**JavaScript与函数式编程**

牢记：函数式编程不是用函数来编程！！！  
  
  
23.4函数式编程   
23.4.1 什么是函数式编程  
  
        什么是函数式编程？如果你这么直白地询问，会发现它竟是一个不太容易解释的概念。许多在程序设计领域有着多年经验的老手，也无法很明白地说清楚函数式编程到底在研究些什么。函数式编程对于熟悉过程式程序设计的程序员来说的确是一个陌生的领域，闭包（closure）,延续（continuation）,和柯里化（currying）这些概念看起来是这么的陌生，同我们熟悉的if、else、while没有任何的相似之处。尽管函数式编程有着过程式无法比拟的优美的数学原型，但它又是那么的高深莫测，似乎只有拿着博士学位的人才玩得转它。  
  
        提示：这一节有点难，但它并不是掌握JavaScript所必需的技能，如果你不想用JavaScript来完成那些用Lisp来完成活儿，或者不想学函数式编程这种深奥的技巧，你完全可以跳过它们，进入下一章的旅程。  
  
        那么回到这个问题，什么是函数式编程？答案很长……

函数式编程第一定律：函数是第一型。  
  
        这句话本身该如何理解？什么才是真正的第一型？我们看下面的数学概念：  
  
        二元方程式 F(x, y) = 0，x, y 是变量， 把它写成 y = f(x)， x是参数，y是返回值，f是由x到y的映射关系，被称为函数。如果又有，G(x, y, z) = 0，或者记为 z = g(x, y)，g是x、y到z的映射关系，也是函数。如果g的参数x, y又满足前面的关系y = f(x)， 那么得到z = g(x, y) = g(x, f(x))，这里有两重含义，一是f(x)是x上的函数，又是函数g的参数，二是g是一个比f更高阶的函数。  
        这样我们就用z = g(x, f(x)) 来表示方程F(x, y) = 0和G(x, y, z) = 0的关联解，它是一个迭代的函数。我们也可以用另一种形式来表示g，记z = g(x, y, f)，这样我们将函数g一般化为一个高阶函数。同前面相比，后面这种表示方式的好处是，它是一种更加泛化的模型，例如T(x,y) = 0和G(x,y,z) = 0的关联解，我们也可以用同样的形式来表示（只要令f=t）。在这种支持把问题的解转换成高阶函数迭代的语言体系中，函数就被称为“第一型”。  
        JavaScript中的函数显然是“第一型”。下面就是一个典型的例子：  
          
        Array.prototype.each = function(closure)  
                {  
                return this.length ? [closure(this[0])].concat(this.slice(1).each(closure)) : [];  
                }  
  
这真是个神奇的魔法代码，它充分发挥了函数式的魅力，在整个代码中只有函数（function）和符号（Symbol）。它形式简洁并且威力无穷。  
[1,2,3,4].each(function(x){return x \* 2})得到[2,4,6,8]，而[1,2,3,4].each(function(x){return x-1})得到[0,1,2,3]。  
  
函数式和面向对象的本质都是“道法自然”。如果说，面向对象是一种真实世界的模拟的话，那么函数式就是数学世界的模拟，从某种意义上说，它的抽象程度比面向对象更高，因为数学系统本来就具有自然界所无法比拟的抽象性。  
  
函数式编程第二定律：闭包是函数式编程的挚友。  
  
闭包，在前面的章节中我们已经解释过了，它对于函数式编程非常重要。它最大的特点是不需要通过传递变量（符号）的方式就可以从内层直接访问外层的环境，这为多重嵌套下的函数式程序带来了极大的便利性，下面是一个例子：  
  
(function outerFun(x)  
{  
        return function innerFun(y)  
        {  
                return x \* y;  
        }  
})(2)(3);  
  
函数式编程第三定律：函数可以被科里化（Currying）。  
  
什么是Currying? 它是一个有趣的概念。还是从数学开始：我们说，考虑一个三维空间方程 F(x, y, z) = 0，如果我们限定z = 0，于是得到 F(x, y, 0) = 0 记为 F’(x, y)。这里F’显然是一个新的方程式，它代表三维空间曲线F(x, y, z)在z = 0平面上的两维投影。记y = f(x, z)， 令z = 0， 得到 y = f(x, 0)，记为 y = f’(x)， 我们说函数f’是f的一个Currying解。  
下面给出了JavaScript的Currying的例子：  
function add(x, y)  
{  
        if(x!=null && y!=null) return x + y;  
                else if(x!=null && y==null) return function(y)  
                {  
                return x + y;  
                }  
                else if(x==null && y!=null) return function(x)  
                {  
                       return x + y;  
                 }  
}  
var a = add(3, 4);  
var b = add(2);  
var c = b(10);  
  
上面的例子中，b=add(2)得到的是一个add()的Currying函数，它是当x = 2时，关于参数y的函数，注意到上面也用到了闭包的特性。  
  
有趣的是，我们可以给任意函数一般化Currying，例如：  
  
function Foo(x, y, z, w)  
{  
        var args = arguments;  
  
        if(Foo.length < args.length)  
                return function()  
                {  
                        return   
args.callee.apply(Array.apply([], args).concat(Array.apply([], arguments)));  
                }  
        else  
                return x + y – z \* w;  
}  
  
函数式编程第四定律：延迟求值和延续。  
        //TODO:这里再考虑下  
          
  
23.4.2 函数式编程的优点  
  
单元测试  
  
严格函数式编程的每一个符号都是对直接量或者表达式结果的引用，没有函数产生副作用。因为从未在某个地方修改过值，也没有函数修改过在其作用域之外的量并被其他函数使用（如类成员或全局变量）。这意味着函数求值的结果只是其返回值，而惟一影响其返回值的就是函数的参数。  
这是单元测试者的梦中仙境(wet dream)。对被测试程序中的每个函数，你只需在意其参数，而不必考虑函数调用顺序，不用谨慎地设置外部状态。所有要做的就是传递代表了边际情况的参数。如果程序中的每个函数都通过了单元测试，你就对这个软件的质量有了相当的自信。而命令式编程就不能这样乐观了，在 Java 或 C++ 中只检查函数的返回值还不够——我们还必须验证这个函数可能修改了的外部状态。  
  
调试  
  
如果一个函数式程序不如你期望地运行，调试也是轻而易举。因为函数式程序的 bug 不依赖于执行前与其无关的代码路径，你遇到的问题就总是可以再现。在命令式程序中，bug 时隐时现，因为在那里函数的功能依赖与其他函数的副作用，你可能会在和 bug 的产生无关的方向探寻很久，毫无收获。函数式程序就不是这样——如果一个函数的结果是错误的，那么无论之前你还执行过什么，这个函数总是返回相同的错误结果。  
一旦你将那个问题再现出来，寻其根源将毫不费力，甚至会让你开心。中断那个程序的执行然后检查堆栈，和命令式编程一样，栈里每一次函数调用的参数都呈现在你眼前。但是在命令式程序中只有这些参数还不够，函数还依赖于成员变量，全局变量和类的状态（这反过来也依赖着这许多情况）。函数式程序里函数只依赖于它的参数，而那些信息就在你注视的目光下！还有，在命令式程序里，只检查一个函数的返回值不能够让你确信这个函数已经正常工作了，你还要去查看那个函数作用域外数十个对象的状态来确认。对函数式程序，你要做的所有事就是查看其返回值！  
沿着堆栈检查函数的参数和返回值，只要发现一个不尽合理的结果就进入那个函数然后一步步跟踪下去，重复这一个过程，直到它让你发现了 bug 的生成点。  
  
并行  
函数式程序无需任何修改即可并行执行。不用担心死锁和临界区，因为你从未用锁！函数式程序里没有任何数据被同一线程修改两次，更不用说两个不同的线程了。这意味着可以不假思索地简单增加线程而不会引发折磨着并行应用程序的传统问题。  
事实既然如此，为什么并不是所有人都在需要高度并行作业的应用中采用函数式程序？嗯，他们正在这样做。爱立信公司设计了一种叫作 Erlang 的函数式语言并将它使用在需要极高抗错性和可扩展性的电信交换机上。还有很多人也发现了 Erlang 的优势并开始使用它。我们谈论的是电信通信控制系统，这与设计华尔街的典型系统相比对可靠性和可升级性要求高了得多。实际上，Erlang 系统并不可靠和易扩展，JavaScript 才是。Erlang 系统只是坚如磐石。  
关于并行的故事还没有就此停止，即使你的程序本身就是单线程的，那么函数式程序的编译器仍然可以优化它使其运行于多个CPU上。请看下面这段代码：  
  
String s1 = somewhatLongOperation1();  
String s2 = somewhatLongOperation2();  
String s3 = concatenate(s1, s2);  
  
在函数编程语言中，编译器会分析代码，辨认出潜在耗时的创建字符串s1和s2的函数，然后并行地运行它们。这在命令式语言中是不可能的，因为在那里，每个函数都有可能修改了函数作用域以外的状态并且其后续的函数又会依赖这些修改。在函数式语言里，自动分析函数并找出适合并行执行的候选函数简单的像自动进行的函数内联化！在这个意义上，函数式风格的程序是“不会过时的技术(future proof)”(即使不喜欢用行业术语，但这回要破例一次)。硬件厂商已经无法让CPU运行得更快了，于是他们增加了处理器核心的速度并因并行而获得了四倍的速度提升。当然他们也顺便忘记提及我们的多花的钱只是用在了解决平行问题的软件上了。一小部分的命令式软件和 100% 的函数式软件都可以直接并行运行于这些机器上。  
  
代码热部署  
  
过去要在 Windows上安装更新，重启计算机是难免的，而且还不只一次，即使是安装了一个新版的媒体播放器。Windows XP 大大改进了这一状态，但仍不理想（我今天工作时运行了Windows Update，现在一个烦人的图标总是显示在托盘里除非我重启一次机器）。Unix系统一直以来以更好的模式运行，安装更新时只需停止系统相关的组件，而不是整个操作系统。即使如此，对一个大规模的服务器应用这还是不能令人满意的。电信系统必须100%的时间运行，因为如果在系统更新时紧急拨号失效，就可能造成生命的损失。华尔街的公司也没有理由必须在周末停止服务以安装更新。  
理想的情况是完全不停止系统任何组件来更新相关的代码。在命令式的世界里这是不可能的。考虑运行时上载一个Java类并重载一个新的定义，那么所有这个类的实例都将不可用，因为它们被保存的状态丢失了。我们可以着手写些繁琐的版本控制代码来解决这个问题，然后将这个类的所有实例序列化，再销毁这些实例，继而用这个类新的定义来重新创建这些实例，然后载入先前被序列化的数据并希望载入代码可以恰到地将这些数据移植到新的实例。在此之上，每次更新都要重新手动编写这些用来移植的代码，而且要相当谨慎地防止破坏对象间的相互关系。理论简单，但实践可不容易。  
对函数式的程序，所有的状态即传递给函数的参数都被保存在了堆栈上，这使的热部署轻而易举！实际上，所有我们需要做的就是对工作中的代码和新版本的代码做一个差异比较，然后部署新代码。其他的工作将由一个语言工具自动完成！如果你认为这是个科幻故事，请再思考一下。多年来 Erlang工程师一直更新着他们的运转着的系统，而无需中断它。  
  
机器辅助的推理和优化  
  
函数式语言的一个有趣的属性就是他们可以用数学方式推理。因为一种函数式语言只是一个形式系统的实现，所有在纸上完成的运算都可以应用于用这种语言书写的程序。编译器可以用数学理论将转换一段代码转换为等价的但却更高效的代码[7]。多年来关系数据库一直在进行着这类优化。没有理由不能把这一技术应用到常规软件上。  
另外，还能使用这些技术来证明部分程序的正确，甚至可能创建工具来分析代码并为单元测试自动生成边界用例！对稳固的系统这种功能没有价值，但如果你要设计心房脉冲产生器 (pace maker)或空中交通控制系统，这种工具就不可或缺。如果你编写的应用程序不是产业的核心任务，这类工具也是你强于竞争对手的杀手锏。  
  
23.4.3 函数式编程的缺点  
  
闭包的副作用  
  
        非严格函数式编程中，闭包可以改写外部环境（在上一章中我们已经见过了），这带来了副作用，当这种副作用频繁出现并经常改变程序运行环境时，错误就变得难以跟踪。  
        //TODO:  
  
递归的形式  
          
        尽管递归通常是一种最简洁的表达形式，但它确实不如非递归的循环来的直观。  
        //TODO:  
  
延迟取值的弱点  
  
        //TODO:

只需要用function与符号就可以进行运算了，不需要其它额外的副作用