

零基础自学用Python 3开发网络爬虫(三): 伪装浏览器君

原文出处: [Jecvay Notes \(@Jecvay\)](#)

- [《零基础自学用Python 3开发网络爬虫\(一\)》](#)
- [《零基础自学用Python 3开发网络爬虫\(二\)》](#)

上一次我自学爬虫的时候, 写了一个简陋的勉强能运行的爬虫alpha. alpha版有很多问题. 比如一个网站上不了, 爬虫却一直在等待连接返回response, 不知道超时跳过; 或者有的网站专门拦截爬虫程序, 我们的爬虫也不会伪装自己成为浏览器正规部队; 并且抓取的内容没有保存到本地, 没有什么作用. 这次我们一个个解决这些小问题.

此外, 在我写这系列文章的第二篇的时候, 我还是一个对http的get和post以及response这些名词一无所知的人, 但是我觉得这样是写不好爬虫的. 于是我参考了 <<计算机网络-自顶向下方法>> 这本书的第二章的大部分内容. 如果你也一样对http的机制一无所知, 我也推荐你找一找这方面的资料来看. 在看的过程中, 安装一个叫做Fiddler的软件, 边学边实践, 观察浏览器是如何访问一个网站的, 如何发出请求, 如何处理响应, 如何进行跳转, 甚至如何通过登录认证. 有句老话说得好, 越会用Fiddler, 就对理论理解更深刻; 越对理论理解深刻, Fiddler就用得越顺手. 最后我们在用爬虫去做各种各样的事情的时候, Fiddler总是最得力的助手之一.

添加超时跳过功能

首先, 我简单地将

Python

```
urlop =  
urllib.request.urlopen(url)
```

```
1 urlop = urllib.request.urlopen(url)
```

改为

Python

```
urlop =  
urllib.request.urlopen(url)
```

```
1 urlop = urllib.request.urlopen(url, timeout = 2)
```

运行后发现, 当发生超时, 程序因为exception中断. 于是我把这一句也放在try .. except 结构里, 问题解决.

支持自动跳转

在爬 <http://baidu.com> 的时候, 爬回来一个没有什么内容的东西, 这个东西告诉我们应该跳转到

http://www.baidu.com . 但是我们的爬虫并不支持自动跳转, 现在我们来加上这个功能, 让爬虫在爬 baidu.com 的时候能够抓取 www.baidu.com 的内容.

首先我们要知道爬 http://baidu.com 的时候他返回的页面是怎么样, 这个我们既可以用 Fiddler 看, 也可以写一个小爬虫来抓取. 这里我抓到的内容如下, 你也应该尝试一下写几行 python 来抓一抓.

XHTML



```
1 <html>
2 <meta http-equiv="refresh" content="0;url=http://www.baidu.com/">
3 </html>
```

看代码我们知道这是一个利用 html 的 meta 来刷新与重定向的代码, 其中的0是等待0秒后跳转, 也就是立即跳转. 这样我们再像上一次说的那样用一个正则表达式把这个url提取出来就可以爬到正确的地方去了. 其实我们上一次写的爬虫已经可以具有这个功能, 这里只是单独拿出来说明一下 http 的 meta 跳转.


伪装浏览器正规军

前面几个小内容都写的比较少. 现在详细研究一下如何让网站们把我们的Python爬虫当成正规的浏览器来访. 因为如果不这么伪装自己, 有的网站就爬不回来了. 如果看过理论方面的知识, 就知道我们是要在 GET 的时候将 User-Agent 添加到header里.

如果没有看过理论知识, 按照以下关键字搜索学习吧 :D

- HTTP 报文分两种: **请求报文**和**响应报文**
- 请求报文的**请求行**与**首部行**
- **GET, POST, HEAD, PUT, DELETE** 方法

我用 IE 浏览器访问百度首页的时候, 浏览器发出去请求报文如下:



```
1 GET http://www.baidu.com/ HTTP/1.1
2 Accept: text/html, application/xhtml+xml, */*
3 Accept-Language: en-US,en;q=0.8,zh-Hans-CN;q=0.5,zh-Hans;q=0.3
4 User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
5 Accept-Encoding: gzip, deflate
6 Host: www.baidu.com
7 DNT: 1
8 Connection: Keep-Alive
9 Cookie: BAIDUID=57F4D171573A6B88A68789EF5DDFE87:FG=1;
uc_login_unique=ccba6e8d978872d57c7654130e714abd; BD_UPN=11263145; BD
```

然后百度收到这个消息后, 返回给我的响应报文如下(有删节):

```
HTTP/1.1 200 OK
Date: Mon, 29 Sep 2014
```

```
HTTP/1.1 200 OK
Date: Mon, 29 Sep 2014 13:07:01 GMT
Content-Type: text/html; charset=utf-8
1 Connection: Keep-Alive
2 Vary: Accept-Encoding
3 Cache-Control: private
4 Cxy_all: baidu+8b13ba5a7289a37fb380e0324ad688e7
5 Expires: Mon, 29 Sep 2014 13:06:21 GMT
6 X-Powered-By: HPHP
7 Server: BWS/1.1
8 BDPAGETYPE: 1
9 BDQID: 0x8d15bb610001fe79
10 BDUSERID: 0
11 Set-Cookie: BDSVRTM=0; path=/
12 Set-Cookie: BD_HOME=0; path=/
13 Content-Length: 80137
14
15 <!DOCTYPE html><!--STATUS OK--><html><head><meta http-equiv="content-type"
16 content="text/html; charset=utf-8"><meta http-equiv="X-UA-Compatible" content="IE=Edge"><link rel="dns-prefetch"
17 href="//s1.bdstatic.com"/><link rel="dns-prefetch" href="//t1.baidu.com"/><link rel="dns-prefetch"
18 href="//t2.baidu.com"/><link rel="dns-prefetch" href="//t3.baidu.com"/><link rel="dns-prefetch" href="//t10.baidu.com"/>
<link rel="dns-prefetch" href="//t11.baidu.com"/><link rel="dns-prefetch" href="//t12.baidu.com"/><link rel="dns-
prefetch" href="//b1.bdstatic.com"/><title>百度一下，你就知道</title><style index="index" > .....这里省略两万
字..... </script></body></html>
```

如果能够看懂这段话的第一句就OK了，别的可以以后再配合 Fiddler 慢慢研究。所以我们要做的就是 Python 爬虫向百度发起请求的时候，顺便在请求里面写上 User-Agent，表明自己是浏览器君。

在 GET 的时候添加 header 有很多方法，下面介绍两种方法。

第一种方法比较简便直接，但是不好扩展功能，代码如下：

Python

```
import urllib.request
```

```
1 import urllib.request
2
3 url = 'http://www.baidu.com/'
4 req = urllib.request.Request(url, headers = {
5     'Connection': 'Keep-Alive',
6     'Accept': 'text/html, application/xhtml+xml, */*',
7     'Accept-Language': 'en-US,en;q=0.8,zh-Hans-CN;q=0.5,zh-Hans;q=0.3',
8     'User-Agent': 'Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko'
9 })
10 oper = urllib.request.urlopen(req)
11 data = oper.read()
12 print(data.decode())
```

第二种方法使用了 build_opener 这个方法，用来自定义 opener，这种方法的好处是可以方便的拓展功能，例如下面的代码就拓展了自动处理 Cookies 的功能。

Python

```
import urllib.request
import http.cookiejar
```

```
1 import urllib.request
2 import http.cookiejar
3
4 # head: dict of header
5 def makeMyOpener(head = {
6     'Connection': 'Keep-Alive',
7     'Accept': 'text/html, application/xhtml+xml, */*',
8     'Accept-Language': 'en-US,en;q=0.8,zh-Hans-CN;q=0.5,zh-Hans;q=0.3',
9     'User-Agent': 'Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko'
10 }):
11     cj = http.cookiejar.CookieJar()
12     opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
13     header = []
14     for key, value in head.items():
15         elem = (key, value)
16         header.append(elem)
17     opener.addheaders = header
18     return opener
19
20 oper = makeMyOpener()
21 uop = oper.open('http://www.baidu.com/', timeout = 1000)
22 data = uop.read()
23 print(data.decode())
```

上述代码运行后通过 Fiddler 抓到的 GET 报文如下所示:

```
GET
http://www.baidu.com/
```

```
1 GET http://www.baidu.com/ HTTP/1.1
2 Accept-Encoding: identity
3 Connection: close
4 Host: www.baidu.com
5 User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
6 Accept: text/html, application/xhtml+xml, */*
7 Accept-Language: en-US,en;q=0.8,zh-Hans-CN;q=0.5,zh-Hans;q=0.3
```

可见我们在代码里写的东西都添加到请求报文里面了。

保存抓回来的报文

顺便说说文件操作。Python 的文件操作还是相当方便的。我们可以讲抓回来的数据 data 以二进制形式保存, 也可以经过 decode() 处理成为字符串后以文本形式保存。改动一下打开文件的方式就能用不同的姿势保存文件了。下面是参考代码:

Python

```
def saveFile(data):
    save_path =
```

```
1 def saveFile(data):
2     save_path = 'D:\\temp.out'
3     f_obj = open(save_path, 'wb') # wb 表示打开方式
4     f_obj.write(data)
```

```
5     f_obj.close()
6
7 # 这里省略爬虫代码
8 # ...
9
10 # 爬到的数据放到 dat 变量里
11 # 将 dat 变量保存到 D 盘下
12 saveFile(dat)
```

[下回](#)我们会用 Python 来爬那些需要登录之后才能看到的信息. 在那之前, 我已经对 Fiddler 稍微熟悉了. 希望一起学习的也提前安装个 Fiddler 玩一下.

1 赞 5 收藏 [评论](#)

零基础自学用Python 3开发网络爬虫(二): 用到的数据结构简介以及爬虫

Ver1.0 alpha

原文出处: [Jecvay Notes \(@Jecvay\)](#)

[上一回](#), 我学会了

1. 用伪代码写出爬虫的主要框架;
2. 用Python的urllib.request库抓取指定url的页面;
3. 用Python的urllib.parse库对普通字符串转符合url的字符串.

这一回, 开始用Python将伪代码中的所有部分实现. 由于文章的标题就是“零基础”, 因此会先把用到的两种数据结构**队列**和**集合**介绍一下. 而对于**正则表达式**部分, 限于篇幅不能介绍, 但给出我比较喜欢的几个参考资料.

Python的队列

在爬虫程序中, 用到了广度优先搜索(BFS)算法. 这个算法用到的数据结构就是队列.

Python的List功能已经足够完成队列的功能, 可以用 `append()` 来向队尾添加元素, 可以用类似数组的方式来获取队首元素, 可以用 `pop(0)` 来弹出队首元素. 但是List用来完成队列功能其实是**低效率**的, 因为List在队首使用 `pop(0)` 和 `insert()` 都是效率比较低的, Python官方建议使用 `collection.deque` 来高效的完成队列任务.

Python

```
from collections import deque
1 from collections import deque
2 queue = deque(["Eric", "John", "Michael"])
3 queue.append("Terry")      # Terry 入队
4 queue.append("Graham")     # Graham 入队
5 queue.popleft()            # 队首元素出队
6 #输出: 'Eric'
7 queue.popleft()            # 队首元素出队
8 #输出: 'John'
9 queue                      # 队列中剩下的元素
10 #输出: deque(['Michael', 'Terry', 'Graham'])
```

(以上例子引用自官方文档)

Python的集合

在爬虫程序中, 为了不重复爬那些已经爬过的网站, 我们需要把爬过的页面的url放进集合中, 在每一次要爬某一个url之前, 先看看集合里面是否已经存在. 如果已经存在, 我们就跳过这个url; 如果不存在, 我们先把url放入集合中, 然后再去爬这个页面.

Python提供了set这种数据结构. set是一种无序的, 不包含重复元素的结构. 一般用来测试是否已经包含了某元素, 或者用来对众多元素们去重. 与数学中的集合论同样, 他支持的运算有**交**, **并**, **差**, **对称差**

创建一个set可以用 `set()` 函数或者花括号 `{}`. 但是创建一个空集是不能使用一个花括号的, 只能用 `set()` 函数. 因为一个空的花括号创建的是一个字典数据结构. 以下同样是Python官网提供的示例.

Python

```
>>> basket = {'apple',
1 >>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2 >>> print(basket)      # 这里演示的是去重功能
3 {'orange', 'banana', 'pear', 'apple'}
4 >>> 'orange' in basket   # 快速判断元素是否在集合内
5 True
6 >>> 'crabgrass' in basket
7 False
8
9 >>> # 下面展示两个集合间的运算.
10 ...
11 >>> a = set('abracadabra')
12 >>> b = set('alacazam')
13 >>> a
14 {'a', 'r', 'b', 'c', 'd'}
15 >>> a - b                # 集合a中包含元素
16 {'r', 'd', 'b'}
17 >>> a | b                # 集合a或b中包含的所有元素
18 {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
19 >>> a & b                # 集合a和b中都包含了的元素
```

```
20 {'a', 'c'}
21 >>> a ^ b          # 不同时包含于a和b的元素
22 {'r', 'd', 'b', 'm', 'z', 'l'}
```

其实我们只是用到其中的**快速判断元素是否在集合内**的功能, 以及集合的**并运算**.

Python的正则表达式

在爬虫程序中, 爬回来的数据是一个字符串, 字符串的内容是页面的html代码. 我们要从字符串中, 提取出页面提到过的所有url. 这就要求爬虫程序要有简单的字符串处理能力, 而正则表达式可以很轻松的完成这一任务.

参考资料

- [正则表达式30分钟入门教程](#)
- [w3cschool 的Python正则表达式部分](#)
- [Python正则表达式指南](#)

虽然正则表达式功能异常强大, 很多实际上用的规则也非常巧妙, 真正熟练正则表达式需要比较长的实践锻炼. 不过我们只需要掌握如何使用正则表达式在一个字符串中, 把所有的url都找出来, 就可以了. 如果实在想要跳过这一部分, 可以在网上找到很多现成的匹配url的表达式, 拿来用即可.

Python网络爬虫Ver 1.0 alpha

有了以上铺垫, 终于可以开始写真正的爬虫了. 我选择的入口地址是Fenng叔的[Startup News](#), 我想Fenng叔刚刚拿到7000万美金融资, 不会介意大家的爬虫去光临他家的小站吧. 这个爬虫虽然可以勉强运行起来, 但是由于缺乏**异常处理**, 只能爬些静态页面, 也不会分辨什么是静态什么是动态, 碰到什么情况应该跳过, 所以工作一会儿就要败下阵来.

Python

```
import re

1 import re
2 import urllib.request
3 import urllib
4
5 from collections import deque
6
7 queue = deque()
8 visited = set()
9
10 url = 'http://news.dbanotes.net' # 入口页面, 可以换成别的
11
12 queue.append(url)
13 cnt = 0
14
15 while queue:
16     url = queue.popleft() # 队首元素出队
17     visited |= {url} # 标记为已访问
18
19     print('已经抓取: ' + str(cnt) + ' 正在抓取 <--- ' + url)
20     cnt += 1
21     url = urllib.request.urlopen(url)
22     if 'html' not in url.getheader('Content-Type'):
23         continue
24
25     # 避免程序异常中止, 用try..catch处理异常
26     try:
27         data = url.read().decode('utf-8')
28     except:
29         continue
30
31     # 正则表达式提取页面中所有队列, 并判断是否已经访问过, 然后加入待爬队列
32     linkre = re.compile('href="(.*?)\"')
33     for x in linkre.findall(data):
34         if 'http' in x and x not in visited:
35             queue.append(x)
36             print('加入队列 ----> ' + x)
```

这个版本的爬虫使用的正则表达式是

Python

```
'href="(.*?)\"'
1 'href="(.*?)\"'
```

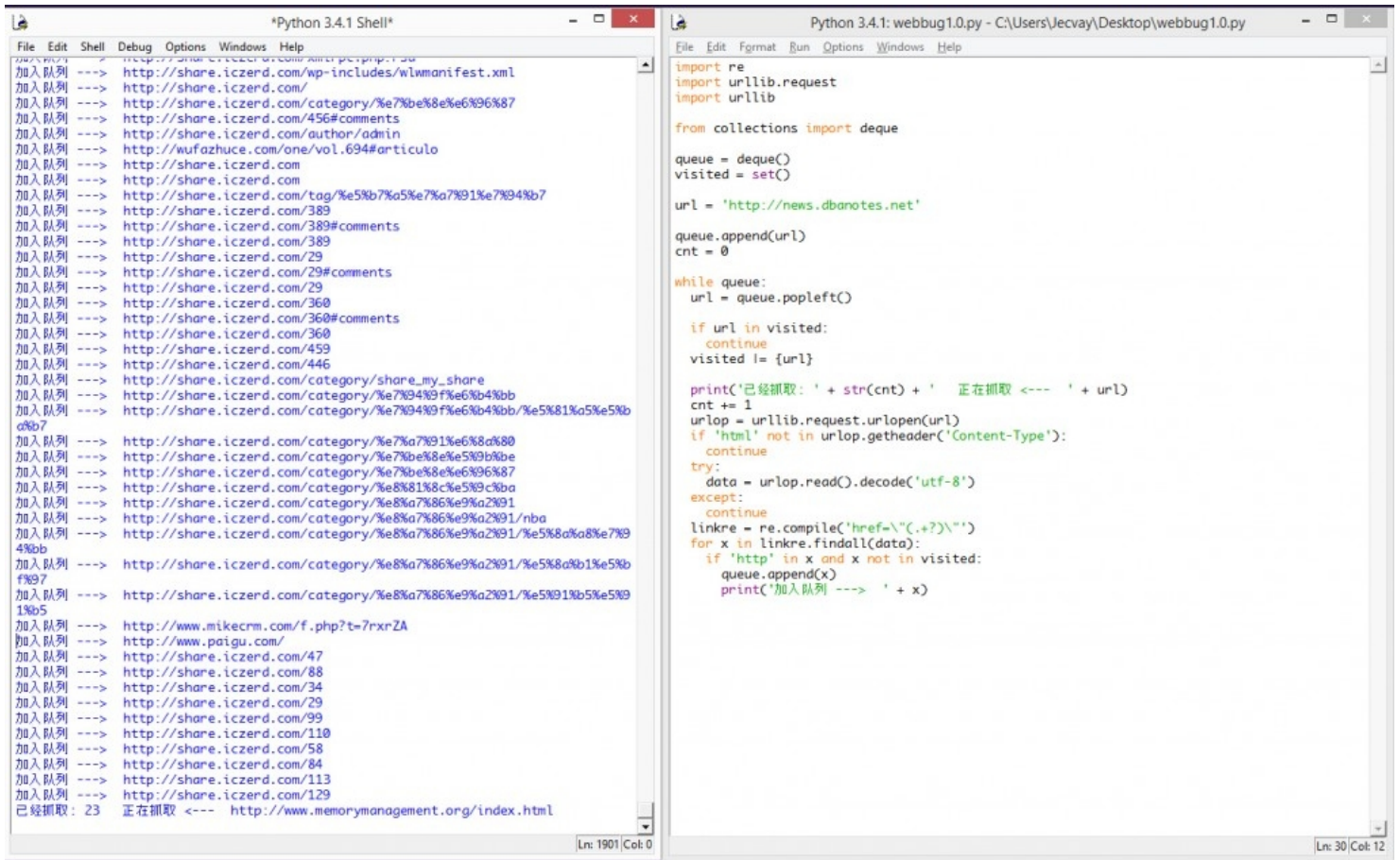
所以会把那些.ico或者.jpg的链接都爬下来. 这样read()了之后碰上decode('utf-8')就要抛出异常. 因此我们用[getheader\(\)](#)函数来获取抓取到的文件类型, 是html再继续分析其中的链接.

Python

```
if 'html' not in
```

```
1 if 'html' not in urlup.getheader('Content-Type'):
2     continue
```

但是即使是这样, 依然有些网站运行decode()会异常. 因此我们把decode()函数用try..catch语句包围住, 这样他就不会导致程序中止. 程序运行效果图如下:



下回预告

爬虫是可以工作了, 但是在碰到连不上的链接的时候, 它并不会超时跳过. 而且爬到的内容并没有进行处理, 没有获取对我们有价值的信息, 也没有保存到本地. 下次我们可以完善这个alpha版本.

1 赞 5 收藏 [评论](#)

零基础自学用Python 3开发网络爬虫(一)

原文出处：[Jecvay Notes \(@Jecvay\)](#)

由于本学期好多神都选了Cisco网络课,而我这等弱渣没选,去蹭了一节发现讲的内容虽然我不懂但是还是无爱.我想既然都本科就出来工作还是按照自己爱好来点技能吧,于是我就不去了.一个人在宿舍没有点计划好的事情做就会很容易虚度,正好这个学期主打网络与数据库开发,那就先学学Python开发爬虫吧.我失散多年的好朋友Jay Loong突然说他会爬虫了,我感到真棒,我也要学:D 因为一个星期有两节Cisco课,所以本系列博文也就一周两更。

选择一门语言

爬虫可以用各种语言写, C++, Java都可以, 为什么要Python? 首先用C++搞网络开发的例子不多(可能是我见得太少), 然后由于Oracle收购了Sun, Java目前虽然在Android开发上很重要, 但是如果Google官司进展不顺利, 那么很有可能用Go语言替代掉Java来做Android开发. 在这计算机速度高速增长年代里, 选语言都要看他爹的业绩, 真是稍不注意就落后于时代. 随着计算机速度的高速发展, 某种语言开发的软件运行的时间复杂度的常数系数已经不像以前那么重要, 我们可以越来越偏爱为程序员打造的而不是为计算机打造的语言. 比如Ruby这种传说中的纯种而又飘逸的的OOP语言, 或者Python这种稍严谨而流行库又非常多的语言, 都大大弱化了针对计算机运行速度而打造的特性, 强化了为程序员容易思考而打造的特性. 所以我选择Python。

选择Python版本

有2和3两个版本, 3比较新, 听说改动大. 根据我在知乎上搜集的观点来看, 我还是倾向于使用"在趋势中将会越来越火"的版本, 而非"目前已经很稳定而且很成熟"的版本. 这是个人喜好, 而且预测不一定准确. 但是如果Python3无法像Python2那么火, 那么整个Python语言就不可避免的随着时间的推移越来越落后, 因此我想其实选哪个的最坏风险都一样, 但是最好回报却是Python3的大. 其实两者区别也可以说大也可以说不大, 最终都不是什么大问题. 我选择的是Python 3。

选择参考资料

由于我是一边学一边写, 而不是我完全学会了之后才开始很有条理的写, 所以参考资料就很重要(本来应该是个人开发经验很重要, 但我是零基础).

- [Python官方文档](#)
- [知乎相关资料\(1\)](#) 这篇非常好, 通俗易懂的总览整个Python学习框架.
- [知乎相关资料\(2\)](#)

写到这里的时候, 上面第二第三个链接的票数第一的回答已经看完了, 他们提到的有些部分(比如爬行的路线不能有回路)我就不写了.

一个简单的伪代码

以下这个简单的伪代码用到了set和queue这两种经典的数据结构, 集与队列. 集的作用是记录那些已经访问过的页面, 队列的作用是进行广度优先搜索.

Python

```
queue Q
set S

1 queue Q
2 set S
3 StartPoint = "http://jecvay.com"
4 Q.push(StartPoint) # 经典的BFS开头
5 S.insert(StartPoint) # 访问一个页面之前先标记他为已访问
6 while (Q.empty() == false) # BFS循环体
7     T = Q.top() # 并且pop
8     for point in PageUrl(T) # PageUrl(T)是指页面T中所有url的集合, point是这个集合中的一个元素.
9         if (point not in S)
10             Q.push(point)
11             S.insert(point)
```

这个伪代码不能执行, 我觉得我写的有的不伦不类, 不类Python也不类C++.. 但是我相信看懂是没问题的, 这就是个最简单的BFS结构. 我是看了知乎里面的那个伪代码之后, 自己用我的风格写了一遍. 你也需要用你的风格写一遍.

这里用到的Set其内部原理是采用了Hash表, 传统的Hash对爬虫来说占用空间太大, 因此有一种叫做[Bloom Filter](#)的数据结构更适合用在这里替代Hash版本的set. 我打算以后再看这个数据结构怎么使用, 现在先跳过, 因为对于零基础的我来说, 这不是重点.

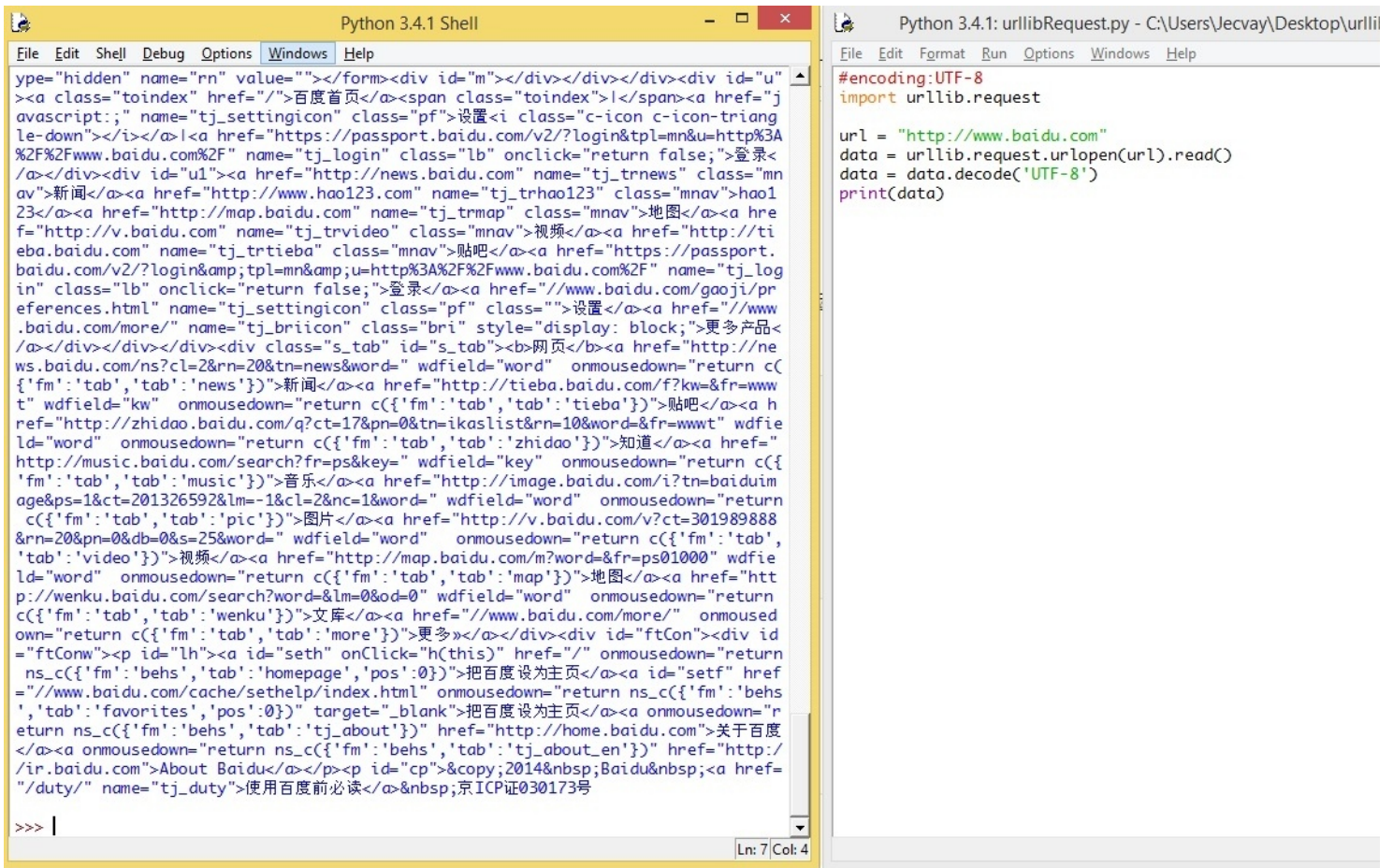
代码实现(一): 用Python抓取指定页面

我使用的编辑器是Idle, 安装好Python3后这个编辑器也安装好了, 小巧轻便, 按一个F5就能运行并显示结果. 代码如下:

Python

```
#encoding:UTF-8
import urllib.request

1 #encoding:UTF-8
2 import urllib.request
3
4 url = "http://www.baidu.com"
5 data = urllib.request.urlopen(url).read()
6 data = data.decode('UTF-8')
7 print(data)
```



urllib.request是一个库, 隶属urllib. [点此打开官方相关文档](#). 官方文档应该怎么使用呢? 首先刚才刚刚提到的这个链接进去的页面有urllib的几个子库, 我们暂时用到了request, 所以我们先看urllib.request部分. 首先看到的是一句话介绍这个库是干什么用的:

The urllib.request module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

然后把我们代码中用到的urlopen()函数部分阅读完.

urllib.request.urlopen(url, data=None, [timeout,], *, cafile=None, capath=None, cadefault=False)

重点部分是返回值, 这个函数返回一个 http.client.HTTPResponse 对象, 这个对象又有各种方法, 比如我们用到的read()方法, 这些方法都可以[根据官方文档的链接链过去](#). 根据官方文档所写, 我用控制台运行完毕上面这个程序后, 又继续运行如下代码, 以更熟悉这些乱七八糟的方法是干什么的.

Python

```
>>> a =
urllib.request.urlopen(full
1 >>> a = urllib.request.urlopen(full_url)
2 >>> type(a)
3 <class 'http.client.HTTPResponse'>
4
5 >>> a.geturl()
6 'http://www.baidu.com/s?word=Jecvay'
7
8 >>> a.info()
9 <http.client.HTTPMessage object at 0x03272250>
10
11 >>> a.getcode()
12 200
```

代码实现(二): 用Python简单处理URL

如果要抓取百度上面搜索关键词为Jecvay Notes的网页, 则代码如下

Python

```
import urllib
import urllib.request
1 import urllib
2 import urllib.request
3
4 data={}
5 data['word']='Jecvay Notes'
6
```

```
7 url_values=urllib.parse.urlencode(data)
8 url="http://www.baidu.com/s?"
9 full_url=url+url_values
10
11 data=urllib.request.urlopen(full_url).read()
12 data=data.decode('UTF-8')
13 print(data)
```

data是一个字典, 然后通过urllib.parse.urlencode()来将data转换为 'word=Jecvay+Notes'的字符串, 最后和url合并为full_url, 其余和上面那个最简单的例子相同. 关于urlencode(), 同样通过官方文档学习一下他是干什么的. 通过查看

1. [urllib.parse.urlencode\(query, doseq=False, safe=", encoding=None, errors=None\)](#)
2. [urllib.parse.quote_plus\(string, safe=", encoding=None, errors=None\)](#)

大概知道他是把一个通俗的字符串, 转化为url格式的字符串.

1 赞 18 收藏 [1 评论](#)

实例详解Django的 select_related 和 prefetch_related 函数对 QuerySet 查询的优化 (一)

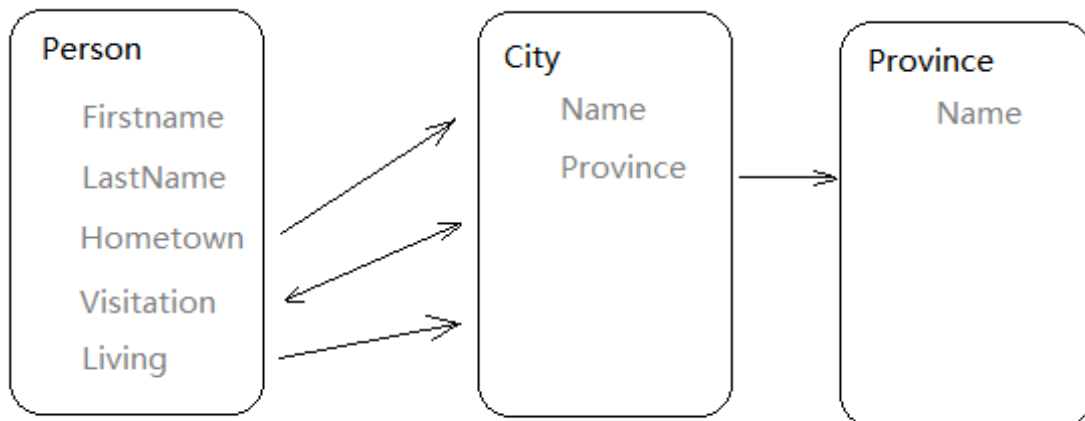
本文作者：[伯乐在线 - 熊锋](#)。未经作者许可，禁止转载！
欢迎加入伯乐在线 [专栏作者](#)。

在数据库有外键的时候，使用 select_related() 和 prefetch_related() 可以很好的减少数据库请求的次数，从而提高性能。本文通过一个简单的例子详解这两个函数的作用。虽然QuerySet的文档中已经详细说明了，但本文试图从QuerySet触发的SQL语句来分析工作方式，从而进一步了解Django具体的运作方式。

本来打算写成一篇单独的文章的，但是写完select_related()之后发现长度已经有点长了，所以还是写成系列，大概在两到三篇。整个完成之后将会在这里添加上其他文章的链接。

1. 实例的背景说明

假定一个个人信息系统，需要记录系统中各个人的故乡、居住地、以及到过的城市。数据库设计如下：



<http://blog.csdn.net/CuGBabyBear>

Models.py 内容如下：

Python

```
from django.db import models

1  from django.db import models
2
3  class Province(models.Model):
4      name = models.CharField(max_length=10)
5      def __unicode__(self):
6          return self.name
7
8  class City(models.Model):
9      name = models.CharField(max_length=5)
10     province = models.ForeignKey(Province)
```

```

11 def __unicode__(self):
12     return self.name
13
14 class Person(models.Model):
15     firstname = models.CharField(max_length=10)
16     lastname = models.CharField(max_length=10)
17     visitation = models.ManyToManyField(City, related_name = "visitor")
18     hometown = models.ForeignKey(City, related_name = "birth")
19     living = models.ForeignKey(City, related_name = "citizen")
20     def __unicode__(self):
21         return self.firstname + self.lastname

```

注1：创建的app名为“QSOptimize”

注2：为了简化起见，qsoptimize_province 表中只有2条数据：湖北省和广东省，qsoptimize_city表中只有三条数据：武汉市、十堰市和广州市

2. select_related()

对于一对一字段（OneToOneField）和外键字段（ForeignKey），可以使用select_related 来对QuerySet进行优化

作用和方法

在对QuerySet使用select_related()函数后，Django会获取相应外键对应的对象，从而在之后需要的时候不必再查询数据库了。以上例说明，如果我们需要打印数据库中的所有市及其所属省份，最直接的做法是：

Python

```

>>> citys =
City.objects.all()
1 >>> citys = City.objects.all()
2 >>> for c in citys:
3 ...     print c.province
4 ...

```

这样会导致线性的SQL查询，如果对象数量n太多，每个对象中有k个外键字段的话，就会导致n*k+1次SQL查询。在本例中，因为有3个city对象就导致了4次SQL查询：

MySQL

```

SELECT
`QSOptimize_city`.`id`,
1 SELECT `QSOptimize_city`.`id`, `QSOptimize_city`.`name`, `QSOptimize_city`.`province_id`
2 FROM `QSOptimize_city`
3
4 SELECT `QSOptimize_province`.`id`, `QSOptimize_province`.`name`
5 FROM `QSOptimize_province`
6 WHERE `QSOptimize_province`.`id` = 1 ;
7
8 SELECT `QSOptimize_province`.`id`, `QSOptimize_province`.`name`
9 FROM `QSOptimize_province`
10 WHERE `QSOptimize_province`.`id` = 2 ;
11

```

```

12 SELECT `QSOptimize_province`.`id`, `QSOptimize_province`.`name`
13 FROM `QSOptimize_province`
14 WHERE `QSOptimize_province`.`id` = 1 ;

```

注：这里的SQL语句是直接从Django的logger:‘django.db.backends’输出出来的

如果我们使用select_related()函数：

Python

```

>>> citys =
City.objects.select_relat
1 >>> citys = City.objects.select_related().all()
2 >>> for c in citys:
3 ...     print c.province
4 ...

```

就只有一次SQL查询，显然大大减少了SQL查询的次数：

MySQL

```

SELECT
`QSOptimize_city`.`id`,
1 SELECT `QSOptimize_city`.`id`, `QSOptimize_city`.`name`,
2 `QSOptimize_city`.`province_id`, `QSOptimize_province`.`id`, `QSOptimize_province`.`name`
3 FROM `QSOptimize_city`
4 INNER JOIN `QSOptimize_province` ON (`QSOptimize_city`.`province_id` = `QSOptimize_province`.`id`) ;

```

这里我们可以看到，Django使用了INNER JOIN来获得省份的信息。顺便一提这条SQL查询得到的结果如下：

Python

```

+-----+-----+-----+-----+
1 +---+-----+-----+-----+
2 | id | name      | province_id | id | name      |
3 +---+-----+-----+-----+
4 | 1 | 武汉市    | 1 | 1 | 湖北省    |
5 | 2 | 广州市    | 2 | 2 | 广东省    |
6 | 3 | 十堰市    | 1 | 1 | 湖北省    |
7 +---+-----+-----+-----+
8 3 rows in set (0.00 sec)

```

使用方法

函数支持如下三种用法：

***fields 参数**

select_related() 接受可变长参数，每个参数是需要获取的外键（父表的内容）的字段名，以及外键的外键的字段名、外键的外键的外键…。若要选择外键的外键需要使用两个下划线“__”来连接。

例如我们要获得张三的现居省份，可以用如下方式：

Python

```
>>> zhangs =
1 >>> zhangs = Person.objects.select_related('living__province').get(firstname=u"张",lastname=u"三")
2 >>> zhangs.living.province
```

触发的SQL查询如下：

MySQL

```
SELECT
`QSOptimize_person`.`i
1 SELECT `QSOptimize_person`.`id`, `QSOptimize_person`.`firstname`,
2 `QSOptimize_person`.`lastname`, `QSOptimize_person`.`hometown_id`, `QSOptimize_person`.`living_id`,
3 `QSOptimize_city`.`id`, `QSOptimize_city`.`name`, `QSOptimize_city`.`province_id`, `QSOptimize_province`.`id`,
4 `QSOptimize_province`.`name`
5 FROM `QSOptimize_person`
6 INNER JOIN `QSOptimize_city` ON (`QSOptimize_person`.`living_id` = `QSOptimize_city`.`id`)
7 INNER JOIN `QSOptimize_province` ON (`QSOptimize_city`.`province_id` = `QSOptimize_province`.`id`)
8 WHERE (`QSOptimize_person`.`lastname` = '三' AND `QSOptimize_person`.`firstname` = '张');
```

可以看到，Django使用了2次 INNER JOIN 来完成请求，获得了city表和province表的内容并添加到结果表的相应列，这样在调用 zhangs.living的时候也不必再次进行SQL查询。

Python

```
+-----+-----+-----+-----+-----+-----+-----+-----+
1 | id | firstname | lastname | hometown_id | living_id | id | name | province_id | id | name |
2 |-----+-----+-----+-----+-----+-----+-----+-----+
3 | 1 | 张 | 三 | 3 | 1 | 1 | 武汉市 | 1 | 1 | 湖北省 |
4 |-----+-----+-----+-----+-----+-----+-----+-----+
5 1 row in set (0.00 sec)
```

然而，未指定的外键则不会被添加到结果中。这时候如果需要获取张三的故乡就会进行SQL查询了：

Python

```
>>>
zhangs.hometown.provi
1 >>> zhangs.hometown.province
```

MySQL

```
SELECT
`QSOptimize_city`.`id`,
1 SELECT `QSOptimize_city`.`id`, `QSOptimize_city`.`name`,
2 `QSOptimize_city`.`province_id`
3 FROM `QSOptimize_city`
4 WHERE `QSOptimize_city`.`id` = 3 ;
5
6 SELECT `QSOptimize_province`.`id`, `QSOptimize_province`.`name`
```

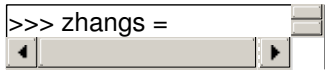


```
7 FROM `QSOptimize_province`  
8 WHERE `QSOptimize_province`.`id` = 1
```

同时，如果不指定外键，就会进行两次查询。如果深度更深，查询的次数更多。

值得一提的是，从Django 1.7开始，`select_related()`函数的作用方式改变了。在本例中，如果要同时获得张三的故乡和现居地的省份，在1.7以前你**只能**这样做：

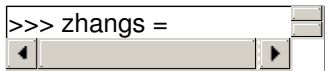
Python



```
>>> zhangs =  
1 >>> zhangs =  
2 Person.objects.select_related('hometown__province','living__province').get(firstname=u"张",lastname=u"三")  
3 >>> zhangs.hometown.province  
>>> zhangs.living.province
```

但是1.7及以上版本，你可以像和`queryset`的其他函数一样进行链式操作：

Python



```
>>> zhangs =  
1 >>> zhangs =  
2 Person.objects.select_related('hometown__province').select_related('living__province').get(firstname=u"张",lastname=u"三")  
3 >>> zhangs.hometown.province  
>>> zhangs.living.province
```

如果你在1.7以下版本这样做了，你只会获得最后一个操作的结果，在本例中就是只有现居地而没有故乡。在你打印故乡省份的时候就会造成两次SQL查询。

depth 参数

`select_related()` 接受`depth`参数，`depth`参数可以确定`select_related`的深度。Django会递归遍历指定深度内的所有的`OneToOneField`和`ForeignKey`。以本例说明：

Python



```
>>> zhangs =  
Person.objects.select_r  
1 >>> zhangs = Person.objects.select_related(depth = d)
```

`d=1` 相当于 `select_related('hometown','living')`

`d=2` 相当于 `select_related('hometown__province','living__province')`

无参数

`select_related()` 也可以不加参数，这样表示要求Django尽可能深的`select_related`。例如：`zhangs = Person.objects.select_related().get(firstname=u"张",lastname=u"三")`。但要注意两点：

1. Django本身内置一个上限，对于特别复杂的表关系，Django可能在你不知道的某处跳出递归，从而与你想要的做法不一样。具体限制是怎么工作的我表示不清楚。

2. Django并不知道你实际要用的字段有哪些，所以会把所有的字段都抓进来，从而会造成不必要的浪费而影响性能。

小结

1. select_related主要针一对一和多对一关系进行优化。
2. select_related使用SQL的JOIN语句进行优化，通过减少SQL查询的次数来进行优化、提高性能。
3. 可以通过可变长参数指定需要select_related的字段名。也可以通过使用双下划线“__”连接字段名来实现指定的递归查询。没有指定的字段不会缓存，没有指定的深度不会缓存，如果要访问的话Django会再次进行SQL查询。
4. 也可以通过depth参数指定递归的深度，Django会自动缓存指定深度内所有的字段。如果要访问指定深度外的字段，Django会再次进行SQL查询。
5. 也接受无参数的调用，Django会尽可能深的递归查询所有的字段。但注意有Django递归的限制和性能的浪费。
6. Django \geq 1.7，链式调用的select_related相当于使用可变长参数。Django $<$ 1.7，链式调用会导致前边的select_related失效，只保留最后一个。

1 赞 收藏 [评论](#)

关于作者：[熊锋](#)



野生业余程序员搞搞Android 玩玩前端 仅此而已 [个人主页](#) · [我的文章](#) · 19 ·

可爱的 Python : Python中的函数式编程，第三部分

英文原文：[Charming Python: Functional programming in Python, Part 3](#)，翻译：[开源中国](#)

摘要： 作者David Mertz在其文章《可爱的Python：“Python中的函数式编程”》中的[第一部分](#)和[第二部分](#)中触及了函数式编程的大量基本概念。本文中他将继续前面的讨论，解释函数式编程的其它功能，如currying和Xoltar Toolkit中的其它一些高阶函数。

表达式绑定

有一位从不满足于解决部分问题读者，名叫Richard Davies，提出了一个问题，问是否可以将所有的绑定全部都转移到一个单个的表达式之中。首先让我们简单看看，我们为什么想这么做，然后再看看由comp.lang.python中的一位朋友提供的一种异常优雅地写表达式的方式。

让我们回想一下功能模块的绑定类。使用该类的特性，我们可以确认在一个给定的范围块内，一个特定的名字仅代表了一个唯一的事物。

具有重新绑定向导的 Python 函数式编程(FP)

Python

```
>>> from functional
import *

1 >>> from functional import *
2 >>> let = Bindings()
3 >>> let.car = lambda lst: lst[0]
4 >>> let.car = lambda lst: lst[2]
5 Traceback (innermost last):
6   File "<stdin>", line 1, in ?
7   File "d:\tools\functional.py", line 976, in __setattr__
8
9   raise BindingError, "Binding '%s' cannot be modified." % name
10 functional.BindingError: Binding 'car' cannot be modified.
11 >>> let.car(range(10))
12 0
```

绑定类在一个模块或者一个功能定义范围内做这些我们希望的事情，但是没有办法在一条表达式内使之工作。然而在ML家族语言(译者注：ML是一种通用的函数式编程语言),在一条表达式内创建绑定是很自然的事。

Haskell 命名绑定表达式

Python

```
-- car (x:xs) = x --
*could* create module-

1 -- car (x:xs) = x -- *could* create module-level binding
2 list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
3
4 -- 'where' clause for expression-level binding
5 firsts1 = [car x | x <- list_of_list] where car (x:xs) = x
```

```

6
7 -- 'let' clause for expression-level binding
8 firsts2 = let car (x:xs) = x in [car x | x <- list_of_list]
9
10 -- more idiomatic higher-order 'map' technique
11 firsts3 = map car list_of_list where car (x:xs) = x
12
13 -- Result: firsts1 == firsts2 == firsts3 == [1,4,7]

```

Greg Ewing 发现用Python的list概念实现同样的效果是有可能的；甚至我们可以用几乎与Haskell语法一样干净的方式做到。

Python 2.0+ 命名绑定表达式

Python

```

>>> list_of_list =
[[1,2,3],[4,5,6],[7,8,9]]
1 >>> list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
2 >>> [car_x for x in list_of_list for car_x in
3 (x[0],)]
4 [1, 4, 7]

```

在列表解析（list comprehension）中将表达式放入一个单项元素（a single-item tuple）中的这个小技巧，并不能为使用带有表达式级绑定的高阶函数提供任何思路。要使用这样的高阶函数，还是需要使用块级（block-level）绑定，就象以下所示：

Python中的使用块级绑定的'map()'

Python

```

>>> list_of_list =
[[1,2,3],[4,5,6],[7,8,9]]
1 >>> list_of_list = [[1,2,3],[4,5,6],[7,8,9]]
2 >>> let = Bindings()
3 >>> let.car = lambda l: l[0]
4 >>> map(let.car,list_of_list)
5 [1, 4, 7]

```

这样真不错，但如果我们想使用函数map()，那么其中的绑定范围可能会比我们想要的更宽一些。然而，我们可以做到的，哄骗列表解析让它替我们做名字绑定，即使其中的列表并不是我们最终想要得到的列表的情况下也没问题：

从Python的列表解析中“走下舞台”

Python

```

# Compare Haskell
expression:
1 # Compare Haskell expression:
2 # result = func car_car
3 #     where
4 #         car (x:xs) = x
5 #         car_car = car (car list_of_list)
6 #         func x = x + x^2
7 >>> [func for x in list_of_list
8 ...
9 for car in (x[0],)]

```

```
10 ...
11 for func in (car+car**2,))[0]
12 2
```

我们对list_of_list列表中第一个元素的第一个元素进行了一次算数运算，而且期间还对该算术运算进行了命名（但其作用域仅仅是在表达式的范围内）。作为一种“优化”，我们可以不用费心创建多于一个元素的列表就能开始运算了，因为我们结尾处用的索引为0，所以我们仅仅选择的是第一个元素。

:

从列表解析中高效地走下舞台

Python

```
# Compare Haskell
expression:
1 # Compare Haskell expression:
2 # result = func car_car
3 #     where
4 #         car (x:xs) = x
5 #         car_car = car (car list_of_list)
6 #         func x = x + x^2
7 >>> [func for x in list_of_list
8 ...
9 for car in (x[0],)
10 ...
11 for func in (car+car**2,))[0]
12 2
```

高阶函数：currying

Python内建的三个最常用的高阶函数是：map()、reduce()和filter()。这三个函数所做的事情——以及谓之“高阶”（higher-order）的原因——是接受其它函数作为它们的（部分）参数。还有一些不属于内置的高阶函数，还会返回函数对象。

藉由函数对象在Python中具有首要地位，Python一直都有能让其使用者构造自己的高阶函数的能力。举个如下所示的小例子：

Python中一个简单函数工厂（function factory）

Python

```
>>> def
foo_factory():
1 >>> def
2 foo_factory():
3 ...
4 def
5 foo():
6 ...
7 print
8 "Foo function from factory"
9 ...
10 return foo
11 ...
12 >>> f = foo_factory()
13 >>> f()
14 Foo function from factory
```

本系列文章的第二部分我讨论过的Xoltar Toolkit中，有一组非常好用的高阶函数。Xoltar的functional模块中提供的绝大多数高阶函数都是在其它各种不同的传统型函数式编程语言中发展出来的高阶函数，其有用性已经过多年的实践验证。

可能其中最著名、最有用和最重要的高阶函数要数curry()了。函数curry()的名字取自于逻辑学家Haskell Curry，前文提及的一种编程语言也是用他姓名当中的名字部分命名的。”currying”背后隐含的意思是，（几乎）每一个函数都可以视为只带一个参数的部分函数（partial function）。要使currying能够用起来所需要做的就是让函数本身的返回值也是个函数，只不过所返回的函数“缩小了范围”或者是“更加接近完整的函数”。这和我在第二部分中提到的闭包特别相似——对经过curry后的返回的后继函数进行调用时一步一步“填入”最后计算所需的更多数据（附加到一个过程（procedure）之上的数据）

现在让我们先用Haskell中一个很简单例子对curry进行讲解，然后在Python中使用functional模块重复展示一下这个简单的例子：

在Haskell计算中使用Curry

Python

```
computation a b c d = (a
+ b^2+ c^3 + d^4)
1 computation a b c d = (a + b^2+ c^3 + d^4)
2 check = 1 + 2^2 + 3^3 + 5^4
3
4 fillOne = computation 1
5 -- specify "a"
6 fillTwo = fillOne 2
7 -- specify "b"
8 fillThree = fillTwo 3
9 -- specify "c"
10 answer = fillThree 5
11 -- specify "d"
12
13 -- Result: check == answer == 657
```

现在使用Python：

在Python计算中使用Curry

Python

```
>>> from functional
1 >>> from functional import curry
2 >>> computation = lambda a,b,c,d: (a + b**2 + c**3 + d**4)
3 >>> computation(1,2,3,5)
4 657
5 >>> fillZero = curry(computation)
6 >>> fillOne = fillZero(1)
7 # specify "a"
8 >>> fillTwo = fillOne(2)
9 # specify "b"
10 >>> fillThree = fillTwo(3)
11 # specify "c"
12 >>> answer = fillThree(5)
13 # specify "d"
14 >>> answer
```

第二部分中提到过的一个简单的计税程序的例子，当时用的是闭包（这次使用`curry()`），可以用来进一步做个对比：

Python中`curry`后的计税程序

Python

```
from functional import *

1 from functional import *
2
3 taxcalc = lambda income,rate,deduct: (income-(deduct))*rate
4
5 taxCurry = curry(taxcalc)
6 taxCurry = taxCurry(50000)
7 taxCurry = taxCurry(0.30)
8 taxCurry = taxCurry(10000)
9 print "Curried taxes due =",taxCurry
10
11 print "Curried expression taxes due =", \
12     curry(taxcalc)(50000)(0.30)(10000)
```

和使用闭包不同，我们需要以特定的顺序（从左到右）对参数进行`curry`处理。当要注意的是，`functional`模块中还包含一个`rcurry()`类，能够以相反的方向进行`curry`处理（从右到左）。从一个层面讲，其中的第二个`print`语句同简单的同普通的`taxcalc(50000,0.30,10000)`函数调用相比只是个微小的拼写方面的变化。但从另一个不同的层面讲，它清晰地一个概念，那就是，每个函数都可以变换成仅仅带有一个参数的函数，这对于刚刚接触这个概念的人来讲，会有一种特别惊奇的感觉。

其它高阶函数

除了上述的`curry`功能，`functional`模块简直就是一个很有意思的高阶函数万能口袋。此外，无论用还是不用`functional`模块，编写你自己的高阶函数真的并不难。至少`functional`模块中的那些高阶函数为你提供了一些很值一看的思路。

它里面的其它高阶函数在很大程度上感觉有点象是“增强”版本的标准高阶函数`map()`、`filter()`和`reduce()`。这些函数的工作模式通常大致如此：将一个或多个函数以及一些列表作为参数接收进来，然后对这些列表参数运行它前面所接收到的函数。在这种工作模式方面，有非常大量很有意思也很有用的摆弄方法。还有一种模式是：拿到一组函数后，将这组函数的功能组合起来创建一个新函数。这种模式同样也有大量的变化形式。下面让我们看看`functional`模块里到底还有哪些其它的高阶函数。

`sequential()`和`also()`这两个函数都是在一系列成分函数（component function）的基础上创建一个新函数。然后这些成分函数可以通过使用相同的参数进行调用。两者的主要区别就在于，`sequential()`需要一个单个的函数列表作为参数，而`also()`接受的是一系列的多个参数。在多数情况下，对于函数的副作用而已这些会很有用，只是`sequential()`可以让你随意选择将哪个函数的返回值作为组合起来后的新函数的返回值。

顺序调用一系列函数(使用相同的参数)

Python


```

>>> def a(x):
...     print x,
1  >>> def a(x):
2  ...     print x,
3  ...     return "a"
4  ...
5  >>> def b(x):
6  ...     print x*2,
7  ...     return "b"
8  ...
9  >>> def c(x):
10 ...     print x*3,
11 ...     return "c"
12 ...
13 >>> r = also(a,b,c)
14 >>> r
15 <functional.sequential instance at 0xb86ac>
16 >>> r(5)
17 5 10 15
18 'a'
19 >>> sequential([a,b,c],main=c)('x')
20 x xx xxx
21 'c'

```

isjoin()和conjoin()这两个函数同equential()和also()在创建新函数并对参数进行多个成分函数的调用方面非常相似。只是disjoin()函数用来查询成分函数中是否有一个函数的返回值（针对给定的参数）为真；conjoin()函数用来查询是否所有的成分函数的返回值都为真。在这些函数中只要条件允许就会使用逻辑短路，因此disjoin()函数可能不会出现某些副作用。joinfuncs()同also()类似，但它返回的是由所有成分函数的返回值组成的一个元组（tuple），而不是选中的某个主函数。

前文所述的几个函数让你可以使用相同的参数对一系列函数进行调用，而any()、all()和 none_of()这三个让你可以使用一个参数列表对同一个函数进行多次调用。在大的结构方面，这些函数同内置的map()、reduce()和filter()有点象。但funtional模块中的这三个高阶函数中都是对一组返回值进行布尔（boolean）运算得到其返回值的。例如：

对一系列返回值的真、假情况进行判断

Python

```

>>> from functional
import *
1  >>> from functional import *
2  >>> isEven = lambda n: (n%2 == 0)
3  >>> any([1,3,5,8], isEven)
4  1
5  >>> any([1,3,5,7], isEven)
6  0
7  >>> none_of([1,3,5,7], isEven)
8  1
9  >>> all([2,4,6,8], isEven)
10 1
11 >>> all([2,4,6,7], isEven)
12 0

```

有点数学基础的人会对这个高阶函数非常感兴趣：iscompose()。将多个函数进行合成（compostion）指的是，将一个函数的返回值同下个函数的输入“链接到一起”。对多个函数进行合成的程序员需要负责保证函数间的输入和输出是相互匹配的，不过这个条件无论是程序员在何时想使用返回值时都是需

要满足的。举个简单的例子和阐明这一点：
创建合成函数

Python

```
>>> def minus7(n):  
return n-7
```

```
1 >>> def minus7(n): return n-7  
2 ...  
3 >>> def times3(n): return n*3  
4 ...  
5 >>> minus7(10)  
6 3  
7 >>> minustimes = compose(times3,minus7)  
8 >>> minustimes(10)  
9 9  
10 >>> times3(minus7(10))  
11 9  
12 >>> timesminus = compose(minus7,times3)  
13 >>> timesminus(10)  
14 23  
15 >>> minus7(times3(10))  
16 23
```

后会有期

衷心希望我对高阶函数的思考能够引起读者的兴趣。无论如何，请动手试一试。试着编写一些你自己的高阶函数；一些可能很有用，很强大。告诉我它如何运行；或许这个系列之后的章节会讨论读者不断提供的新观点，新想法。

1 赞 收藏 [评论](#)

可爱的 Python : Python中函数式编程，第二部分

英文原文：[Charming Python: Functional programming in Python, Part 2](#)，翻译：[开源中国](#)

摘要： 本专栏继续让David对Python中的函数式编程(FP)进行介绍。读完本文，可以享受到使用不同的编程范型（paradigm）解决问题所带来的乐趣。David在本文中对FP中的多个中级和高级概念进行了详细的讲解。

一个对象就是附有若干过程（procedure）的一段数据。。。一个闭包（closure）就是附有一段数据的一个过程（procedure）。

在我讲解函数式编程的上一篇文章，[第一部分](#)，中，我介绍了FP中的一些基本概念。 本文将更加深入的对这个内容十分丰富的概念领域进行探讨。在我们探讨的大部分内容中，Bryn Keller的“Xoltar Toolkit”为我们提供一些非常有价值的帮助作用。Keller将FP中的许多强项集中到了一个很棒且很小的模块中，他在这个模块中用纯Python代码实现了这些强项。除了 *functional* 模块外，Xoltar Toolkit 还包含了一个 *延迟 (lazy)* 模块，对“仅在需要时”才进行求值提供了支持。许多传统的函数式语言中也都具有延迟求值的手段，这样，使用Xoltar Toolkit中的这些组件，你就可以做到使用象Haskell这样的函数式语言能够做到的大部分事情了。

绑定 (Binding)

有心的读者会记得，我在第一部分中所述的函数式技术中指出过Python的一个局限。具体讲，就是Python中没有任何手段禁止对用来指代函数式表达式的名字进行重新绑定。在FP中，名字一般是理解为对比较长的表达式的简称，但这里面隐含了一个诺言，就是“同一个表达式总是具有同一个值”。如果对用来指代的名字重新进行绑定，就会违背这个诺言。例如，假如我们如以下所示，定义了一些要用在函数式程序中的简记表达式：

Python中由于重新绑定而引起问题的FP编程片段

Python

```
>>> car = lambda lst: lst[0]
```

```
1 >>> car = lambda lst: lst[0]
2 >>> cdr = lambda lst: lst[1:]
3 >>> sum2 = lambda lst: car(lst)+car(cdr(lst))
4 >>> sum2(range(10))
5 1
6 >>> car = lambda lst: lst[2]
7 >>> sum2(range(10))
8 5
```

非常不幸，程序中完全相同的表达式 `sum2(range(10))` 在两个不同的点求得的值却不相同，尽管在该表达式的参数中根本没有使用任何可变的（mutable）变量。

幸运的是， *functional* 模块提供了一个叫做 *Bindings* (由鄙人向Keller进行的提议，proposed to Keller by yours truly) 的类，可以用来避免这种重新绑定（至少可以避免意外的重新绑定，Python并不阻止任何拿定主意就是要打破规则的程序员）。尽管要用 *Bindings* 类就需要使用一些额外的语法，但这么做就能让这种事故不太容易发生。Keller在 *functional* 模块里给出的例子中，有个 *Bindings* 的实例名字叫做 *let*（我推测这么叫是为了仿照ML族语言中的 *let* 关键字）。例如，我们可以这么做：

Python中对重新绑定进行监视后的FP编程片段

Python

```
>>> from functional
import *

1 >>> from functional import *
2 >>> let = Bindings()
3 >>> let.car = lambda lst: lst[0]
4 >>> let.car = lambda lst: lst[2]
5 Traceback (innermost last):
6   File "<stdin>", line 1, in ?
7   File "d:\tools\functional.py", line 976, in __setattr__
8     raise BindingError, "Binding '%s' cannot be modified." % name
9 functional.BindingError: Binding 'car' cannot be modified.
10 >>> car(range(10))
11 0
```

显而易见，在真正的程序中应该去做一些事情，捕获这种“BindingError”异常，但发出这些异常这件事，就能够避免产生这一大类的问题。

functional模块随同Bindings一起还提供了一个叫做namespace的函数，这个函数从Bindings实例中弄出了一个命名空间（实际就是个字典）。如果你想计算一个表达式，而该表达式是在定义于一个Bindings中的一个（不可变）命名空间中时，这个函数就可以很方便地拿来使用。Python的eval()函数允许在命名空间中进行求值。举个例子就能说明这一切：

Python中使用不可变命名空间的FP编程片段

Python

```
>>> let = Bindings()
# "Real world" function

1 >>> let = Bindings()    # "Real world" function names
2 >>> let.r10 = range(10)
3 >>> let.car = lambda lst: lst[0]
4 >>> let.cdr = lambda lst: lst[1:]
5 >>> eval('car(r10)+car(cdr(r10))', namespace(let))
6 >>> inv = Bindings()    # "Inverted list" function names
7 >>> inv.r10 = let.r10
8 >>> inv.car = lambda lst: lst[-1]
9 >>> inv.cdr = lambda lst: lst[:-1]
10 >>> eval('car(r10)+car(cdr(r10))', namespace(inv))
11 17
```

FP中有一个特别有引人关注的概念叫做闭包。实际上，闭包充分引起了很多程序员的关注，即使通常意义上的非函数式编程语言，比如Perl和Ruby，都包含了闭包这一特性。此外，Python 2.1 目前一定会添加上词法域（lexical scoping），这样一来就提供的闭包的绝大多数功能。

那么，闭包到底是什么？Steve Majewski最近在Python新闻组中对这个概念的特性提出了一个准确的描述：

就是说，闭包就象是FP的Jekyll，OOP（面向对象编程）的Hyde（或者可能是将这两个角色互换）（译者注：Jekyll和Hyde是一部小说中的两个人物）。和象对象实例类似，闭包是一种把一堆数据和一些功能打包一起进行传递的手段。

先让我们后退一小步，看看对象和闭包都能解决一些什么样的问题，然后再看看在两样都不用的情况

下这些问题是如何得到解决的。函数返回的值通常是由它在计算过程中使用的上下文决定的。最常见可能也是最显然的指定该上下文的方式就是给函数传递一些参数，让该函数对这些参数进行一些运算。但有时候在参数的“背景”（background）和“前景”（foreground）两者之间也有一种自然的区分，也就是说，函数在某特定时刻正在做什么和函数“被配置”为处于多种可能的调用情况之下这两者之间有不同之处。

在集中处理前景的同时，有多种方式进行背景处理。一种就是“忍辱负重”，每次调用时都将函数需要的每个参数传递给函数。这通常就相对于在函数调用链中不断的将很多值（或者是一个具有很多字段的数据结构）传上传下，就是因为在链中的某个地方可能会用到这些值。下面举个简单的例子：

用了货船变量的Python代码片段

Python

```
>>> def a(n):  
...     add7 = b(n)  
  
1 >>> def a(n):  
2 ...     add7 = b(n)  
3 ...     return add7  
4 ...  
5 >>> def b(n):  
6 ...     i = 7  
7 ...     j = c(i,n)  
8 ...     return j  
9 ...  
10 >>> def c(i,n):  
11 ...     return i+n  
12 ...  
13 >>> a(10)    # Pass cargo value for use downstream  
14 17
```

在上述的货船变量例子中，函数b()中的变量n毫无意义，就只是为了传递给函数c()。另一种办法是使用全局变量：

使用全局变量的Python代码片段

Python

```
>>> N = 10  
>>> def addN(i):  
  
1 >>> N = 10  
2 >>> def addN(i):  
3 ...     global N  
4 ...     return i+N  
5 ...  
6 >>> addN(7)  # Add global N to argument  
7 17  
8 >>> N = 20  
9 >>> addN(6)  # Add global N to argument  
10 26
```

全局变量N只要你想调用ddN()就可以直接使用，就不需要显式地传递这个全局背景“上下文”了。有个稍微更加Python化的技巧，可以用来在定义函数时，通过使用缺省参数将一个变量“冻结”到该函数中：

使用冻结变量的Python代码片段

Python

```

>>> N = 10
>>> def addN(i, n=N):
1 >>> N = 10
2 >>> def addN(i, n=N):
3 ...     return i+n
4 ...
5 >>> addN(5) # Add 10
6 15
7 >>> N = 20
8 >>> addN(6) # Add 10 (current N doesn't matter)
9 16

```

我们冻结的变量实质上就是个闭包。我们将一些数据“附加”到了addN()函数之上。对于一个完整的闭包而言，在函数addN()定义时所出现的数据，应该在该函数被调用时也可以拿到。然而，本例中（以及更多更健壮的例子中），使用缺省参数让足够的数据可用非常简单。函数addN()不再使用的变量因而对计算结构捕获产生丝毫影响。

现在让我们再看一个用OOP的方式解决一个稍微更加现实的问题。今年到了这个时候，让我想起了颇具“面试”风格的计税程序，先收集一些数据，数据不一定有什么特别的顺序，最后使用所有这些数据进行一个计算。让我们为这种情况些个简化版本的程序：

Python风格的计税类/实例

Python

```

class TaxCalc:
    def
1 class TaxCalc:
2     def taxdue(self):return (self.income-self.deduct)*self.rate
3 taxclass = TaxCalc()
4 taxclass.income = 50000
5 taxclass.rate = 0.30
6 taxclass.deduct = 10000
7 print"Pythonic OOP taxes due =", taxclass.taxdue()

```

在我们的TaxCalc类 (或者更准确的讲，在它的实例中)，我们先收集了一些数据，数据的顺序随心所欲，然后所有需要的数据收集完成后，我们可以调用这个对象的一个方法，对这堆数据进行计算。所有的一切都呆在一个实例中，而且，不同的实例可以拥有一堆不同的数据。能够创建多个实例，而多个实例仅仅是数据不同，这通过“全局变量”和“冻结变量”这两种方法是无法办到的。”货船”方法能够做到这一点，但从那个展开的例子中我们能够看出，它可能不得不在开始时就传递多个数值。讨论到这里，注意到OOP风格的消息传递方式可能会如何解决这一问题会非常有趣(Smalltalk或者Self与此类似，我所用过的好几种xBase的变种OOP语言也是类似的)：

Smalltalk风格的(Python) 计税程序

Python

```

class TaxCalc:
    def
1 class TaxCalc:
2     def taxdue(self):return (self.income-self.deduct)*self.rate
3     def setIncome(self,income):
4         self.income = income

```



```

5     return self
6     def setDeduct(self,deduct):
7         self.deduct = deduct
8         return self
9     def setRate(self,rate):
10        self.rate = rate
11        return self
12 print"Smalltalk-style taxes due =", \
13     TaxCalc().setIncome(50000).setRate(0.30).setDeduct(10000).taxdue()

```

每个“setter”方法都返回self可以让我们将每个方法调用的结果当作“当前”对象进行处理。这和FP中的闭包方式有些相似。

通过使用Xoltar toolkit，我们可以生成完整的闭包，能够将数据和函数结合起来，获得我们所需的特性；另外还可以让多个闭包（以前成为对象）包含不同的数据：

Python的函数式风格的计税程序

Python

```

from functional import *

1  from functional import *
2
3  taxdue      = lambda: (income-deduct)*rate
4  incomeClosure = lambda income,taxdue: closure(taxdue)
5  deductClosure = lambda deduct,taxdue: closure(taxdue)
6  rateClosure  = lambda rate,taxdue: closure(taxdue)
7
8  taxFP = taxdue
9  taxFP = incomeClosure(50000,taxFP)
10 taxFP = rateClosure(0.30,taxFP)
11 taxFP = deductClosure(10000,taxFP)
12 print"Functional taxes due =",taxFP()
13
14 print"Lisp-style taxes due =", \
15     incomeClosure(50000,
16         rateClosure(0.30,
17             deductClosure(10000, taxdue))))()

```

我们所定义每个闭包函数可以获取函数定义范围内的任意值，然后将这些值绑定到改函数对象的全局范围之中。然而，一个函数的全局范围并不一定就是真正的模块全局范围，也和不同的闭包的“全局”范围不相同。闭包就是“将数据带”在了身边。

在我们的例子中，我们利用了一些特殊的函数把特定的绑定限定到了一个闭包作用范围之内(income, deduct, rate)。要想修改设计，将任意的绑定限定在闭包之中，也非常简单。只是为了好玩，在本例子中我们也使用了两种稍微不同的函数式风格。第一种风格连续将多个值绑定到了闭包的作用范围；通过允许taxFP成为可变的变量，这些“添加绑定”的代码行可以任意顺序出现。然而，如果我们想要使用tax_with_Income这样的不可变名字，我们就需要以特定的顺序来安排这几行进行绑定的代码，将靠前的绑定结果传递给下一个绑定。无论在哪种情况下，在全部所需数据都绑定进闭包范围之后，我们就可以调用“种子”（seeded）方法了。

第二种风格在我看来，更象是Lisp（那些括号最象了）。除去美学问题，这第二种风格有两点值得注意。第一点就是完全避免了名字绑定，变成了一个单个的表达式，连语句都没有使用（关于为什么不使用语句很重要，请参见 P第一部分）。

第二点是闭包的“Lips”风格的用法和前文给出的“Smalltalk”风格的信息传递何其相似。实际上两者都

在调用taxdue()函数/方法的过程中积累了所有值(如果以这种原始的方式拿不到正确的数据，两种方式都会出错)。“Smalltalk”风格的方法中每一步传递的是一个对象，而“Lisp”风格的方法中传递是持续进行的。但实际上，函数式编程和面向对象式编程两者旗鼓相当。

在本文中，我们干掉了函数式编程领域中更多的内容。剩下的要比以前（本小节的题目是个小玩笑；很不幸，这里还没有解释过尾递归的概念）少多了（或者可以证明也简单多了？）。阅读functional模块中的源代码是继续探索FP中大量概念的一种非常好的方法。该模块中的注释很完备，在注释里为模块中的大多数方法/类提供了相关的例子。其中有很多简化性的元函数（meta-function）本专栏里并没有讨论到的，使用这些元函数可以大大简化对其它函数的结合（combination）和交互（interaction）的处理。对于想继续探索函数式范型的Python程序员而言，这些绝对值得好好看看。

[可爱的 Python : Python中的函数式编程，第三部分](#)

1 赞 收藏 [评论](#)

可爱的 Python : Python中函数式编程，第一部分

英文原文：[Charming Python: Functional programming in Python, Part 1](#)，翻译：[开源中国](#)

摘要：虽然人们总把Python当作过程化的，面向对象的语言，但是他实际上包含了函数化编程中，你需要的任何东西。这篇文章主要讨论函数化编程的一般概念，并说明用Python来函数化编程的技术。

我们最好从艰难的问题开始出发：“到底什么是函数化编程呢？”其中一个答案可能是这样的，函数化编程就是你在使用Lisp这样的语言时所做的（还有Scheme, Haskell, ML, OCAML, Mercury, Erlang和其他一些语言）。这是一个保险的回答，但是它解释得并不清晰。不幸的是对于什么是函数化编程，很难能有一个协调一致的定义，即使是从函数化变成本身出发，也很难说明。这点倒很像盲人摸象。不过，把它拿来和命令式编程（imperative programming）做比较也不错（命令式编程就像你在用C, Pascal, C++, Java, Perl, Awk, TCL和很多其他类似语言时所做的，至少大部分一样）。

让我们回想一下功能模块的绑定类。使用该类的特性，我们可以确认在一个给定的范围块内，一个特定的名字仅代表了一个唯一的事物。

我个人粗略总结了一下，认为函数式编程至少应该具有下列几点中的多个特点。在谓之为函数式的语言中，要做到这些就比较容易，但要做到其它一些事情不是很难就是完全不可能：

- 函数具有首要地位 (对象)。也就是说，能对“数据”做什么事，就要能对函数本身做到那些事（比如将函数作为参数传递给另外一个函数）。
- 将递归作为主要的控制结构。在有些函数式语言中，都不存在其它的“循环”结构。
- 列表处理作为一个重点（例如，Lisp语言的名字）。列表往往是通过对于子列表进行递归取代了循环。
- “纯”函数式语言会完全避免副作用。这么做就完全弃绝了命令式语言中几乎无处不在的这种做法：将第一个值赋给一个变量之后为了跟踪程序的运行状态，接着又将另外一个值赋给同一个变量。
- 函数式编程不是不鼓励就是完全禁止使用 *语句*，而是通过对表达式(换句话说，就是函数加上参数) 求值 (evaluation of expressions) 完成任务. 在最纯粹的情形下，一个程序就是一个表达式（再加上辅助性的定义）
- 函数式编程中最关心的是要对 *什么* 进行计算，而不是要 *怎么* 来进行计算。
- 在很多函数式编程语言中都会用到“高阶”（higher order）函数 (换句话说，高阶函数就是对对函数进行运算的函数进行运算的函数)。

函数式编程的倡导者们认为，所有这些特性都有助于更快地编写出更多更简洁并且更不容易出Bug的代码。而且，计算机科学、逻辑学和数学这三个领域中的高级理论家发现，函数式编程语言和程序的形式化特性在证明起来比命令式编程语言和程序要简单很多。

Python内在的函数式功能

自Python 1.0起，Python就已具有了以上所列中的绝大多数特点。但是就象Python所具有的大多数特性一样，这些特点出现在了一种混合了各种特性的语言 中。和Python的OOP（面向对象编程）特性非常象，你想用多少就用多少，剩下的都可以不管（直到你随后需要用到它们为止）。在Python 2.0

中，加入了列表解析（*list comprehensions*）这个非常好用的“语法糖”。尽管列表解析没有添加什么新功能，但它让很多旧功能看起来好了不少。

Python中函数式编程的基本要素包括`functionsmap()`、`reduce()`、`filter()`和`lambda`算子（operator）。在Python 1.x中，`apply()`函数也可以非常方便地拿来将一个函数的列表返回值直接用于另外一个函数。Python 2.0为此提供了一个改进后的语法。可能有点让人惊奇，使用如此之少的函数（以及基本的算子）几乎就足以写出任何Python程序了；更加特别的是，几乎用不着什么执行流程控制语句。

所有(`if`,`elif`,`else`,`assert`,`try`,`except`,`finally`,`for`,`break`,`continue`,`while`,`def`)这些都都能通过仅仅使用函数式编程中的函数和算子就能以函数式编程的风格处理好。尽管真正地在程序中完全排除使用所有流程控制命令可能只在想参加“Python混乱编程”大赛（可将Python代码写得跟Lisp代码非常象）时才有意义，但这对理解函数式编程如何通过函数和递归表达流程控制很有价值。

剔除流程控制语句

剔除练习首先要考虑的第一件事是，实际上，Python会对布尔表达式求值进行“短路”处理。这就为我们提供了一个`if/elif/else`分支语句的表达式版（假设每个分支只调用一个函数，不是这种情况时也很容易组织成重新安排成这种情况）。这里给出怎么做：

对Python中的条件调用进行短路处理

Python

```
# Normal statement-
based flow control

1 # Normal statement-based flow control
2 if <cond1>: func1()
3 elif <cond2>: func2()
4 else:      func3()
5
6 # Equivalent "short circuit" expression
7 (<cond1> and func1()) or (<cond2> and func2()) or (func3())
8
9 # Example "short circuit" expression
10 >>> x = 3
11 >>> def pr(s): return s
12 >>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
13 'other'
14 >>> x = 2
15 >>> (x==1 and pr('one')) or (x==2 and pr('two')) or (pr('other'))
16 'two'
```

我们的表达式版本的条件调用看上去可能不算什么，更象是个小把戏；然而，如果我们注意到`lambda`算子必须返回一个表达式，这就更值得关注了。既然如我们所示，表达式能够通过短路包含一个条件判断，那么，`lambda`表达式就是个完全通用的表达条件判断返回值的手段了。我们来一个例子：

Python中短路的Lambda

Python

```
>>> pr = lambda s:s
>>> namenum =

1 >>> pr = lambda s:s
2 >>> namenum = lambda x: (x==1 and pr("one")) \
3 ....         or (x==2 and pr("two")) \
4 ....         or (pr("other"))
5 >>> namenum(1)
6 'one'
7 >>> namenum(2)
8 'two'
9 >>> namenum(3)
10 'other'
```

将函数作为具有首要地位的对象

前面的例子已经表明了Python中函数具有首要地位，但有点委婉。当我们用lambda操作创建一个函数对象时，我们所得到的东西是完全通用的。就其本质而言，我们可以将我们的对象同名“pr”和“namenum”绑定到一起，以完全相同的方式，我们也完全可以数字23或者字符串“spam”同这些名字绑定到一起。但是，就象我们可以无需将其绑定到任何名字之上就能直接使用数字23（也就是说，它可以用作函数的参数）一样，我们也可以直接使用我们使用lambda创建的函数对象，而无需将其绑定到任何名字之上。在Python中，函数就是另外一种我们能够就像某种处理的值。

我们对具有首要地位的对象做的比较多的事情就是，将它们作为参数传递给函数式编程固有的函数map()、reduce()和filter()。这三个函数接受的第一个参数都是一个函数对象。

- map() 针对指定给它的一个或多个列表中每一项对应的内容，执行一次作为参数传递给它的那个函数，最后返回一个结果列表。
- reduce() 针对每个后继项以及最后结果的累积结果，执行一次作为参数传递给它的那个函数；例如，reduce(lambda n,m:n*m, range(1,10))是求“10的阶乘”的意思（换言之，将每一项和前面所得的乘积进行相乘）
- filter() 使用那个作为参数传递给它的函数，对一个列表中的所有项进行“求值”，返回一个由所有能够通过那个函数测试的项组成的经过遴选后的列表。

我们经常也会把函数对象传递给我们自己定义的函数，不过一般情况下这些自定义的函数就是前文提及的内建函数的某种形式的组合。

通过组合使用这三种函数式编程内建的函数，能够实现范围惊人的“执行流程”操作(全都不用语句，仅仅使用表达式实现)。

Python中的函数式循环

替换循环语言和条件状态语言块同样简单。for可以直接翻译成map()函数。正如我们的条件执行，我们会需要简化语句块成简单的函数调用（我们正在接近通常能做的）：

替换循环

Python

```
for e in lst: func(e)    #
statement-based loop
```

```
1 for e in lst: func(e)    # statement-based loop
2 map(func,lst)          # map()-based loop
```

通过这种方法，对有序程序流将有一个相似的函数式方式。那就是，命令式编程几乎是由大量“做这，然后做那，之后做其它的”语句组成。map()让我们只要做这样：

Map-based 动作序列

Python

```
# let's create an
execution utility function

1 # let's create an execution utility function
2 do_it = lambda f: f()
3
4 # let f1, f2, f3 (etc) be functions that perform actions
5
6 map(do_it, [f1,f2,f3]) # map()-based action sequence
```

通常，我们的整个主要的程序都可以使用一个map表达式加上一些函数列表的执行来完成这个程序。最高级别的函数的另一个方便的特性是你可以把它们放在一个列表里。

翻译while会稍稍复杂一些，但仍然可以直接地完成：

Python中的函数式“while”循环

Python

```
# statement-based while
loop

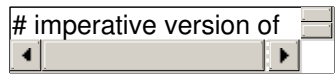
1 # statement-based while loop
2 while <cond>:
3     <pre-suite>
4     if <break_condition>:
5         break
6     else:
7         <suite>
8
9 # FP-style recursive while loop
10 def while_block():
11     <pre-suite>
12     if <break_condition>:
13         return 1
14     else:
15         <suite>
16     return 0
17
18 while_FP = lambda: (<cond> and while_block()) or while_FP()
19 while_FP()
```

在翻译while循环时，我们仍然需要使用while_block()函数，这个函数本身里面可以包含语句而不是仅仅包含表达式。但我们可能还能够对这个函数再进行更进一步的剔除过程（就像前面模版中的对if/else进行短路处理一样）。还有，<cond>很难对普通的测试有什么用，比如while myvar==7，既然循环体（在设计上）不能对任何变量的值进行修改（当然，在while_block()中可以修改全局变量）。有一种方法可以用来为 while_block()添加更有用的条件判断，让 while_block()返回一个有意

义的值，然后将这个返回值同循环结束条件进行比较。现在应该来看一个剔除其中语句的具体例子了：

Python中'echo'循环

Python



```
# imperative version of "echo()"
1 # imperative version of "echo()"
2 def echo_IMP():
3     while 1:
4         x = raw_input("IMP -- ")
5         if x == 'quit':
6             break
7         else:
8             print x
9 echo_IMP()
10
11 # utility function for "identity with side-effect"
12 def monadic_print(x):
13     print x
14     return x
15
16 # FP version of "echo()"
17 echo_FP = lambda: monadic_print(raw_input("FP -- "))=='quit' or echo_FP()
18 echo_FP()
```

在上面的例子中我们所做的，就是想办法将一个涉及I/O、循环和条件判断的小程序，表达为一个递归方式的纯粹的表达式（确切地说，表达为一个可以在需要的情况下传递到别的地方的函数对象）。我们 *的确* 仍然使用了实用函数monadic_print()，但这个函数是完全通用的，而且可以用于以后我们可能会创建的每个函数式程序的表达式中（它的代价是一次性的）。请注意，任何包含monadic_print(x)的表达式 *值* 都是一样的，好像它只是包含了一个x而已。函数式编程中（特别是在Haskell中）的函数有一种叫做“monad”（一元）的概念，这种一元函数“实际什么都不做，只是在执行过程中产生一个副作用”。

避免副作用

在做完这些没有非常明智的理由陈述，并把晦涩的嵌套表达式代替他们之后，一个很自然的问题是“为什么要这样做？！” 我描述的函数式编程在Python中 都实现了。但是最重要的特性和一个有具体用处——就是避免副作用（或至少它们阻止如monads的特殊区域）。程序错误的大部分——并且这些问题驱使程序员去debug——出现是因为在程序的运行中变量获取了非期望的值。函数式编程简单地通过从不给变量赋值而绕过了这个问题。

现在让我们看一段非常普通的命令式代码。这段代码的目的是打印出乘积大于25的一对一对数字所组成的一个列表。组成每对数字的每一个数字都是取自另外的两个列表。这种事情和很多程序员在他们的编程中经常做的一些事情比较相似。命令式的解决方式有可能就象下面这样：

命令式的“打印大乘积”的Python代码

Python

```
# Nested loop  
procedural style for
```

```
1 # Nested loop procedural style for finding big products  
2 xs = (1,2,3,4)  
3 ys = (10,15,3,22)  
4 bigmuls = []  
5 # ...more stuff...  
6 for x in xs:  
7     for y in ys:  
8         # ...more stuff...  
9         if x*y > 25:  
10             bigmuls.append((x,y))  
11             # ...more stuff...  
12 # ...more stuff...  
13 print bigmuls
```

这个项目足够小了，好像没有地方会出什么差错。但有可能在这段代码中我们会嵌入一些同时完成其它任务的代码。用“more stuff”（其它代码）注释掉的部分，就是有可能存在导致出现bug的副作用的地方。在那三部分的任何一点上，变量xs、ys、bigmuls、x、y都有可能在这段按照理想情况简化后的代码中取得一个出人意料的值。还有，这段代码执行完后，后继代码有可能需要也有可能不需要对所有这些变量中的值有所预期。显而易见，将这段代码封装到函数/实例中，小心处理变量的作用范围，就能够避免这种类型的错误。你也可以总是将使用完毕的变量del掉。但在实践中，这里指出的这种类型的错误很常见。

以一种函数式的途径一举消除这些副作用所产生的错误，这样就达到了我们的目的。一种可能的代码如下：

以函数式途径达到我们的目的

Python

```
bigmuls = lambda xs,ys:  
filter(lambda (x,y):x*y >
```

```
1 bigmuls = lambda xs,ys: filter(lambda (x,y):x*y > 25, combine(xs,ys))  
2 combine = lambda xs,ys: map(None, xs*len(ys), dupelms(ys,len(xs)))  
3 dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]*n, lst))  
4 print bigmuls((1,2,3,4),(10,15,3,22))
```

在例子中我们绑定我们的匿名（lambda）函数对象到变量名，但严格意义上讲这并不是必须的。我们可以用简单的嵌套定义来代替之。这不仅是为了代码的可读性，我们才这样做的；而且是因为combine()函数在任何地方都是一个非常好的功能函数（函数从两个输入的列表读入数据生成一个相应的pair列表）。函数dupelms()只是用来辅助函数combine()的。即使这个函数式的例子跟命令式的例子显得要累赘些，不过一旦你考虑到功能函数的重用，则新的bigmuls()中代码就会比命令式的那个要稍少些。

这个函数式例子的真正优点在于：在函数中绝对没有改变变量的值。这样就**不可能**在之后的代码（或者从之前的代码）中产生不可预期的副作用。显然，在函数中没有副作用，并不能保证代码的正确性，但它仍然是一个优点。无论如何请注意，Python（不像很多其它的函数式语言）不会阻止名字bigmuls, combine和dupelms的再次绑定。如果combine()运行在之后的程序中意味着有所不同时，所有的预测都会失效。你可能会需要新建一个单例类来包含这个不变的绑定（也就是说，s.bigmuls之类的）；但是这一例并没有空间来做这些。

一个明显值得注意的是，我们特定的目标是定制Python 2的一些特性。而不是命令式的或函数式编程的例子，最好的（也是函数式的）方法是：

Python

```
print [(x,y) for x in  
(1,2,3,4) for y in
```

```
1 print [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]
```

结束语

我已经列出了把每一个Python控制流替换成一个相等的函数式代码的方法（在程序中减少副作用）。高效翻译一个特定的程序需要一些额外的思考，但我们已经看出内置的函数式功能是全面且完善的。在接下来的文章里，我们会看到更多函数式编程的高级技巧；并且希望我们接下来能够摸索到函数式编程风格的更多优点和缺点。

[可爱的 Python : Python中函数式编程，第二部分](#)

[可爱的 Python : Python中的函数式编程，第三部分](#)

1 赞 1 收藏 [1 评论](#)