飞林沙

公告

2009年微软最有影响力开发者
2010年微软社区精英

昵称：飞林沙
园龄：6年7个月
荣誉：推荐博客
粉丝：354
关注：22
+加关注

随笔分类(89)
C#相关(11)
C语言相关(18)
Javascript相关(19)
PHP相关(9)
Silverlight相关(4)
Web开发相关(2)
WF相关(9)
乱七八糟的生活(2)
软件工程相关(4)
设计模式相关(8)
项目管理相关(3)

随笔档案(148)
2012年12月 (1)
2012年9月 (1)
2012年5月 (2)
2011年12月 (1)
2011年11月 (1)
2011年9月 (1)
2011年7月 (1)

函数式编程扫盲篇

1. 概论

在过去的近十年的时间里，面向对象编程大行其道。以至于在大学的教育里，老师也只会教给我们两种编程模型，面向过程和面向对象。

孰不知，在面向对象产生之前，在面向对象思想产生之前，函数式编程已经有了数十年的历史。

那么，接下来，就让我们回顾这个古老又现代的编程模型，让我们看看究竟是什么魔力将这个概念，将这个古老的概念，在21世纪的今天再次拉入了我们的视野。

2. 什么是函数式编程

在维基百科中，已经对函数式编程有了很详细的介绍。

那我们就来摘取一下Wiki上对Functional Programming的定义：

> In computer science, **functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

简单地翻译一下，也就是说**函数式编程是一种编程模型，他将计算机运算看做是数学中函数的计算，并且避免了状态以及变量的概念。**

接下来，我们就来剖析下函数式编程的一些特征。

3. 从并发说开来

说来惭愧，我第一个真正接触到函数式编程，要追溯到两年以前的《Erlang程序设计》，我们知道Erlang是一个支持高并发，有着强大容错性的函数式编程语言。

因为时间太久了，而且一直没有过真正地应用，所以对Erlang也只是停留在一些感性认识上。在我眼里，Erlang对高并发的支持体现在两方面，第一，**Erlang对轻量级进程的支持**（请注意此处进程并不等于操作系统的进程，而只是Erlang内部的一个单位单元），第二，**就是变量的不变性**。

4. 变量的不变性

在《Erlang程序设计》一书中，对变量的不变性是这样说的，Erlang是目前唯一变量不变性的语言。具体的话我记不清了，我不知道是老爷子就是这么写的，还是译者的问题。我在给这本书写书评的时候吹毛求疵地说：

2011年6月 (1)
2011年4月 (1)
2011年3月 (9)
2011年1月 (3)
2010年12月 (1)
2010年11月 (4)
2010年10月 (1)
2010年8月 (1)
2010年7月 (6)
2010年5月 (1)
2010年4月 (16)
2010年3月 (4)
2010年2月 (10)
2010年1月 (24)
2009年12月 (5)
2009年10月 (3)
2009年8月 (1)
2009年6月 (1)
2009年5月 (5)
2009年4月 (16)
2009年3月 (7)
2009年2月 (4)
2009年1月 (9)
2008年11月 (2)
2008年9月 (3)
2008年8月 (2)

积分与排名
积分 - 301600
排名 - 319

阅读排行榜
1. git rebase小计(转)(51890)
2. 函数式编程扫盲篇(39885)
3. 说说Python程序的执行过程(18265)
4. 重温设计模式（三）——职责链模式(chain of responsibility)(17037)
5. 总结字符串比较函数(13814)

> 我对这句话有异议，切不说曾经的Lisp，再到如今的F#都对赋值操作另眼相看，低人一等。单说如今的Java和C#，提供的final和readonly一样可以支持变量的不变性，而这个唯一未免显得有点太孤傲了些。

让我们先来看两段程序，首先是我们常见的一种包含赋值的程序：

```
class Account:
    def __init__(self,balance):
        self.balance = balance
    def desposit(self,amount):
        self.balance = self.balance + amount
        return self.balance
    def despositTwice(self):
        self.balance = self.balance * 2
        return self.balance

if __name__ == '__main__':
    account = Account(100)
    print(account.desposit(10))
    print(account.despositTwice())
```

这段程序本身是没有问题的，但是我们考虑这样一种情况，现在有多个进程在同时跑这一个程序，那么程序就会被先desposit 还是先 despositTwice所影响。

但是如果我们采用这样的方式：

```
def makeAccount(balance):
    global desposit
    global despositTwice
    def desposit(amount):
        result = balance + amount
        return result
    def despositTwice():
        result = balance * 2
        return result
    def dispatch(method):
        return eval(method)
    return dispatch

if __name__ == '__main__':
    handler = makeAccount(100)
    print(handler('desposit')(10))
    print(handler('despositTwice')())
```

这时我们就会发现，无论多少个进程在跑，因为我们本身没有赋值操作，所以都不会影响到我们的最终结果。

但是这样也像大家看到的一样，采用这样的方式没有办法保持状态。

这也就是我们在之前概念中看到的无状态性。

5. 再看函数式编程的崛起

既然已经看完了函数式编程的基本特征，那就让我们来想想数十年后函数式编程再次崛起的幕后原因。

一直以来，作为函数式编程代表的Lisp，还是Haskell，更多地都是在大学中，在实验室中应用，而很少真的应用到真实的生产环境。

先让我们再来回顾一下伟大的摩尔定律：

> 1、集成电路芯片上所集成的电路的数目，每隔18个月就翻一番。
>
> 2、微处理器的性能每隔18个月提高一倍，而价格下降一半。
>
> 3、用一个美元所能买到的电脑性能，每隔18个月翻两番。

一如摩尔的预测，整个信息产业就这样飞速地向前发展着，但是在近年，我们却可以发现摩尔定律逐渐地失效了，芯片上元件的尺寸是不可能无限地缩小的，这就意味着芯片上所能集成的电子元件的数量一定会在某个时刻达到一个极限。那么当技术达到这个极限时，我们又该如何适应日益增长的计算需求，电子元件厂商给出了答案，就是多核。

多核并行程序设计就这样被推到了前线，而命令式编程天生的缺陷却使并行编程模型变得非常复杂，无论是信号量，还是锁的概念，都使程序员不堪其重。

就这样，函数式编程终于在数十年后，终于走出实验室，来到了真实的生产环境中，无论是冷门的Haskell，Erlang，还是Scala，F#，都是函数式编程成功的典型。

6. 函数式编程的第一型

我们知道，对象是面向对象的第一型，那么函数式编程也是一样，函数是函数式编程的第一型。

我们在函数式编程中努力用函数来表达所有的概念，完成所有的操作。

在面向对象编程中，我们把对象传来传去，那在函数式编程中，我们要做的是把函数传来传去，而这个，说成术语，我们把他叫做高阶函数。

那我们就来看一个高阶函数的应用，熟悉js的同学应该对下面的代码很熟悉，让哦我们来写一个在电子电路中常用的滤波器的示例代码。

```
def Filt(arr,func):
    result = []
    for item in arr:
```

```
        result.append(func(item))
    return result

def MyFilter(ele):
    if ele < 0 :
        return 0
    return ele

if __name__ == '__main__':
    arr = [-5,3,5,11,-45,32]
    print('%s' % (Filt(arr,MyFilter)))
```

哦，之前忘记了说，什么叫做高阶函数，我们给出定义：

> 在数学和计算机科学中，**高阶函数**是至少满足下列一个条件的函数：
>
> - 接受一个或多个函数作为输入
> - 输出一个函数

那么，毫无疑问上面的滤波器，就是高阶函数的一种应用。

在函数式编程中，函数是基本单位，是第一型，他几乎被用作一切，包括最简单的计算，甚至连变量都被计算所取代。在函数式编程中，变量只是一个名称，而不是一个存储单元，这是函数式编程与传统的命令式编程最典型的不同之处。

让我们看看，变量只是一个名称，在上面的代码中，我们可以这样重写主函数：

```
if __name__ == '__main__':
    arr = [-5,3,5,11,-45,32]
    func = MyFilter
    print('%s' % (Filt(arr,func)))
```

当然，我们还可以把程序更精简一些，利用函数式编程中的利器，map,filter和reduce：

```
if __name__ == '__main__':
    arr = [-5,3,5,11,-45,32]
    print('%s' % (map(lambda x : 0 if x<0 else x ,arr)))
```

这样看上去是不是更赏心悦目呢?

这样我们就看到了，函数是我们编程的基本单位。

7. 函数式编程的数学本质

忘了是谁说过：一切问题，归根结底到最后都是数学问题。

编程从来都不是难事儿，无非是细心，加上一些函数类库的熟悉程度，加上经验的堆积，而真正困难的，是如何把一个实际问题，转换成一个数学模

型。这也是为什么微软，Google之类的公司重视算法，这也是为什么数学建模大赛在大学计算机系如此被看重的原因。

先假设我们已经凭借我们良好的数学思维和逻辑思维建立好了数学模型，那么接下来要做的是如何把数学语言来表达成计算机能看懂的程序语言。

这里我们再看在第四节中，我们提到的赋值模型，同一个函数，同一个参数，却会在不同的场景下计算出不同的结果，这是在数学函数中完全不可能出现的情况，**f(x) = y**，那么这个函数无论在什么场景下，都会得到同样的结果，这个我们称之为函数的确定性。

这也是赋值模型与数学模型的不兼容之处。而函数式编程取消了赋值模型，则使数学模型与编程模型完美地达成了统一。

8. 函数式编程的抽象本质

相信每个程序员都对抽象这个概念不陌生。

在面向对象编程中，我们说，类是现实事物的一种抽象表示。那么抽象的最大作用在我看来就在于抽象事物的重用性，一个事物越具体，那么他的可重用性就越低，因此，我们再打造可重用性代码，类，类库时，其实在做的本质工作就在于提高代码的抽象性。而再往大了说开来，程序员做的工作，就是把一系列过程抽象开来，反映成一个通用过程，然后用代码表示出来。

在面向对象中，我们把事物抽象。而在函数式编程中，我们则是在将函数方法抽象，第六节的滤波器已经让我们知道，函数一样是可重用，可置换的抽象单位。

那么我们说函数式编程的抽象本质则是将函数也作为一个抽象单位，而反映成代码形式，则是高阶函数。

9.状态到底怎么办

我们说了一大堆函数式编程的特点，但是我们忽略了，这些都是在理想的层面，我们回头想想第四节的变量不变性，确实，我们说，函数式编程是无状态的，可是在我们现实情况中，状态不可能一直保持不变，而状态必然需要改变，传递，那么我们在函数式编程中的则是将其保存在函数的参数中，作为函数的附属品来传递。

ps：在Erlang中，进程之间的交互传递变量是靠"信箱"的收发信件来实现，其实我们想一想，从本质而言，也是将变量作为一个附属品来传递么！

我们来看个例子，我们在这里举一个求x的n次方的例子，我们用传统的命令式编程来写一下：

```
def expr(x,n):
    result = 1
    for i in range(1,n+1):
        result = result * x
    return result

if __name__ == '__main__':
```

```
    print(expr(2,5))
```

这里，我们一直在对result变量赋值，但是我们知道，在函数式编程中的变量是具有不变性的，那么我们为了保持result的状态，就需要将result作为函数参数来传递以保持状态：

```
def expr(num,n):
    if n==0:
        return 1
    return num*expr(num,n-1)

if __name__ == '__main__':
    print(expr(2,5))
```

呦，这不是递归么！

10. 函数式编程和递归

递归是函数式编程的一个重要的概念，循环可以没有，但是递归对于函数式编程却是不可或缺的。

在这里，我得承认，我确实不知道我该怎么解释递归为什么对函数式编程那么重要。我能想到的只是递归充分地发挥了函数的威力，也解决了函数式编程无状态的问题。（如果大家有其他的意见，请赐教）

递归其实就是将大问题无限地分解，直到问题足够小。

而递归与循环在编程模型和思维模型上最大的区别则在于：

循环是在描述我们该如何地去解决问题。

递归是在描述这个问题的定义。

那么就让我们以斐波那契数列为例来看下这两种编程模型。

先说我们最常见的递归模型，这里，我不采用动态规划来做临时状态的缓存，只是说这种思路：

```
def Fib(a):
    if a==0 or a==1:
        return 1
    else:
        return Fib(a-2)+Fib(a-1)
```

递归是在描述什么是斐波那契数列，这个数列的定义就是一个数等于他的前两项的和，并且已知Fib(0)和Fib(1)等于1。而程序则是用计算机语言来把这个定义重新描述了一次。

那接下来，我们看下循环模型：

```
def Fib(n):
```

```
a=1
b=1
n = n - 1
while n>0:
    temp=a
    a=a+b
    b=temp
    n = n-1
return b
```

这里则是在描述我们该如何求解斐波那契数列，应该先怎么样再怎么样。

而我们明显可以看到，递归相比于循环，具有着更加良好的可读性。

但是，我们也不能忽略，递归而产生的StackOverflow，而赋值模型呢？我们懂的，函数式编程不能赋值，那么怎么办？

11. 尾递归，伪递归

我们之前说到了递归和循环各自的问题，那怎么来解决这个问题，函数式编程为我们抛出了答案，尾递归。

什么是尾递归，用最通俗的话说：**就是在最后一部单纯地去调用递归函数**，这里我们要注意"单纯"这个字眼。

那么我们说下尾递归的原理，**其实尾递归就是不要保持当前递归函数的状态，而把需要保持的东西全部用参数给传到下一个函数里，这样就可以自动清空本次调用的栈空间。**这样的话，占用的栈空间就是常数阶的了。

在看尾递归代码之前，我们还是先来明确一下递归的分类，我们将**递归分成"树形递归"和"尾递归"，什么是树形递归，就是把计算过程逐一展开，最后形成的是一棵树状的结构**，比如之前的斐波那契数列的递归解法。

那么我们来看下斐波那契尾递归的写法：

```
def Fib(a,b,n):
    if n==0:
        return b
    else:
        return Fib(b,a+b,n-1)
```

这里看上去有些难以理解，我们来解释一下：传入的a和b分别是前两个数，那么每次我都推进一位，那么b就变成了第一个数，而a+b就变成的第二个数。

这就是尾递归。其实我们想一想，这不是在描述问题，而是在寻找一种问题的解决方案，和上面的循环有什么区别呢？我们来做一个从尾递归到循环的转换把！

最后返回b是把，那我就先声明了，b=0

要传入a是把，我也声明了，a=1

要计算到n==0是把，还是循环while n!=0

每一次都要做一个那样的计算是吧，我用临时变量交换一下。temp=b ; b=a+b;a=temp。

那么按照这个思路一步步转换下去，是不是就是我们在上面写的那段循环代码呢?

那么这个尾递归，其实本质上就是个"伪递归"，您说呢?

既然我们可以优化，对于大多数的函数式编程语言的编译器来说，他们对尾递归同样提供了优化，使尾递归可以优化成循环迭代的形式，使其不会造成堆栈溢出的情况。

12. 惰性求值与并行

第一次接触到惰性求值这个概念应该是在Haskell语言中，看一个最简单的惰性求值，我觉得也是最经典的例子:

在Haskell里，有个repeat关键字，他的作用是返回一个无限长的List，那么我们来看下:

```
take 10 (repeat 1)
```

就是这句代码，如果没有了惰性求值，我想这个进程一定会死在那里，可是结果却是很正常，返回了长度为10的List，List里的值都是1。这就是惰性求值的典型案例。

我们看这样一段简单的代码:

```
def getResult():
    a = getA()   //Take a long time
    b = getB()   //Take a long time
    c = a + b
```

这段代码本身很简单，在命令式程序设计中，编译器（或解释器）会做的就是逐一解释代码，按顺序求出a和b的值，然后再求出c。

可是我们从并行的角度考虑，求a的值是不是可以和求b的值并行呢? 也就是说，直到执行到a+b的时候我们编译器才意识到a和b直到现在才需要，那么我们双核处理器就自然去发挥去最大的功效去计算了呢!

这才是惰性求值的最大威力。

当然，惰性求值有着这样的优点也必然有着缺点，我记得我看过一个例子是最经典的:

```
def Test():
    print('Please enter a number:')
    a = raw_input()
```

可是这段代码如果惰性求值的话，第一句话就不见得会在第二句话之前执行了。

13. 函数式编程总览

我们看完了函数式编程的特点，我们想想函数式编程的应用场合。

1. 数学推理

2. 并行程序

那么我们总体地说，其实函数式编程最适合地还是解决局部性的数学小问题，要让函数式编程来做CRUD，来做我们传统的逻辑性很强的Web编程，就有些免为其难了。

就像如果要用Scala完全取代今天的Java的工作，我想恐怕效果会很糟糕。而让Scala来负责底层服务的编写，恐怕再合适不过了。

而在一种语言中融入多种语言范式，最典型的C#。在C# 3.0中引入Lambda表达式，在C# 4.0中引入声明式编程，我们某些人在嘲笑C#越来越臃肿的同时，却忽略了，这样的语法糖，带给我们的不仅仅是代码书写上的遍历，更重要的是编程思维的一种进步。

好吧，那就让我们忘记那些C#中Lambda背后的实现机制，在C#中，还是在那些更纯粹地支持函数式编程的语言中，尽情地去体验函数式编程带给我们的快乐把！

« 上一篇：mongodb小结(转)
» 下一篇：载入Haskell的函数

posted @ 2011-03-07 23:12 飞林沙 阅读(39885) 评论(50) 编辑 收藏

评论列表

#1楼 2011-03-08 07:16 蛙蛙王子
没人抢沙发，我抢
支持(0) 反对(0)

#2楼 2011-03-08 08:19 南京--XLuo(不觉流年似水)
获益匪浅
支持(0) 反对(0)

#3楼 2011-03-08 08:48 Kain
获益匪浅
支持(0) 反对(0)

#4楼 2011-03-08 09:13 EasyShop

楼主用的几个示例貌似是python,
那么用c#也一样可以实现"函数式编程"了?

另外一个疑问:
除了理论上可以简化并发编程,函数式编程不见得比传统编程思想更有优势,假如我可以把"状态变量"影响的副作用控制在小范围内,个人觉得有时候循环比递归表达更直白,更容易调试.

等楼主继续科普....

支持(1) 反对(0)

#5楼 2011-03-08 09:49 地狱门神

感觉并行计算重要的不是循环或者递归。
而是要把集群操作抽象出来，由编译器或者JIT去决定内部计算细节。在这个要求下，需要消除副作用，保证计算结果和计算顺序无关。

支持(0) 反对(0)

#6楼 2011-03-08 09:56 深蓝医生

的确是思维的大转变!

支持(0) 反对(0)

#7楼[楼主] 2011-03-08 10:07 飞林沙

@EasyShop

1.C#本身并不是函数式的语言，但是从C#3.0开始，引入Lambda表达式，虽然说Lambda本身的实现是匿名方法，而匿名方法本身是委托，儿委托的本身又是一个类的实现，但是从思维上，我们确实也可以来实现函数式编程，写个嘴简单的吧。

```
List<int> list = new List<int>(){1,2,3,4,5};
list.Sort((a,b)=>a>b)
```

好久不写C#了，身边也没环境，类库方法有点忘记了....大概是这个意思吧

2.具体是哪个直白，哪个更容易理解，更贴近人的思维，其实我更适应的也是顺序式的程序设计，毕竟这么多年养成的习惯。就像当我们刚刚从C迁移到C#和Java上一样，我们都可能没有办法去适应面向对象的思维，不是么?

支持(0) 反对(0)

#8楼[楼主] 2011-03-08 10:13 飞林沙

@地狱门神

恩，对，从并行计算的角度来说循环还是递归并不重要。

支持(0) 反对(0)

#9楼 2011-03-08 10:24 bidaas

要底层都是递归了,对系统的要求得多高啊?我可不希望一软件到客户那一运行就看着内存占用往上飙

支持(0) 反对(0)

#10楼[楼主] 2011-03-08 10:43 飞林沙

@bidaas

递归不是实现函数式编程的唯一方式，而且就算递归，您也可以看看尾递归哦！

支持(0) 反对(0)

#11楼 2011-03-08 11:09 yzx226

必须顶！

支持(0) 反对(0)

#12楼 2011-03-08 11:23 toEverybody

C#的Andser好象说过,C#是解决我们在各种应用苦于学习多种语法的困境,如操作数据库要用Sql,调底层要用C++....这是C#要解决的问题,可是他忽略了性能的带来的问题

支持(0) 反对(0)

#13楼 2011-03-08 11:52 徐少侠

引用

toEverybody：C#的Andser好象说过,C#是解决我们在各种应用苦于学习多种语法的困境,如操作数据库要用Sql,调底层要用C++....这是C#要解决的问题,可是他忽略了性能的带来的问题

性能，在他成为问题之前不是问题。

支持(1) 反对(0)

#14楼 2011-03-08 12:34 Jeffrey Zhao

@toEverybody

我用一句话陈述回应你的一句话陈述：有个屁性能问题。

支持(0) 反对(0)

#15楼 2011-03-08 12:58 横刀天笑

沙沙又换领域了啊～～

我觉得函数编程最大的亮点是编程的思考方式的转变，什么变量不变性，什么惰性求值都是浮云。它让我们从另一种角度思考问题。
不管并行时代是不是来临，学习一下函数编程都能对日常的开发和设计工作起到点拨的作用～～

沙沙，继续～～

支持(0) 反对(0)

#16楼 2011-03-08 13:38 蛙蛙王子

花了一个多小时看了下这篇帖子，有几个问题要请教一下:
global desposit里的gloabal是啥意思?
map,filter和reduce能详细讲讲不?
print('%s' % (map(lambda x : 0 if x<0 else x ,arr))) 这玩意括号括的多了看着晕呀。
算法设计里经常用动态规划把递归整成迭代来提高性能，函数式编程正好相反，这会降低性能吧?
11.尾递归,伪递归里"占用的栈控件就是常数阶的了。"应该是栈空间。
12.惰性求值里"可是这段代码如果惰性求职的话，第一句话就不见得会在第二句话之前执行了。"应该是求值
惰性求值的例子里，如果getA先执行完，而b还没执行完，那么c=a+b的执行就执行不了吧，这时候执行c=a+b的线程是等待状态吗?

本篇精华是：编程从来都不是难事儿，无非是细心，加上一些函数类库的熟悉程度，加上经验的堆积，而真正困难的，是如何把一个实际问题，转换成一个数学模型。

总体来说，莎莎这篇帖子可以作为函数式编程的经典入门文章了，强大呀。

支持(0) 反对(0)

#17楼 2011-03-08 13:52 BlueDream

学习了~

支持(0) 反对(0)

#18楼[楼主] 2011-03-08 13:59 飞林沙

@蛙蛙王子

1. 在Python中，eval只能找到两样东西，一个是公有的，一个是局部的。所以上面的代码为了能够找到withdraw方法，就只能把withdraw在外部声明为global，当然另外一种办法是在eval的同一作用域里声明一个withdraw的变量

2. map,filter,reduce就是三个函数...蛙蛙在网上随便搜一下就一大堆了，其实C#里也有挺多类似的实现的，比如sort，当然，这个蛙蛙可以自己写下，应该不难的。。。。

3. 那段一大堆括号的代码，我只是想说明，用函数式编程，用map可以把代码写成很短，我看着其实也有点迷糊

4. 递归和迭代的区别我在文里说了，当然递归确实很占用空间，容易Stackoverflow，但是有时候，递归确实比迭代有更好的可读性，比如我在文中写的那个斐波那契。这个只能说采用个折中了，或者换个思路，用尾递归。毕竟一般的函数式语言解释器会做一定的优化，将之优化成迭代的形式