽管各种语言有差异，但是大致上来说，FP的共同点在于：“没有副作用”（Side Effect）、“第一级函数”（First-Class Function）。“没有副作用”是指在表达式（expression）内不可以造成值的改变；“第一级函数”是指函数被当作一般值对待，而不是次级公民，也就是说，函数可当作“传入参数”或“传出结果”。

基本上，遵守上述两点进行程序编写，差不多就可以称为FP。而且这两点和OOP是没有冲突的，所以同时采用OOP和FP的编程风格，是有可能的。尽管FP重度爱好者似乎都对OOP没有特别的好感，甚至会对OOP提出批评。究竟FP和OOP之间是互补还是竞争，究竟采用FP之后，还有必要使用OOP吗？这是值得探讨的话题。

FP和我们惯用的编写程序风格，有相当大的差异。Imperative Programming认为程序的执行，就是一连串状态的改变；但FP将程序的运作，视为数学函数的计算，且避免“状态”和“可变数据”。但是，没有状态，没有可变数据，程序要如何运作呢？事实上，FP使用函数，而函数可以“自动”帮我们保存数据。Imperative Programming的资料大量放在heap中，但FP则是放在栈（stack）内（或者由栈指向heap）。而现在普遍流行的GC（垃圾收集），源自于FP语言。

在讨论FP时，也常常会讨论到递归（recursion）。为何递归对于FP相当重要？因为递归可以用来保存状态。以斐波那契数列（1, 1, 2, 3, 5, 8, 13…）来说，每个值是前两个值的和，想制造

出斐波那契数列，imperative编程的做法会用循环，而FP的做法会用递归。

这个时候，你可能会说，用递归写出斐波那契数列或计算阶乘，这样的程序你曾经用C或C++或Java写过。恭喜你，你确实用过FP，只是你当时不自知而已。

递归可以保存状态，可以让程序变得相当精简，但是成本（时间与内存）也很高。所以，许多时候，函数式语言会希望我们将程序写成尾端递归（Tail Recursion），以便编译器自动将它编译成内存的直接跳跃（也就是循环）。

为了提升效率，许多函数式语言会纳入imperative的某些做法（例如允许副作用），这类的FPL被称为不纯（Impure）的函数式编程语言，例如Ocaml、F#、LISP、REBOL。当然也有一些语言坚持Pure Functional的做法，例如Erlang、Haskell、Occam、Oz。

以往纯的函数式语言会被某些人认为“不食人间烟火”，而不纯的函数式语言，则被认为比较实际、实用，但是最近大家的看法似乎有了改变。主要是以Erlang为首的纯函数式语言，似乎更能充分展现出FP的优势。除了可以标新立异当作IT上流社会炫耀表征之外，究竟采用FP有何优势？

首先是，单元测试（Unit）变得相当容易。对OOP来说，单元测试是以类别为单元，这种单元其实不小，而且要测试完整也不见得很容易。对于FP来说，函数是单元测试的单位。因为函数不可以有副作用，所以对于函数来说，我们要注意的只有输入（自变量）和输出（传出值），且传出值只受到自变量的影响。

这使得单元测试相当容易，只要管自变量的结果正确与否就好，不需要管函数调用的次序正确与否，或者外部状态是否做好正确的设定。如果是像C、Java或C#这类语言，检查函数的传出值是不够的，因为函数执行过程中可能会改变外部状态。但是对于FP来说，就不用担心这一点。

想除错，就必须能让此错误可以重现（reproduce），然后定位（locate）错误的地方。对FP来说，由于没有外部状态的因素干扰，所以上述这两点都相当容易就可以做到。Erlang的某个函数只要会出错，就一定每次都会出错，所以可以“重现”；C语言的某个函数出错，却不见得每次都出错，相当麻烦。一旦知道某个函数出错，你可以快速地在Erlang函数内找出问题所在，而予以修正；但是对C语言来说，外部状态影响太多，不容易除错。

FP相当适合写concurrency的程序。没有共享内存，没有执行绪，不需要担心critical section，不必使用mutex等上锁机制。由于没有外部状态的问题，FP的程序也相当适合进行程序代码“热插拔”或“热部署”（Hot Code Deployment），你可以不需要关闭你的软件系统，可以直接部署新的程序模块。

FP is as FP does

到底FP有哪些常见的特色？阿甘（Forrest Gump）说过“Stupid is as stupid does”。这个句型相当好用，套用到FP，就是“FP is as FP does”。我们可以透过FP做了些什么（does），来了解FP是什么（is）。

【Higher-Order Functions】。函数式语言有相当大的威力来自较高次方函数，特别是函数式语言的链接库往往会有许多较高次方函数，可以帮助你进行数据处理（例如特殊排序法、数据对应、数据过滤）、事件处理。

【Currying】链接库往往将函数定义得比较一般化，具有通用性。这样的函数，需要传入比较多的参数。利用Currying的方式，可以定义出“特殊化”的函数，当然非函数式语言也做得到这一点，但不可能像函数式语言的语法这么精简。

【Lazy Evaluation】表示式的执行可以拖延到真正需要执行时才执行，这就是Lazy Evaluation。

【Continuations】利用Continuation，可以将一个函数的传出值，传进另一个函数当作传入值，也可以产生循序执行的效果。利用continuation的方式，可以让原本没有状态的技术，有了状态，让应用更好写。利用Web原本是无状态的，如果将Continuation用到Web的开发上，会使得开发变得容易许多。

【Pattern Matching】模式比对的方式，可以让系统自动帮我们进行分支（branch），与变量的指定（assignment）。有了模式比对，FP可以降低依赖（imperative语言的）switch/case与（对象导向语言的）多型，而且写出来的程序代码也不会像switch/case那样一大块。

【Closure】Closure让函数在离开之后，其context依然保留（而不会像call stack内的frame一样，被丢弃）。有了Closure，就可以设计“传出值是函数”的函数。

【List Processing】FP的始祖语言LISP，名称的意思正是List Processing，目的是要进行方便的List处理（List是数据的集合）。许多函数式语言都有好用的List处理语法（例如List Comprehensions、取出List头部元素、插入List头部）与Lisp处理函数（例如map、filter）。

【Meta-Programming】许多FP语言都可以有提供方便的Meta-Programming工具，让你可以设计自己的DSL，来辅助软件开发。■

（本文略有删节。）

作者简介

自由作家，出生并成长于台湾，毕业于清华大学计算机科学研究所硕士班，曾任软件工程师、培训单位讲师、出版社编辑。

■ 责任编辑:赵健平（zhaojp@csdn.net）