

Python 入门到精通(3): VS 2015 搭建开发环境

原文出处: [头头哥](#)

在第一篇《[Python 入门到精通\(1\): Windows 搭建 Python 开发环境](#)》很多园友提到希望使用visual studio 2013/visual studio 2015 python做demo，这里略带一句，其实就“学习python”而言，比较建议使用pycharm，pycharm可以设置VS或者eclipse等多种IDE的编码，可以让绝大部分其他语言使用IDE的习惯者更容易上手。这一点兼容确实做的很好。不过既然这么多园友要求使用vs开发python的话，就介绍一下visual studio 2015开发python的环境。众所周知visual studio号称“世界第一IDE”，自然是有所亮点，相信微软的设计师在设计visual studio 2015这款产品时也是本着用户体验出发的。所以visual studio 2015搭建python开发环境非常快速便捷。

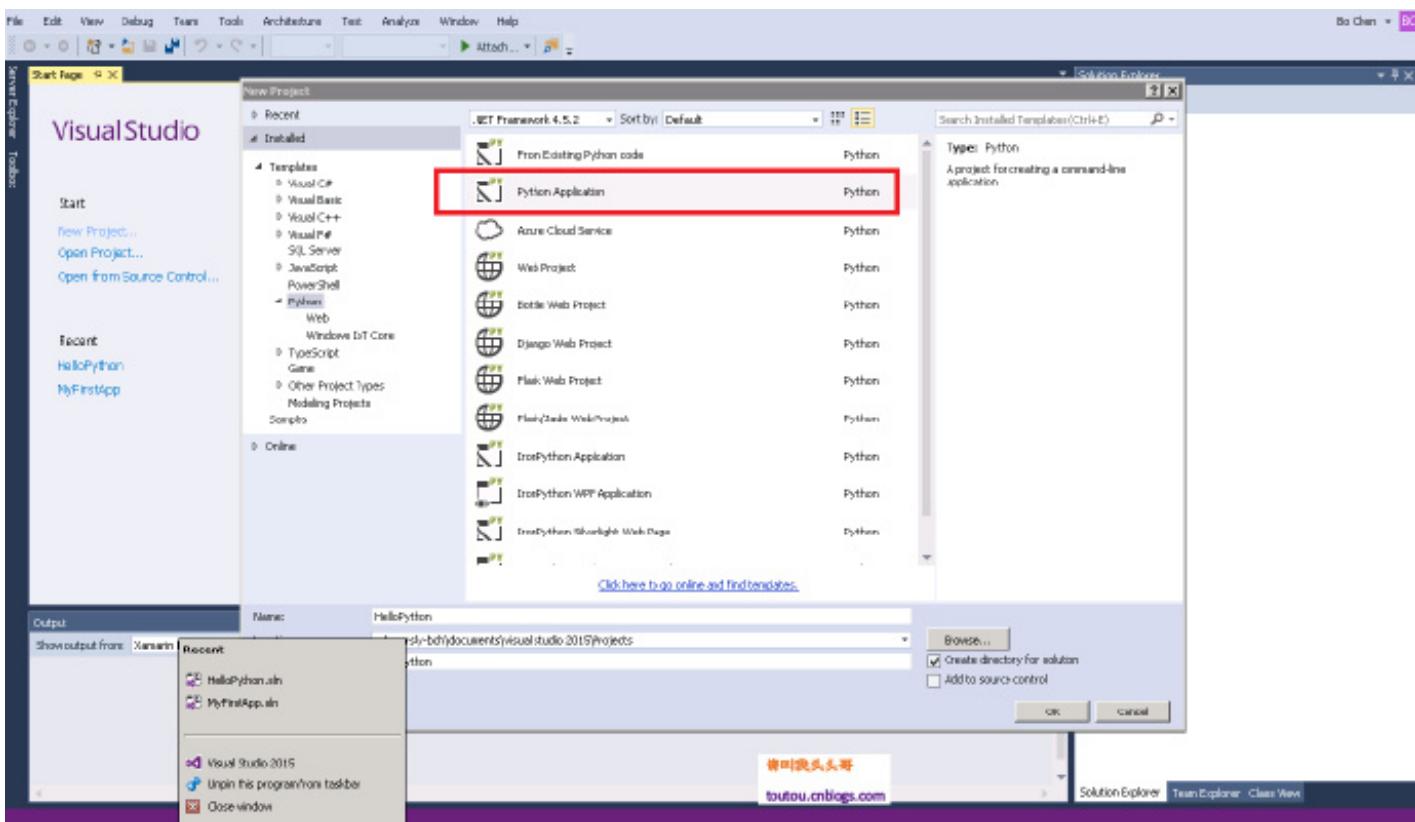
写在前面

python你不去认识它，可能没什么，一旦你认识了它，你就会爱上它。

正文开始

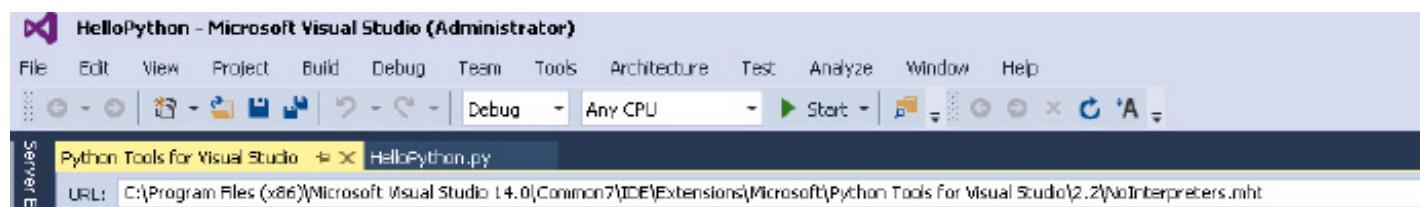
正式开始之前，首先你电脑得有visual studio2013/2015，如果没有的话就先去下载一个[visual studio](#)

1. 创建工程：



2. 下载环境：

创建好工作以后，点击运行，就会出现下面这个界面，然后点击下载，并安装



We didn't find any interpreters

Installing a Python interpreter will let you run your project, and will include useful libraries and tools.

Download and install Python

This will install the latest version (32-bit) from the [Python homepage](#).

CPython is the most commonly used interpreter and has the most up-to-date set of language features.

Help me choose one

There are other interpreters and bundles that may better suit your needs.

This link will take you to a web page that explains the differences and will help you download and install a bundle.

I already have one

Sorry, but we did not detect it automatically. We can use it, but you'll need to add it yourself.

This link will take you to a web page that will help you add your existing interpreter.

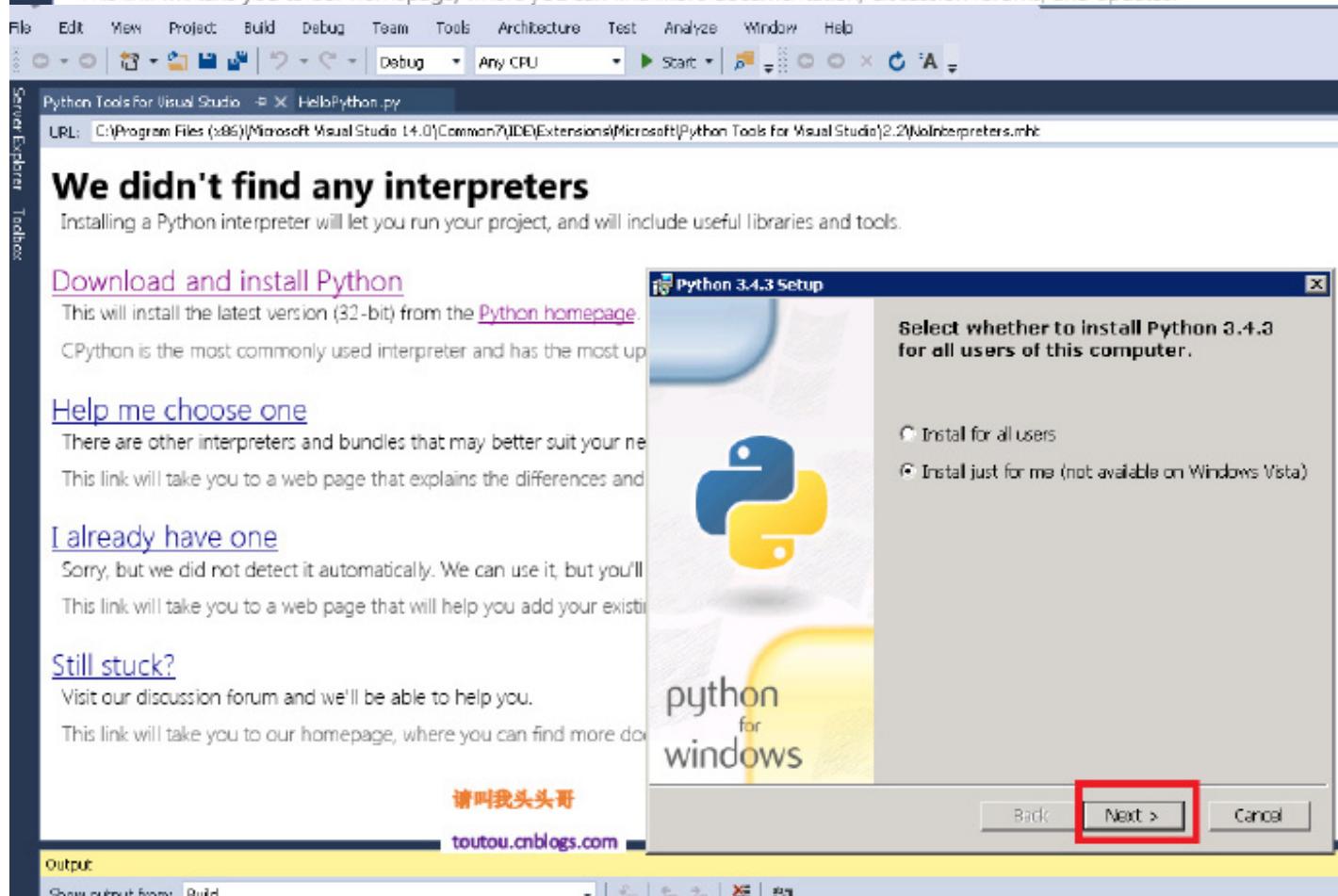
Still stuck?

Visit our discussion forum and we'll be able to help you.

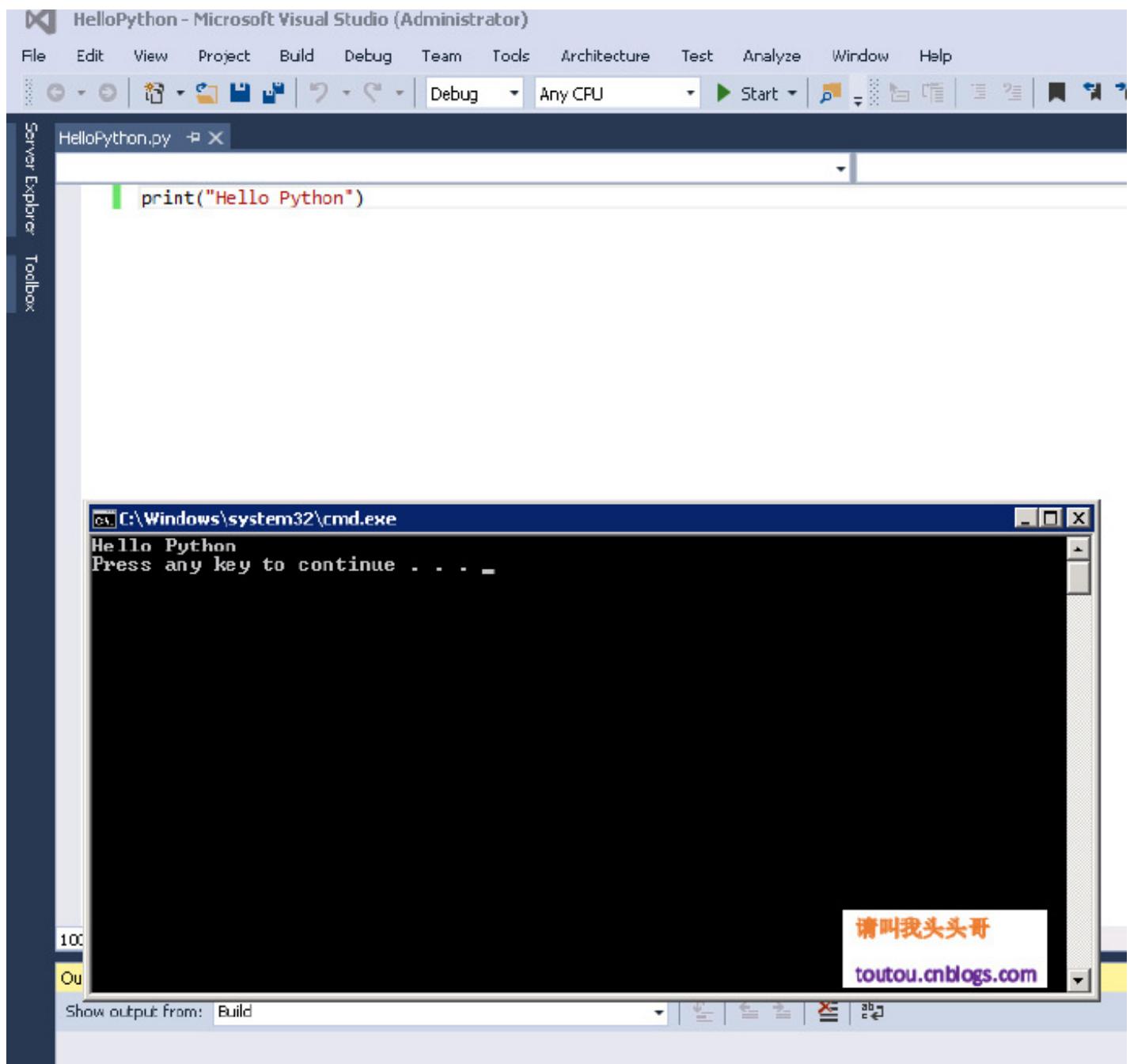
请叫我头头哥

toutou.cnblogs.com

This link will take you to our homepage, where you can find more documentation, discussion forums, and updates.



2. 测试环境：



OK，成功运行！

博客总结

好了，关于visual studio开发python就简单介绍这么多。这篇博客只是为了回应众多园友的支持和响应才写的，同时也非常感谢你们的支持。其实无论是visual studio还是pycharm都是一样的，IDE只是一个工具，如果大家更喜欢用visual studio学习python的话，可以通过这篇博客配置好visual studio开发python的环境，然后将以后博客中的demo在visual studio中练手，其效果是一样的。各位看个如果还是觉得VS更顺手，其实也可以用VS的。只是我当初刚接触python的时候用的是pycharm所以习惯了。再次重申，这篇博客的确很简易，也只是为了响应大家的号召。至于其他知识点我会在接下来继续更新！

Python 入门到精通(2): 基本语法 (1)

原文出处: [头头哥](#)

郑重承诺

我承认，现在园子里烂尾的系列博文比比皆是，[在上一篇博文中也有园友对Python这个系列存在质疑，在这里我告诉大家，只要python这个系列的博文在园子里不沉\(至少有园友关注，如果python确实在博客园吃不开的话，我就只好转战其他python社区了\)，我一定会尽自己最大的努力写到所了解的最大领域。大家不必担心，我能做到的就是一周更新1~2篇\(毕竟我只敢保证周末能出一两篇，工作日怕有加班的情况，同为苦逼IT，你懂得\)](#)

正文开始：Python基本语法

1. 定义变量：

代码正文：

Python

```
x=1  
y=2
```

```
1 x=1  
2 y=2  
3 z=x+y
```

Python定义变量的方式呢很简单，就是上面这段代码，相信只要稍微懂点数学的人都能看懂这段代码的含义。其实现在在国外很多大学都是把Python作为计算机语言入门的第一门语言，因为python语言可以说是人类的语言，很容易上手，一眼就能看懂(不过大部分语言都是这样，入门容易深入难，要持之以恒。)

代码讲解：

The screenshot shows the PyCharm IDE interface. The top bar displays the title "HelloWorld - [E:\PythonSoft\PythonSelfCode\HelloWorld]" and the menu options: File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help. The left sidebar shows the project structure under "HelloWorld": "FirstWork.py" and "External Libraries". The main editor window titled "FirstWork.py" contains the following Python code:

```
x=1  
y=2  
z=x+y  
print(z)
```

The bottom panel shows the "Run" tab with the command "C:\Python27\python.exe E:/PythonSoft/PythonSelfCode/HelloWorld/FirstWork.py" and the output "3". To the right of the output, there is a watermark: "请叫我头头哥" and "TOUTOU.CNBLOGS.COM".

2. 判断语句：

代码正文：

Python

```
# coding=utf-8
```

```
score=90  
1 # coding=utf-8  
2 score=90  
3 if score>=90:  
4     print("你真棒")  
5     print("优秀")  
6 elif score>=80:  
7     print("良好")  
8 elif score>=60:  
9     print("及格")
```

Python语言非常便于大家理解，就连判断语句也是如此。

代码讲解：

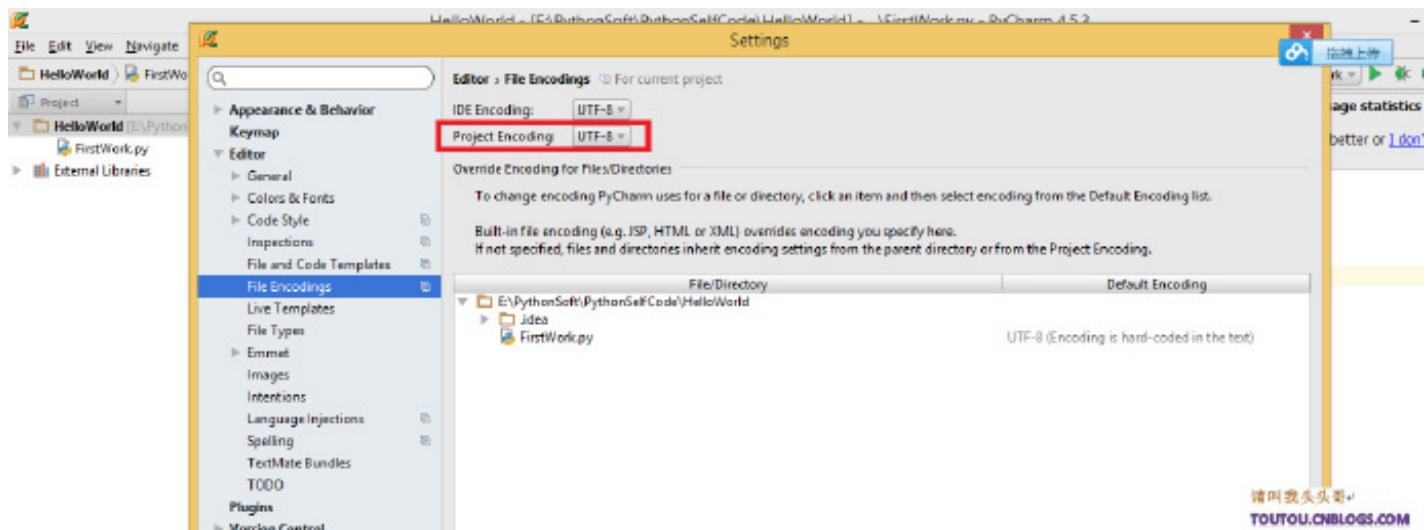
The screenshot shows the PyCharm interface with the file 'FirstWork.py' open. The code contains the following if-elif block:

```
# coding=utf-8
score=90
if score>=90:
    print("你真棒")
    print("优秀")
elif score>80:
    print("良好")
elif score>60:
    print("及格")
```

Annotations in red and yellow highlight specific parts of the code:

- A red arrow points to the first line of the script with the text: "因为代码中包含中文，所以需要加上 # coding=utf-8 不然会报错。这个大家在练习这里的时候可以本着“重在参与”的原则去掉这个运行试试" (Because the code contains Chinese, you need to add # coding=utf-8 or it will报错. When practicing here, you can remove it and run it.)
- A red arrow points to the 'if' keyword with the text: "判断条件后面的执行语句必须得有缩进" (The execution statements after the condition judgment must have indentation).
- A red arrow points to the 'elif' keyword with the text: "elif相当于else if" (elif is equivalent to else if).
- A yellow box highlights the 'elif' block with the text: "判断语句后面的执行语句，默认的是不会有{}来确定开始和结束，非常简单，就是从条件语句后面的第一个缩进开始，最后一个缩进结束。比如这段代码输出的就是90后面的两个语句" (The execution statements after the judgment statement are default to not having {} to determine the start and end, very simple, starting from the first indentation after the judgment statement, and ending at the last indentation. For example, the output of this code is the two statements after 90).
- A blue box in the top right corner says: "Help Improve PyCharm by sending an JetBrain s.r.o. Please click Agree if you want to help make otherwise. Decline".
- The bottom status bar says: "请叫我头头哥" and "TOUTOU.CNBLOGS.COM".

值得一提的是默认情况下，代码中有中文需要注意，不光是在运行时，在运行后也是需要设置的，因为默认的输出中文会乱码。大家可以在这里设置。File>>Settings>>Editor>>File Encodings>>Project Encodings 改成UTF-8 详情见下图：



3. 循环：

代码正文：

Python

```
for i in range(0,3):
    print(i)
```

```
1 for i in range(0,3):
2     print(i)
```

```

3 # print("Index"+i)
4 print("Index {0} {1}".format(i,"cnblogs"))
5 print("end")

```

Python的循环语法可能让家用起来觉得有点不习惯，但是相信很容易就可以理解的，只是有一点点出入而已。

代码讲解：

The screenshot shows the PyCharm IDE interface with the following annotations:

- range表示范围，表示的范围在 0 <= i < 3**: Points to the range(0, 3) call in the code.
- print如果需要进行拼接，正确的语法是需要用 format 的，format支持多个拼接，但是需要与前面的{}的索引对应 python不支持直接使用"index"+i 这种写法**: Points to the print("Index "+i) line, explaining the use of format instead of string concatenation.
- for循环执行代码结束的标志就是没有缩进**: Points to the end of the for loop block.

Output window (Run tab) shows the execution results:

```

C:\Python27\python.exe E:/PythonSoft/PythonSelfCode/HelloWorld/FirstWork.py
0
Index 0 cnblogs
1
Index 1 cnblogs
2
Index 2 cnblogs
end

```

4. 定义函数def：

代码正文：

Python

```

def HelloCNBlogs():
    print("Hello cnblogs")

1 def HelloCNBlogs():
2     print("Hello cnblogs")
3
4 def GetMax(x,y):
5     if x>y:
6         return x
7     else:
8         return y
9
10 HelloCNBlogs()
11 print(GetMax(9,3))

```

代码讲解：

The screenshot shows the PyCharm IDE interface with a project named 'HelloWorld' and a file named 'FirstWork.py'. The code in the editor is:

```
def HelloCNBlogs():
    print("Hello cnblogs")

def GetMax(x,y):
    if x>y:
        return x
    else:
        return y

HelloCNBlogs()
print(GetMax(3,5))
```

Annotations in red and blue highlight specific parts of the code:

- Red text and arrows point to the `def` keyword in the first function definition and the `return` statements in the `GetMax` function.
- Blue text and arrows point to the opening brace of the `GetMax` function and the closing brace of the `GetMax` function.
- Blue text and arrows point to the `print` statement in the `GetMax` function and the `print` statement at the bottom of the script.
- A yellow bar at the top right of the editor window contains a message from JetBrains asking for anonymous usage statistics.
- In the bottom left corner, the 'Run' tool window shows the output of the script:

Run	Environment	C:\Python27\python.exe E:\PythonSoft\PythonSelfCode\HelloWorld\FirstWork.py
		Hello cnblogs
		9
		Process finished with exit code 0

A red arrow points from the '输出结果' (Output Result) text to the '9' in the Run window.

5.00面向对象class：

代码正文：

Python

```
class FirstTest:
    def __init__(self, name):
        self._name = name
    def SayFirst(self):
        print("Hello {0}".format(self._name))
F = FirstTest("CNBlogs")
F.SayFirst()
```

代码讲解：

可以看见在定义类的时候也离不开冒号

`_init_是类的构造函数
(如果是其他语言这里定义
构造函数可能是直接
写FirstTest(),self有点
像其他语言的this)`

实例化FirstTest,因为
FirstTest的构造函数
里有个参数，所以在实例
化的时候必须传入一个参数

调用实例化对象F的方法

运行结果

```

class FirstTest:
    def __init__(self, name):
        self._name = name
    def SayFirst(self):
        print("Hello {0}".format(self._name))
F = FirstTest("CNBlogs")
F.SayFirst()

```

6. 继承：

代码正文：

Python

```

class FirstTest:
    def __init__(self, name):
        self._name = name
    def SayFirst(self):
        print("Hello {0}".format(self._name))

class SecondTest(FirstTest):
    def __init__(self, name):
        FirstTest.__init__(self, name)
    def SaySecond(self):
        print("Good {0}".format(self._name))

S = SecondTest("CNBlogs");
S.SayFirst()
S.SaySecond();

```

代码讲解：

构造函数

Python继承是直接在当前类后面加上(父类)就行了,不像其他语言是:父类

实例化时传入参数

输出结果

```

HelloWorld - [E:\PythonSoft\PythonSelfCode\HelloWorld] - ..\FirstWork.py - PyCharm 4.5.3
File Edit View Navigate Code Refactor Run Tools VCS Window Help
HelloWorld > FirstWork.py
Project + - External Libraries
HelloWorld [E:\PythonSoft\PythonSelfCode\HelloWorld]
FirstWork.py
External Libraries
Help improve PyCharm by sending anonymous usage statistics to JetBrains s.r.o.
Please click Agree if you want to help make PyCharm better otherwise, more...

```

```

class FirstTest:
    def __init__(self, name):
        self._name = name
    def SayFirst(self):
        print("Hello {0}".format(self._name))

class SecondTest(FirstTest):
    def __init__(self, name):
        FirstTest.__init__(self, name)
    def SaySecond(self):
        print("Good {0}".format(self._name))

S=SecondTest("CNBlogs")
S.SayFirst()
S.SaySecond()

```

请叫我头头哥
TOU TOU .CNBLOGS.COM

```

Run FirstWork
C:\Python27\python.exe E:/PythonSoft/PythonSelfCode/HelloWorld/FirstWork.py
Hello CNBlogs
Good CNBlogs
Process finished with exit code 0

```

7.引入其他文件的类:

代码正文:

Python

```

# 第一种引入的方法
# import FirstWork

1 # 第一种引入的方法
2 # import FirstWork
3 #
4 # S=FirstWork.SecondTest("CNBlogs");
5 # S.SayFirst()
6 # S.SaySecond();
7
8 #第二种引入方法
9
10 from FirstWork import SecondTest
11
12 ST=SecondTest("CNBlogs");
13 ST.SayFirst()
14 ST.SaySecond();

```

上面代码中我引入了6里面的FirstWork.py文件里的SecondTest这个类，这里我们可以看见，引入其他文件的类有两种方法，至于他们的区别也很明显，这里就不多说了，大家可以根据实际需求选择。相信大家也发现了Python中引入其他文件中的类的时候，用到import颇有点其他语言的using使用命名空间的感觉。没错，其实import就是引入命名空间

代码讲解：

The screenshot shows the PyCharm IDE interface. The top bar includes File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help menus. The title bar says "HelloWorld - [E:\PythonSoft\PythonSelfCode\HelloWorld] - ..\LoadOther.py - PyCharm 4.5.3". The left sidebar shows a project tree with "HelloWorld" containing "FirstWork.py" and "LoadOther.py". The main editor window contains Python code:

```
# 第一种引入方法
# import FirstWork
#
# S=FirstWork.SecondTest("CHLog")
# S.SayFirst()
# S.SaySecond()

# 第二种引入方法

from FirstWork import SecondTest

ST=SecondTest("CHLog")
ST.SayFirst()
ST.SaySecond()
```

The bottom right corner of the editor has a watermark: "提问我从头学~ TOUTYOU.CNLOGS.COM". Below the editor is the "Run" tool window titled "FirstWork" with the command "C:\Python27\python.exe E:/PythonSoft/PythonSelfCode/HelloWorld/FirstWork.py" and output "Calls CHLog
Good CHLog".

Pycharm 快捷键

在上面的演示中可以看出来python注释的用法是#而不再是//

Pycharm常见快捷键：

- Ctrl+/注释(取消注释)选择的行
- Shift + Enter开始新行
- Ctrl + Enter智能换行
- TAB Shift+TAB缩进/取消缩进所选择的行
- Ctrl + Alt + I自动缩进行
- Ctrl + Y删除当前插入符所在的行
- Ctrl + D 复制当前行、或者选择的块
- Ctrl + Shift + J合并行
- Ctrl + Shift + V从最近的缓存区里粘贴
- Ctrl + Delete删除到字符结尾
- Ctrl + Backspace删除到字符的开始
- Ctrl + NumPad+/-展开或者收缩代码块
- Ctrl + Shift + NumPad+展开所有的代码块
- Ctrl + Shift + NumPad-收缩所有的代码块

博客总结

关于python的基本语法就介绍这么多，大家如果有什么疑问或者补充的可以踊跃发言。这个系列我不能说一天更新一篇(毕竟同为男人，都需要养家糊口敲代码)，只要python这个系列的博文在园子里不沉(至少有园友关注，如果python确实在博客园吃不开的话，我就只好转战其他python社区了)，我就一定会坚持做完(尽量保证一周最少更新1~2篇)。在上篇博客中很多园友提到希望使用visual studio 2013/visual studio 2015 python做demo，这里略带一句，其实就“学习python”而言，比较建议使用

pycharm, pycharm可以设置VS或者eclipse等多种IDE的编码，可以让绝大部分其他语言使用IDE的习惯者更容易上手。这一点兼容确实做的很好。如果大家确实更习惯或者要求使用visual studio的话，只要有这个需求，我会在稍后的博客中单独介绍下visual studio开发python。

1 赞 3 收藏 [1 评论](#)

Python 入门到精通(1): Windows 搭建 Python 开发环境

原文出处：[头头哥](#)

写在前面

从大学开始玩python到现在参加工作，已经有5年了，现在的公司是一家.net的公司用到python的比较少，最近公司有新项目需要用到python,领导希望我来跟其他同事training，就有了这篇博客，打算将python的training弄成一个简易的python系列，供大家入门使用。Python语言自从20世纪90年代初诞生至今，它逐渐被广泛应用于处理系统管理任务和Web编程。今天就让我们来搭建一个python的开发环境，Windows搭建python开发环境。一切从“Hello world”开始。

python你不去认识它，可能没什么，一旦你认识了它，你就会爱上它。

基本概念

Python（英语发音：/paɪθən/），是一种面向对象、解释型计算机程序设计语言，由Guido van Rossum于1989年发明，第一个公开发行版发行于1991年。

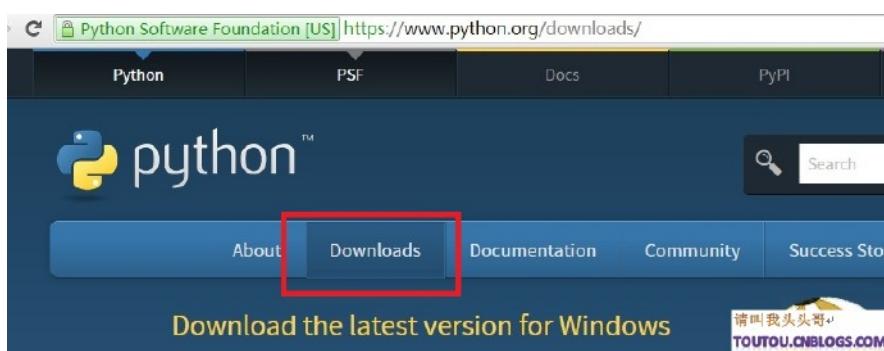
Python是纯粹的自由软件，源代码和解释器CPython遵循 GPL(GNU General Public License)协议[1]。

Python语法简洁清晰，特色之一是强制用空白符(white space)作为语句缩进。

Python具有丰富和强大的库。它常被昵称为胶水语言，能够把用其他语言制作的各种模块（尤其是C/C++）很轻松地联结在一起。常见的一种应用情形是，使用Python快速生成程序的原型（有时甚至是程序的最终界面），然后对其中有特别要求的部分，用更合适的语言改写，比如3D游戏中的图形渲染模块，性能要求特别高，就可以用C/C++重写，而后封装为Python可以调用的扩展类库。需要注意的是在您使用扩展类库时可能需要考虑平台问题，某些可能不提供跨平台的实现。

Windows搭建python开发环境

首先需要去[python的官网](#)下载环境。鼠标移动到Downloads的tab上，在这里可以下载。



python的环境还是很人性化的，没有那么多罗里吧嗦的配置什么的，下载好以后直接无脑next就行了，直到finish.

Python IDE

优秀的Python IDE有很多，这里我就介绍几款相对我来说比较常用的！排名不分先后！

- [pycharm](#)
- [VIM](#)
- [Eclipse with PyDev](#)
- [Sublime Text](#)
- [Komodo Edit](#)
- [PyScripter](#)
- [Interactive Editor for Python](#)

这里就以pycharm为例



pycharm默认的是可以用30天，这里在给大家共享一个注册码(注册码来源：百度知道：)

Python

用户名: yueteng3527
注册码:

用户名: yueteng3527

注册码：

```
===== <a href="https://www.baidu.com/s?wd=LICENSE&tn=44039180_cpr&fenlei=mv6quAkxTZn0lZRqlHckPjm4nH00T1d9nyNWPACdujbknAPhnvPb0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0l&bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1fsPWR4njcvP6" target="_blank" rel="nofollow">LICENSE</a> BEGIN =====  
2 g33347-12042010  
4 00001FMHemWls"6wozMZhaz3lgXKJX  
5 2lnV2l6kSO48hgGLa9JNgjQ5oKz1Us  
6 FFR8k"nGzJHzjQT6IBG1fbQZn9!Vi  
7 ===== <a href="https://www.baidu.com/s?  
8 wd=LICENSE&tn=44039180_cpr&fenlei=mv6quAkxTZn0lZRqlHckPjm4nH00T1d9nyNWPACdujbknAPhnvPb0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0l&bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1fsPWR4njcvP6" target="_blank" rel="nofollow">LICENSE</a> <a href="https://www.baidu.com/s?  
9 wd=END&tn=44039180_cpr&fenlei=mv6quAkxTZn0lZRqlHckPjm4nH00T1d9nyNWPACdujbknAPhnvPb0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0l&bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1fsPWR4njcvP6" target="_blank" rel="nofollow">END</a> =====
```

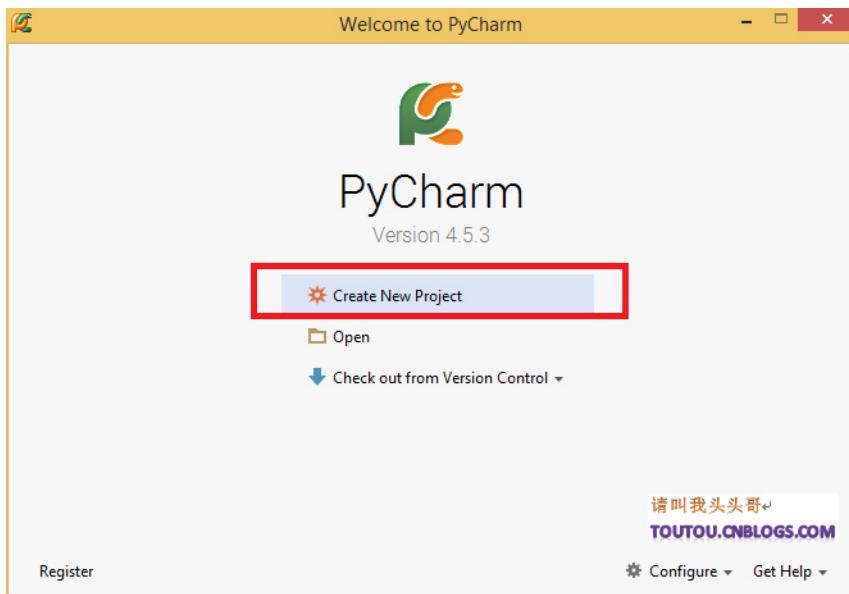
从Hello World开始

Hello world比较简单，搭建好环境之后基本可以一气呵成的，这里我就直接贴图了。

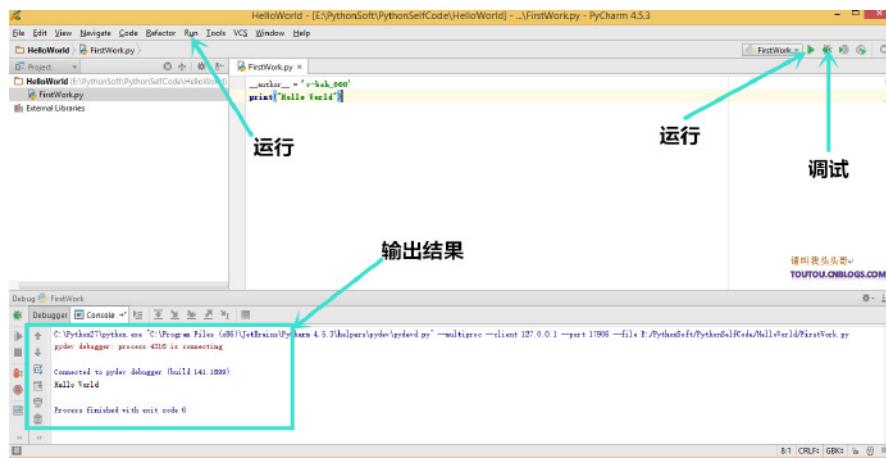
首先打开安装好的pycharm



点击create创建一个工程



print一个"Hello World"



博客总结

因为这个系列是需要给一些完全没有接触过Python的同事training用的，所以园子里想学Python的朋友们赶紧跟上，不一定能带你超神，但是肯定可以让你完完全全的爱上Python。如果你对Python有兴趣，就请加关注吧。

1 赞 17 收藏 [评论](#)

用 RAKE 和 Maui 做 NLP 关键词提取的教程

本文由 [伯乐在线 - 许世豪](#) 翻译, [黄利民](#) 校稿。未经许可, 禁止转载!

英文出处: [Alyona Medelyan](#)。欢迎加入[翻译组](#)。

Alyona 经营着一家总部在新西兰的 NLP 咨询公司——Entopix, 她有计算语言学硕士和计算机博士学位, 是主题提取工具 Maui 的作者。

视频简介: https://www.youtube.com/watch?v=Chwm_AFZg-c

1 引言

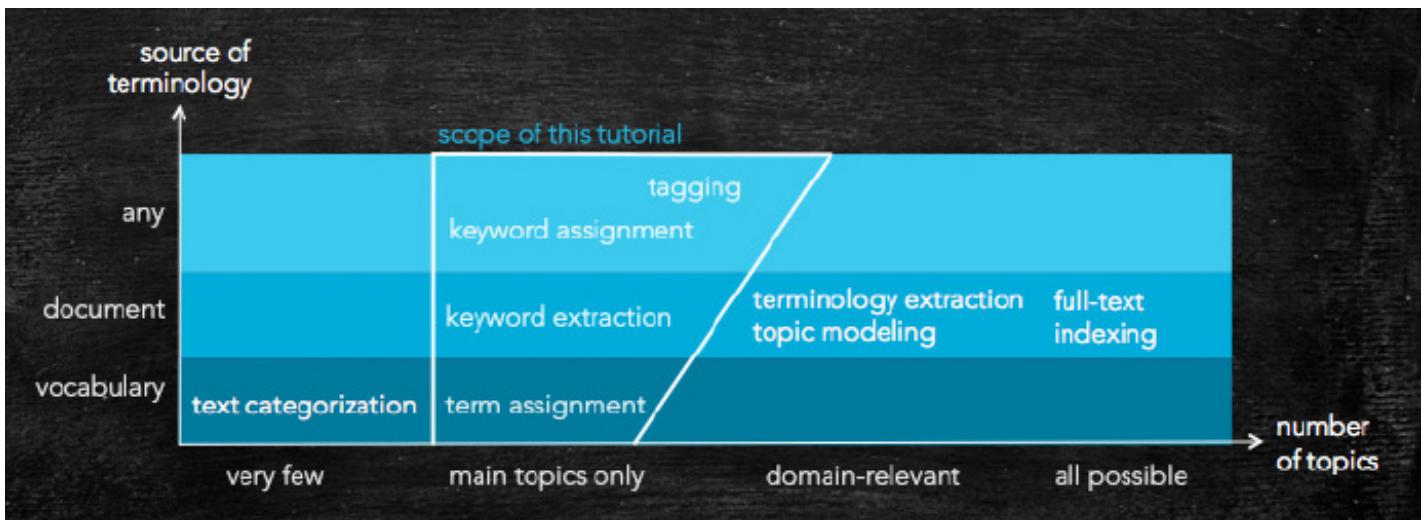
在这个教程中, 你将会学到如何用Python和Java自动提取关键词, 而且你将理解与之相关的任务, 例如有控制词表的关键短语提取 (换句话说是把文本分类到各种可能类别的超大集合中), 还有术语提取。

本教程如下组织: 首先, 我们讨论要一点背景——什么是关键词, 一个关键词算法如何工作? 然后我们用一个叫做Rake的Python库举一个简单但在很多情况下很有用的关键词提取的例子。最后, 我们展示一个叫做Maui的Java工具如何用机器学习方法提取关键词。

1.1 为什么提取关键词

处理文档时, 提取关键词是最重要的工作之一。读者受益于关键词, 因为他们可以快速判断一篇文章是否值得一读。网站创立者从中受益, 因为他们可以根据话题集中相似的内容。算法开发者从关键词受益, 因为关键词降低文本维度来显出最重要的特征。这是一些有帮助的例子。

根据定义, **关键词是一篇文档中表达的主要话题**。[这个术语有点迷惑](#), 所以下面的图片在词汇的源头和被每个文档的话题数量方面比较了相关的任务。



这个教程中, 我们会关注两个具体的任务并且评价它们:

- 在给定文本中出现的最重要的词和短语
- 从与给定文本匹配的预定词表中, 识别一系列主题

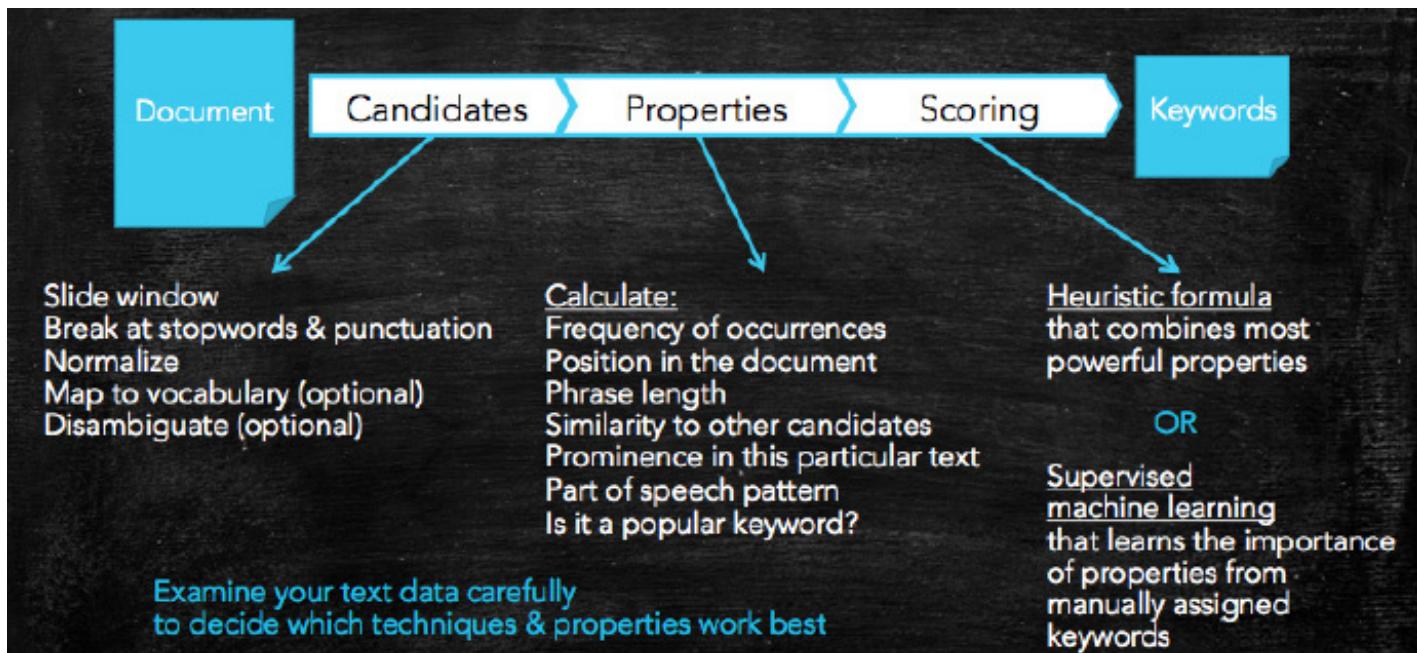
如何多个文档间词汇一致性重要，我推荐你使用一个词表——或学科词表、分类词典，除非由于某些原因做不到这点。

给对文本分类（另一个做文本工作时流行的任务）有兴趣的人的几句话：如果类别的数量很大，你将会很难收集足够的训练集用于有监督的分类。因此，如果有一百多个类别，并且你可以给出这些类别的名字（而不是抽象类别），那么你面临是细粒度的分类。我们可以把这个任务看作带有受控词汇表的关键词提取，或者术语分配。所以，读下去，这个教程也是针对你的！

2 关键词提取如何工作？

通常关键词提取算法有三个主要成分：

1. 候选词选择：这里，我们提取所有可能是关键词的词，词组，术语或概念（取决于任务）。
2. 特性计算：对于每个候选词，我们需要计算表示它是否为关键词的特性。比如，一个候选词
3. 对关键词评分并选择：所有候选词可以通过把各性质结合进一个公式来评分，或者用机器学习技术来决定一个候选词是一个关键词的概率。然后一个分数或概率的阈值或者对关键词数量的限制用来选择最终的关键词集合。



最终，像候选词最小频率的参数，它的最小和最大词长，或者用来使候选词标准化的词干提取器，都有助于调整算法对于特定数据集的性能。

3 Python下用RAKE提取关键词

对于Python用户，有一个易用的关键词提取库叫做RAKE，其全称是 Rapid Automatic Keyword Extraction。算法本身表述在 Michael W. Berry 的 [《文本挖掘和理论》](#)一书中（[免费pdf](#)）。这里，我们使用已有的[Python实现](#)。这里有个[修改过的版本](#)，它使用了自然语言处理工具NLTK处理一些计算。对于这个教程，我已经 [fork了原始的RAKE仓库并且将其拓展成了RAKE教程](#)，来使用额外的参数评价它的性能。

3.1 配置RAKE

首先你需要从这个地址获取RAKE仓库 <https://github.com/zelandiya/RAKE-tutorial>.

Shell

```
$ git clone https://github.com/zelan
1 $ git clone https://github.com/zelandiya/RAKE-tutorial
```

然后，按照*rake_tutorial.py*中的步骤，import RAKE，并且为这个教程的”幕后”部分import operator：

Python

```
import rake
import operator
1 import rake
2 import operator
```

3.2 对一小段文本使用RAKE

首先，我们用一个通向一个[停止词表](#)的路径初始化RAKE并且设置一些参数。

Python

```
rake_object =
rake.Rake("SmartStopli
1 rake_object = rake.Rake("SmartStoplist.txt", 5, 3, 4)
```

现在，我们有一个提取关键词的RAKE对象，其中：

- 每个词至少有5个字符
- 每个短语至少有3个词
- 每个关键词至少在文本中出现4次

这些参数取决于你手上的文本，并且仔细选择这些参数是很关键的（试着用默认参数运行这个例子你就会明白）。更多信息在下一节。

接下来，我们已经有了存储在一个变量中的一段文本（在这个例子中，我们从一个文件中读取），我们可以应用RAKE并且打印关键词。

Python

```
sample_file =
open("data/docs/fao_tes
1 sample_file = open("data/docs/fao_test/w2167e.txt", 'r')
2 text = sample_file.read()
3 keywords = rake_object.run(text)
4 print "Keywords:", keywords
```

输出应该看起来像这样：

Python

```
Keywords: Keywords: [('household food' 
```

```
1 Keywords: Keywords: [('household food security', 7.711414565826329), ('indigenous groups living', 7.4), ('national  
forest programmes', 7.249539170506913), ('wood forest products', 6.844777265745007)...
```

这里，我们没有每个关键词的名字和它对应于这个算法的分数。

3.3 RAKE: 幕后

这一次，我们将会使用一个短文本片段，并且我们可以在这里使用默认参数：

Python

```
stoppath = "SmartStoplist.txt"   
1 stoppath = "SmartStoplist.txt"  
2  
3 rake_object = rake.Rake(stoppa  
4  
5 text = "Compatibility of systems of linear constraints over the set of natural numbers. Criteria of compatibility " \  
6 "of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered." \  
7 "Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generating" \  
8 " sets of solutions for all types of systems are given. These criteria and the corresponding algorithms " \  
9 "for constructing a minimal supporting set of solutions can be used in solving all the considered types of " \  
10 "systems and systems of mixed types."
```

首先，RAKE把文本分割成句子，并且生成候选词：

Python

```
sentenceList =   
1 sentenceList = rake.split_sentences(text)  
2 stopwordpattern = rake.build_stop_word_regex(stoppa  
3 phraseList = rake.generate_candidate_keywords(sentenceList, stopwordpattern)
```

这里，各种标点符号将会被认为句子边界。大多数情况这都很有用，但是对于标点是真实短语的一部分的情况（例，.Net或Dr. Who）没有用。

列在停止词文件中的所有词将会被认为是短语边界。这帮助我们生成包含一个或更多非停止词的候选词，比如这篇文本中的“compatibility”、“systems”、“linear constraints”、“set”，、“natural numbers”和“criteria”。大多数候选词将会是有效的，但是，对于停止词是短语的一部分的情况不会有用。例如，“new”列在RAKE的停止词中。这意味着“New York”或“New Zealand”都不会是一个关键词。

第二，RAKE计算每个候选词的属性，是各候选词的分数的总和。候选词是怎么打分的？根据候选词的出现频率，以及典型长度。

Python

```
wordscores =  
rake.calculate_word_sc
```

```
1 wordscores = rake.calculate_word_scores(phraseList)  
2 keywordcandidates = rake.generate_candidate_keyword_scores(phraseList, wordscores)
```

这里的一个问题是，候选词并没有标准化。所有我们可能会有看起来完全一样的关键词，比如：

「small scale production 和 small scale producers」，或者 「skim milk powder 和 skimmed milk powder」。在理想情况下，关键词提取算法应首先应用于词干和其他标准化的关键词。

最后，我们根据RAKE的分数给候选关键词排序。关键词然后可以既是分数排名前五的候选词，也可以是超过一个选定分数阈值的，或者排名第三的，如同下面的例子：

Python

```
sortedKeywords =  
sorted(keywordcandidat
```

```
1 sortedKeywords = sorted(keywordcandidates.iteritems(), key=operator.itemgetter(1), reverse=True)  
2 totalKeywords = len(sortedKeywords)  
3  
4 for keyword in sortedKeywords[0:(totalKeywords / 3)]:  
5     print "Keyword: ", keyword[0], ", score: ", keyword[1]
```

正常的输出信息如下：

Python

```
Keyword: minimal  
generating sets , score:
```

```
1 Keyword: minimal generating sets , score: 8.66666666667  
2 Keyword: linear diophantine equations , score: 8.5  
3 Keyword: minimal supporting set , score: 7.66666666667  
4 Keyword: minimal set , score: 4.66666666667  
5 Keyword: linear constraints , score: 4.5  
6 Keyword: upper bounds , score: 4.0  
7 Keyword: natural numbers , score: 4.0  
8 Keyword: nonstrict inequations , score: 4.0
```

这还有两个有用的脚本。第一个用有文档和他们的手工分分配的关键词的目录，和应该评价的排名靠前的关键词的数量，评价了 RAKE 的准确度。例如：

Python

```
$ python  
evaluate_rake.py
```

```
1 $ python evaluate_rake.py data/docs/fao_test/ 10  
2 ...  
3 Precision 4.44 Recall 5.17 F-Measure 4.78
```

精度（Precision）告诉我们在这些被提取的关键词中正确的百分比，回召（Recall）告诉我们在所有正确的关键词中，正确提取的百分比，F量度是两者的结合。

为了改善RAKE的性能，我们可以运行我为这个教程准备好的另一个脚本。它循环运转，每次使用不同的参数集，并且评估每次运转的关键词的质量。然后它返回在这个数据集上性能最好的参数。例

如：

Python

```
$ python optimize_rake.py
1 $ python optimize_rake.py data/docs/fao_test/ 10
2 Best result at 5.56
3 with min_char_length 3
4 max_words_length 5
5 min_keyword_frequency 6
```

这些值表示在这样长的文档上，RAKE最好不包括超过5个词的候选词，并且只考虑出现少于6词的候选词。

总结一下，RAKE是一个简单的关键词提取库，它主要解决找包含频繁词的多词短语。它的强大之处在于它的易用性，它的缺点是它有限的准确度，参数配置的必须，还有它抛弃很多有效短语并且不归一化候选词。

4 用Java写的Maui提取关键词

[Maui](#)表示多用途自动主题索引。它是一个GPL许可用Java写的库，它的核心是机器学习工具包[Weka](#)。它是在[多年的研究](#)后，对[关键词提取算法KEA](#)的再现。跟RAKE相比，Maui支持：

- 不仅单从文本，还可以参照受控词表，提取关键词
- 通过用手工选择的关键词训练Maui改善准确度

4.1 配置Maui

Maui的源码在Github和Maven Central可以下载，但是最简单的方法是从Github[下载](#)Maui完整jar包，然后把jar复制到RAKE-tutorial工作目录。

4.2 Mauti：从文本提取关键词

为了比较，我们把Maui用于我们之前给RAKE使用的同一段文本。然而，因为Maui需要一个训练模型，我们首先需要创建一个训练模型。为了训练Maui，我们执行下列命令：

```
$ java -Xmx1024m -jar maui-standalone-1.1-
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar train -l data/docs/fao_train/ -m
   data/models/keyword_extraction_model -v none -o 2
```

这些参数解释如下，train用来表明我们在训练一个模型

- -l 表示文档和他们的手工关键词的路径
- -m 表示模型输出路径，
- -v none表示没有词表或者执行关键词提取，

- -o 2 表示抛弃任何候选词出现少于两次的候选词。

因为训练目录非常大，我相应增加了Java的堆空间。

一旦这个命令完成（它需要几分钟），我们就有了一个训练模型并且我们可以把它用在我们的RAKE例子中的同样文档，如下：

```
$ java -Xmx1024m -jar maui-standalone-1.1-
```

```
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar run data/docs/fao_test/w2167e.txt -m  
1 data/models/keyword_extraction_model -v none -n 8
```

run命令既可以用于一个文件路径或者一个文本串。输出应该像这样：

Python

```
Keyword: food security  
0.4168253968253969
```

```
1 Keyword: food security 0.4168253968253969  
2 Keyword: household 0.361691628264209  
3 Keyword: food production 0.3374377787854522  
4 Keyword: nutrition 0.2319136508627857  
5 Keyword: household food security and nutrition 0.21550774200419887  
6 Keyword: rural development 0.1905299881996819  
7 Keyword: forest resources 0.13882003874950102  
8 Keyword: trees 0.13051174278199784
```

我们可以通过运行test命令评价关键词质量，它会使用Maui内置的评价：

```
$ java -Xmx1024m -jar maui-standalone-1.1-
```

```
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar test -l data/docs/fao_test/ -m  
1 data/models/keyword_extraction_model -v none -n 8  
2 ...  
3 INFO MauiTopicExtractor - Avg. number of correct keyphrases per document: 2 +/- 1.03  
4 INFO MauiTopicExtractor - Precision: 25 +/- 12.91  
5 INFO MauiTopicExtractor - Recall: 26.16 +/- 15.07  
6 INFO MauiTopicExtractor - F-Measure: 25.57
```

你可以得到显著提升的性能，如果你在整个[手工标注文档](#)集上训练Maui。但是确保从训练集中去掉你用于测试的文件。

如果你对用Maui提取术语有兴趣，仅用提高概率阈值。然后记数一个文档集合中最常用的一些词。

Python

```
$ java -Xmx1024m -jar maui-standalone-1.1-
```

```
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar test -l data/docs/fao_test/ -m  
1 data/models/keyword_extraction_model -v none -n 100
```

```
2
3 $ cd data/docs/fao_test/
4 $ cat *.maui | sort | uniq -c | sort -n -r | head -n 20
5 14 FAO
6 11 used
7 11 training
8 11 supply
9 11 environment
10 10 market
11 10 developing countries
12 9 important
13 9 government
14 9 consumption
15 9 Asia
16 8 world
17 8 species
18 8 services
```

这给出了在这些文档中使用的术语种类的一个很好的指示。

4.3 Maui：用受控词表做关键词提取

假设我们在对一个文档集合的每个文档提取关键词。如果我们从文档的文本中提取关键词，他们受制于不一致性。一个作者可能会谈论“栽培树林”，另一个作者会说“人造树林”。为了对两个文档一致地使用相同关键词，用受控词表：一个术语列表、分类词典或者分类系统，是个好主意。用词表的另一个好处是它包含有助于提取过程的语义。比如，通过知道文档中“栽培树林”和“人工树林”等候选词是相关的，算法可以区分有联系的主题与联系少的主题。

Maui可以与任何RDF SKOS格式的词表工作，并且还有很多[各领域的这样的可用词表](#)。

在Maui中使用一个词表很容易。使用-v来指定词表文件的路径并且用-f指定它的格式，比如“skos”：

```
$ java -Xmx1024m -jar maui-standalone-1.1-
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar train -l data/docs/fao_train/ -m
2 data/models/term_assignment_model -v data/vocabulary/agrovoc_en.rdf.gz -f skos
3 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar run data/docs/fao_test/w2167e.txt -m
4 data/models/term_assignment_model -v data/vocabulary/agrovoc_en.rdf.gz -f skos
5
6 Keyword: Food security 0.5300188323917138
7 Keyword: Foods 0.4959033690020568
8 Keyword: Forestry 0.2994235942964757
9 Keyword: Forest products 0.29543300574208564
10 Keyword: Forest management 0.2359568256207246
11 Keyword: Community forestry 0.211907148358341
12 Keyword: Food production 0.19937233666335424
13 Keyword: Households 0.19101089588377723
14 Keyword: Forest resources 0.18298670742855433
   Keyword: Natural resources 0.15789472654988765
```

这些关键词的两个主要优势是 a)它们与一个唯一的关联于这些短语实际含义的ID绑定，并且 b)他们对于任何由同一词表分析的文档会是一致的。

```
$ java -Xmx1024m -jar maui-standalone-1.1-
```

```
1 $ java -Xmx1024m -jar maui-standalone-1.1-SNAPSHOT.jar test -l data/docs/fao_test/ -m  
2 data/models/term_assignment_model -v data/vocabulary/agrovoc_en.rdf.gz -f skos  
3 ...  
4 INFO MauiTopicExtractor - Avg. number of correct keyphrases per document: 2.94 +/- 1.57  
5 INFO MauiTopicExtractor - Precision: 29.38 +/- 15.69  
6 INFO MauiTopicExtractor - Recall: 36.18 +/- 17.6  
7 INFO MauiTopicExtractor - F-Measure: 32.43
```

这里，在50个文档上训练后，我们使用默认特征和参数测试了Maui。在[更多文档](#)上训练并且也调整了参数后，性能可以提升35%甚至更多。

5 接下来有什么？

我们已经学到了关键词提取的核心准则并且用专门为这个任务设计的Python和Java库实验了。不论你使用的库或者编程语言，你现在可以开始把这些应用到你的项目中但是如果你想继续学习，这有一些选项：

- 了解更多关于Maui如何工作的问题，可以参考[我的博士论文](#)，其中详细解释了各个方面。在[Maui的网站](#)上也有一个文档。
- 运行更多实验，你可以下载并且尝试不同的[关键词提取数据集和词表](#)。
- 对比最先进的算法评估你自己的关键词提取算法，你可以在数据集上在[SemEval的关键词提取测试](#)中对它进行性能测试

如果你在上面某个中遇到问题，尽管通过AirPair网站来联系我。

1 赞 1 收藏 [评论](#)

关于作者：[许世豪](#)



学生党。往服务器后端发展中..... (微博：@Hexus世豪) [个人主页](#) · [我的文章](#) · 10

从头开始实现神经网络：入门

本文由 [伯乐在线 - fzs](#) 翻译, [唐尤华](#) 校稿。未经许可, 禁止转载!

英文出处: [Denny Britz](#)。欢迎加入[翻译组](#)。

[获取代码: 接下来, 为了匹配文章的内容, 所有的代码都会在Github上以Python笔记的形式提供。](#)

本文中我们会从头实现一个简单的3层神经网络。我们不会推导所有的数学公式, 但会给我们正在做的事情一个相对直观的解释。我也会给出你研读所需的资源链接。

这里假设你已经比较熟悉微积分和机器学习的概念了。比如, 你知道什么是分类和正则化。当然你也应该了解一点优化技巧, 如梯度下降是如何工作的。但是即使你对上面提到的任何一个概念都不熟悉, 你仍然会发现本文的有趣所在。

但是为什么要从头实现一个神经网络呢? 即使你打算将来使用像[PyBrain](#)这样的神经网络库, 从头实现神经网络仍然是一次非常有价值的练习。它会帮助你理解神经网络的工作原理, 而这是设计有效模型的必备技能。

需要注意的是这里的示例代码并不是十分高效, 它们本就是用来帮助理解的。在接下来的文章中, 我会探索如何使用[Theano](#)写一个高效的神经网络实现。

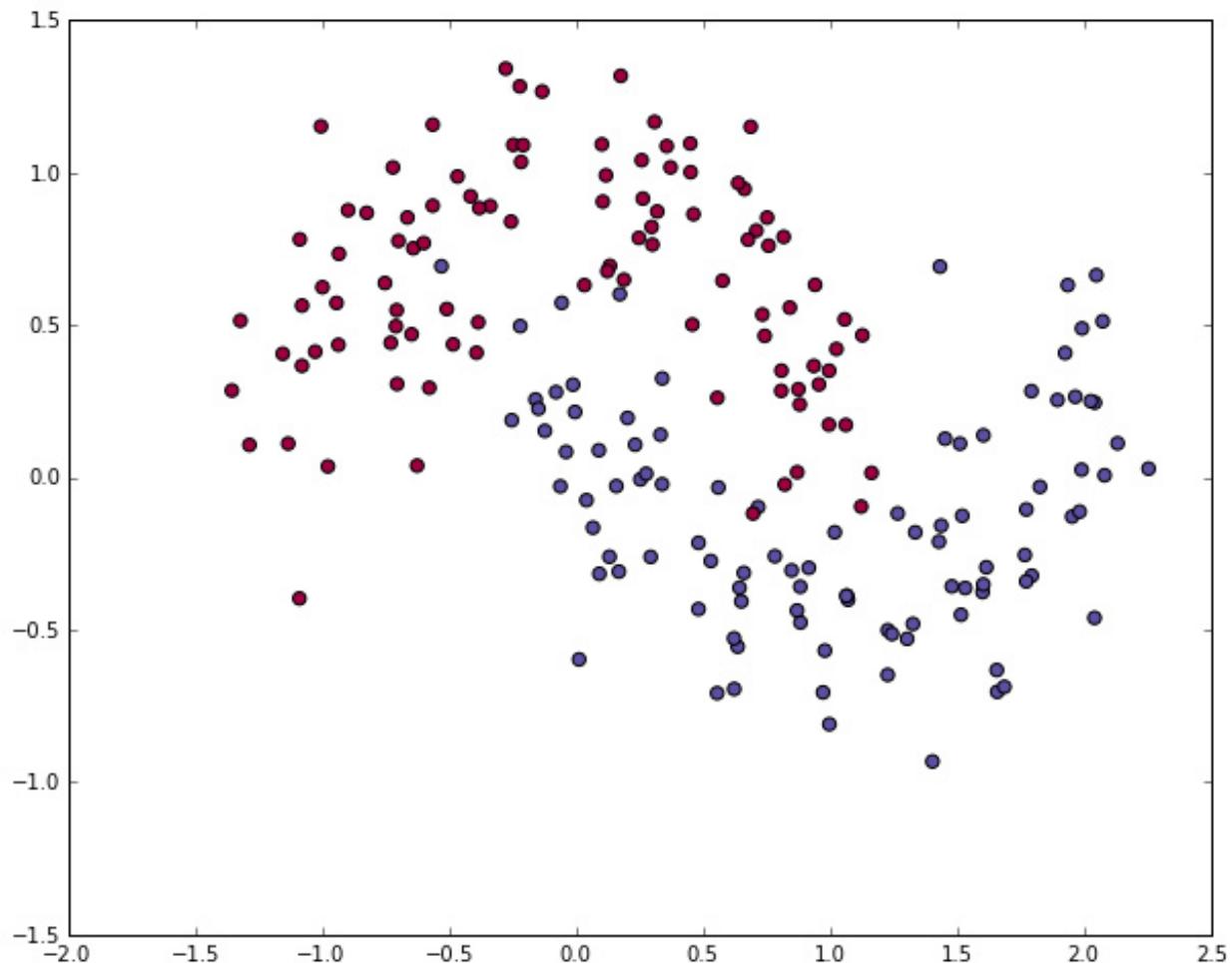
产生数据集

让我们从力所能及的产生数据集开始吧。幸运的是, [scikit-learn](#)提供了一些很有用的数据集产生器, 所以我们不需要自己写代码了。我们将从[make_moons](#) 函数开始。

Python

```
# Generate a dataset  
and plot it
```

1 # Generate a dataset and plot it
2 np.random.seed(0)
3 X, y = sklearn.datasets.make_moons(200, noise=0.20)
4 plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)



产生的数据集中有两类数据，分别以红点和蓝点表示。你可以把蓝点看作是男性病人，红点看作是女性病人， x 和 y 轴表示药物治疗。

我们的目标是，在给定 x 和 y 轴的情况下训练机器学习分类器以预测正确的分类（男女分类）。注意，数据并不是线性可分的，我们不能直接画一条直线以区分这两类数据。这意味着线性分类器，比如 Logistic 回归，将不适用于这个数据集，除非手动构建在给定数据集表现很好的非线性特征（比如多项式）。

事实上，这也是神经网络的主要优势。你不用担心[特征构建](#)，神经网络的隐藏层会为你学习特征。

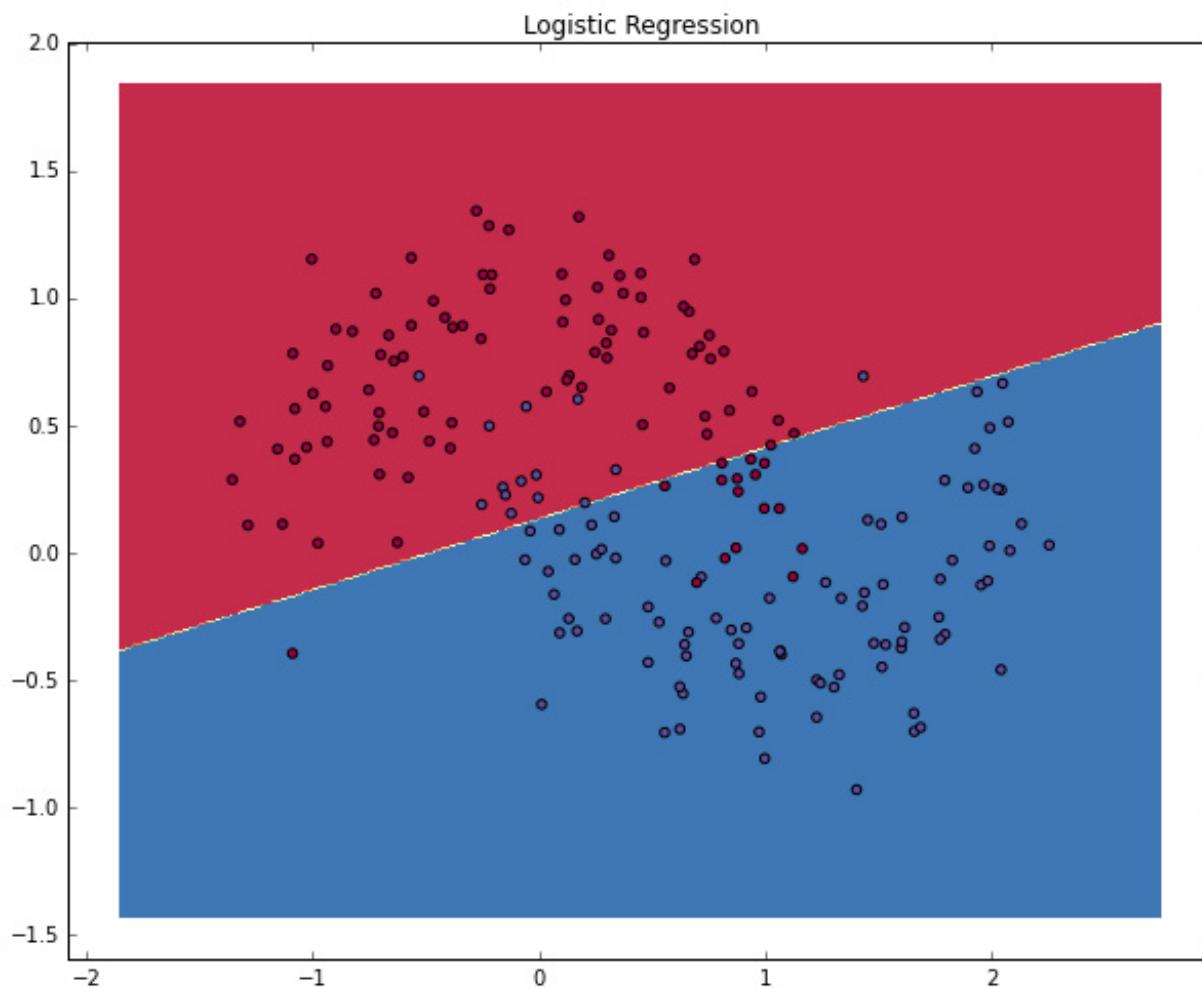
Logistic回归

为了证明这个观点，我们来训练一个Logistic回归分类器。它的输入是 x 和 y 轴的值，输出预测的分类（0或1）。为了简单，我们使用scikit-learn库里的Logistic回归类。

Python

```
# Train the logistic regression classifier
# Train the logistic regression classifier
2 clf = sklearn.linear_model.LogisticRegressionCV()
3 clf.fit(X, y)
4
```

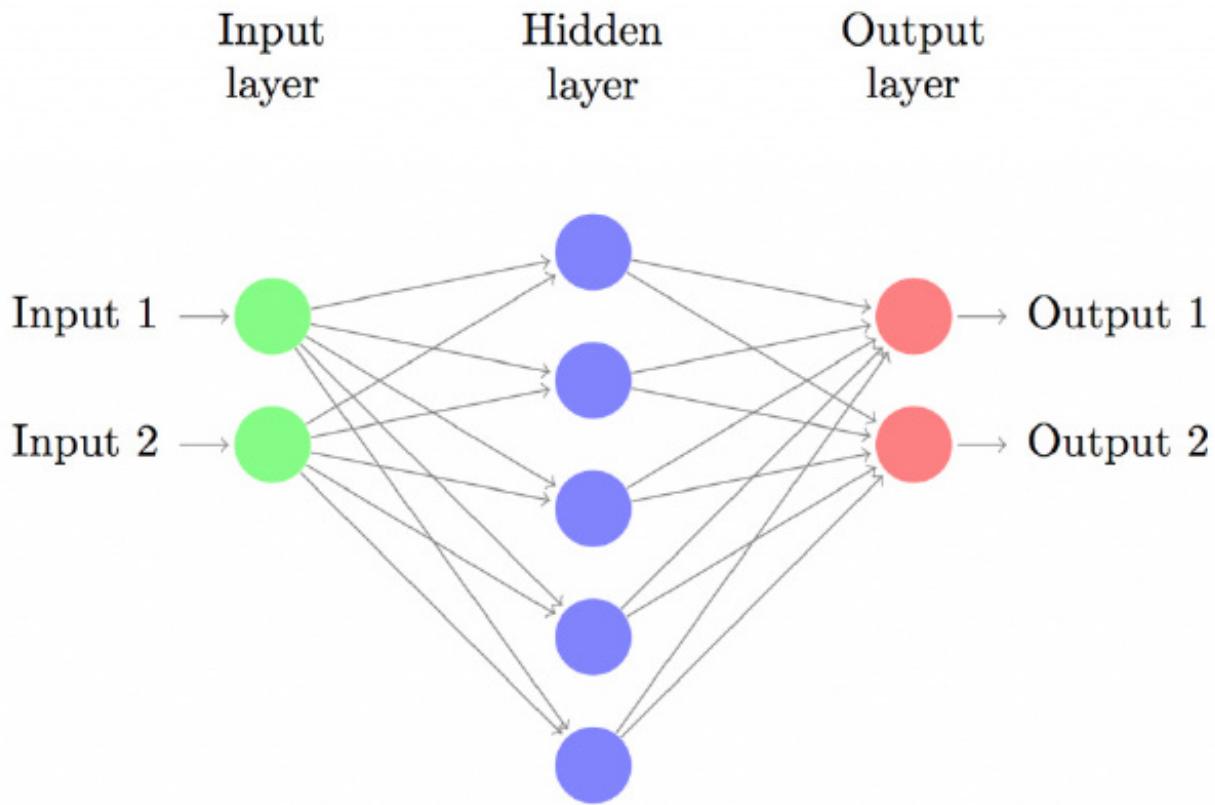
```
5 # Plot the decision boundary  
6 plot_decision_boundary(lambda x: clf.predict(x))  
7 plt.title("Logistic Regression")
```



上图展示了Logistic回归分类器学习到的决策边界。使用一条直线尽量将数据分离开来，但它并不能捕捉到数据的“月形”特征。

训练神经网络

让我们来建立具有一个输入层、一个隐藏层、一个输出层的三层神经网络。输入层的结点数由数据维度决定，这里是2维。类似地，输出层的结点数由类别数决定，也是2。（因为我们只有两类输出，实际中我们会避免只使用一个输出结点预测0和1，而是使用两个输出结点以使网络以后能很容易地扩展到更多类别）。网络的输入是x和y坐标，输出是概率，一个是0（女性）的概率，一个是1（男性）的概率。它看起来像下面这样：



我们可以为隐藏层选择维度（结点数）。放入隐藏层的结点越多，我们能训练的函数就越复杂。但是维度过高也是有代价的。首先，预测和学习网络的参数就需要更多的计算。参数越多就意味着我们可能会过度拟合数据。

如何选择隐藏层的规模？尽管有一些通用的指导和建议，但还是依赖于具体问题具体分析，与其说它是一门科学不如说是一门艺术。我们稍后会在隐藏层的结点数上多做一点事情，然后看看它会对输出有什么影响。

我们还需要为隐藏层挑选一个激活函数。激活函数将该层的输入转换为输出。一个非线性激活函数允许我们拟合非线性假设。常用的激活函数有[tanh](#)、[the sigmoid](#)函数或者是[ReLUs](#)。这里我们选择使用在很多场景下都能表现很好的tanh函数。这些函数的一个优点是它们的导数可以使用原函数值计算出来。例如， $\tanh x$ 的导数是 $1 - \tanh^2 x$ 。这个特性是很有用的，它使得我们只需要计算一次 $\tanh x$ 值，之后只需要重复使用这个值就可以得到导数值。

因为我们想要得到神经网络输出概率，所以输出层的激活函数就要是[softmax](#)。这是一种将原始分数转换为概率的方法。如果你很熟悉logistic回归，可以把softmax看作是它在多类别上的一般化。

神经网络如何预测

神经网络使用前向传播进行预测。前向传播只不过是一堆矩阵相乘并使用我们上面定义的激活函数了。假如 x 是该网络的2维输入，我们将按如下计算预测值（也是二维的）：

$$z_1 = xW_1 + b_1$$

$$a_1 = \tanh(z_1)$$

$$z_2 = a_1W_2 + b_2$$

$$a_2 = \hat{y} = \text{softmax}(z_2)$$

z_i 是输入层、 a_i 是输出层。 W_1, b_1, W_2, b_2 是需要从训练数据中学习的网络参数。你可以把它们看作是神经网络各层之间数据转换矩阵。看着上文的矩阵相乘，我们可以计算出这些矩阵的维度。如果我们的隐藏层中使用500个结点，那么有 $W_1 \in \mathbb{R}^{2 \times 500}, b_1 \in \mathbb{R}^{500}, W_2 \in \mathbb{R}^{500 \times 2}, b_2 \in \mathbb{R}^2$.

现在你明白了为什么增大隐藏层的规模会导致需要训练更多参数。

学习参数

学习该网络的参数意味着要找到使训练集上错误率最小化的参数(W_1, b_1, W_2, b_2)。但是如何定义错误率呢？我们把衡量错误率的函数叫做损失函数（loss function）。输出层为softmax时多会选择交叉熵损失（cross-entropy loss）。假如我们有N个训练例子和C个分类，那么预测值(\hat{y})相对真实标签值的损失就由下列公式给出：

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{n \in N} \sum_{i \in C} y_{n,i} \log \hat{y}_{n,i}$$

这个公式看起来很复杂，但实际上它所做的事情不过是把所有训练例子求和，然后加上预测值错误的损失。所以， \hat{y} （预测值）距离 y （真实标签值）越远，损失值就越大。

要记住，我们的目标是找到能最小化损失函数的参数值。我们可以使用梯度下降方法找到最小值。我会实现梯度下降的一种最普通的版本，也叫做有固定学习速率的批量梯度下降法。诸如SGD（随机梯度下降）或minibatch梯度下降通常在实践中有更好的表现。所以，如果你是认真的，这些可能才是你的选择，最好还能逐步衰减学习率。

作为输入，梯度下降需要一个与参数相关的损失函数的梯度（导数矢量）： $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2}$ 。为了计算这些梯度，我们使用了著名的后向传播算法。这个算法是从输出计算梯度的一种很高效的方法。在这里我不会深入讲解后向传播如何工作，但是在网络上流传有很多很优秀的讲解（参见[这里](#)或是[这里](#)）。

应用后向传播公式我们发现以下内容（这点你要相信我）：

$$\delta_3 = y - \hat{y}$$

$$\delta_2 = (1 - \tanh^2 z_2) \circ \delta_3 W_2^T$$

$$\frac{\partial L}{\partial W_2} = a_1^T \delta_3$$

$$\frac{\partial L}{\partial b_2} = \delta_3$$

$$\frac{\partial L}{\partial W_1} = x^T \delta_2$$

$$\frac{\partial L}{\partial b_1} = \delta_2$$

实现

现在我们要准备开始实现网络了。我们从定义梯度下降一些有用的变量和参数开始：

Python

```
1 num_examples = len(X) # training set size  
2 nn_input_dim = 2 # input layer dimensionality  
3 nn_output_dim = 2 # output layer dimensionality  
4  
5 # Gradient descent parameters (I picked these by hand)  
6 epsilon = 0.01 # learning rate for gradient descent  
7 reg_lambda = 0.01 # regularization strength
```

首先要实现我们上面定义的损失函数。以此来衡量我们的模型工作得如何：

Python

```
# Helper function to  
evaluate the total loss  
1 # Helper function to evaluate the total loss on the dataset  
2 def calculate_loss(model):  
3     W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']  
4     # Forward propagation to calculate our predictions  
5     z1 = X.dot(W1) + b1  
6     a1 = np.tanh(z1)  
7     z2 = a1.dot(W2) + b2  
8     exp_scores = np.exp(z2)  
9     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)  
10    # Calculating the loss  
11    correct_logprobs = -np.log(probs[range(num_examples), y])  
12    data_loss = np.sum(correct_logprobs)  
13    # Add regularization term to loss (optional)  
14    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))  
15    return 1./num_examples * data_loss
```

还要实现一个辅助函数来计算网络的输出。它的工作就是传递前面定义的前向传播并返回概率最高的类别。

Python

```
# Helper function to  
predict an output (0 or  
1 # Helper function to predict an output (0 or 1)  
2 def predict(model, x):  
3     W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']  
4     # Forward propagation  
5     z1 = x.dot(W1) + b1  
6     a1 = np.tanh(z1)  
7     z2 = a1.dot(W2) + b2  
8     exp_scores = np.exp(z2)  
9     probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)  
10    return np.argmax(probs, axis=1)
```

最后是训练神经网络的函数。它使用上文中发现的后向传播导数实现批量梯度下降。

Python

```
# This function learns
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradient descent
# - print_loss: If True, print the loss every 1000 iterations
5 def build_model(nn_hdim, num_passes=20000, print_loss=False):
6
7     # Initialize the parameters to random values. We need to learn these.
8     np.random.seed(0)
9     W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
10    b1 = np.zeros((1, nn_hdim))
11    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
12    b2 = np.zeros((1, nn_output_dim))
13
14    # This is what we return at the end
15    model = {}
16
17    # Gradient descent. For each batch...
18    for i in xrange(0, num_passes):
19
20        # Forward propagation
21        z1 = X.dot(W1) + b1
22        a1 = np.tanh(z1)
23        z2 = a1.dot(W2) + b2
24        exp_scores = np.exp(z2)
25        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
26
27        # Backpropagation
28        delta3 = probs
29        delta3[range(num_examples), y] -= 1
30        dW2 = (a1.T).dot(delta3)
31        db2 = np.sum(delta3, axis=0, keepdims=True)
32        delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
33        dW1 = np.dot(X.T, delta2)
34        db1 = np.sum(delta2, axis=0)
35
36        # Add regularization terms (b1 and b2 don't have regularization terms)
37        dW2 += reg_lambda * W2
38        dW1 += reg_lambda * W1
39
40        # Gradient descent parameter update
41        W1 += -epsilon * dW1
42        b1 += -epsilon * db1
43        W2 += -epsilon * dW2
44        b2 += -epsilon * db2
45
46        # Assign new parameters to the model
47        model = { 'W1': W1, 'b1': b1, 'W2': W2, 'b2': b2}
48
49        # Optionally print the loss.
50        # This is expensive because it uses the whole dataset, so we don't want to do it too often.
51        if print_loss and i % 1000 == 0:
52            print "Loss after iteration %i: %f" %(i, calculate_loss(model))
53
54    return model
```

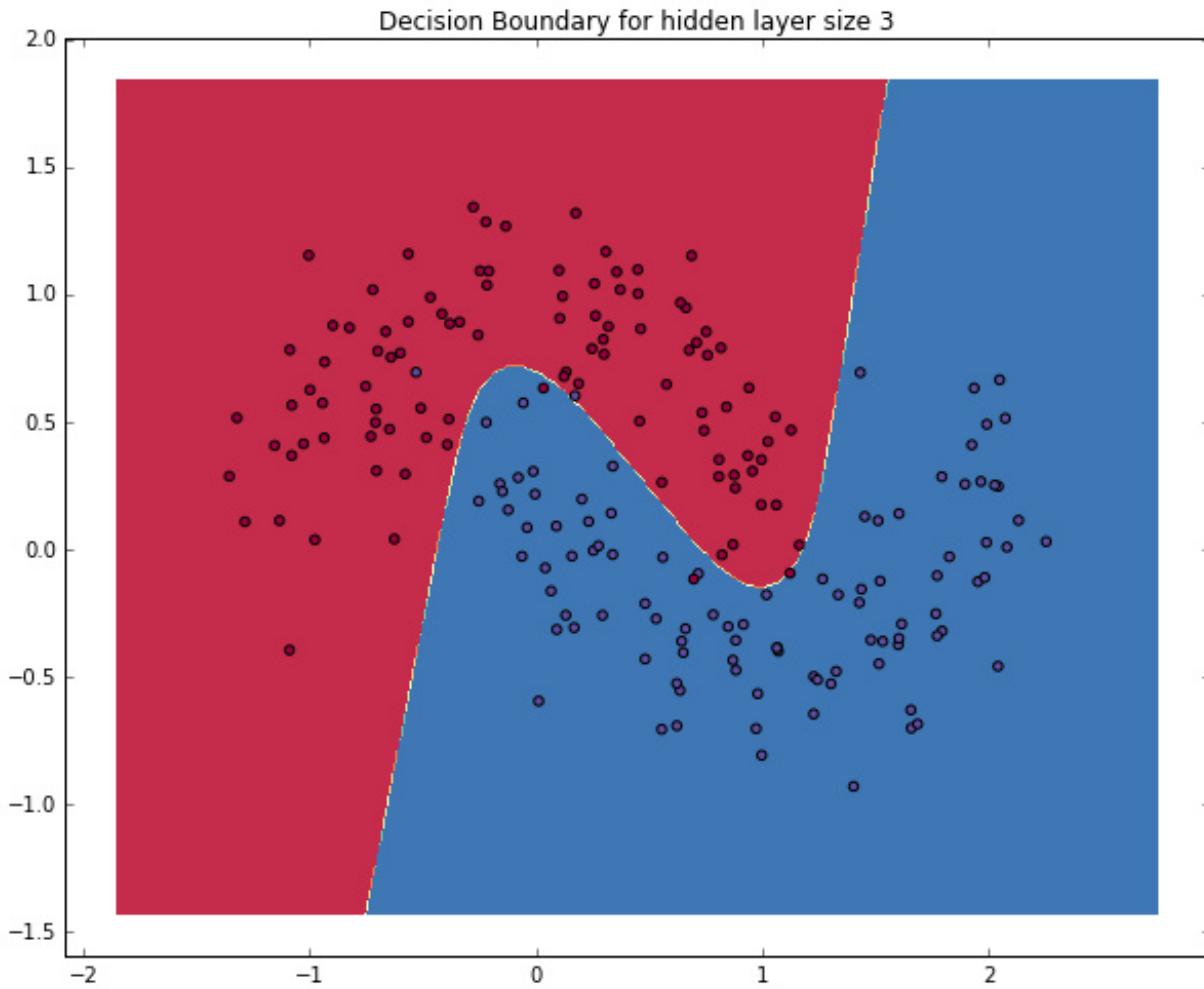
隐藏层规模为3的神经网络

一起来看看假如我们训练了一个隐藏层规模为3的神经网络会发什么。

Python

```
# Build a model with a  
3-dimensional hidden layer
```

1 # Build a model with a 3-dimensional hidden layer
2 model = build_model(3, print_loss=True)
3
4 # Plot the decision boundary
5 plot_decision_boundary(lambda x: predict(model, x))
6 plt.title("Decision Boundary for hidden layer size 3")



耶！这看起来结果相当不错。我们的神经网络能够找到成功区分类别的决策边界。

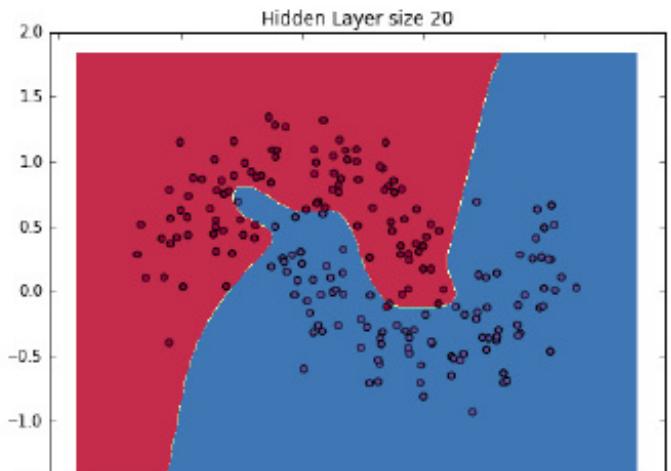
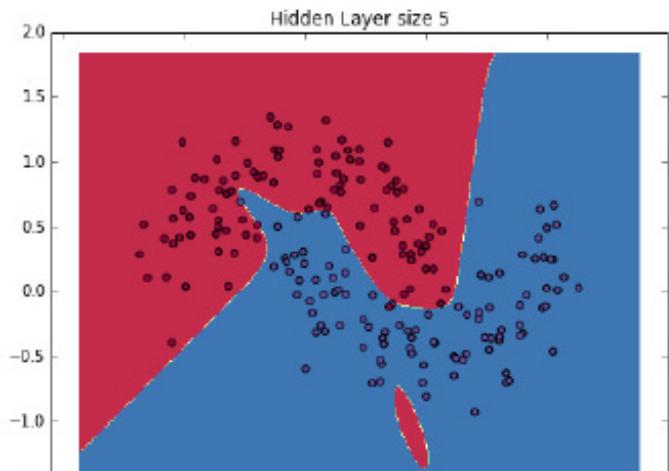
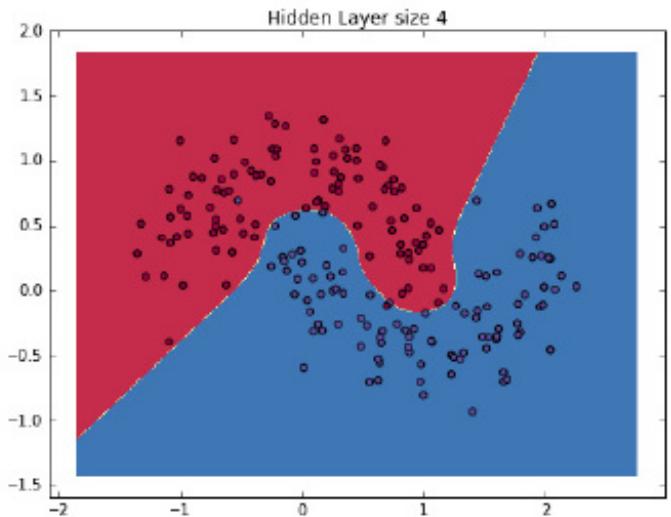
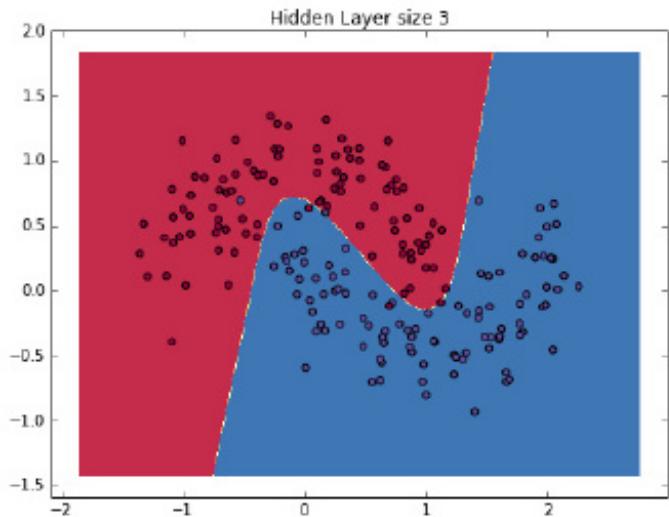
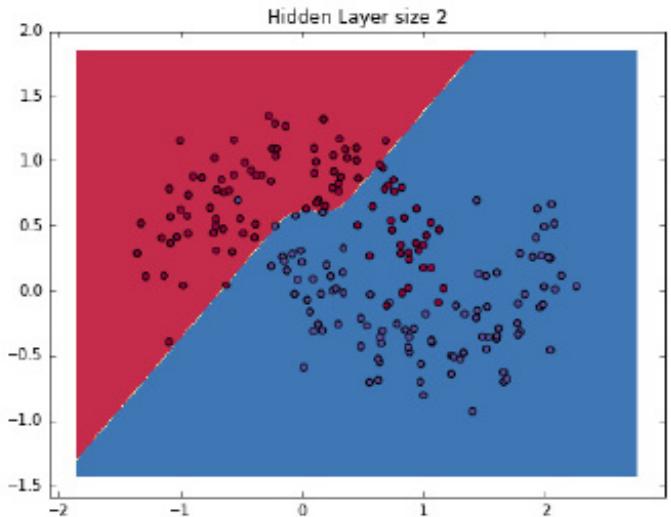
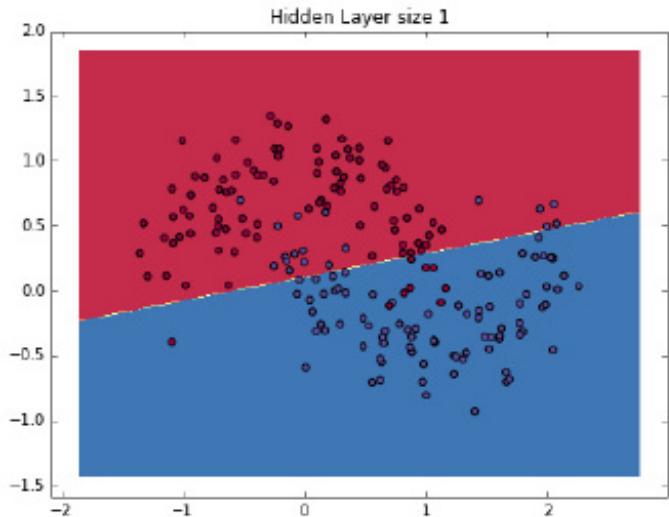
变换隐藏层的规模

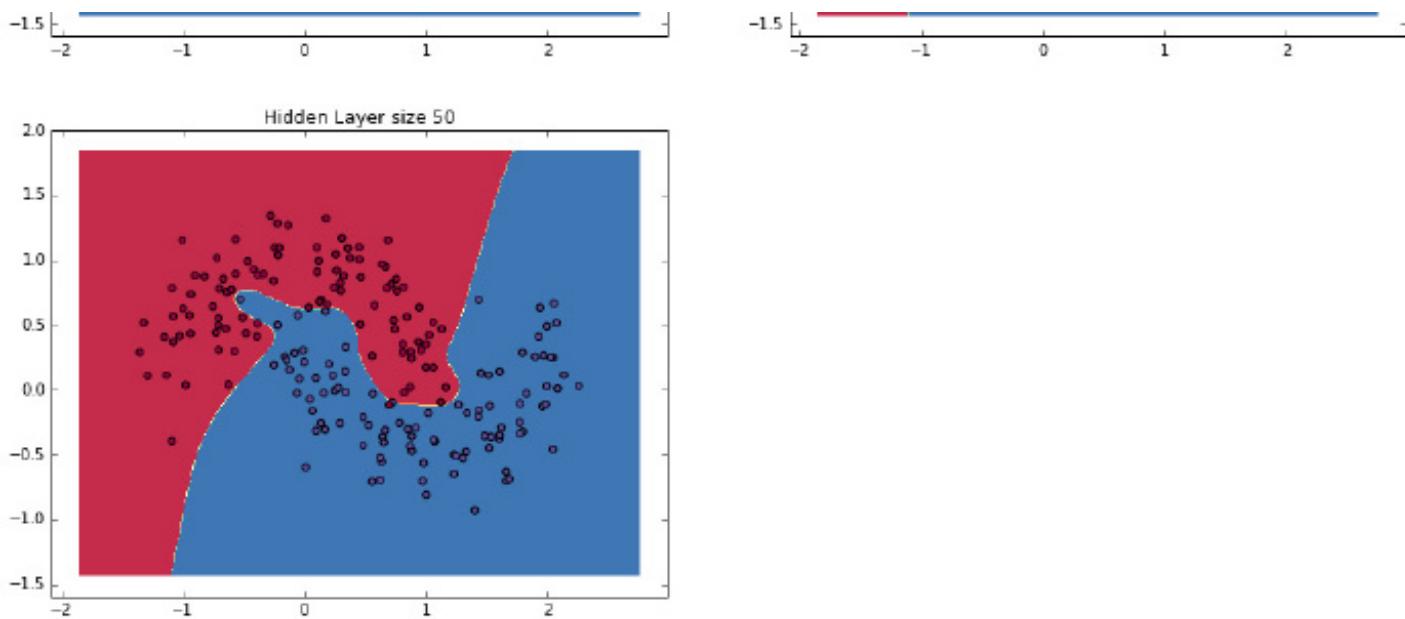
在上述例子中，我们选择了隐藏层规模为3。现在来看看改变隐藏层规模会对结果造成怎样的影响。

Python

```
plt.figure(figsize=(16, 32))
```

```
1 plt.figure(figsize=(16, 32))
2 hidden_layer_dimensions = [1, 2, 3, 4, 5, 20, 50]
3 for i, nn_hdim in enumerate(hidden_layer_dimensions):
4     plt.subplot(5, 2, i+1)
5     plt.title('Hidden Layer size %d' % nn_hdim)
6     model = build_model(nn_hdim)
7     plot_decision_boundary(lambda x: predict(model, x))
8 plt.show()
```





可以看到，低维隐藏层能够很好地捕捉到数据的总体趋势。更高的维度则更倾向于过拟合。它们更像是在“记忆”数据而不是拟合数据的大体形状。假如我们打算在独立测试集上评测该模型（你也应当这样做），隐藏层规模较小的模型会因为能更好的泛化而表现更好。虽然我们可以使用更强的泛化来抵消过拟合，但是为隐藏层选择一个合适的规模无疑是更加“经济”的方案。

练习

这里有一些练习帮助你进一步熟悉这些代码：

使用minibatch梯度下降代替批量梯度下降来训练网络 ([更多信息](#))。minibatch梯度下降在实际中一般都会表现的更好。

例子中我们使用的是固定学习速率的梯度下降 。请为梯度下降的学习速率实现一个退火算法 ([更多信息](#))。

这里我们使用的是tanh 作为隐藏层的激活函数。请尝试一下其他激活函数（比如上文中提到过的那些）。注意改变激活函数就意味着改变后向传播导数。

请扩展网络从两类输出到三类输出。你还需要为此产生一个近似数据集。

将这个网络扩展到四层。实验一下每层的规模。添加另一个隐藏层意味着前向传播和后向传播的代码都需要调整。

[所有的代码都在Github上以iPython手册的形式提供](#)。如有疑问或反馈，请留在评论中。

3 赞 23 收藏 [11 评论](#)

关于作者： [fzr](#)



微博: @fzr-fzr) [个人主页](#) · [我的文章](#) · 25

用 Python 从零开始写一个简单的解释器（1）

本文由 [伯乐在线 - fzr](#) 翻译, [黄利民](#) 校稿。未经许可, 禁止转载!
英文出处: [Jay Conrod](#)。欢迎加入[翻译组](#)。

大学里计算机科学最吸引我的地方就是编译器。最神奇的是，编译器是如何读出我写的那些烂代码，并且还能生成那么复杂的程序。当我终于选了一门编译方面的课程时，我发现这个过程比我想的要简单得多。

在本系列的文章中，我会试着通过为一种基本命令语言IMP写一个解释器，来展示这种简易性。因为IMP是一个简单广为人知的语言，所以打算用Python写这个解释器。Python代码看起来很像伪代码，所以即使你不认识Python，你也能理解它。解析可以通过一套从头开始实现的解析器组合完成（在本系列的下一篇文中会有解释）。除了sys（用于I/O）、re（用于解析正则表达式）以及unittest（用于确保一切工作正常）库，没有使用其他额外的库。

IMP 语言

在开始写之前，我们先来讨论一下将要解释的语言。IMP是拥有下面结构的最小命令语言：

赋值语句（所有变量都是全局的，而且只能存储整数）：

Python

```
x := 1
```

```
1 x := 1
```

条件语句：

Python

```
if x = 1 then  
y := 2
```

```
1 if x = 1 then  
2 y := 2  
3 else  
4 y := 3  
5 end
```

while循环：

Python

```
while x < 10 do  
x := x + 1
```

```
1 while x < 10 do  
2 x := x + 1  
3 end
```

复合语句（分号分隔）：

Python

```
x := 1;  
y := 2
```

```
1 x := 1;  
2 y := 2
```

OK，所以它只是一门工具语言，但你可以很容易就把它扩展成比Lua或python更有用的语言。我希望能把这份教程能保持尽量简单。

下面这个例子是计算阶乘的程序：

Python

```
n := 5;  
p := 1;
```

```
1 n := 5;  
2 p := 1;  
3 while n > 0 do  
4   p := p * n;  
5   n := n - 1  
6 end
```

IMP没有读取输入的方式，所以初始状态必须是在程序最开始写一系列的赋值语句。也没有打印结果的方式，所以解释器必须在程序的结尾打印所有变量的值。

解释器的结构

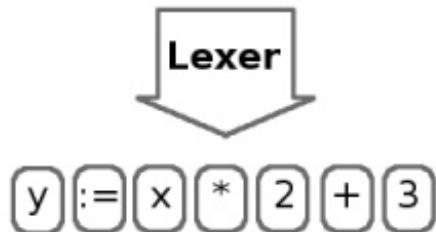
解释器的核心是「中间表示」（Intermediate representation, IR）。这就是如何在内存中表示IMP程序。因为IMP是一个很简单的语言，中间表示将直接对应于语言的语法；每一种表达和语句都有对应的类。在一种更复杂的语言中，你不仅需要一个「语法表示」，还需要一个更容易分析或运行的「语义表示」。

解释器将会执行三个阶段：

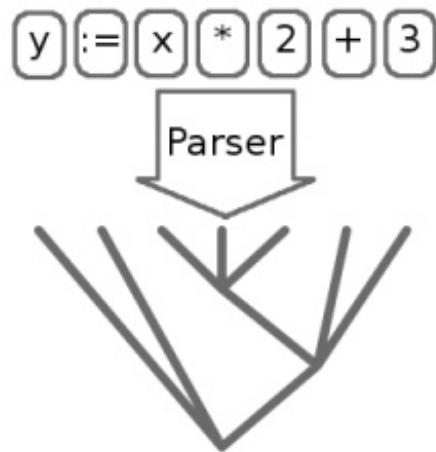
1. 将源码中的字符分割成标记符（token）
2. 将标记符组织成一棵抽象语法树（AST）。抽象语法树就是中间表示。
3. 评估这棵抽象语法树，并在最后打印这棵树的状态

将字符串分割成标记符的过程叫做「词法分析」，通过一个词法分析器完成。关键字是很短，易于理解的字符串，包含程序中最基本的部分，如数字、标识符、关键字和操作符。词法分析器会除去空格和注释，因为它们都会被解释器忽略。

```
y := x * 2 + 3
```



将标记符组织成抽象语法树（AST）的过程称为「解析过程」。解析器将程序的结构提取成一张我们可以评估的表格。



实际执行这个解析过的抽象语法树的过程称为评估。这实际上是这个解析器中最简单的部分了。

本文会把重点放在词法分析器上。我们将编写一个通用的词汇库，然后用它来为IMP创建一个词法分析器。下一篇文章将会重点打造一个语法分析器和评估计算器。

词汇库

词法分析器的操作相当简单。它是基于正则表达式的，所以如果你不熟悉它们，你可能需要[读一些资料](#)。简单来说，正则表达式就是一种能描述其他字符串的特殊的格式化的字符串。你可以使用它们去匹配电话号码或是邮箱地址，或者是像我们遇到在这种情况下，不同类型的标记符。

词法分析器的输入可能只是一个字符串。简单起见，我们将整个输入文件都读到内存中。输出是一个标记符列表。每个标记符包括一个值（它代表的字符串）和一个标记（表示它是一个什么类型的标记符）。语法分析器会使用这两个数据来决定如何构建一棵抽象语法树。

由于不论何种语言的词法分析器，其操作都大同小异，我们将创建一个通用的词法分析器，包括一个正则表达式列表和对应的标签（tag）。对每一个表达式，它都会检查是否和当前位置的输入文本匹配。如果匹配，匹配文本就会作为一个标记符被提取出来，并且被加上该正则表达式的标签。如果该正则表达式没有标签，那么这段文本将会被丢弃。这样免得我们被诸如注释和空格之类的垃圾字符干扰。如果没有匹配的正则表达式，程序就要报错并终止。这个过程会不断循环直到没有字符可匹配。

下面是一段来自词汇库的代码：

```
import sys
import re

1 import sys
2 import re
3
4 def lex(characters, token_exprs):
5     pos = 0
6     tokens = []
7     while pos < len(characters):
8         match = None
9         for token_expr in token_exprs:
10             pattern, tag = token_expr
11             regex = re.compile(pattern)
12             match = regex.match(characters, pos)
13             if match:
14                 text = match.group(0)
15                 if tag:
16                     token = (text, tag)
17                     tokens.append(token)
18                 break
19             if not match:
20                 sys.stderr.write('Illegal character: %sn' % characters[pos])
21                 sys.exit(1)
22             else:
23                 pos = match.end(0)
24     return tokens
```

注意，我们遍历正则表达式的顺序很重要。`lex`会遍历所有的表达式，然后接受第一个匹配成功的表达式。这也就意味着，当使用词法分析器时，我们应当首先考虑最具体的表达式（像那些匹配算子 (*matching operator*) 和关键词），其次才是比较一般的表达式（像标识符和数字）。

词法分析器

给定上面的`lex`函数，为IMP定义一个词法分析器就非常简单了。首先我们要做的就是为标记符定义一系列的标签。IMP只需要三个标签。RESERVED表示一个保留字或操作符。INT表示一个文字整数。ID代表标识符。

Python

```
import lexer

1 import lexer
2
3 RESERVED = 'RESERVED'
4 INT      = 'INT'
5 ID       = 'ID'
```

接下来定义词法分析器将会用到的标记符表达式。前两个表达式匹配空格和注释。它们没有标签，所以`lex`会丢弃它们匹配到的所有字符。

Python

```
token_exprs = [
    (r'[ \t]+',
```

```
1 token_exprs = [  
2     (r'[ nt]+', None),  
3     (r'#[^n]*', None),
```

然后，只剩下所有的操作符和保留字了。记住，每个正则表达式前面的“r”表示这个字符串是“raw”；Python不会处理任何转义字符。这使我们可以在字符串中包含进反斜线，正则表达式正是利用这一点来转义操作符比如“+”和“*”。

Python

```
(r':=',
RESERVED),
```



```
1 (r':=', RESERVED),
2 (r'(', RESERVED),
3 (r')', RESERVED),
4 (r';', RESERVED),
5 (r'+', RESERVED),
6 (r'-', RESERVED),
7 (r '**', RESERVED),
8 (r'/', RESERVED),
9 (r'<=', RESERVED),
10 (r'<', RESERVED),
11 (r'>=', RESERVED),
12 (r'>', RESERVED),
13 (r'=', RESERVED),
14 (r'!=', RESERVED),
15 (r'and', RESERVED),
16 (r'or', RESERVED),
17 (r'not', RESERVED),
18 (r'if', RESERVED),
19 (r'then', RESERVED),
20 (r'else', RESERVED),
21 (r'while', RESERVED),
22 (r'do', RESERVED),
23 (r'end', RESERVED),
```

最后，轮到整数和标识符的表达式。要注意的是，标识符的正则表达式会匹配上面的所有的保留字，所以它一定要留到最后。

Python

```
(r'[0-9]+',
INT),
```



```
1 (r'[0-9]+', INT),
2 (r'[A-Za-z][A-Za-z0-9_]*', ID),
3 ]
```

既然正则表达式已经定义好了，我们还需要创建一个实际的lexer函数。

Python

```
def imp_lex(characters):
```



```
1 def imp_lex(characters):
2     return lexer.lex(characters, token_exprs)
```

如果你对这部分感兴趣，这里有一些驱动代码可以测试输出：

Python

```
import sys
from imp_lexer import *

1 import sys
2 from imp_lexer import *
3
4 if __name__ == '__main__':
5     filename = sys.argv[1]
6     file = open(filename)
7     characters = file.read()
8     file.close()
9     tokens = imp_lex(characters)
10    for token in tokens:
11        print token
```

继续.....

在本系列的下一篇篇文章中，我会讨论解析器组合，然后描述如何使用他们从lexer中生成的标记符列表建立抽象语法树。

如果你对于实现IMP解释器很感兴趣，你可以从[这里下载全部的源码](#)。

在源码包含的示例文件中运行解释器：

Python

```
python imp.py hello.imp
```

```
1 python imp.py hello.imp
```

运行单元测试：

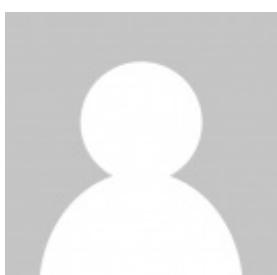
Python

```
python test.py
```

```
1 python test.py
```

1 赞 8 收藏 [7 评论](#)

关于作者：[fzr](#)



Python函数式编程指南（4）：生成器

原文出处：[AstralWind](#)

4. 生成器(generator)

4.1. 生成器简介

首先请确信，生成器就是一种迭代器。生成器拥有next方法并且行为与迭代器完全相同，这意味着生成器也可以用于Python的for循环中。另外，对于生成器的特殊语法支持使得编写一个生成器比自定义一个常规的迭代器要简单不少，所以生成器也是最常用到的特性之一。

从Python 2.5开始，[PEP 342：通过增强生成器实现协同程序]的实现为生成器加入了更多的特性，这意味着生成器还可以完成更多的工作。这部分我们会在稍后的部分介绍。

4.2. 生成器函数

4.2.1. 使用生成器函数定义生成器

如何获取一个生成器？首先来看一小段代码：

Python

```
>>> def get_0_1_2():
...     yield 0
1 >>> def get_0_1_2():
2 ...     yield 0
3 ...     yield 1
4 ...     yield 2
5 ...
6 >>> get_0_1_2
7 <function get_0_1_2 at 0x00B2CB70>
```

我们定义了一个函数get_0_1_2，并且可以查看到这确实是函数类型。但与一般的函数不同的是，get_0_1_2的函数体内使用了关键字yield，这使得get_0_1_2成为了一个生成器函数。生成器函数的特性如下：

调用生成器函数将返回一个生成器；

Python

```
>>> generator =
get_0_1_2()
1 >>> generator = get_0_1_2()
2 >>> generator
3 <generator object get_0_1_2 at 0x00B1C7D8>
```

第一次调用生成器的next方法时，生成器才开始执行生成器函数（而不是构建生成器时），直到遇到yield时暂停执行（挂起），并且yield的参数将作为此次next方法的返回值；

Python

```
>>> generator.next()
```

```
0
```

```
1 >>> generator.next()
```

```
2 0
```

之后每次调用生成器的next方法，生成器将从上次暂停执行的位置恢复执行生成器函数，直到再次遇到yield时暂停，并且同样的，yield的参数将作为next方法的返回值；

Python

```
>>> generator.next()
```

```
1
```

```
1 >>> generator.next()
```

```
2 1
```

```
3 >>> generator.next()
```

```
4 2
```

如果当调用next方法时生成器函数结束（遇到空的return语句或是到达函数体末尾），则这次next方法的调用将抛出StopIteration异常（即for循环的终止条件）；

Python

```
>>> generator.next()
```

```
Traceback (most recent)
```

```
1 >>> generator.next()
```

```
2 Traceback (most recent call last):
```

```
3   File "<stdin>", line 1, in <module>
```

```
4 StopIteration
```

生成器函数在每次暂停执行时，函数体内的所有变量都将被封存(freeze)在生成器中，并将在恢复执行时还原，并且类似于闭包，即使是同一个生成器函数返回的生成器，封存的变量也是互相独立的。我们的小例子中并没有用到变量，所以这里另外定义一个生成器来展示这个特点：

Python

```
>>> def fibonacci():
```

```
...   a = b = 1
```

```
1 >>> def fibonacci():
```

```
2 ...   a = b = 1
```

```
3 ...   yield a
```

```
4 ...   yield b
```

```
5 ...   while True:
```

```
6 ...     a, b = b, a+b
```

```
7 ...     yield b
```

```
8 ...
```

```
9 >>> for num in fibonacci():
```

```
10 ...   if num > 100: break
```

```
11 ...   print num,
```

```
12 ...
```

```
13 1 1 2 3 5 8 13 21 34 55 89
```

看到while True可别太吃惊，因为生成器可以挂起，所以是延迟计算的，无限循环并没有关系。这个例子中我们定义了一个生成器用于获取斐波那契数列。

4.2.2. 生成器函数的FAQ

接下来我们来讨论一些关于生成器的有意思的话题。

1. 你的例子里生成器函数都没有参数，那么生成器函数可以带参数吗？

当然可以啊亲，而且它支持函数的所有参数形式。要知道生成器函数也是函数的一种：）

Python

```
>>> def
counter(start=0):
1 >>> def counter(start=0):
2 ... while True:
3 ...     yield start
4 ...     start += 1
5 ...
```

这是一个从指定数开始的计数器。

2. 既然生成器函数也是函数，那么它可以使用return输出返回值吗？

不行的亲，是这样的，生成器函数已经有默认的返回值——生成器了，你不能再另外给一个返回值；对，即使是return None也不行。但是它可以使用空的return语句结束。如果你坚持要为它指定返回值，那么Python将在定义的位置赠送一个语法错误异常，就像这样：

Python

```
>>> def
i_wanna_return():
1 >>> def i_wanna_return():
2 ...     yield None
3 ...     return None
4 ...
5 File "<stdin>", line 3
6 SyntaxError: 'return' with argument inside generator
```

3. 好吧，那人家需要确保释放资源，需要在try...finally中yield，这会是神马情况？（我就是想玩你）我在finally中还yield了一次！

Python会在真正离开try...finally时再执行finally中的代码，而这里遗憾地告诉你，暂停不算哦！所以结局你也能猜到吧！

Python

```
>>> def play_u():
...     try:
1 >>> def play_u():
2 ...     try:
3 ...         yield 1
4 ...         yield 2
5 ...         yield 3
6 ...     finally:
7 ...         yield 0
```

```
8 ...
9  >>> for val in play_u(): print val,
10 ...
11 1 2 3 0
```

*这与return的情况不同。return是真正的离开代码块，所以会在return时立刻执行finally子句。

*另外，“在带有finally子句的try块中yield”定义在PEP 342中，这意味着只有Python 2.5以上版本才支持这个语法，在Python 2.4以下版本中会得到语法错误异常。

4. 如果我需要在生成器的迭代过程中接入另一个生成器的迭代怎么办？写成下面这样好傻好天真。。

Python

```
>>> def
sub_generator():
1 >>> def sub_generator():
2 ...   yield 1
3 ...   yield 2
4 ...   for val in counter(10): yield val
5 ...
```

这种情况的语法改进已经被定义在[PEP 380：委托至子生成器的语法]中，据说会在Python 3.3中实现，届时也可能回馈到2.x中。实现后，就可以这么写了：

Python

```
>>> def
sub_generator():
1 >>> def sub_generator():
2 ...   yield 1
3 ...   yield 2
4 ...   yield from counter(10)
5 File "<stdin>", line 4
6     yield from counter(10)
7           ^
8 SyntaxError: invalid syntax
```

看到语法错误木有？现在我们还是天真一点吧~

有问题？请回复此文：)

4.3. 协同程序(coroutine)

协同程序（协程）一般来说是指这样的函数：

- 彼此间有不同的局部变量、指令指针，但仍共享全局变量；
- 可以方便地挂起、恢复，并且有多个入口点和出口点；
- 多个协同程序间表现为协作运行，如A的运行过程中需要B的结果才能继续执行。

协程的特点决定了同一时刻只能有一个协同程序正在运行（忽略多线程的情况）。得益于此，协程可以直接传递对象而不需要考虑资源锁、或是直接唤醒其他协程而不需要主动休眠，就像是内置了锁的线程。在符合协程特点的应用场景，使用协程无疑比使用线程要更方便。

从另一方面说，协程无法并发其实也将它的应用场景限制在了一个很狭窄的范围，这个特点使得协程

更多的被拿来与常规函数进行比较，而不是与线程。当然，线程比协程复杂许多，功能也更强大，所以我建议大家牢牢地掌握线程即可：Python线程指南

这一节里我也就不列举关于协程的例子了，以下介绍的方法了解即可。

Python 2.5对生成器的增强实现了协程的其他特点，在这个版本中，生成器加入了如下方法：

1. send(value):

send是除next外另一个恢复生成器的方法。Python 2.5中，yield语句变成了yield表达式，这意味着yield现在可以有一个值，而这个值就是在生成器的send方法被调用从而恢复执行时，调用send方法的参数。

Python

```
>>> def repeater():
...     n = 0
...
1  >>> def repeater():
2  ...     n = 0
3  ...     while True:
4  ...         n = (yield n)
5  ...
6  >>> r = repeater()
7  >>> r.next()
8  0
9  >>> r.send(10)
10 10
```

*调用send传入非None值前，生成器必须处于挂起状态，否则将抛出异常。不过，未启动的生成器仍可以使用None作为参数调用send。

*如果使用next恢复生成器，yield表达式的值将是None。

2. close():

这个方法用于关闭生成器。对关闭的生成器后再次调用next或send将抛出StopIteration异常。

3. throw(type, value=None, traceback=None):

这个方法用于在生成器内部（生成器的当前挂起处，或未启动时在定义处）抛出一个异常。

*别为没见到协程的例子遗憾，协程最常见的用处其实也是生成器。

4.4. 一个有趣的库：pipe

这一节里我要向诸位简要介绍pipe。pipe并不是Python内置的库，如果你安装了easy_install，直接可以安装它，否则你需要自己下载它：<http://pypi.python.org/pypi/pipe>

之所以要介绍这个库，是因为它向我们展示了一种很有新意的使用迭代器和生成器的方式：流。pipe将可迭代的数据看成是流，类似于linux，pipe使用'|'传递数据流，并且定义了一系列的“流处理”函数用于接受并处理数据流，并最终再次输出数据流或者是将数据流归纳得到一个结果。我们来看一些例子。

第一个，非常简单的，使用add求和：

Python

```
>>> from pipe import *  
>>> range(5) | add  
1 >>> from pipe import *  
2 >>> range(5) | add  
3 10
```

求偶数和需要使用到where，作用类似于内建函数filter，过滤出符合条件的元素：

Python

```
>>> range(5) |  
where(lambda x: x % 2  
1 >>> range(5) | where(lambda x: x % 2 == 0) | add  
2 6
```

还记得我们定义的斐波那契数列生成器吗？求出数列中所有小于10000的偶数和需要用到take_while，与itertools的同名函数有类似的功能，截取元素直到条件不成立：

Python

```
>>> fib = fibonacci  
>>> fib() |  
1 >>> fib = fibonacci  
2 >>> fib() | where(lambda x: x % 2 == 0) |  
3 ... | take_while(lambda x: x < 10000) |  
4 ... | add  
5 3382
```

需要对元素应用某个函数可以使用select，作用类似于内建函数map；需要得到一个列表，可以使用as_list：

Python

```
>>> fib() | select(lambda  
x: x ** 2) |  
1 >>> fib() | select(lambda x: x ** 2) | take_while(lambda x: x < 100) | as_list  
2 [1, 1, 4, 9, 25, 64]
```

pipe中还包括了更多的流处理函数。你甚至可以自己定义流处理函数，只需要定义一个生成器函数并加上修饰器Pipe。如下定义了一个获取元素直到索引不符合条件的流处理函数：

Python

```
>>> @Pipe  
... def  
1 >>> @Pipe  
2 ... def take_while_idx(iterable, predicate):  
3 ... for idx, x in enumerate(iterable):  
4 ... if predicate(idx): yield x  
5 ... else: return  
6 ...
```

使用这个流处理函数获取fib的前10个数字：

Python

```
>>> fib() |  
take_while_idx(lambda  
1 >>> fib() | take_while_idx(lambda x: x < 10) | as_list  
2 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

更多的函数就不在这里介绍了，你可以查看pipe的源文件，总共600行不到的文件其中有300行是文档，文档中包含了大量的示例。

pipe实现起来非常简单，使用Pipe装饰器，将普通的生成器函数（或者返回迭代器的函数）代理在一个实现了__ror__方法的普通类实例上即可，但是这种思路真的很有趣。

函数式编程指南全文到这里就全部结束了，希望这一系列文章能给你带来帮助。希望大家都能看到一些结构式编程之外的编程方式，并且能够熟练地在恰当的地方使用：）

明天我会整理一个目录放上来方便查看，并且列出一些供参考的文章。遗憾的是这些文章几乎都是英文的，请努力学习英语吧 - -#

1 赞 5 收藏 [评论](#)

Python函数式编程指南（3）：迭代器

原文出处：[AstralWind](#)

3. 迭代器

3.1. 迭代器(Iterator)概述

迭代器是访问集合内元素的一种方式。迭代器对象从集合的第一个元素开始访问，直到所有的元素都被访问一遍后结束。

迭代器不能回退，只能往前进行迭代。这并不是什么很大的缺点，因为人们几乎不需要在迭代途中进行回退操作。

迭代器也不是线程安全的，在多线程环境中对可变集合使用迭代器是一个危险的操作。但如果小心谨慎，或者干脆贯彻函数式思想坚持使用不可变的集合，那这也不是什么大问题。

对于原生支持随机访问的数据结构（如tuple、list），迭代器和经典for循环的索引访问相比并无优势，反而丢失了索引值（可以使用内建函数enumerate()找回这个索引值，这是后话）。但对于无法随机访问的数据结构（比如set）而言，迭代器是唯一的访问元素的方式。

迭代器的另一个优点就是它不要求你事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合，比如几个G的文件，或是斐波那契数列等等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)。

迭代器更大的功劳是提供了一个统一的访问集合的接口。只要是实现了__iter__()方法的对象，就可以使用迭代器进行访问。

3.2. 使用迭代器

使用内建的工厂函数iter(iterable)可以获取迭代器对象：

Python

```
>>> lst = range(2)
>>> it = iter(lst)

1 >>> lst = range(2)
2 >>> it = iter(lst)
3 >>> it
4 <listiterator object at 0x00BB62F0>
```

使用迭代器的next()方法可以访问下一个元素：

Python

```
>>> it.next()
0

1 >>> it.next()
```

如果是Python 2.6+，还有内建函数next(iterator)可以完成这一功能：

Python

```
>>> next(it)
1
```

```
1 >>> next(it)
2 1
```

如何判断迭代器还有更多的元素可以访问呢？Python里的迭代器并没有提供类似has_next()这样的方法。

那么在这个例子中，我们已经访问到了最后一个元素1，再使用next()方法会怎样呢？

Python

```
>>> it.next()
Traceback (most recent)
```

```
1 >>> it.next()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 StopIteration
```

Python遇到这样的情况时将会抛出StopIteration异常。事实上，Python正是根据是否检查到这个异常来决定是否停止迭代的。

这种做法与迭代前手动检查是否越界相比各有优点。但Python的做法总有一些利用异常进行流程控制的嫌疑。

了解了这些情况以后，我们就能使用迭代器进行遍历了。

Python

```
it = iter(lst)
try:
```

```
1 it = iter(lst)
2 try:
3   while True:
4     val = it.next()
5     print val
6 except StopIteration:
7   pass
```

实际上，因为迭代操作如此普遍，Python专门将关键字for用作了迭代器的语法糖。在for循环中，Python将自动调用工厂函数iter()获得迭代器，自动调用next()获取元素，还完成了检查StopIteration异常的工作。上述代码可以写成如下的形式，你一定非常熟悉：

Python

```
for val in lst:
    print val
```

```
1 for val in lst:
2   print val
```

首先Python将对关键字in后的对象调用iter函数获取迭代器，然后调用迭代器的next方法获取元素，直到抛出StopIteration异常。对迭代器调用iter函数时将返回迭代器自身，所以迭代器也可以用于for语句中，不需要特殊处理。

常用的几个内建数据结构tuple、list、set、dict都支持迭代器，字符串也可以使用迭代操作。你也可以自己实现一个迭代器，如上所述，只需要在类的__iter__方法中返回一个对象，这个对象拥有一个next()方法，这个方法能在恰当的时候抛出StopIteration异常即可。但是需要自己实现迭代器的时候不多，即使需要，使用生成器会更轻松。下一篇我们将讨论生成器的部分。

*异常并不是非抛出不可的，不抛出该异常的迭代器将进行无限迭代，某些情况下这样的迭代器很有用。这种情况下，你需要自己判断元素并中止，否则就死循环了！

使用迭代器的循环可以避开索引，但有时候我们还是需要索引来进行一些操作的。这时候内建函数enumerate就派上用场咯，它能在iter函数的结果前加上索引，以元组返回，用起来就像这样：

Python

```
for idx, ele in  
enumerate(lst):  
    print(idx, ele)
```

3.3. 生成器表达式(Generator expression)和列表解析(List Comprehension)

绝大多数情况下，遍历一个集合都是为了对元素应用某个动作或是进行筛选。如果看过本文的第二部分，你应该还记得有内建函数map和filter提供了这些功能，但Python仍然为这些操作提供了语言级的支持。

Python

```
(x+1 for x in lst) #生成器  
表达式，返回迭代器。  
1 (x+1 for x in lst) #生成器表达式，返回迭代器。外部的括号可在用于参数时省略。  
2 [x+1 for x in lst] #列表解析，返回list
```

如你所见，生成器表达式和列表解析（注：这里的翻译有很多种，比如列表展开、列表推导等等，指的是同一个意思）的区别很小，所以人们提到这个特性时，简单起见往往只描述成列表解析。然而由于返回迭代器时，并不是一开始就计算所有的元素，这样能得到更多的灵活性并且可以避开很多不必要的计算，所以除非你明确希望返回列表，否则应该始终使用生成器表达式。接下来的文字里我就不分这两种形式了：）

你也可以为列表解析提供if子句进行筛选：

Python

```
(x+1 for x in lst if x!=0)  
1 (x+1 for x in lst if x!=0)
```

或者提供多条for子句进行嵌套循环，嵌套次序就是for子句的顺序：

Python

```
((x, y) for x in range(3) for  
y in range(x))  
1 ((x, y) for x in range(3) for y in range(x))
```

列表解析就是鲜明的Pythonic。我常遇到两个使用列表解析的问题，本应归属于最佳实践，但这两个问题非常典型，所以不妨在这里提一下：

第一个问题是，因为对元素应用的动作太复杂，不能用一个表达式写出来，所以不使用列表解析。这是典型的思想没有转变的例子，如果我们将动作封装成函数，那不就是一个表达式了么？

第二个问题是，因为if子句里的条件需要计算，同时结果也需要进行同样的计算，不希望计算两遍，就像这样：

Python

```
(x.doSomething() for x in  
lst if x.doSomething()>0)  
1 (x.doSomething() for x in lst if x.doSomething()>0)
```

这样写确实很糟糕，但组合一下列表解析即可解决：

Python

```
(x for x in  
(y.doSomething() for y in lst) if x>0)  
1 (x for x in (y.doSomething() for y in lst) if x>0)
```

内部的列表解析变量其实也可以用x，但为清晰起见我们改成了y。或者更清楚的，可以写成两个表达式：

Python

```
tmp = (x.doSomething()  
for x in lst)  
1 tmp = (x.doSomething() for x in lst)  
2 (x for x in tmp if x > 0)
```

列表解析可以替代绝大多数需要用到map和filter的场合，可能正因为此，著名的静态检查工具pylint将map和filter的使用列为了警告。

3.4. 相关的库

Python内置了一个模块itertools，包含了很多函数用于creating iterators for efficient looping（创建更有效率的循环迭代器），这说明很是霸气，这一小节就来浏览一遍这些函数并留下印象吧，需要这些功能的时候隐约记得这里面有就好。这一小节的内容翻译自itertools模块官方文档。

3.4.1. 无限迭代

- count(start, [step])

从start开始，以后每个元素都加上step。step默认值为1。

count(10) → 10 11 12 13 14 ...

- cycle(p)

迭代至序列p的最后一个元素后，从p的第一个元素重新开始。

cycle('ABCD') → A B C D A B C D ...

- repeat(elem [,n])

将elem重复n次。如果不指定n，则无限重复。

repeat(10, 3) → 10 10 10

3.4.2. 在最短的序列参数终止时停止迭代

- chain(p, q, ...)

迭代至序列p的最后一个元素后，从q的第一个元素开始，直到所有序列终止。

chain('ABC', 'DEF') → A B C D E F

- compress(data, selectors)

如果bool(selectors[n])为True，则next()返回data[n]，否则跳过data[n]。

compress('ABCDEF', [1,0,1,0,1,1]) → A C E F

- dropwhile(pred, seq)

当pred对seq[n]的调用返回False时才开始迭代。

dropwhile(lambda x: x<5, [1,4,6,4,1]) → 6 4 1

- takewhile(pred, seq)

dropwhile的相反版本。

takewhile(lambda x: x<5, [1,4,6,4,1]) → 1 4

- ifilter(pred, seq)

内建函数filter的迭代器版本。

ifilter(lambda x: x%2, range(10)) → 1 3 5 7 9

- ifilterfalse(pred, seq)

ifilter的相反版本。

ifilterfalse(lambda x: x%2, range(10)) → 0 2 4 6 8

- imap(func, p, q, ...)

内建函数map的迭代器版本。

imap(pow, (2,3,10), (5,2,3)) → 32 9 1000

- `starmap(func, seq)`

将`seq`的每个元素以变长参数(*args)的形式调用`func`。

`starmap(pow, [(2,5), (3,2), (10,3)]) -> 32 9 1000`

- `izip(p, q, ...)`

内建函数`zip`的迭代器版本。

`izip('ABCD', 'xy') -> Ax By`

- `izip_longest(p, q, ..., fillvalue=None)`

`izip`的取最长序列的版本，短序列将填入`fillvalue`。

`izip_longest('ABCD', 'xy', fillvalue='-') -> Ax By C- D-`

- `tee(it, n)`

返回n个迭代器`it`的复制迭代器。

- `groupby(iterable[, keyfunc])`

这个函数功能类似于SQL的分组。使用`groupby`前，首先需要使用相同的`keyfunc`对`iterable`进行排序，比如调用内建的`sorted`函数。然后，`groupby`返回迭代器，每次迭代的元素是元组(key值, `iterable`中具有相同key值的元素的集合的子迭代器)。或许看看Python的排序指南对理解这个函数有帮助。

`groupby([0, 0, 0, 1, 1, 1, 2, 2, 2]) -> (0, (0 0 0)) (1, (1 1 1)) (2, (2 2 2))`

3.4.3. 组合迭代器

- `product(p, q, ... [repeat=1])`

笛卡尔积。

`product('ABCD', repeat=2) -> AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`

- `permutations(p[, r])`

去除重复的元素。

`permutations('ABCD', 2) -> AB AC AD BA BC BD CA CB CD DA DB DC`

- `combinations(p, r)`

排序后去除重复的元素。

`combinations('ABCD', 2) -> AB AC AD BC BD CD`

- `combinations_with_replacement()`

排序后，包含重复元素。

`combinations_with_replacement('ABCD', 2) -> AA AB AC AD BB BC BD CC CD DD`

此篇结束。

1 赞 2 收藏 [评论](#)

Python函数式编程指南（2）：函数

原文出处：[AstralWind](#)

2. 从函数开始

2.1. 定义一个函数

如下定义了一个求和函数：

Python

```
def add(x, y):
    return x + y
1 def add(x, y):
2     return x + y
```

关于参数和返回值的语法细节可以参考其他文档，这里就略过了。

使用lambda可以定义简单的单行匿名函数。lambda的语法是：

Python

```
lambda args: expression
1 lambda args: expression
```

参数(args)的语法与普通函数一样，同时表达式(expression)的值就是匿名函数调用的返回值；而lambda表达式返回这个匿名函数。如果我们给匿名函数取个名字，就像这样：

Python

```
lambda_add = lambda x, y:
x + y
1 lambda_add = lambda x, y: x + y
```

这与使用def定义的求和函数完全一样，可以使用lambda_add作为函数名进行调用。然而，提供lambda的目的是为了编写偶尔为之的、简单的、可预见不会被修改的匿名函数。这种风格虽然看起来很酷，但并不是一个好主意，特别是当某一天需要对它进行扩充，再也无法用一个表达式写完时。如果一开始就需要给函数命名，应该始终使用def关键字。

2.2. 使用函数赋值

事实上你已经见过了，上一节中我们将lambda表达式赋值给了add。同样，使用def定义的函数也可以赋值，相当于为函数取了一个别名，并且可以使用这个别名调用函数：

Python

```
add_a_number_to_another_one_by_using_plus
```

```
1 add_a_number_to_another_one_by_using_plus_operator = add
2 print add_a_number_to_another_one_by_using_plus_operator(1, 2)
```

既然函数可以被变量引用，那么将函数作为参数和返回值就是很寻常的做法了。

2.3. 闭包

闭包是一类特殊的函数。如果一个函数定义在另一个函数的作用域中，并且函数中引用了外部函数的局部变量，那么这个函数就是一个闭包。下面的代码定义了一个闭包：

Python

```
def f():
    n = 1
```

```
1 def f():
2     n = 1
3     def inner():
4         print n
5     inner()
6     n = 'x'
7     inner()
```

函数inner定义在f的作用域中，并且在inner中使用了f中的局部变量n，这就构成了一个闭包。闭包绑定了外部的变量，所以调用函数f的结果是打印1和'x'。这类似于普通的模块函数和模块中定义的全局变量的关系：修改外部变量能影响内部作用域中的值，而在内部作用域中定义同名变量则将遮蔽（隐藏）外部变量。

如果需要在函数中修改全局变量，可以使用关键字global修饰变量名。Python 2.x中没有关键字为在闭包中修改外部变量提供支持，在3.x中，关键字nonlocal可以做到这一点：

Python

```
#Python 3.x supports
`nonlocal`
```

```
1 #Python 3.x supports `nonlocal`
2 def f():
3     n = 1
4     def inner():
5         nonlocal n
6         n = 'x'
7     print(n)
8     inner()
9     print(n)
```

调用这个函数的结果是打印1和'x'，如果你有一个Python 3.x的解释器，可以试着运行一下。

由于使用了函数体外定义的变量，看起来闭包似乎违反了函数式风格的规则即不依赖外部状态。但是由于闭包绑定的是外部函数的局部变量，而一旦离开外部函数作用域，这些局部变量将无法再从外部访问；另外闭包还有一个重要的特性，每次执行至闭包定义处时都会构造一个新的闭包，这个特性使得旧的闭包绑定的变量不会随第二次调用外部函数而更改。所以闭包实际上不会被外部状态影响，完全符合函数式风格的要求。（这里有一个特例，Python 3.x中，如果同一个作用域中定义了两个闭包，由于可以修改外部变量，他们可以相互影响。）

虽然闭包只有在作为参数和返回值时才能发挥它的真正威力，但闭包的支持仍然大大提升了生产率。

2.4. 作为参数

如果你对OOP的模板方法模式很熟悉，相信你能很快速地学会将函数当作参数传递。两者大体是一致的，只是在这里，我们传递的是函数本身而不再是实现了某个接口的对象。

我们先来给前面定义的求和函数add热热身：

Python

```
print add('三角形的树', '北  
极')  
1 print add('三角形的树', '北极')
```

与加法运算符不同，你一定很惊讶于答案是'三角函数'。这是一个内置的彩蛋...bazinga!

言归正传。我们的客户有一个从0到4的列表：

Python

```
lst = range(5) #[0, 1, 2, 3,  
4]  
1 lst = range(5) #[0, 1, 2, 3, 4]
```

虽然我们在上一小节里给了他一个加法器，但现在他仍然在为如何计算这个列表所有元素的和而苦恼。当然，对我们而言这个任务轻松极了：

Python

```
amount = 0  
for num in lst:  
1 amount = 0  
2 for num in lst:  
3     amount = add(amount, num)
```

这是一段典型的指令式风格的代码，一点问题都没有，肯定可以得到正确的结果。现在，让我们试着用函数式的风格重构一下。

首先可以预见的是求和这个动作是非常常见的，如果我们把这个动作抽象成一个单独的函数，以后需要对另一个列表求和时，就不必再写一遍这个套路了：

Python

```
def sum_(lst):  
    amount = 0  
1 def sum_(lst):  
2     amount = 0  
3     for num in lst:  
4         amount = add(amount, num)  
5     return amount  
6  
7 print sum_(lst)
```

还能继续。sum_函数定义了这样一种流程：

1. 使用初始值与列表的第一个元素相加；
2. 使用上一次相加的结果与列表的下一个元素相加；
3. 重复第二步，直到列表中没有更多元素；
4. 将最后一次相加的结果返回。

如果现在需要求乘积，我们可以写出类似的流程——只需要把相加换成相乘就可以了：

Python

```
def multiply(lst):  
    product = 1  
  
1 def multiply(lst):  
2     product = 1  
3     for num in lst:  
4         product = product * num  
5     return product
```

除了初始值换成了1以及函数add换成了乘法运算符，其他的代码全部都是冗余的。我们为什么不把这个流程抽象出来，而将加法、乘法或者其他函数作为参数传入呢？

Python

```
def reduce_(function,  
lst, initial):  
  
1 def reduce_(function, lst, initial):  
2     result = initial  
3     for num in lst:  
4         result = function(result, num)  
5     return result  
6  
7 print reduce_(add, lst, 0)
```

现在，想要算出乘积，可以这样做：

Python

```
print reduce_(lambda x, y:  
x * y, lst, 1)  
  
1 print reduce_(lambda x, y: x * y, lst, 1)
```

那么，如果想要利用reduce_找出列表中的最大值，应该怎么做呢？请自行思考：）

虽然有模板方法这样的设计模式，但那样的复杂度往往使人们更情愿到处编写循环。将函数作为参数完全避开了模板方法的复杂度。

Python有一个内建函数reduce，完整实现并扩展了reduce_的功能。本文稍后的部分包含了有用的内建函数的介绍。请注意我们的目的是没有循环，使用函数替代循环是函数式风格区别于指令式风格的最显而易见的特征。

*像Python这样构建于类C语言之上的函数式语言，由于语言本身提供了编写循环代码的能力，内置函数虽然提供函数式编程的接口，但一般在内部还是使用循环实现的。同样的，如果发现内建函数无

法满足你的循环需求，不妨也封装它，并提供一个接口。

2.5. 作为返回值

将函数返回通常需要与闭包一起使用（即返回一个闭包）才能发挥威力。我们先看一个函数的定义：

Python

```
def map_(function, lst):  
    result = []  
  
1 def map_(function, lst):  
2     result = []  
3     for item in lst:  
4         result.append(function(item))  
5     return result
```

函数map_封装了最常见的一种迭代：对列表中的每个元素调用一个函数。map_需要一个函数参数，并将每次调用的结果保存在一个列表中返回。这是指令式的做法，当你知道了列表解析(list comprehension)后，会有更好的实现。

这里我们先略过map_的蹩脚实现而只关注它的功能。对于上一节中的lst，你可能发现最后求乘积结果始终是0，因为lst中包含了0。为了让结果看起来足够大，我们来使用map_为lst中的每个元素加1：

Python

```
lst = map_(lambda x:  
add(1, x), lst)  
  
1 lst = map_(lambda x: add(1, x), lst)  
2 print reduce_(lambda x, y: x * y, lst, 1)
```

答案是120，这还远远不够大。再来：

Python

```
lst = map_(lambda x:  
add(10, x), lst)  
  
1 lst = map_(lambda x: add(10, x), lst)  
2 print reduce_(lambda x, y: x * y, lst, 1)
```

囧，事实上我真的没有想到答案会是360360，我发誓没有收周鸿祎任何好处。

现在回头看看我们写的两个lambda表达式：相似度超过90%，绝对可以使用抄袭来形容。而问题不在于抄袭，在于多写了很多字符有木有？如果有一个函数，根据你指定的左操作数，能生成一个加法函数，用起来就像这样：

Python

```
lst = map_(add_to(10),  
lst) #add_to(10)返回一  
  
1 lst = map_(add_to(10), lst) #add_to(10)返回一个函数，这个函数接受一个参数并加上10后返回
```

写起来应该会舒服不少。下面是函数add_to的实现：

Python

```
def add_to(n):  
    return lambda x:
```

```
1 def add_to(n):  
2     return lambda x: add(n, x)
```

通过为已经存在的某个函数指定数个参数，生成一个新的函数，这个函数只需要传入剩余未指定的参数就能实现原函数的全部功能，这被称为偏函数。Python内置的functools模块提供了一个函数partial，可以为任意函数生成偏函数：

Python

```
functools.partial(func[,  
*args][, **keywords])
```

```
1 functools.partial(func[, *args][, **keywords])
```

你需要指定要生成偏函数的函数、并且指定数个参数或者命名参数，然后partial将返回这个偏函数；不过严格的说partial返回的不是函数，而是一个像函数一样可直接调用的对象，当然，这不会影响它的功能。

另外一个特殊的例子是装饰器。装饰器用于增强甚至干脆改变原函数的功能，我曾写过一篇文档介绍装饰器，地址在这里：<http://www.cnblogs.com/huxi/archive/2011/03/01/1967600.html>。

*题外话，单就例子中的这个功能而言，在一些其他的函数式语言中（例如Scala）可以使用名为柯里化(Currying)的技术实现得更优雅。柯里化是把接受多个参数的函数变换为接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。如下的伪代码所示：

Python

```
#不是真实的代码  
def add(x)(y): #柯里化
```

```
1 #不是真实的代码  
2 def add(x)(y): #柯里化  
3     return x + y  
4  
5 lst = map_(add(10), lst)
```

通过将add函数柯里化，使得add接受第一个参数x，并返回一个接受第二个参数y的函数，调用该函数与前文中的add_to完全相同（返回 $x + y$ ），且不再需要定义add_to。看上去是不是更加清爽呢？遗憾的是Python并不支持柯里化。

2.6. 部分内建函数介绍

- reduce(function, iterable[, initializer])

这个函数的主要功能与我们定义的reduce_相同。需要补充两点：

它的第二个参数可以是任何可迭代的对象（实现了__iter__()方法的对象）；

如果不指定第三个参数，则第一次调用function将使用iterable的前两个元素作为参数。
由reduce和一些常见的function组合成了下面列出来的内置函数：

Python

```
all(iterable) ==  
reduce(lambda x, y:  
    
```

```
1 all(iterable) == reduce(lambda x, y: bool(x and y), iterable)  
2 any(iterable) == reduce(lambda x, y: bool(x or y), iterable)  
3 max(iterable[, args...][, key]) == reduce(lambda x, y: x if key(x) > key(y) else y, iterable_and_args)  
4 min(iterable[, args...][, key]) == reduce(lambda x, y: x if key(x) < key(y) else y, iterable_and_args)  
5 sum(iterable[, start]) == reduce(lambda x, y: x + y, iterable, start)
```

- map(function, iterable, ...)

这个函数的主要功能与我们定义的map_相同。需要补充一点：

map还可以接受多个iterable作为参数，在第n次调用function时，将使用iterable1[n], iterable2[n], ...作为参数。

- filter(function, iterable)

这个函数的功能是过滤出iterable中所有以元素自身作为参数调用function时返回True或bool(返回值)为True的元素并以列表返回，与系列第一篇中的my_filter函数相同。

- zip(iterable1, iterable2, ...)

这个函数返回一个列表，每个元素都是一个元组，包含(iterable1[n], iterable2[n], ...)。

例如：zip([1, 2], [3, 4]) -> [(1, 3), (2, 4)]

如果参数的长度不一致，将在最短的序列结束时结束；如果不提供参数，将返回空列表。

除此之外，你还可以使用本文2.5节中提到的functools.partial()为这些内置函数创建常用的偏函数。

另外，pypi上有一个名为functional的模块，除了这些内建函数外，还额外提供了更多的有意思的函数。但由于使用的场合并不多，并且需要额外安装，在本文中就不介绍了。但我仍然推荐大家下载这个模块的纯Python实现的源代码看看，开阔思维嘛。里面的函数都非常短，源文件总共只有300行不到，地址在这里：<http://pypi.python.org/pypi/functional>

此篇结束：）

1 赞 2 收藏 [1 评论](#)

Python函数式编程指南（1）：概述

原文出处：[AstralWind](#)

1. 函数式编程概述

1.1. 什么是函数式编程？

函数式编程使用一系列的函数解决问题。函数仅接受输入并产生输出，不包含任何能影响产生输出的内部状态。任何情况下，使用相同的参数调用函数始终能产生同样的结果。

在一个函数式的程序中，输入的数据“流过”一系列的函数，每一个函数根据它的输入产生输出。函数式风格避免编写有“边界效应”(side effects)的函数：修改内部状态，或者是其他无法反应在输出上的变化。完全没有边界效应的函数被称为“纯函数式的”(purely functional)。避免边界效应意味着不使用在程序运行时可变的数据结构，输出只依赖于输入。

可以认为函数式编程刚好站在了面向对象编程的对立面。对象通常包含内部状态（字段），和许多能修改这些状态的函数，程序则由不断修改状态构成；函数式编程则极力避免状态改动，并通过在函数间传递数据流进行工作。但这并不是说无法同时使用函数式编程和面向对象编程，事实上，复杂的系统一般会采用面向对象技术建模，但混合使用函数式风格还能让你额外享受函数式风格的优点。

1.2. 为什么使用函数式编程？

函数式的风格通常被认为有如下优点：

- 逻辑可证

这是一个学术上的优点：没有边界效应使得更容易从逻辑上证明程序是正确的（而不是通过测试）。

- 模块化

函数式编程推崇简单原则，一个函数只做一件事情，将大的功能拆分成尽可能小的模块。小的函数更易于阅读和检查错误。

- 组件化

小的函数更容易加以组合形成新的功能。

- 易于调试

细化的、定义清晰的函数使得调试更加简单。当程序不正常运行时，每一个函数都是检查数据是否正确的接口，能更快速地排除没有问题的代码，定位到出现问题的地方。

- 易于测试

不依赖于系统状态的函数无须在测试前构造测试桩，使得编写单元测试更加容易。

- 更高的生产率

函数式编程产生的代码比其他技术更少（往往是其他技术的一半左右），并且更容易阅读和维护。

1.3. 如何辨认函数式风格？

支持函数式编程的语言通常具有如下特征，大量使用这些特征的代码即可被认为是函数式的：

- 函数是一等公民

函数能作为参数传递，或者是作为返回值返回。这个特性使得模板方法模式非常易于编写，这也促使了这个模式被更频繁地使用。

以一个简单的集合排序为例，假设lst是一个数集，并拥有一个排序方法sort需要将如何确定顺序作为参数。

如果函数不能作为参数，那么lst的sort方法只能接受普通对象作为参数。这样一来我们需要首先定义一个接口，然后定义一个实现该接口的类，最后将该类的一个实例传给sort方法，由sort调用这个实例的compare方法，就像这样：

Python

```
#伪代码
interface Comparator {  
    compare(o1, o2)  
}  
  
1 #伪代码
2 interface Comparator {
3     compare(o1, o2)
4 }
5 lst = list(range(5))
6 lst.sort(Comparator())
7     compare(o1, o2) {
8         return o2 - o1 //逆序
9 })
```

可见，我们定义了一个新的接口、新的类型（这里是一个匿名类），并new了一个新的对象只为了调用一个方法。如果这个方法可以直接作为参数传递会怎样呢？看起来应该像这样：

Python

```
def compare(o1, o2):
    return o2 - o1 #逆序  
  
1 def compare(o1, o2):
2     return o2 - o1 #逆序
3 lst = list(range(5))
4 lst.sort(compare)
```

请注意，前一段代码已经使用了匿名类技巧从而省下了不少代码，但仍然不如直接传递函数简单、自然。

- 匿名函数(lambda)

lambda提供了快速编写简单函数的能力。对于偶尔为之的行为，lambda让你不再需要在编码时跳转到其他位置去编写函数。

lambda表达式定义一个匿名的函数，如果这个函数仅在编码的位置使用到，你可以现场定义、直接

使用：

Python

```
lst.sort(lambda o1, o2:  
o1.compareTo(o2))  
1 lst.sort(lambda o1, o2: o1.compareTo(o2))
```

相信从这个小小的例子你也能感受到强大的生产效率：）

- 封装控制结构的内置模板函数

为了避开边界效应，函数式风格尽量避免使用变量，而仅仅为了控制流程而定义的循环变量和流程中产生的临时变量无疑是最需要避免的。

假如我们需要对刚才的数集进行过滤得到所有的正数，使用指令式风格的代码应该像是这样：

Python

```
lst2 = list()  
for i in range(len(lst)): #  
1 lst2 = list()  
2 for i in range(len(lst)): #模拟经典for循环  
3   if lst[i] > 0:  
4     lst2.append(lst[i])
```

这段代码把从创建新列表、循环、取出元素、判断、添加至新列表的整个流程完整的展示了出来，俨然把解释器当成了需要手把手指导的傻瓜。然而，“过滤”这个动作是很常见的，为什么解释器不能掌握过滤的流程，而我们只需要告诉它过滤规则呢？

在Python里，过滤由一个名为filter的内置函数实现。有了这个函数，解释器就学会了如何“过滤”，而我们只需要把规则告诉它：

Python

```
lst2 = filter(lambda n: n > 0,  
lst)  
1 lst2 = filter(lambda n: n > 0, lst)
```

这个函数带来的好处不仅仅是少写了几行代码这么简单。

封装控制结构后，代码中就只需要描述功能而不是做法，这样的代码更清晰，更可读。因为避开了控制结构的干扰，第二段代码显然能让你更容易了解它的意图。

另外，因为避开了索引，使得代码中不太可能触发下标越界这种异常，除非你手动制造一个。函数式编程语言通常封装了数个类似“过滤”这样的常见动作作为模板函数。唯一的缺点是这些函数需要少量的学习成本，但这绝对不能掩盖使用它们带来的好处。

- 闭包(closure)

闭包是绑定了外部作用域的变量（但不是全局变量）的函数。大部分情况下外部作用域指的是外部函

数。

闭包包含了自身函数体和所需外部函数中的“变量名的引用”。引用变量名意味着绑定的是变量名，而不是变量实际指向的对象；如果给变量重新赋值，闭包中能访问到的将是新的值。

闭包使函数更加灵活和强大。即使程序运行至离开外部函数，如果闭包仍然可见，则被绑定的变量仍然有效；每次运行至外部函数，都会重新创建闭包，绑定的变量是不同的，不需要担心在旧的闭包中绑定的变量会被新的值覆盖。

回到刚才过滤数集的例子。假设过滤条件中的 0 这个边界值不再是固定的，而是由用户控制。如果没有闭包，那么代码必须修改为：

Python

```
class greater_than_helper:
    def __init__(self, minval):
        self.minval = minval
    def is_greater_than(self, val):
        return val > self.minval
def my_filter(lst, minval):
    helper = greater_than_helper(minval)
    return filter(helper.is_greater_than, lst)
```

请注意我们现在已经为过滤功能编写了一个函数my_filter。如你所见，我们需要在别的地方（此例中是类greater_than_helper）持有另一个操作数minval。

如果支持闭包，因为闭包可以直接使用外部作用域的变量，我们就不需要greater_than_helper了：

Python

```
def my_filter(lst,
              minval):
    def my_filter(lst, minval):
        return filter(lambda n: n > minval, lst)
```

可见，闭包在不影响可读性的同时也省下了不少代码量。

函数式编程语言都提供了对闭包的不同程度的支持。在Python 2.x中，闭包无法修改绑定变量的值，所有修改绑定变量的行为都被看成新建了一个同名的局部变量并将绑定变量隐藏。Python 3.x中新加入了一个关键字nonlocal以支持修改绑定变量。但不管支持程度如何，你始终可以访问（读取）绑定变量。

- 内置的不可变数据结构

为了避开边界效应，不可变的数据结构是函数式编程中不可或缺的部分。不可变的数据结构保证数据的一致性，极大地降低了排查问题的难度。

例如，Python中的元组(tuple)就是不可变的，所有对元组的操作都不能改变元组的内容，所有试图修改元组内容的操作都会产生一个异常。

函数式编程语言一般会提供数据结构的两种版本（可变和不可变），并推荐使用不可变的版本。

- 递归

递归是另一种取代循环的方法。递归其实是函数式编程很常见的形式，经常可以在一些算法中见到。但之所以放到最后，是因为实际上我们一般很少用到递归。如果一个递归无法被编译器或解释器优化，很容易就会产生栈溢出；另一方面复杂的递归往

往让人感觉迷惑，不如循环清晰，所以众多最佳实践均指出使用循环而非递归。
这一系列短文中都不会关注递归的使用。

1 赞 3 收藏 [评论](#)

Python Guide 系列 2.1：结构化你的项目

本文由 [伯乐在线 - mtunique](#) 翻译, [艾凌风](#) 校稿。未经许可, 禁止转载!

英文出处: docs.python-guide.org。欢迎加入[翻译组](#)。

所谓“结构”就是使项目清楚地直接地干净地优雅地达到他的目的。我们要考虑如何最大限度的利用python的特性来写干净、有效的代码。实际上, “structure”意味着写干净的代码（逻辑关系，依赖关系明确），以及如何在文件系统中组织文件和文件夹。

哪个函数应该在哪个模块里？项目的数据流是怎样的？什么特性和功能应该放在一起，什么样的应该隔离？广义上说，通过回答这些问题你可以开始计划成品是什么样子的。

在这一节，我们会仔细研究python的module（模块）和import（导入）系统，因为它们是增强项目结构化的核心内容。然后我们讨论如何构建可扩展可测试的可靠代码。

结构是关键

由于import和module是python掌控的，结构化一个Python项目相对比较容易。这里所指的容易，意味着没有很多约束而且python的模块导入模型比较容易掌握。因此，留给你的就剩下一些架构性的工作，编写项目的不同模块并负责它们之间的交互。 Easy structuring of a project means it is also easy to do it poorly. Some signs of a poorly structured project include: 容易结构化的项目同样意味着它的结构化容易做的不好。。结构性差的项目，其特征包括：

- 多重且混乱的循环依赖关系：如果furn.py中你的类Table和Chair需要import worker.py中的Carpenter来回答例如table.isdomeby()这种问题，相反Carpenter类需要import Table和Chair类来回答carpenter.whatdo()这种问题，所以你有一个循环依赖项。在这种情况下你将不得不借助于一些不太可靠的技巧，比如在方法或函数内部使用import语句。
- 隐藏耦合：每一次改变Table的实现都会打破20个互不相关的测试用例，因为它打破了Carpenter的代码，所以需要小心操作来适应改变。这意味着在Carpenter的代码中你有太多关于Table的假设，反过来也是如此。
- 大量使用全局状态或上下文：彼此间不使用显示传递（高度、宽度、类型、材料）参数，Table和Carpenter都依赖全局变量，可以被修改而且被不同的引用修改。你需要仔细检查所有访问这些全局变量的地方，来理解为什么一个矩形变成一个正方形，并发现远程模板代码也修改上下文，弄乱了桌子的尺寸参数。
- 面条式代码：多层嵌套的if语句for循环，大量复制粘贴代码，没有适当分割的代码被称为面条式代码。python的具有意义的缩进（其最具争议的特性之一）使它难以维持这种代码。所以好消息是你也许看不到太多这种面条式代码。
- python中更可能出现混沌代码：如果没有适当的结构，它会包括几百个相似的小逻辑块，类或对象。如果你记不住在手头的任务中你是否必须用FurnitureTable, AssetTable或Table甚至TableNew，你就可能陷入混沌的代码中。

模块

Python 的模块是最主要的抽象层之一，也算最自然的一个。抽象层允许我们把代码分成不同的部

分，每部分包含着相关的数据和功能。

例如：一层控制用户的行为的接口，而另一层处理低级别的数据操作。分离这两层的最自然的方式是将所有接口功能重新组合到一个文件里，所有低级别操作在另一个文件。在这种情况下，接口文件需要import低级别操作的文件。通过from ... import语句来完成。

只要你一用import语句，你就可以用这个module了。可以是内置的模块比如os和sys，可以是已经安装到环境中的第三方模块，也可以是你项目的内部模块。

为了和编码风格保存一致，模块名要短，使用小写字母，一定要避免使用特殊符号，如点(.)，问号(?)。所以要避免的像my.spam.pu这样的文件名！这种命名方式会干扰python查找模块。

在my.spam.py这个例子中，python想要在my文件夹中查找spam.py文件，而不是我们想要的。在python文档中还有一个关于应该如何使用点的[例子](#)。

你可以将模块命名为my_spam.py。尽管可以使用，但还是不应该经常在模块名中看到下划线。

除了一些命名的限制，在将一个python文件当做一个模块方面没有什么特别的要求。但是你需要理解import的机制来正确使用这一概念和避免一些问题。

具体而言，如果modu.py文件和调用方在同一目录中，import modu语句将能寻找到适当的文件。如果找不到他，python解释器将在“path”中递归查找，如果没有找到将raise ImportError异常。

一旦发现了modu.py，python解释器会在一个隔离的作用域内执行这个模块。任何modu.py中的顶级语句将被执行，包括其他import。函数和类的定义将被存储在module的字典中。

然后，该模块的变量、函数和类将通过模块的命名空间提供给调用方。在python中这是特别有用和强大的核心概念。

在很多语言中，包含文件的指令的作用是：由预处理器找到文件中的所有代码并复制到调用方的代码中。在Python是不同的：包含的代码被隔离在一个模块命名空间中，这就意味着你一般不需要担心包含的代码产生不良影响，例如覆盖具有相同名称的现有函数。

也可以通过用特殊语法的import语句来模拟更标准的行为：from modu import *。普遍认为这是不好的做法。**使用`*`使得代码难以阅读并且使得依赖关系没有进行足够的划分**.**

用from modu import func这种方式可以准确定位你想要imprint的函数并把它引入全局命名空间。比import *的危害小的多，因为它明确地显示什么被引入全局命名空间中，相比import modu的唯一优势是他可以少打点字。

非常糟糕

Python

```
[...]
from modu import *
1 [...]
2   from modu import *
3   [...]
4   x = sqrt(4) # sqrt是modu的？内置的？上面定义的？
```

好一点

Python

```
from modu import sqrt  
[...]  
1 from modu import sqrt  
2 [...]  
3 x = sqrt(4) # sqrt如果不在中间定义那可能是modu的一部分
```

最好

Python

```
import modu  
[...]  
1 import modu  
2 [...]  
3 x = modu.sqrt(4) # sqrt明显是modu命名空间的一部分
```

[Code Style](#)一节中提到可读性是python的主要特点之一。可读性意味着避免无用的文字和散乱的结构，因此要花费一些精力在达到一定程度的简洁。但是不能太简介，否则就晦涩难懂了。要能够立刻告诉一个类或函数来自哪里，比如modu.func这种。这能大大提高代码的可读性和可理解性，除了最简单的单文件项目。

包

python提供了一个非常简单的包系统，可以简单的将一个目录扩展为一个包。任何有 `__init__.py` 文件的目录都可以被认为是一个python包。包中不同的模块可以像普通的模块一样被引入。`__init__.py`文件有一个特殊的作用，收集所有包范围的定义。

`import pack.modu`语句可以引入pack/目录里的modu.py文件。此语句将在pack中查找`__init__.py`文件，执行所有其顶层的语句。然后他将查找名为pack/modu.py的文件并执行文件中的所有顶级语句。这些操作中，所有modu.py中的变量、函数和类的定义可以通过pack.modu命名空间获得。

一个常见的问题是将太多代码写在了`__init__.py`文件中。当项目的复杂性增长时，可能在深层的目录结构中可能会有子包甚至子子包。在这种情况下，从子子包中import一个简单的项目同样需要执行所有在遍历树中遇到的`__init__.py`文件。

`__init__.py`文件是空的这很正常。如果包的模块和子模块不需要共享任何代码这甚是是一个好的做法。最后，介绍一种方便的语法，可以用来引入深层嵌套的包：`import very.deep.module as mod`。这样可以用`mod`来代替冗长的`very.deep.module`。

面向对象编程

Python有时被描述为一种面向对象的编程语言。这可能会让人误解需要加以澄清。

在python中，一切都是对象。这是什么意思，例如：函数是一级对象。函数、类、字符串等在python中都是对象：像任何对象一样，他们有类型，他们可以被作为函数参数传递，他们可能有方法和属

性。这样理解的话，python是一种面向对象的语言。

但是不像java。python没有将面向对象编程作为主要的编程范式。对于python项目不是面向对象的（也就是没有使用或很少使用类的定义、类的继承或任何其他特定于面向对象编程的机制）是完全可行的。

此外，在模块部分，python处理模块和命名空间的方式给开发者很自然的方式去确保抽象层的封装和分离，这成为了使用面向对象的最常见的原因。因此，当没有被要求时，python程序员有更多空间来不使用面向对象。

实际上确实存在一些场合，应当避免在不必要的时候使用“面向对象”。若要将“状态”和“功能”结合起来，通过自定义类的方式自然是受用。不过问题在于，正如我们在讨论函数式编程时指出的那样，函数式的“状态”和类的“状态”根本就不一回事。

通常在一些架构中，典型的例子是web应用程序，会生成 Python程序的多个实例，使得可以在同一时间对外部请求进行响应。在这种情况下，实例化的对象持有着某种状态，也就是说持有一些环境的静态信息，这很容易出现并发问题或争态条件。有时，在初始化的对象（通常是用`__init__()`方法）与实际使用对象之间，环境可能发生了改变，而且保留的状态可能已经过时。例如，请求可能加载一个item到内存中的，并将其标记为已读。在同一时间如果另一个请求需要删这个item，可能会发生这样的事情：第一个进程加载了item后被删除了，然后我们把已经删除的对象标记为了已读。

这个问题或者其他问题让我们产生这样一个想法，使用无状态函数或许是一个更好的编程范式。另一种方式是建议使用隐式上下文和副作用尽可能少的函数和过程。函数的隐式上下文是由全局变量和在函数内部访问可以访问的持久层中的项组成。副作用是函数会使其隐式上下文发生改变，如果一个函数保存或删除了全局变量或持久层中的数据，我们把这种行为称之为副作用。

把带有上下文和副作用的函数从仅仅包含逻辑的函数（纯函数）中小心的剥离出来，会带来如下的益处：

- 纯函数都具有确定性：给出一个固定的输入，输出总是会相同。
- 纯函数更容易更改或替换，如果它们需要重构或优化的话。
- 纯函数的测试与单元测试更容易编写：很少需要复杂的上下文设置和之后的数据清洗。
- 纯函数更容易操纵，修饰，分发。

总之，一些架构中纯函数比类和对象能更有效地进行模块化构建。因为他们没有任何上下文或副作用。很明显，在许多情况下面向对象是有用的，甚至是必要的，例如当开发图形化桌面应用程序或游戏，有需要的操纵的东西（窗口、按钮、人物、车辆）需要在计算机的内存中具有相对较长的生命周期。

修饰器

python语言提供简单但功能强大的语法：“修饰器”。装饰器是一个函数或者类，它可以包装（或修饰）一个函数或方法。装饰器函数或方法将取代原来的“未装饰”的函数或方法。因为在python中函数是一级对象，它可以被“手动操作”，但是用@decorator语法更清晰，因此要首选这种方式。

```
def foo():
    # do something

1 def foo():
2     # do something
3
4 def decorator(func):
5     # 操作 func
6     return func
7
8 foo = decorator(foo) # 手动修饰
9
10 @decorator
11 def bar():
12     # Do something
13 # bar() 已经被修饰
```

这种机制是对分离关注点和避免外部非相关的逻辑‘污染’函数或方法的核心逻辑来说是有用的。有些功能如果用装饰器来实现会更好，缓存就是一个很好的例子：你想要将耗时的函数结果存储在一个表中并直接使用他们而不是重复计算他们。这显然不是函数逻辑的一部分。

动态类型

python是动态类型的，这意味着变量并没有固定的类型。事实上，在python中，变量和很多其他语言非常不同，特别是静态类型语言。变量不是电脑内存中的一段，他们是指向对象的'tags'或'names'。因此可能变量“a”被设为1，然后变成了“a string”，然后又变成了一个函数。

这样不好

Python

```
a = 1
a = 'a string'

1 a = 1
2 a = 'a string'
3 def a():
4     pass # Do something
```

这样好

Python

```
count = 1
msg = 'a string'

1 count = 1
2 msg = 'a string'
3 def func():
4     pass # Do something
```

python的动态类型通常被认为是不可靠的，确实会带来复杂的，难以调试的代码。命名为“a”的可能是很多不同的东西，开发者或维护者需要在代码中跟踪它确保它没有被设为完全无关的对象。一些方法有助于避免这种问题：避免为不同的事物使用相同的变量名

使用剪短的函数或方法有助于降低使用同名代表两个不同事物所带来的风险。

甚至对于相关的事物，也最好使用不同的名称，如果它们类型不同的话

这样不好

Python

```
items = 'a b c d' # 这是  
一个字符串...
```

```
1 items = 'a b c d' # 这是一个字符串...  
2 items = items.split(' ') # ...变成了列表  
3 items = set(items) # ...又变成了集合
```

重用名称并不会提高效率：赋值的时候无论怎样都会去创建新的对象。然而，随着复杂性的上升，赋值语句被其他代码分开，包括“if”分支和循环，将越来越难以确定变量的类型是什么。一些编码实践，比如函数式编程，建议永远不会重新分配一个变量。在java中，可以使用final关键字，python没有final关键字而且无论如何这都是违反python的哲学的。不过，避免多次为同一个变量赋值是一个好习惯，而且可以有助于掌握可变类型和不可变类型的概念

可变类型和不可变类型

python提供两种内置或用户定义的类型。可变类型是内容允许修改的。典型的可变类型是list和dict：所有的list都有可变方法，比如list.append()或list.pop()，并且可以就地修改。字典也是一样的。不可变类型没有提供改变其内容的方法。例如：设置为6的整数变量x没有“increment”方法。如果你想要计算x+1，你必须创建另一个整数并给他一个名称。

Python

```
my_list = [1, 2, 3]  
my_list[0] = 4
```

```
1 my_list = [1, 2, 3]  
2 my_list[0] = 4  
3 print my_list # [4, 2, 3] &lt;- The same list as changed  
4  
5 x = 6  
6 x = x + 1 # The new x is another object
```

这种差异的一个后果是可变类型不是“稳定的”，并因此不能用作字典的键。可变性质的东西用可变类型，固定不变的用不可变类型这有助于阐明代码的目的。

例如，类似列表的不可变类型是元组，通过类似(1, 2)这种方式创建。此元组是一对，不能就地更改，并且可以用作键的字典。python中一件令初学者吃惊的事情是，字符串类型是不可变的。这意味着，当需要组合一个字符串时，把每一部分都放到列表中（是可变的）会比较好，然后当需要整个字符串的时候再把他们连('join')起来。然而，有一件事要注意，列表推导比在循环调用append()来构造列表要更好和更快。

不好

Python

```
# create a concatenated  
string from 0 to 19 (e.g.
```

```
1 # create a concatenated string from 0 to 19 (e.g. "012..1819")
2 nums = ""
3 for n in range(20):
4     nums += str(n) # slow and inefficient
5 print nums
```

好

Python

```
# create a concatenated  
string from 0 to 19 (e.g.
```

```
1 # create a concatenated string from 0 to 19 (e.g. "012..1819")
2 nums = []
3 for n in range(20):
4     nums.append(str(n))
5 print "".join(nums) # much more efficient
```

最佳

Python

```
# create a concatenated  
string from 0 to 19 (e.g.
```

```
1 # create a concatenated string from 0 to 19 (e.g. "012..1819")
2 nums = [str(n) for n in range(20)]
3 print "".join(nums)
```

关于字符串最后要提的是，使用 `join()` 不是总是最好。比如，当你要用预先确定数目的字符串创建一个新的字符串时，使用加法运算符确实是更快，但在上述情况下或添加到现有的字符串的情况下用你应该首选 `join()`。

Python

```
foo = 'foo'  
bar = 'bar'
```

```
1 foo = 'foo'
2 bar = 'bar'
3
4 foobar = foo + bar # This is good
5 foo += 'ooo' # This is bad, instead you should do:
6 foo = ".join([foo, 'ooo'])"
```

注意除了[str.join\(\)](#) 和 `+`，你也可以使用[%](#)格式运算符来串联预先确定数目的字符串。然而[PEP 3101](#)，建议用 `str.format()` 方法取代`%`运算符。

Python

```
foo = 'foo'  
bar = 'bar'
```

```
1 foo = 'foo'
2 bar = 'bar'
```

```
3
4 foobar = '%s%s' % (foo, bar) # It is OK
5 foobar = '{0}{1}'.format(foo, bar) # It is better
6 foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # It is best
```

2 赞 4 收藏 [评论](#)

关于作者： [mtunique](#)



微博：@孟涛_hust GitHub:mtunique 个人网站：mtunique.com [个人主页](#) · [我的文章](#) · 12

Python HOWTOs 官方文档：Socket 编程

本文由 [伯乐在线 - 高世界](#) 翻译, [艾凌风](#) 校稿。未经许可, 禁止转载!

英文出处: docs.python.org。欢迎加入[翻译组](#)。

摘要

几乎所有地方都用到了sockets，但它们可能是被严重误解的技术之一。本文是关于Sockets的概述。它并不是一篇教程——要让sockets运行起来，你仍旧需要做点工作。本文并没有涵盖全部的要点（而这样的要点有很多），但我希望它可以给你足够的背景知识，让你能像样地使用sockets。

Sockets

我只打算谈谈INET（比如IPv4）sockets，但是99%使用中的sockets都是它。而且我只谈流（比如TCP）——除非你真的知道你在干什么（这样的话本HOWTO不适合你啦），使用流socket要比别的更稳定，性能更好。我将揭开socket是什么的神秘面纱、还有关于如何使用阻塞和非阻塞sockets的提示。但是，我会先从阻塞sockets开始谈起，在处理非阻塞sockets之前，你得知道它们（阻塞sockets）是如何工作的。

理解这些事情麻烦之一是，根据不同上下文，socket可以代表很多略有不同的东西。所以首先，咱们先区分一下“客户端”socket——会话的一个终端，和“服务器”socket，（服务器socket）更像一个接线员。客户端应用程序（比如说浏览器）只使用“客户端”sockets；web服务器在通信时使用“服务器”sockets和客户端sockets这两者。

历史

在各种各样的IPC里，sockets是目前最流行的。对于任意指定平台，可能有其他的IPC更快，但对跨平台通信而言，sockets是唯一的一个。

作为BSD风格的Unix的一部分，它们被创造于伯克利。它们像烈火般蔓延般在互联网上传播。因为——和INET的结合使得世界上任意机器通信变得难以置信地简单（至少跟其他方案比）。

创建一个socket

大体来讲，当你点击了把你带到这个页面的链接时，你的浏览器做了类似以下的事：

Python

```
# create an INET,
STREAMing socket
# create an INET, STREAMing socket
2 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 # now connect to the web server on port 80 - the normal http port
4 s.connect(("www.python.org", 80))
```

连接完成后，请求页面的文本时就可以使用sockets发送请求了。接着它读取响应，然后销毁。对，就是销毁。客户端sockets一般只用来做一次数据交换（或一个少量的有序的数据交换）。

web服务器那边更复杂些。首先， web服务器创建一个“服务器socket”：

Python

```
# create an INET,
STREAMing socket
1 # create an INET, STREAMing socket
2 serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 # bind the socket to a public host, and a well-known port
4 serversocket.bind((socket.gethostname(), 80))
5 # become a server socket
6 serversocket.listen(5)
```

有几件事要注意：我们使用`socket.gethostname()`，这样外面就能访问到socket了。如果我们用的是`s.bind('localhost', 80)` 或 `s.bind('127.0.0.1', 80)`，我们仍然得到一个“服务器”socket，但它只能在同一台机器上访问它了。`s.bind(('', 80))`意味着socket可以被这台机器拥有的任意地址访问。

第二个要注意的是：小数值的端口号一般留给“众所周知”的服务（HTTP, SNMP等）。如果你随便玩玩，用一个更好的大数值吧（4位）。

最后，传给`listen`的参数告诉socket库，在拒绝外面的连接之前，我们想让它最多有5个连接请求（通常最多就这么大）队列。如果后面的代码写得正确，应该就够了。

现在我们有了「服务器」socket了，监听在80端口，我们可以进入web服务器主循环了：

Python

```
while True:
    # accept connections
1 while True:
2     # accept connections from outside
3     (clientsocket, address) = serversocket.accept()
4     # now do something with the clientsocket
5     # in this case, we'll pretend this is a threaded server
6     ct = client_thread(clientsocket)
7     ct.run()
```

实际上在这个循环里有3种通用做法 —— 分发一个线程来处理`clientsocket`，创建一个新进程来处理`clientsocket`，或者重构本应用，使用非阻塞sockets，然后用`select`多路复用“服务器”socket和活动的`clientsockets`。以后再详细说这个。现在要理解一个要点：“服务器”socket只做这件事。它不发送任何数据。也不接收任何数据。它只生产客户端sockets。每当别的客户端socket使用`connect()`连接我们绑定的主机和端口时，都会生成一个`clientsocket`。一生成`clientsocket`，我们就返回去监听更多的连接。两个“客户端”总是可以通信——它们用的动态分配的端口会在通信结束时被回收掉。

进程间通信 (IPC)

如果你需要在同一台机器上快速地进程间通信，你应该看一下管道或者共享内存。如果你确实想用AF_INET sockets，那就把“服务器”socket绑定到'localhost'。在大多数平台上，这么做会绕过很多网络层，从而变快很多。

Python

See also: The 

1 See also: The multiprocessing integrates cross-platform IPC into a higher-level API.

参见：[multiprocessing](#)把跨平台进程间通信封装成了更高级别的API。

使用Socket

首先要注意到的事情是，web浏览器的”客户端”socket和web服务器的”客户端”socket是同一个东西。也就是说，这是一个对等网络（p2p）通信。或者换句话说，作为一个设计者，你必须决定通信的规则。通常，连接socket通过发送请求或登录来启动通信。但这是你设计的——不是sockets的规定。

目前通信有两套动作可用。你可以使用send和recv，或者把客户端socket转成类似文件的东西，然后使用read和write。Java给它的socket提供的就是后一种方式。在这我不打算讲它，但是要提醒你，你需要对sockets使用flush。sockets带有是缓冲的”文件”，一个常见的错误就是写入了一些东西，然后去读取响应。但是如果我没有flush，你可能要永远地等下去了，因为请求可能还在输出缓冲里。

现在咱们看看sockets的主要难点吧——网络缓冲的send和recv操作。它们并不一定处理你传递给它们（或期望从其得到）的所有的字节，因为它们的注意力主要集中在处理网络缓冲。通常，当分配的网络缓冲被填充（send）了或空（recv）了，它们就会返回。告诉你处理了多少字节。当消息被完全处理后你需要再次调用它们。

当recv返回0字节时，意味着另一端已经关闭（或者正在关闭）了连接。从这个连接上你再也接收不到数据了。但你可能可以成功发送数据；等下我会详细讲这点。

像HTTP这样的协议每次传输都只使用一个socket。客户端发送请求，然后读取回复。就这样。然后丢弃socket。就是说客户端接收到0字节就知道回复结束了。

但是，如果你打算在以后的传输中重用socket，你就得知道scoket里是没有EOT的。我再重复一遍：如果socket send或recv返回0字节，那这个连接就断开了。如果连接没有断开，就会永远的等下去，因为socket不会告诉你没有东西可读（目前）。现在如果你多想一下，你就会发现一个基本的事实：消息必须是固定长度的（呸），或带分隔符的（耸肩），或指示长度（好多啦），或者当连接关闭时结束。任你选择，（但有的方式比别的好）。

假设你不想关闭连接，最简单的解决方案是使用固定长度的消息：

Python

```
class MySocket:  
    """demonstration  
  
1 class MySocket:  
2     """demonstration class only  
3     - coded for clarity, not efficiency  
4     """  
5  
6     def __init__(self, sock=None):  
7         if sock is None:  
8             self.sock = socket.socket(  
9                 socket.AF_INET, socket.SOCK_STREAM)  
10            else:  
11                self.sock = sock
```

```

12
13     def connect(self, host, port):
14         self.sock.connect((host, port))
15
16     def mysend(self, msg):
17         totalsent = 0
18         while totalsent < MSGLEN:
19             sent = self.sock.send(msg[totalsent:])
20             if sent == 0:
21                 raise RuntimeError("socket connection broken")
22             totalsent = totalsent + sent
23
24     def myreceive(self):
25         chunks = []
26         bytes_recd = 0
27         while bytes_recd < MSGLEN:
28             chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
29             if chunk == b"":
30                 raise RuntimeError("socket connection broken")
31             chunks.append(chunk)
32             bytes_recd = bytes_recd + len(chunk)
33         return b"".join(chunks)

```

这里的发送代码适用于任何消息模式——使用Python发送字符串，可以使用`len()`来判断字符串长度（甚至当字符串包含字符时）。多数情况下接收代码比较复杂。（使用C的话，不会太坏，除了当消息包含时你不能使用`strlen`）

最简单的改进是让消息的第一个字符表示消息类型，并让类型来决定长度。现在你需要进行两次`recv`了——首先(至少)要获取第一个字符，这样你就能知道长度了，然后循环获取剩下的。如果你打算使用分隔符，就得使用一个大小任意的块来接收数据，（4096或8192通常比较适合网络缓冲大小），然后从接收到的数据里搜索分隔符。

有个麻烦的事要注意：如果你的通信协议允许连续发送多个消息（不需要某种回复），然后传任意大的块给`recv`，你可能会读到下一个消息的头。你必须把它先存起来，直到需要用到它的时候再使用。

在消息前加个表示它的长度（就是说，5个数字类型的字符）的前缀更复杂些，因为（信不信由你）一次`recv`可能不能全部读够5字符。在测试时，你可能能侥幸避免；但在网络繁忙时，你的代码很快就会挂掉，除非你使用两个`recv`循环——第一个用来决定长度，第二个读取消息的数据部分。好“恶心”。同样“恶心”的是，你会发现`send`也不能一次性解决。尽管你已经读了本文，最后还会在这上面栽跟头的。

为了节省篇幅，塑造你的人格，（并且保持我自己的竞争地位），这些改进会作为练习留给读者解决。让我们继续。

二进制数据

完全可以通过socket发送二进制数据。主要的问题在于，不是所有的机器都使用相同的二进制格式。举个例子，摩托罗拉的芯片使用两个十六进制字节00 01来表示16位的整数1。然而，英特尔和DEC，字节就反过来了一——同样是1就用01 00表示。socket库必须调用`ntohl`, `htonl`, `ntohs`, `htons`把转换16和32位整数。这里的“n”表示网络，“h”表示主机，“s”表示短整型，“l”表示长整型。当网络字节序和主机字节序相同时，它们什么也不做，否则，它们就相应的交换字节。

对于现如今的32位机器，使用`ascii`表示二进制数据通常比二进制表示法要小。这是因为在数据传输的

大量时间里，数据流的内容要么是0，要么是1。用字符表示“0”需要两个字节，而用二进制表示要4个字节。当然，这种情况对于固定长度的消息并不适用。所以在选择数据表示法时一定要好好考虑。

断开连接

严格来说，在关闭socket之前，你应该先shutdown它。shutdown就是给另一端的socket一个通知。它可以表示“我不会再发送数据啦，但我还在接收呢”，或者“我不在接收啦，解放啦！”，取决于传递给它的参数。然而，大多数socket库，对程序员忽略这个礼节已经很习惯了，通常close就跟先shutdown(); close()一样。所以，在大多数情况下，一个明确的shutdown就不需要了。

有效的使用shutdown的一种方式是在类HTTP通信中。客户端发送一个请求，然后调用shutdown(1)。这样就告诉服务器“这个客户端已经发送结束了，但还在接收呢。”了。服务器可以通过接收到0字节来判断“EOF”。它（服务器）就可以认为它（客户端）已经完成了请求。然后服务器发送一个回复。如果成功发送完成后，客户端实际上仍然在接收。

Python把这个自动shutdown的传统更进一步，也就是说，当一个socket被垃圾回收时，如果需要，它会自动close。但依赖这个是个非常坏的习惯。如果socket没有调用close就消失了，另一端的socket就会一直挂起，它会认为你只是变慢了而已。当结束时请close掉socket。

当sockets挂掉的时候

可能使用阻塞socket最坏的事就是遇到另一端的socket挂了（没有调用close）。你的socket就很可能被挂起。TCP是可靠的协议，它会等待很久，直到放弃了这个连接。如果你是使用线程，整个线程就死了。你帮不了什么忙。只要你没有做什么蠢事，比如在阻塞读的时候锁，线程就不会消耗太多的资源。不要尝试去杀死线程——部分原因是，线程比进程高效，线程避免了分配自动回收的资源的开销。换句话说，如果你设法去结束线程，你的整个进程可能会被弄糟。

非阻塞socket

如果你已经理解了前面说的，你就知道了使用socket的原理。你还是以非常相似的方式去调用相同的函数。事实上，if you do it right, your app will be almost inside-out.

在Python里，要用socket.setblocking(0)来设置非阻塞。在C里，更复杂了，（首先，你要从BSD风格的O_NONBLOCK和几乎难以分辨的Posix风格的O_NDELAY选择，O_NDELAY跟TCP_NODELAY完全不同），但它的原理一致。你要在创建socket之后，使用之前做这件事。（实际上，如果你已经抓狂了，你可以转回去再看看。）

主要的区别是，send, recv, connect和accept会在未完成前返回。你（当然）有很多选择。你可以检查返回值和错误码，一般这样做会让你抓狂的。不信你找个时间试试。你的应用会变得越来越臃肿，bug不断，还浪费CPU。所以，咱们跳过这个愚蠢的方案用正确的吧。

那就是用select。

在C里，使用select相当复杂。在Python里，它是块甜点，但它跟C版本的很像，如果你理解了Python里的select，在C里你也不会有太大困难：

```
ready_to_read, ready_to_write, in_error
```

```
1 ready_to_read, ready_to_write, in_error =
2     select.select(
3         potential_readers,
4         potential_writers,
5         potential_errs,
6         timeout)
```

传给select三个列表：第一个包含你想要读的所有socket；第二个包含你想要写的所有socket；最后一个（通常置空）包含那些你想要检查错误的socket。应当注意，一个socket可以在多个列表里。select调用是阻塞的，但可以给它一个超时设置。一般明智的做法是——给它一个合理长的超时时间（比如一分钟），除非有个更好的原因让你不这样做。

在返回值里，就能取到三个列表啦。它们包含了确实可读，可写和出错的socket。每一个列表（可能为空）都是相应传入的列表的子集。

如果有个socket在输出的可读列表中，你几乎可以肯定对这个socket调用recv会返回些东西。同理可证可写列表可以send些东西。也许不能recv或send你想要的全部，但聊胜于无。（实际上，任何正常的socket将作为可写的socket被返回——这只表示出口网络缓冲空间是可用的。）

如果你有一个“服务器”socket，把它放入可能可读列表里。如果返回的可读列表中有它，accept（几乎必然）可成功调用。如果你创建了一个连接别人的新的socket，把它放入可能可写列表里，如果它在可写列表里出现了，表示它已经连接上了。

实际上，select对于阻塞socket也很方便好用。它是判断是否阻塞的一种方式——socket会在缓冲里有数据时返回可读。然而，这并不能解决这个问题：判断另一端是否完成，或忙于处理别的事。

可移植警告：在Unix上，select能处理socket和file。在Windows上不要尝试这个。在Windows上，select只能处理socket。另外说下在C里，很多socket高级选项在Windows上是有区别的。事实上，在Windows上，我通常使用线程处理socket（它工作得非常，非常好）。

1 赞 5 收藏 [1 评论](#)

关于作者：[高世界](#)



我翻译得越多，发现知道的越少，我就要更多地翻译。论得的地的正确用法。我是php开发者，对python, c/c++, linux感兴趣。[个人主页](#) · [我的文章](#) · 17 ·

一起写一个Web服务器（3）

本文由[伯乐在线 - 高世界](#)翻译，[董利民](#)校稿。未经许可，禁止转载！

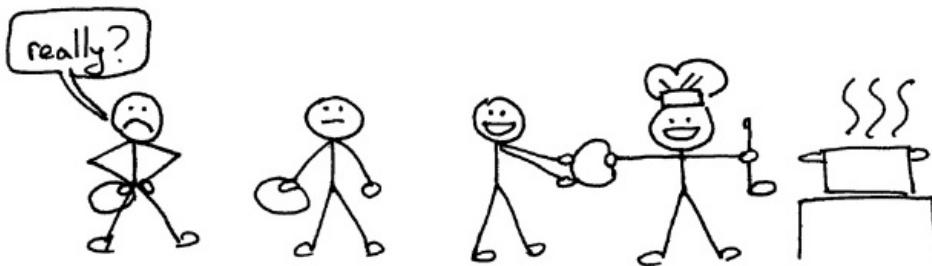
英文出处：[ruslan spivak](#)。欢迎加入[翻译组](#)。

- [一起写一个 Web 服务器（2）](#)
- [一起写一个 Web 服务器（1）](#)

“发明创造时，我们学得最多”——Piaget

在本系列第二部分，你已经创造了一个可以处理基本的HTTP GET请求的WSGI服务器。我还问了你一个问题，“怎么让服务器在同一时间处理多个请求？”在本文中你将找到答案。那么，系好安全带加大马力。你马上就乘上快车啦。准备好Linux、Mac OS X（或任何类unix系统）和Python。本文的所有源码都能在GitHub上找到。

首先咱们回忆下一个基本的Web服务器长什么样，要处理客户端请求它得做什么。你在第一部分和第二部分创建的是一个迭代的服务器，每次处理一个客户端请求。除非已经处理了当前的客户端请求，否则它不能接受新的连接。有些客户端对此就不开心了，因为它们必须要排队等待，而且如果服务器繁忙的话，这个队伍会很长。

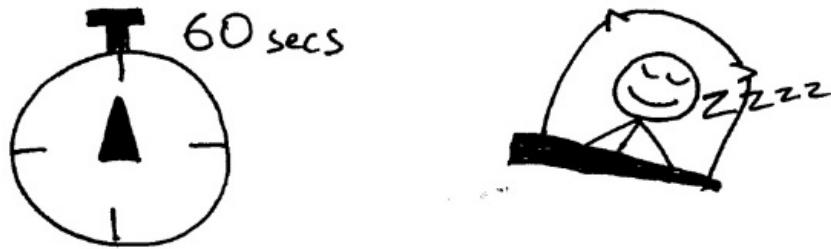


以下是迭代服务器webserver3a.py的代码：

Python

```
#####
# Iterative server - webserver3a.py          #
#                                                 #
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
#####
6 import socket
7
8 SERVER_ADDRESS = (HOST, PORT) = "", 8888
9 REQUEST_QUEUE_SIZE = 5
10
11 def handle_request(client_connection):
12     request = client_connection.recv(1024)
13     print(request.decode())
14     http_response = b"""
15 HTTP/1.1 200 OK
16
17 Hello, World!
18 """
19     client_connection.sendall(http_response)
20
21 def serve_forever():
22     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
24     listen_socket.bind(SERVER_ADDRESS)
25     listen_socket.listen(REQUEST_QUEUE_SIZE)
26     print('Serving HTTP on port {port} ...'.format(port=PORT))
27
28     while True:
29         client_connection, client_address = listen_socket.accept()
30         handle_request(client_connection)
31         client_connection.close()
32
33 if __name__ == '__main__':
34     serve_forever()
```

要观察服务器同一时间只处理一个客户端请求，稍微修改一下服务器，在每次发送给客户端响应后添加一个60秒的延迟。添加这行代码就是告诉服务器睡眠60秒。



以下是睡眠版的服务器webserver3b.py代码：

Python

```
#####
# Iterative server - webserver3b.py
#
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X
#
# - Server sleeps for 60 seconds after sending a response to a client
#####
import socket
import time

SERVER_ADDRESS = (HOST, PORT) = "", 8888
REQUEST_QUEUE_SIZE = 5

def handle_request(client_connection):
    request = client_connection.recv(1024)
    print(request.decode())
    http_response = b"""
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, World!
"""

    client_connection.sendall(http_response)
    time.sleep(60) # sleep and block the process for 60 seconds

def serve_forever():
    listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listen_socket.bind(SERVER_ADDRESS)
    listen_socket.listen(REQUEST_QUEUE_SIZE)
    print('Serving HTTP on port {}'.format(PORT))

    while True:
        client_connection, client_address = listen_socket.accept()
        handle_request(client_connection)
        client_connection.close()

if __name__ == '__main__':
    serve_forever()
```

启动服务器：

Python

```
$ python webserver3b.py
$ python webserver3b.py
```

现在打开一个新的控制台窗口，运行以下curl命令。你应该立即就会看到屏幕上打印出了“Hello, World!”字符串：

Python

```
$ curl  
http://localhost:8888/hello  
1 $ curl http://localhost:8888/hello  
2 Hello, World!  
3  
4 And without delay open up a second terminal window and run the same curl command:
```

立刻再打开一个控制台窗口，然后运行相同的curl命令：

Python

```
$ curl  
http://localhost:8888/hello  
1 $ curl http://localhost:8888/hello
```

如果你是在60秒内做的，那么第二个curl应该不会立刻产生任何输出，而是挂起。而且服务器也不会在标准输出打印出新请求体。在我的Mac上看起来像这样（在右下角的黄色高亮窗口表示第二个curl命令正挂起，等待服务器接受这个连接）：

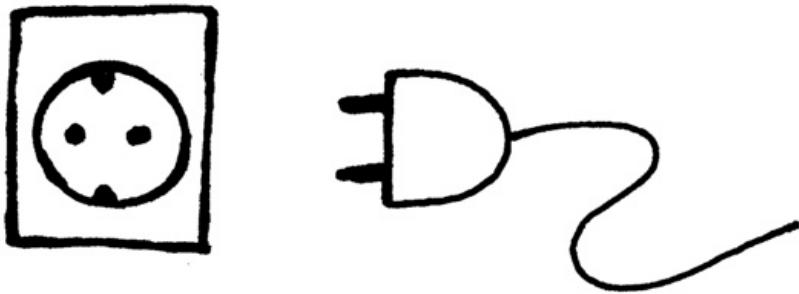
```
(lsbows)Ruslans-MacBook-Air:part3 rspivak$ python webserver3b.py  
Serving HTTP on port 8888 ...  
GET /hello HTTP/1.1  
User-Agent: curl/7.37.1  
Host: localhost:8888  
Accept: */*  
  
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello  
Hello, World!  
  
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello  
  
Session: 0 3 2 1:bash 2:bash- 3:curl*
```

当你等待足够长时间（大于60秒）后，你会看到第一个curl终止了，第二个curl在屏幕上打印出“Hello, World!”，然后挂起60秒，然后再终止：

```
(lsbows)Ruslans-MacBook-Air:part3 rspivak$ python webserver3b.py  
Serving HTTP on port 8888 ...  
GET /hello HTTP/1.1  
User-Agent: curl/7.37.1  
Host: localhost:8888  
Accept: */*  
  
GET /hello HTTP/1.1  
User-Agent: curl/7.37.1  
Host: localhost:8888  
Accept: */*  
  
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello  
Hello, World!  
Ruslans-MacBook-Air:~ rspivak$  
  
Ruslans-MacBook-Air:~ rspivak$ curl http://localhost:8888/hello  
Hello, World!  
Ruslans-MacBook-Air:~ rspivak$  
  
Session: 0 3 2 1:bash 2:bash- 3:bash*
```

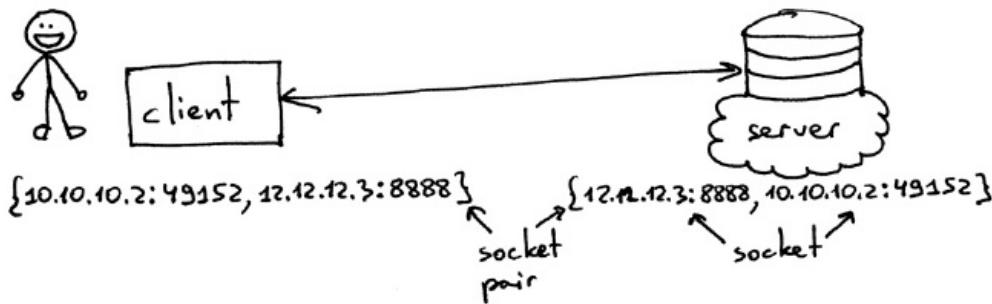
它是这么工作的，服务器完成处理第一个curl客户端请求，然后睡眠60秒后开始处理第二个请求。这些都是顺序地，或者迭代地，一步一步地，或者，在我们例子中是一次一个客户端请求地，发生。

咱们讨论点客户端和服务器的通信吧。为了让两个程序能够网络通信，它们必须使用socket。你在[第一部分](#)和[第二部分](#)已经见过socket了，但是，socket是什么呢？



socket就是通信终端的一种抽象，它允许你的程序使用文件描述符和别的程序通信。本文我将详细谈谈在Linux/Mac OS X上的TCP/IP socket。理解socket的一个重要的概念是TCP socket对。

TCP的socket对是一个4元组，标识着TCP连接的两个终端：本地IP地址、本地端口、远程IP地址、远程端口。一个socket对唯一地标识着网络上的TCP连接。标识着每个终端的两个值，IP地址和端口号，通常被称为socket。



所以，元组{10.10.10.2:49152, 12.12.12.3:8888}是客户端TCP连接的唯一标识着两个终端的socket对。元组{12.12.12.3:8888, 10.10.10.2:49152}是服务器TCP连接的唯一标识着两个终端的socket对。标识TCP连接中服务器终端的两个值，IP地址12.12.12.3和端口8888，在这里就是指socket（同样适用于客户端终端）。

服务器创建一个socket并开始接受客户端连接的标准流程经历通常如下：

socket



bind



listen



accept



1. 服务器创建一个TCP/IP socket。在Python里使用下面的语句即可：

Python

```
listen_socket =  
    socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. 服务器可能会设置一些socket选项（这是可选的，上面的代码就设置了，为了在杀死或重启服务器后，立马就能再次重用相同的地址）。

Python

```
listen_socket.setsockopt  
    (socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

3. 然后，服务器绑定指定地址，bind函数分配一个本地地址给socket。在TCP中，调用bind可以指定一个端口号，一个IP地址，两者都，或者两者都不指定。

Python

```
listen_socket.bind(SERVER  
    _ADDRESS)
```

4. 然后，服务器让这个socket成为监听socket。

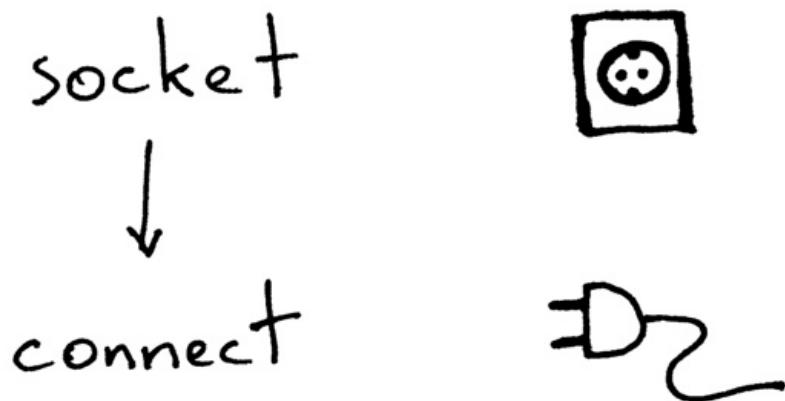
Python

```
listen_socket.listen(REQUEST_QUEUE_SIZE)  
1 listen_socket.listen(REQUEST_QUEUE_SIZE)
```

listen方法只会被服务器调用。它告诉内核它要接受这个socket上的到来的连接请求了。

做完这些后，服务器开始循环地一次接受一个客户端连接。当有连接到达时，accept调用返回已连接的客户端socket。然后，服务器从这个socket读取请求数据，在标准输出上把数据打印出来，并回发一个消息给客户端。然后，服务器关闭客户端连接，准备好再次接受新的客户端连接。

下面是客户端使用TCP/IP和服务器通信要做的：



以下是客户端连接服务器，发送请求并打印响应的示例代码：

Python

```
import socket  
1 import socket  
2  
3 # create a socket and connect to a server  
4 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
5 sock.connect(('localhost', 8888))  
6  
7 # send and receive some data  
8 sock.sendall(b'test')  
9 data = sock.recv(1024)  
10 print(data.decode())
```

创建socket后，客户端需要连接服务器。这是通过connect调用做到的：

Python

```
sock.connect(('localhost',  
8888))  
1 sock.connect(('localhost', 8888))
```

客户端仅需提供要连接的远程IP地址或主机名和远程端口号即可。

可能你注意到了，客户端不用调用bind和accept。客户端没必要调用bind，是因为客户端不关心本地IP地址和本地端口号。当客户端调用connect时内核的TCP/IP栈自动分配一个本地IP地址和本地端口。本地端口被称为暂时端口（ephemeral port），也就是，short-lived端口。



服务器上标识着一个客户端连接的众所周知的服务的端口被称为well-known端口（举例来说，80就是HTTP，22就是SSH）。操起Python shell，创建个连接到本地服务器的客户端连接，看看内核分配给你创建的socket的暂时的端口是多少（在这之前启动webserver3a.py或webserver3b.py）：

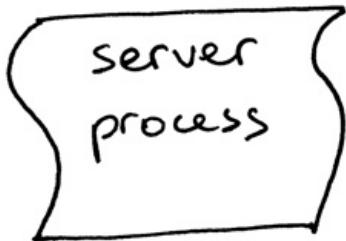
Python

```
>>> import socket  
>>> sock =  
1 >>> import socket  
2 >>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
3 >>> sock.connect(('localhost', 8888))  
4 >>> host, port = sock.getsockname()[:2]  
5 >>> host, port  
6 ('127.0.0.1', 60589)
```

上面这个例子中，内核分配了60589这个暂时端口。

在我开始回答第二部分提出的问题前，我需要快速讲一下几个重要的概念。你很快就知道为什么重要了。两个概念是进程和文件描述符。

什么是进程？进程就是一个正在运行的程序的实例。比如，当服务器代码执行时，它被加载进内存，运行起来的程序实例被称为进程。内核记录了进程的一堆信息用于跟踪，进程ID就是一个例子。当你运行服务器 webserver3a.py 或 webserver3b.py 时，你就在运行一个进程了。



在控制台窗口运行webserver3b.py：

Python

```
$ python webserver3b.py  
1 $ python webserver3b.py
```

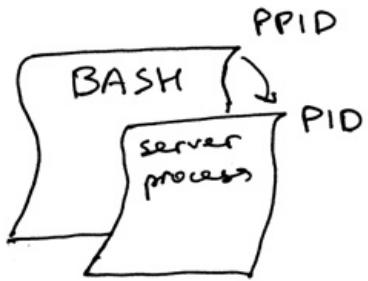
在别的控制台窗口使用ps命令获取这个进程的信息：

Python

```
$ ps | grep webserver3b  
| grep -v grep  
1 $ ps | grep webserver3b | grep -v grep  
2 7182 ttys003 0:00.04 python webserver3b.py
```

ps命令表示你确实运行了一个Python进程webserver3b。进程创建时，内核分配给它一个进程ID，也就是 PID。在UNIX里，每个用户进

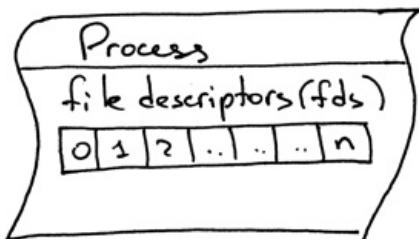
程都有个父进程，父进程也有它自己的进程ID，叫做父进程ID，或者简称PPID。假设默认你是在BASH shell里运行的服务器，那新进程的父进程ID就是BASH shell的进程ID。



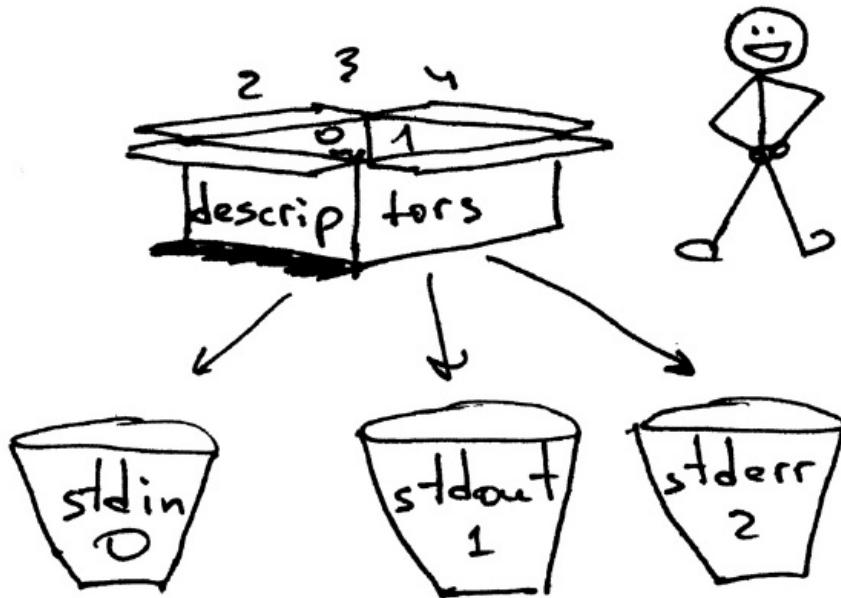
自己试试，看看它是怎么工作的。再启动Python shell，这将创建一个新进程，使用os.getpid() 和 os.getppid() 系统调用获取Python shell进程的ID和父进程ID (BASH shell的PID)。然后，在另一个控制台窗口运行ps命令，使用grep查找PPID(父进程ID，我的是3148)。在下面的截图你可以看到在我的Mac OS X上，子Python shell进程和父BASH shell进程的关系：

```
>>> import os
>>> os.getpid()
10236
>>> os.getppid()
3148
>>>
Ruslans-MacBook-Air:~ rspivak$ ps -opid,ppid,args | grep 3148 | grep -v grep
3148 1391 -bash
10236 3148 /usr/local/Cellar/python/2.7.9/Frameworks/Python.framework/Versions/2.7/Resources/Python.app/Contents/MacOS/Python
Ruslans-MacBook-Air:~ rspivak$
```

另一个要了解的重要概念是文件描述符。那么什么是文件描述符呢？文件描述符是当打开一个存在的文件，创建一个文件，或者创建一个socket时，内核返回的非负整数。你可能已经听过啦，在UNIX里一切皆文件。内核使用文件描述符来追踪进程打开的文件。当你需要读或写文件时，你就用文件描述符标识它好啦。Python给你包装成更高级别的对象来处理文件（和socket），你不必直接使用文件描述符来标识一个文件，但是，在底层，UNIX中是这样标识文件和socket的：通过它们的整数文件描述符。



默认情况下，UNIX shell分配文件描述符0给进程的标准输入，文件描述符1给进程的标准输出，文件描述符2给标准错误。



就像我前面说的，虽然Python给了你更高级别的文件或者类文件的对象，你仍然可以使用对象的fileno()方法来获取对应的文件描述符。回到Python shell来看看怎么做：

Python

```
>>> import sys
>>> sys.stdin
1 >>> import sys
2 >>> sys.stdin
3 <open file '<stdin>', mode 'r' at 0x102beb0c0>
4 >>> sys.stdin.fileno()
5 0
6 >>> sys.stdout.fileno()
7 1
8 >>> sys.stderr.fileno()
9 2
```

虽然在Python中处理文件和socket，通常使用高级的文件/socket对象，但有时候你需要直接使用文件描述符。下面这个例子告诉你如何使用write系统调用写一个字符串到标准输出，write使用整数文件描述符做为参数：

Python

```
>>> import sys
>>> import os
1 >>> import sys
2 >>> import os
3 >>> res = os.write(sys.stdout.fileno(), 'hellon')
4 hello
```

有趣的是——应该不会惊讶到你啦，因为你已经知道在UNIX里一切皆文件——socket也有一个分配给它的文件描述符。再说一遍，当你创建一个socket时，你得到的是一个对象而不是非负整数，但你也可以使用我前面提到的fileno()方法直接访问socket的文件描述符。

还有一件事我想说下：你注意到了吗？在第二个例子webserver3b.py中，当服务器进程在60秒的睡眠时你仍然可以用curl命令来连接。当然啦，curl没有立刻输出什么，它只是在那挂起。但为什么服务器不接受连接，客户端也不立刻被拒绝，而是能连接服务器呢？答案就是socket对象的listen方法和它的BACKLOG参数，我称它为 REQUEST_QUEUE_SIZE(请求队列长度)。BACKLOG参数决定了内核为进入的连接请求准备的队列长度。当服务器webser3b.py睡眠时，第二个curl命令可以连接到服务器，因为内核在服务器socket的进入连接请求队列上有足够的可用空间。

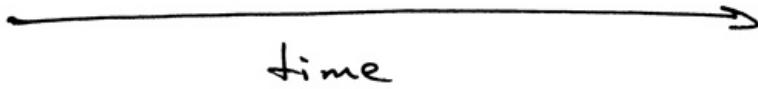
然而增加BACKLOG参数不会神奇地让服务器同时处理多个客户端请求，设置一个合理大点的backlog参数挺重要的，这样accept调用就不用等新连接建立起来，立刻就能从队列里获取新的连接，然后开始处理客户端请求啦。

吼吼！你已经了解了非常多的背景知识啦。咱们快速简要重述到目前为止你都学了什么（如果你都知道啦就温习一下吧）。



- 迭代服务器
- 服务器socket创建流程 (socket, bind, listen, accept)
- 客户端连接创建流程 (socket, connect)
- socket对
- socket
- 临时端口和众所周知端口
- 进程
- 进程ID (PID) , 父进程ID (PPID) , 父子关系。
- 文件描述符
- listen方法的BACKLOG参数的意义

现在我准备回答第二部分问题的答案了：“怎样才能让服务器同时处理多个请求？”或者换句话说，“怎样写一个并发服务器？”



在Unix上写一个并发服务器最简单的方法是使用fork()系统调用。



下面就是新的牛逼闪闪的并发服务器webserver3c.py的代码，它能同时处理多个客户端请求（和咱们迭代服务器例子webserver3b.py一样，每个子进程睡眠60秒）：



Python

```
#####
# Concurrent server - webserver3c.py
#
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X
#
# - Child process sleeps for 60 seconds after handling a client's request
# - Parent and child processes close duplicate descriptors
#
#####
1 import os
2 import socket
3 import time
4
5 SERVER_ADDRESS = (HOST, PORT) = "", 8888
6 REQUEST_QUEUE_SIZE = 5
7
8 def handle_request(client_connection):
9     request = client_connection.recv(1024)
10    print(
11        'Child PID: {pid}. Parent PID {ppid}'.format(
12            pid=os.getpid(),
13            ppid=os.getppid(),
14        )
15    )
16    print(request.decode())
17    http_response = b"""
18        HTTP/1.1 200 OK
19
20        Hello, World!
21 """
22    client_connection.sendall(http_response)
23    time.sleep(60)
24
25 def serve_forever():
26     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
27     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
28     listen_socket.bind(SERVER_ADDRESS)
29     listen_socket.listen(REQUEST_QUEUE_SIZE)
30     print('Serving HTTP on port {}'.format(PORT))
31     print('Parent PID (PPID): {}'.format(pid=os.getpid()))
```

```

41
42 while True:
43     client_connection, client_address = listen_socket.accept()
44     pid = os.fork()
45     if pid == 0: # child
46         listen_socket.close() # close child copy
47         handle_request(client_connection)
48         client_connection.close()
49         os._exit(0) # child exits here
50     else: # parent
51         client_connection.close() # close parent copy and loop over
52
53 if __name__ == '__main__':
54     serve_forever()

```

在深入讨论for如何工作之前，先自己试试，看看服务器确实可以同时处理多个请求，不像webserver3a.py和webserver3b.py。用下面命令启动服务器：

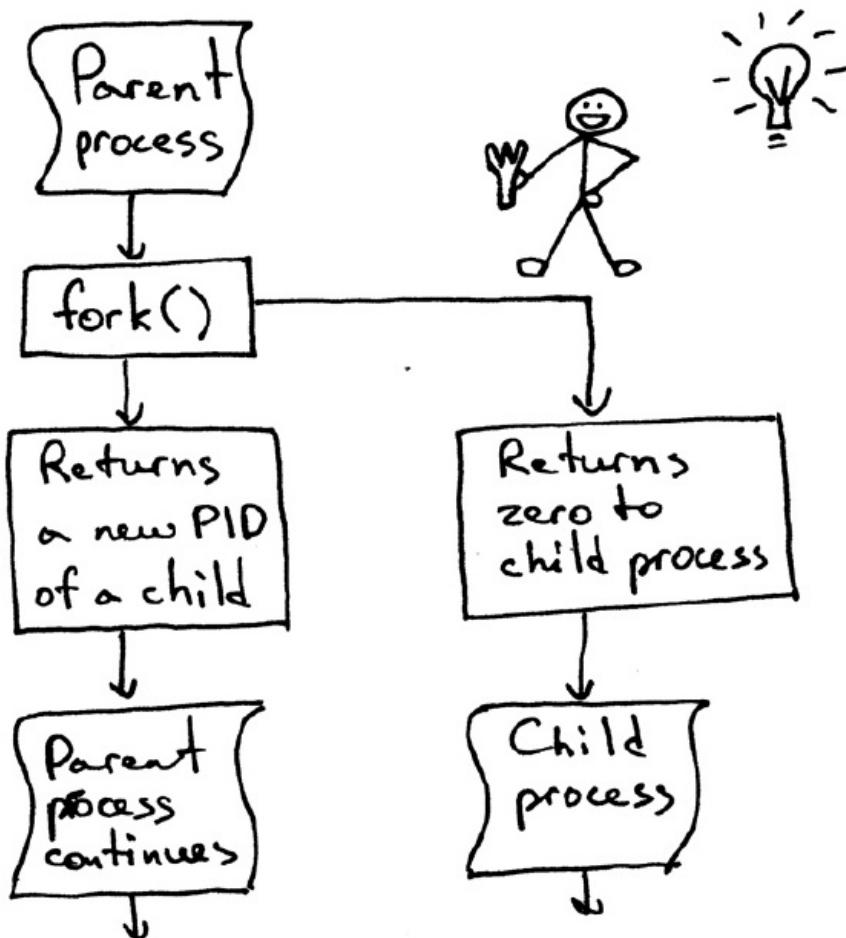
Python

```
$ python webserver3c.py
```

```
1 $ python webserver3c.py
```

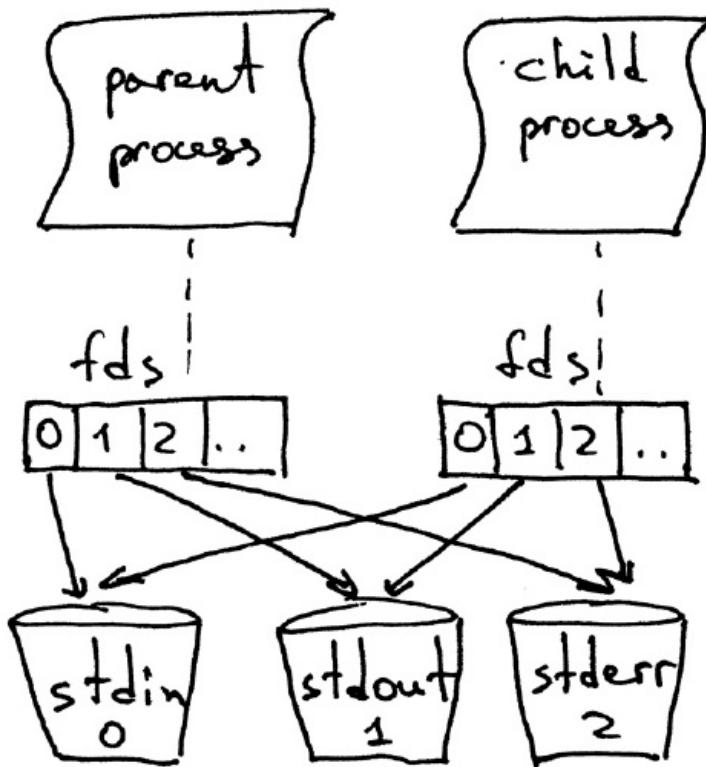
像你以前那样试试用两个curl命令，自己看看，现在虽然服务器子进程在处理客户端请求时睡眠60秒，但不影响别的客户端，因为它们是被不同的完全独立的进程处理的。你应该能看到curl命令立刻就输出了“Hello, World!”，然后挂起60秒。你可以接着想运行多少curl命令就运行多少（嗯，几乎是任意多），它们都会立刻输出服务器的响应“Hello, Wrold”，而且不会有明显的延迟。试试看。

理解fork()的最重要的点是，你fork了一次，但它返回了两次：一个是在父进程中，一个是在子进程中。当你fork了一个新进程，子进程返回的进程ID是0。父进程中fork返回的是子进程的PID。



我仍然记得当我第一次知道它使用它时我对fork是有多着迷。它就像魔法一样。我正读着一段连续的代码，然后“duang”的一声：代码克隆了自己，然后就有两个相同代码的实例同时运行。我想除了魔法无法做到，我是认真哒。

当父进程fork了一个新的子进程，子进程就获取了父进程文件描述符的拷贝：



你可能已经注意到啦，上面代码里的父进程关闭了客户端连接：

Python

```
else: # parent
    ...
1 else: # parent
2 client_connection.close() # close parent copy and loop over
```

那么，如果它的父进程关闭了同一个socket，子进程为什么还能从客户端socket读取数据呢？答案就在上图。内核使用描述符引用计数来决定是否关闭socket。只有当描述符引用计数为0时才关闭socket。当服务器创建一个子进程，子进程获取了父进程的文件描述符拷贝，内核增加了这些描述符的引用计数。在一个父进程和一个子进程的场景中，客户端socket的描述符引用计数就成了2，当父进程关闭了客户端连接socket，它仅仅把引用计数减为1，不会引发内核关闭这个socket。子进程也把父进程的listen_socket拷贝给关闭了，因为子进程不用管接受新连接，它只关心处理已经连接的客户端的请求：

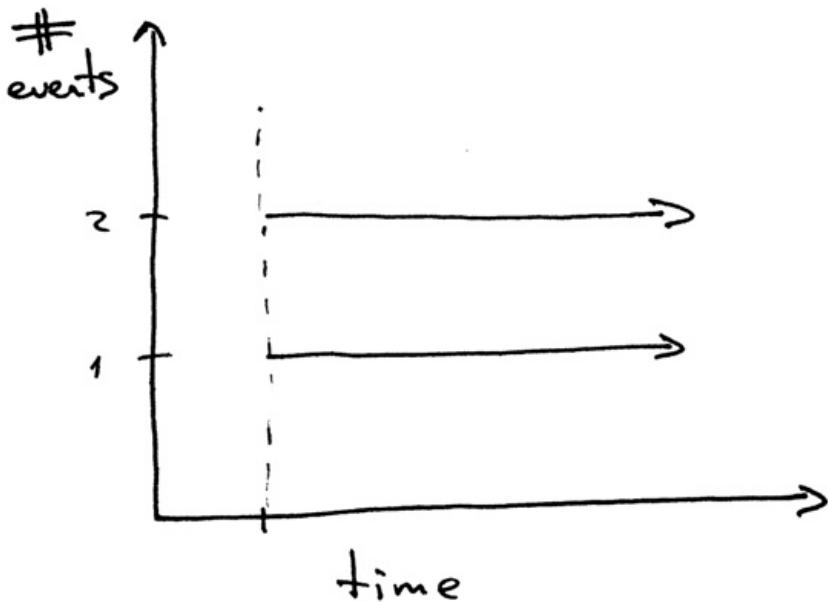
Python

```
listen_socket.close() #
close child copy
1 listen_socket.close() # close child copy
```

本文后面我会讲下如果不关闭复制的描述符会发生什么。

你从并发服务器源码看到啦，现在服务器父进程唯一的角色就是接受一个新的客户端连接，fork一个新的子进程来处理客户端请求，然后重复接受另一个客户端连接，就没有别的事做啦。服务器父进程不处理客户端请求——它的小弟（子进程）干这事。

跑个题，我们说两个事件并发到底是什么意思呢？



当我们说两个事件并发时，我们通常表达的是它们同时发生。简单来说，这也不错，但你要知道严格定义是这样的：

Python

如果你不能通过观察程序来知道哪个先发生

1 如果你不能通过观察程序来知道哪个先发生的，那么这两个事件就是并发的。

又到了简要重述目前为止已经学习的知识点和概念的时间啦.



- 在Unix下写一个并发服务器最简单的方法是使用fork()系统调用
- 当一个进程fork了一个新进程时，它就变成了那个新fork产生的子进程的父进程。
- 在调用fork后，父进程和子进程共享相同的文件描述符。
- 内核使用描述符引用计数来决定是否关闭文件/socket。
- 服务器父进程的角色是：现在它干的所有活就是接受一个新连接，fork一个子进来处理这个请求，然后循环接受新连接。

咱们来看看，如果在父进程和子进程中你不关闭复制的socket描述符会发生什么吧。以下是个修改后的版本，服务器不关闭复制的描述符，webserver3d.py：

Python

```
#####
1 ##### Concurrent server - webserver3d.py #####
2 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X      #
3 # import os
4 import socket
5 SERVER_ADDRESS = (HOST, PORT) = "", 8888
6 REQUEST_QUEUE_SIZE = 5
7
8
9 def handle_request(client_connection):
10    request = client_connection.recv(1024)
11    http_response = b"""
12    HTTP/1.1 200 OK
13
14
15
```

```

16
17 Hello, World!
18 """
19   client_connection.sendall(http_response)
20
21 def serve_forever():
22     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
24     listen_socket.bind(SERVER_ADDRESS)
25     listen_socket.listen(REQUEST_QUEUE_SIZE)
26     print('Serving HTTP on port {port} ...'.format(port=PORT))
27
28     clients = []
29     while True:
30         client_connection, client_address = listen_socket.accept()
31         # store the reference otherwise it's garbage collected
32         # on the next loop run
33         clients.append(client_connection)
34         pid = os.fork()
35         if pid == 0: # child
36             listen_socket.close() # close child copy
37             handle_request(client_connection)
38             client_connection.close()
39             os._exit(0) # child exits here
40         else: # parent
41             # client_connection.close()
42             print(len(clients))
43
44 if __name__ == '__main__':
45     serve_forever()

```

启动服务器：

Python

```
$ python webserver3d.py
```

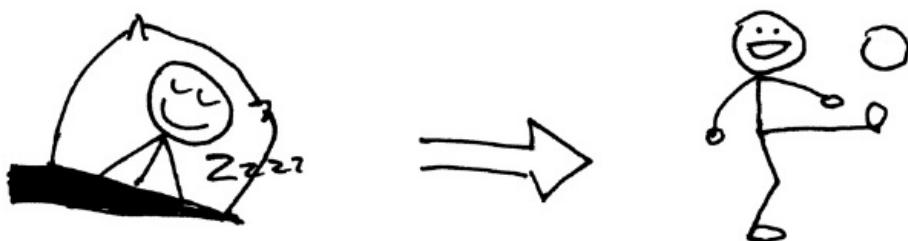
```
1 $ python webserver3d.py
```

使用curl去连接服务器：

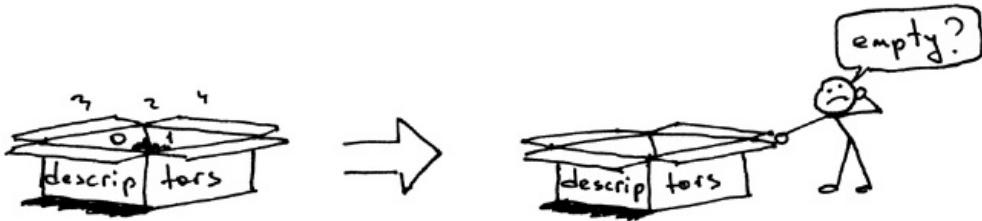
Python

```
$ curl http://localhost:8888/hello
1 $ curl http://localhost:8888/hello
2 Hello, World!
```

好的，curl打印出来并发服务器的响应，但是它不终止，一直挂起。发生了什么？服务器不再睡眠60秒了：它的子进程开心地处理了客户端请求，关闭了客户端连接然后退出啦，但是客户端curl仍然不终止。



那么，为什么curl不终止呢？原因就在于复制的文件描述符。当子进程关闭了客户端连接，内核减少引用计数，值变成了1。服务器子进程退出，但是客户端socket没有被内核关闭掉，因为引用计数不是0啊，所以，结果就是，终止数据包（在TCP/IP说法中叫做FIN）没有发送给客户端，所以客户端就保持在线啦。这里还有个问题，如果服务器不关闭复制的文件描述符然后长时间运行，最终会耗尽可用文件描述符。



使用Control-C停止webserver3d.py，使用shell内建的命令ulimit检查一下shell默认设置的进程可用资源：

Python

```
$ ulimit -a
core file size
```

- 1 \$ ulimit -a
- 2 core file size (blocks, -c) 0
- 3 data seg size (kbytes, -d) unlimited
- 4 scheduling priority (-e) 0
- 5 file size (blocks, -f) unlimited
- 6 pending signals (-i) 3842
- 7 max locked memory (kbytes, -l) 64
- 8 max memory size (kbytes, -m) unlimited
- 9 open files (-n) 1024
- 10 pipe size (512 bytes, -p) 8
- 11 POSIX message queues (bytes, -q) 819200
- 12 real-time priority (-r) 0
- 13 stack size (kbytes, -s) 8192
- 14 cpu time (seconds, -t) unlimited
- 15 max user processes (-u) 3842
- 16 virtual memory (kbytes, -v) unlimited
- 17 file locks (-x) unlimited

看到上面的了咩，我的Ubuntu上，进程的最大可打开文件描述符是1024。

现在咱们看看怎么让服务器耗尽可用文件描述符。在已存在或新的控制台窗口，调用服务器最大可打开文件描述符为256：

Python

```
$ ulimit -n 256
```

- 1 \$ ulimit -n 256

在同一个控制台上启动webserver3d.py：

Python

```
$ python webserver3d.py
```

- 1 \$ python webserver3d.py

使用下面的client3.py客户端来测试服务器。

Python

```
#####
# Test client - client3.py
# Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X #
#####
1 import argparse
2 import errno
3 import os
4 import socket
5
6 SERVER_ADDRESS = 'localhost', 8888
7 REQUEST = b""
```

```
13 GET /hello HTTP/1.1
14 Host: localhost:8888
15
16 """
17
18 def main(max_clients, max_conns):
19     socks = []
20     for client_num in range(max_clients):
21         pid = os.fork()
22         if pid == 0:
23             for connection_num in range(max_conns):
24                 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
25                 sock.connect(SERVER_ADDRESS)
26                 sock.sendall(REQUEST)
27                 socks.append(sock)
28                 print(connection_num)
29             os._exit(0)
30
31 if __name__ == '__main__':
32     parser = argparse.ArgumentParser(
33         description='Test client for LSBAWS.',
34         formatter_class=argparse.ArgumentDefaultsHelpFormatter,
35     )
36     parser.add_argument(
37         '--max-conns',
38         type=int,
39         default=1024,
40         help='Maximum number of connections per client.'
41     )
42     parser.add_argument(
43         '--max-clients',
44         type=int,
45         default=1,
46         help='Maximum number of clients.'
47     )
48     args = parser.parse_args()
49     main(args.max_clients, args.max_conns)
```

在新的控制台窗口里，启动client3.py，让它创建300个连接同时连接服务器。

Python

```
$ python client3.py --max-
clients=300
1 $ python client3.py --max-clients=300
```

很快服务器就崩了。下面是我电脑上抛异常的截图：

```
248
249
250
251
252
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
    File "webserver3d.py", line 43, in serve_forever
      File "/usr/lib/python2.7/socket.py", line 202, in accept
        socket.error: [Errno 24] Too many open files
```

教训非常明显啦——服务器应该关闭复制的描述符。但即使关闭了复制的描述符，你还没有接触到底层，因为你的服务器还有个问题，僵尸！



是哒，服务器代码就是产生了僵尸。咱们看下是怎么产生的。再次运行服务器：

Python

```
$ python webserver3d.py
$ python webserver3d.py
```

在另一个控制台窗口运行下面的curl命令：

Python

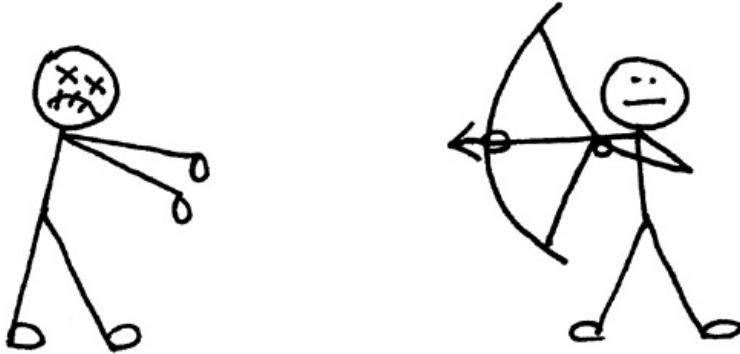
```
$ curl
http://localhost:8888/hello
$ curl http://localhost:8888/hello
```

现在运行ps命令，显示运行着的Python进程。以下是我的Ubuntu电脑上的ps输出：

Python

```
$ ps auxw | grep -i python | grep -v grep
$ ps auxw | grep -i python | grep -v grep
vagrant 9099 0.0 1.2 31804 6256 pts/0 S+ 16:33 0:00 python webserver3d.py
vagrant 9102 0.0 0.0 0 0 pts/0 Z+ 16:33 0:00 [python] </defunct>
```

你看到上面第二行了咩？它说PId为9102的进程的状态是Z+，进程的名称是。这个就是僵尸啦。僵尸的问题在于，你杀死不了他们啊。



即使你试着用 \$ kill -9 来杀死僵尸，它们还是会幸存下来哒，自己试试看看。

僵尸到底是什么呢？为什么咱们的服务器会产生它们呢？僵尸就是一个进程终止了，但是它的父进程没有等它，还没有接收到它的终止状态。当一个子进程比父进程先终止，内核把子进程转成僵尸，存储进程的一些信息，等着它的父进程以后获取。存储的信息通常就是进程ID，进程终止状态，进程使用的资源。嗯，僵尸还是有用的，但如果服务器不好好处理这些僵尸，系统就会越来越堵塞。咱们看看怎么做到的。首先停止服务器，然后新开一个控制台窗口，使用ulimit命令设置最大用户进程为400（确保设置打开文件更高，比如500吧）：

Python

```
$ ulimit -u 400
$ ulimit -n 500
```

1 \$ ulimit -u 400
2 \$ ulimit -n 500

在同一个控制台窗口运行webserver3d.py：

Python

```
$ python webserver3d.py
```

1 \$ python webserver3d.py

新开一个控制台窗口，启动client3.py，让它创建500个连接同时连接到服务器：

Python

```
$ python client3.py --max-clients=500
```

1 \$ python client3.py --max-clients=500

然后，服务器又一次崩了，是OSError的错误：抛了资源临时不可用的异常，当试图创建新的子进程时但创建不了时，因为达到了最大子进程数限制。以下是我的电脑的截图：

```
186
187
188
189
190
191
Traceback (most recent call last):
  File "webserver3d.py", line 58, in <module>
    serve_forever()
  File "webserver3d.py", line 47, in serve_forever
    pid = os.fork()
OSError: [Errno 11] Resource temporarily unavailable
```

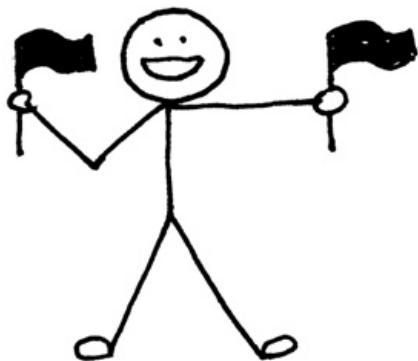
看到了吧，如果你不处理好僵尸，服务器长时间运行就会出问题。我会简短讨论下服务器应该怎样处理僵尸问题。

咱们简要重述下目前为止你已经学习到主要知识点：

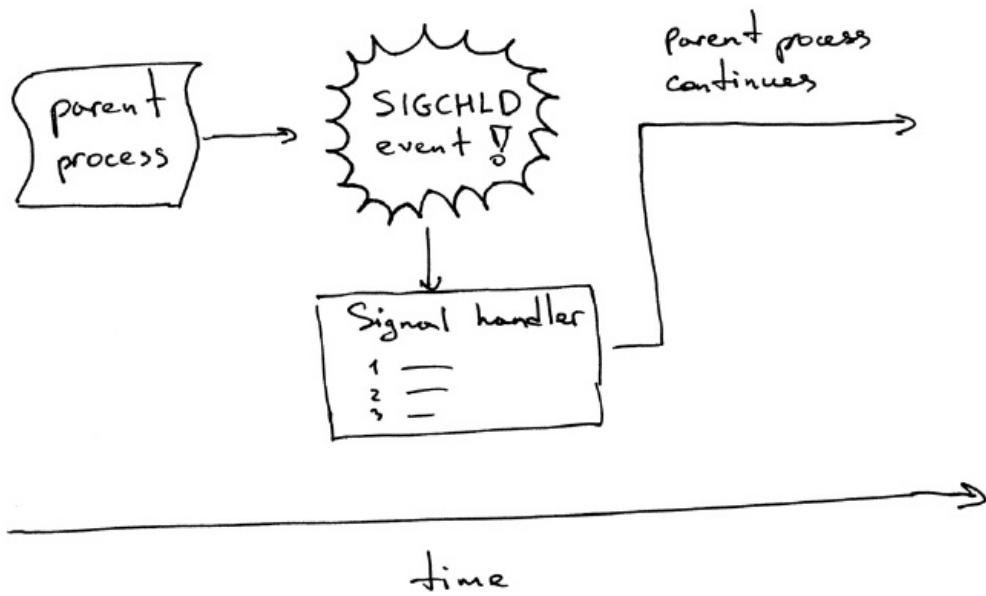


- 如果不关闭复制描述符，客户端不会终止，因为客户端连接不会关闭。
- 如果不关闭复制描述符，长时间运行的服务器最终会耗尽可用文件描述符（最大打开文件）。
- 当fork了一个子进程，然后子进程退出了，父进程没有等它，而且没有收集它的终止状态，它就变成僵尸了。
- 僵尸要吃东西，我们的场景中，就是内存。服务器最终会耗尽可用进程（最大用户进程），如果不处理好僵尸的话。
- 僵尸杀不死的，你需要等它们。

那么，处理好僵尸的话，要做什么呢？要修改服务器代码去等僵尸，获取它们的终止状态。通过调用wait系统调用就好啦。不幸的是，这不完美，因为如果调用wait，然而没有终止的子进程，wait就会阻塞服务器，实际上就是阻止了服务器处理新的客户端连接请求。有其他办法吗？当然有啦，其中之一就是使用信号处理器和wait系统调用组合。



以下是如何工作的。当一个子进程终止了，内核发送SIGCHLD信号。父进程可以设置一个信号处理器来异步地被通知，然后就能wait子进程获取它的终止状态，因此阻止了僵尸进程出现。



顺便说下，异步事件意味着父进程不会提前知道事件发生的时间。

修改服务器代码，设置一个SIGCHLD事件处理器，然后在事件处理器里wait终止的子进程。webserver3e.py代码如下：

Python

```
#####
# Concurrent server - webserver3e.py
#
```

```

4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X      #
5 ######
6 import os
7 import signal
8 import socket
9 import time
10
11 SERVER_ADDRESS = (HOST, PORT) = "", 8888
12 REQUEST_QUEUE_SIZE = 5
13
14 def grim_reaper(signum, frame):
15     pid, status = os.wait()
16     print(
17         'Child {pid} terminated with status {status}'
18         '\n'.format(pid=pid, status=status)
19     )
20
21 def handle_request(client_connection):
22     request = client_connection.recv(1024)
23     print(request.decode())
24     http_response = b"""
25 HTTP/1.1 200 OK
26
27 Hello, World!
28 """
29     client_connection.sendall(http_response)
30     # sleep to allow the parent to loop over to 'accept' and block there
31     time.sleep(3)
32
33 def serve_forever():
34     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
35     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
36     listen_socket.bind(SERVER_ADDRESS)
37     listen_socket.listen(REQUEST_QUEUE_SIZE)
38     print('Serving HTTP on port {}...'.format(PORT))
39
40     signal.signal(signal.SIGCHLD, grim_reaper)
41
42     while True:
43         client_connection, client_address = listen_socket.accept()
44         pid = os.fork()
45         if pid == 0: # child
46             listen_socket.close() # close child copy
47             handle_request(client_connection)
48             client_connection.close()
49             os._exit(0)
50         else: # parent
51             client_connection.close()
52
53 if __name__ == '__main__':
54     serve_forever()

```

启动服务器：

Python

```
$ python webserver3e.py
1 $ python webserver3e.py
```

使用老朋友curl给修改后的并发服务器发送请求：

Python

```
$ curl
http://localhost:8888/hello
1 $ curl http://localhost:8888/hello
```

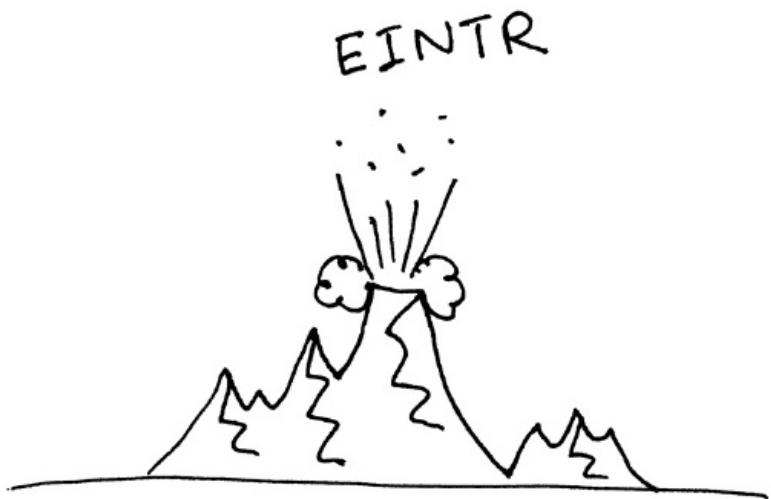
观察服务器：

```
Serving HTTP on port 8888 ...
GET /hello HTTP/1.1
User-Agent: curl/7.35.0
Host: localhost:8888
Accept: */*

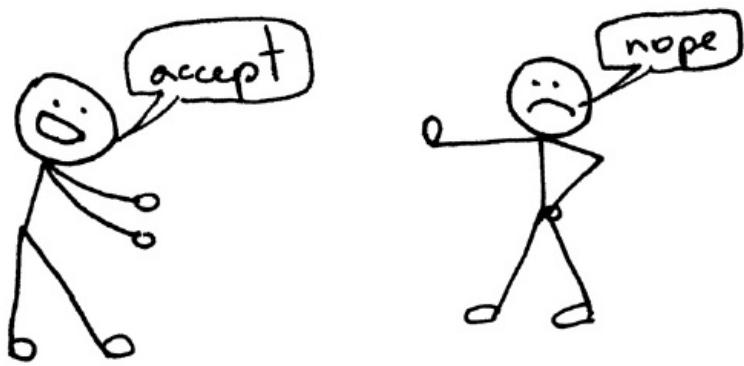
Child 9951 terminated with status 0

Traceback (most recent call last):
  File "webserver3e.py", line 62, in <module>
    serve_forever()
  File "webserver3e.py", line 51, in serve_forever
    client_connection, client_address = listen_socket.accept()
  File "/usr/lib/python2.7/socket.py", line 202, in accept
    sock, addr = self._sock.accept()
socket.error: [Errno 4] Interrupted system call
```

刚才发生了什么? accept调用失败了, 错误是EINTR。



当子进程退出, 引发SIGCHLD事件时, 父进程阻塞在accept调用, 这激活了事件处理器, 然后当事件处理器完成时, accept系统调用就中断了:



别着急, 这个问题很好解决。你要做的就是重新调用accept。以下是修改后的代码:

Python

```
#####
#| | |
#####
```

1 #####

```

2 # Concurrent server - webserver3f.py          #
3 #                                         #
4 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X    #
5 ######
6 import errno
7 import os
8 import signal
9 import socket
10
11 SERVER_ADDRESS = (HOST, PORT) = "", 8888
12 REQUEST_QUEUE_SIZE = 1024
13
14 def grim_reaper(signum, frame):
15     pid, status = os.wait()
16
17 def handle_request(client_connection):
18     request = client_connection.recv(1024)
19     print(request.decode())
20     http_response = b"""
21 HTTP/1.1 200 OK
22
23 Hello, World!
24 """
25     client_connection.sendall(http_response)
26
27 def serve_forever():
28     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
29     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
30     listen_socket.bind(SERVER_ADDRESS)
31     listen_socket.listen(REQUEST_QUEUE_SIZE)
32     print('Serving HTTP on port {}'.format(PORT))
33
34     signal.signal(signal.SIGCHLD, grim_reaper)
35
36     while True:
37         try:
38             client_connection, client_address = listen_socket.accept()
39         except IOError as e:
40             code, msg = e.args
41             # restart 'accept' if it was interrupted
42             if code == errno.EINTR:
43                 continue
44             else:
45                 raise
46
47         pid = os.fork()
48         if pid == 0: # child
49             listen_socket.close() # close child copy
50             handle_request(client_connection)
51             client_connection.close()
52             os._exit(0)
53         else: # parent
54             client_connection.close() # close parent copy and loop over
55
56 if __name__ == '__main__':
57     serve_forever()

```

启动修改后的webserver3f.py：

Python

```
$ python webserver3f.py
1 $ python webserver3f.py
```

使用curl给修改后的服务器发送请求：

Python

```
$ curl
http://localhost:8888/hello
1 $ curl http://localhost:8888/hello
```

看到了吗？没有EINTR异常啦。现在，验证一下吧，没有僵尸了，带wait的SIGCHLD事件处理器也能处理好子进程了。怎么验证呢？只要运行ps命令，看看没有Z+状态的进程（没有进程）。太棒啦！没有僵尸在四周跳的感觉真安全呢！



- 如果fork了子进程并不wait它，它就成僵尸了。
- 使用SIGCHLD事件处理器来异步的wait终止了的子进程来获取它的终止状态
- 使用事件处理器时，你要明白，系统调用会被中断的，你要做好准备对付这种情况

嗯，目前为止，一次都好。没有问题，对吧？好吧，几乎滑。再次跑下webserver3f.py，这次不用curl请求一次了，改用client3.py来创建128个并发连接：

Python

```
$ python client3.py --max-clients 128  
1 $ python client3.py --max-clients 128
```

现在再运行ps命令

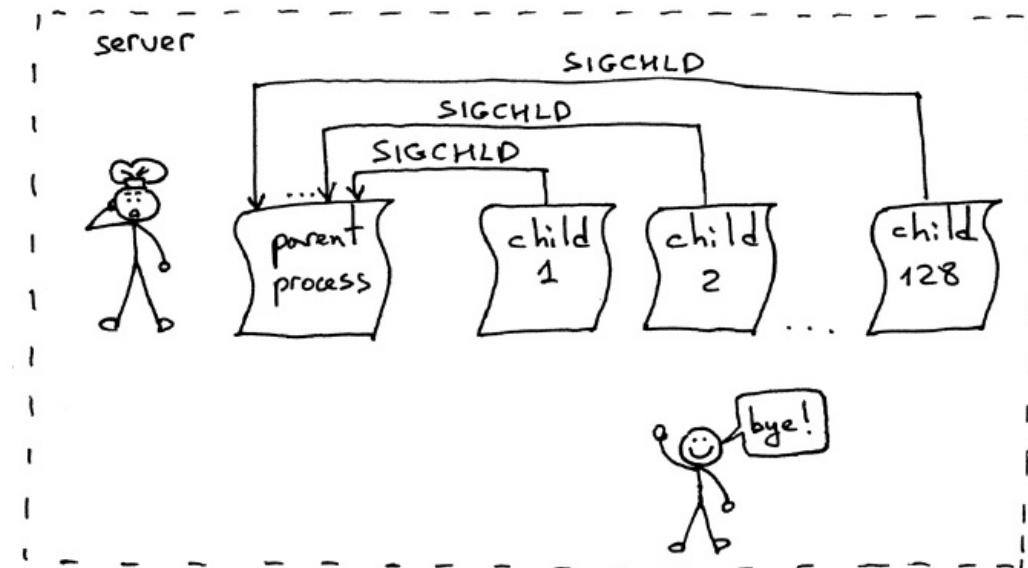
Python

```
$ ps auxw | grep -i python |  
grep -v grep  
1 $ ps auxw | grep -i python | grep -v grep
```

看到了吧，少年，僵尸又回来了！



这次又出什么错了呢？当你运行128个并发客户端时，建立了128个连接，子进程处理了请求然后几乎同时终止了，这就引发了SIGCHLD信号洪水般的发给父进程。问题在于，信号没有排队，父进程错过了一些信号，导致了一些僵尸到处跑没人管：



解决方案就是设置一个SIGCHLD事件处理器，但不用wait了，改用waitpid系统调用，带上WNOHANG参数，循环处理，确保所有的终止的子进程都被处理掉。以下是修改后的webserver3g.py：

Python

```
#####
1 ##### Concurrent server - webserver3g.py #####
2 # Tested with Python 2.7.9 & Python 3.4 on Ubuntu 14.04 & Mac OS X      #
3 #
4 ##### import errno
5 ##### import os
6 ##### import signal
7 ##### import socket
10
11 SERVER_ADDRESS = (HOST, PORT) = "", 8888
12 REQUEST_QUEUE_SIZE = 1024
13
14 def grim_reaper(signum, frame):
15     while True:
16         try:
17             pid, status = os.waitpid(
18                 -1,          # Wait for any child process
19                 os.WNOHANG # Do not block and return EWOULDBLOCK error
20             )
21         except OSError:
22             return
23
24         if pid == 0: # no more zombies
25             return
26
27 def handle_request(client_connection):
28     request = client_connection.recv(1024)
29     print(request.decode())
30     http_response = b"""
31 HTTP/1.1 200 OK
32
33 Hello, World!
34 """
35     client_connection.sendall(http_response)
36
37 def serve_forever():
38     listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
39     listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
40     listen_socket.bind(SERVER_ADDRESS)
41     listen_socket.listen(REQUEST_QUEUE_SIZE)
42     print('Serving HTTP on port {port} ...'.format(port=PORT))
43
44     signal.signal(signal.SIGCHLD, grim_reaper)
45
46     while True:
47         try:
48             client_connection, client_address = listen_socket.accept()
49         except IOError as e:
50             code, msg = e.args
51             # restart 'accept' if it was interrupted
52             if code == errno.EINTR:
53                 continue
54             else:
55                 raise
56
57         pid = os.fork()
58         if pid == 0: # child
59             listen_socket.close() # close child copy
60             handle_request(client_connection)
61             client_connection.close()
62             os._exit(0)
63         else: # parent
64             client_connection.close() # close parent copy and loop over
65
66 if __name__ == '__main__':
67     serve_forever()
```

启动服务器：

Python

```
$ python webserver3g.py
```

```
1 $ python webserver3g.py
```

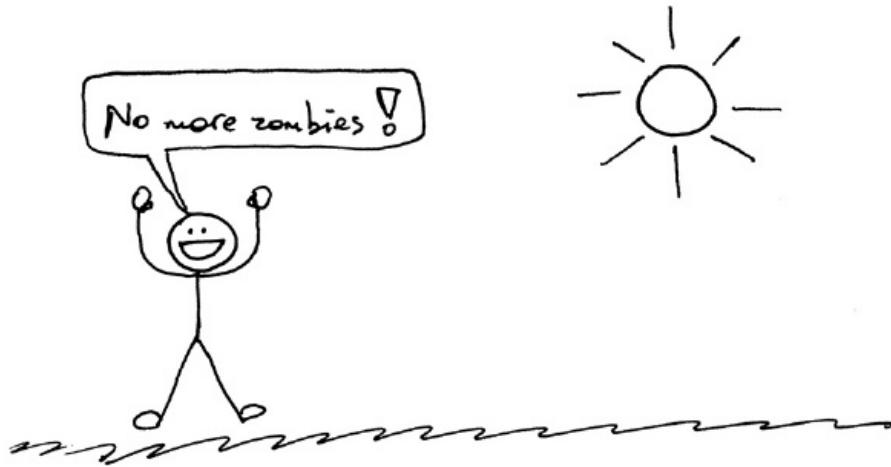
使用测试客户端client3.py：

Python

```
$ python client3.py --max-clients 128
```

```
1 $ python client3.py --max-clients 128
```

现在验证一下没有僵尸了吧。哈！没有僵尸的日子真好！



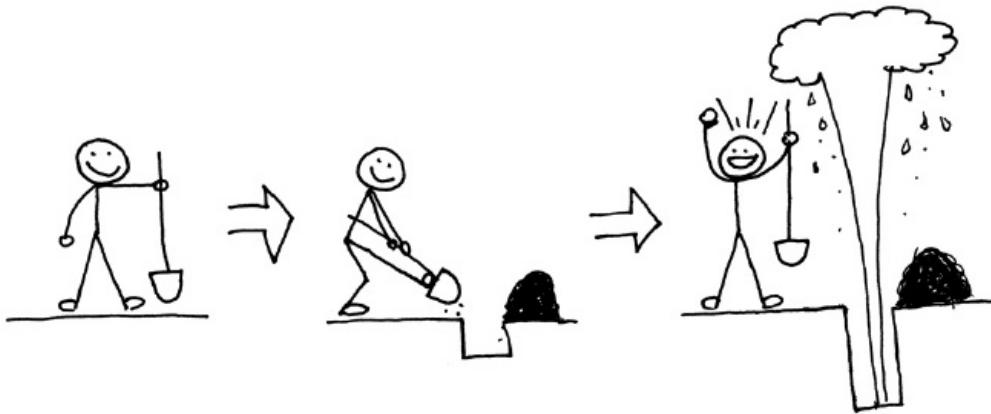
恭喜！这真是段很长的旅程啊，希望你喜欢。现在你已经拥有了自己的简单并发服务器，而且这个代码有助于你在将来的工作中开发一个产品级的Web服务器。

我要把它留作练习，你来修改第二部分的WSGI服务器，让它达到并发。你在这里可以找到修改后的版本。但是你要自己实现后再看我的代码哟。你已经拥有了所有必要的信息，所以，去实现它吧！

接下来做什么呢？就像Josh Billings说的那样，

像邮票那样——用心做一件事，直到完成。

去打好基础吧。质疑你已经知道的，保持深入研究。



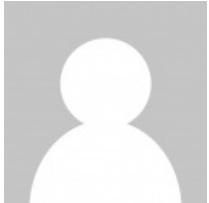
如果你只学方法，你就依赖方法。但如果你学会原理，你可以发明自己的方法。——爱默生

以下是我挑出来对本文最重要的几本书。它们会帮你拓宽加深我提到的知识。我强烈建议你想言设法弄到这些书：从朋友那借也好，从本地图书馆借，或者从亚马逊买也行。它们是守护者：

1. Unix网络编程，卷1：socket网络API（第三版）
2. UNIX环境高级编程，第三版
3. Linux编程接口：Linux和UNIX系统编辑手册
4. TCP/IP详解，卷1：协议（第二版）
5. The Little Book of SEMAPHORES (2nd Edition): The Ins and Outs of Concurrency Control and Common Mistakes. Also available for free on the author's site [here](#).

1 赞 23 收藏 8 [评论](#)

关于作者：[高世界](#)



我翻译得越多，发现知道的越少，我就要更多地翻译。论得的地的正确用法。我是php开发者，对python, c/c++, linux感兴趣。[个人主页](#) · [我的文章](#) · [17](#)

Python解释器简介 (5) : 深入主循环

本文由 [伯乐在线 - v7](#) 翻译, [黄利民](#) 校稿。未经许可, 禁止转载!
英文出处: [akaptur](#)。欢迎加入[翻译组](#)。

- [Python解释器简介 \(1\) : 函数对象](#)
- [Python解释器简介 \(2\) : 代码对象](#)
- [Python解释器简介 \(3\) : 理解字节码](#)
- [Python解释器简介 \(4\) : 动态语言](#)

本文将会带领大家了解 CPython 3.3 中的 Python 解释器。我们首先一起来看 Python 解释器的一个简短的高层概述, 然后对解释器实现过程中的一些有意思的代码块进行深入的探讨。我已经把这里探讨的函数名和文件名囊括进来了, 你可以在源码中找到它们自行阅读深究。

概述

我们从 Python 虚拟机 (又叫 Python 解释器) 的一个高层概述开始。

Python 虚拟机有一个栈帧的调用栈。一个栈帧包含了给出代码块的信息和上下文, 其中包括最后执行的字节码指令、全局和局部的命名空间、异常状态和调用栈帧的引用。每个栈帧有两个与其相关联的栈: block 栈和数据栈, 其中 block 栈在一些控制流 (比如异常处理) 中使用。Python 虚拟机的主要工作就是操作这三个类型的栈。

具体一些, 我们假设有下面这样一段代码, 解释器执行到被标记的行。下面便是当前情况下调用栈、block 栈以及数据栈的情况。

main.py

Python

```
def foo():
    x = 1

1 def foo():
2     x = 1
3     def bar(y):
4         z = y + 2 # <--- (3) ... and the interpreter is here.
5         return z
6     return bar(x) # <--- (2) ... which is returning a call to bar ...
7 foo()           # <--- (1) We're in the middle of a call to foo ...
8
9 main.py
```

Python

```
c -----
a | bar Frame
      |
      |-----+
      |-----+
      |-----+
      |-----+
      |-----+-----+
      |-----+-----+-----+
      |-----+-----+-----+-----+
```

```
1 c -----
2 a | bar Frame          | -> block stack: []
3   | | (newest)          | -> data stack: [1, 2]
4   | -----+
5   |-----+-----+-----+-----+
5   |-----+-----+-----+-----+-----+
```

```
6 s |           | -> data stack: [<Function foo.<locals>.bar at 0x10d389680>, 1]
7 t -----
8 a | main (module) Frame   | -> block stack: []
9 c | (oldest)          | -> data stack: [<Function foo at 0x10d3540e0>]
10 k -----
```

在这一时刻，解释器在嵌套函数的中间位置调用 bar 函数。此时在调用栈中有三个栈帧：模块层级的栈帧、foo 函数的栈帧以及 bar 函数的栈帧。当 bar 函数完成动作返回，调用栈中与 bar 函数关联的栈帧将会弹栈。通常每一个模块都会有一个与其对应的拥有新作用域的栈帧，函数调用和类定义也是如此。注意，每一次函数调用都会创建一个栈帧，在递归函数中，每一层的递归调用都会拥有自己的一个栈帧。

每一个栈帧都有自己的数据栈和 block 栈。独立的数据栈和 block 栈使解释器可以中断或恢复栈帧，这与生成器相似。

这里的情况示意很清楚了，我们深入到代码内部看一下。

堆栈结构对象 frameobj.c 创建一个 ceval.c 文件中定义的 PyEval_EvalCodeEx 栈帧。这个栈帧在 ceval.c 文件中执行 PyEval_EvalFrameEx 栈帧。

栈帧都从哪儿来？

ceval.c 文件中的 PyEval_EvalCodeEx 函数创建了新的栈帧。我们在下面摘录了执行 code 对象的 PyEval_EvalCodeEx 函数。这个函数首先创建了一个新的栈帧，之后解析命令行参数（如果有的话）。倘若 code 对象是生成器，那么函数返回新的生成器；否则，栈帧将会运行直到返回，而返回值将被传递到上层。

ceval.c

C

```
PyObject *
PyEval_EvalCodeEx(PyObject *_co, PyObject *globals, PyObject *locals,
                  PyObject **args, int argcount, PyObject **kws, int kwcount,
                  PyObject **defs, int defcount, PyObject *kwdefs, PyObject *closure)
{
    PyCodeObject* co = (PyCodeObject*)_co;
    PyFrameObject *f;
    PyObject *retval = NULL;
    PyObject **fastlocals, **freevars;
    PyThreadState *tstate = PyThreadState_GET();
    /* [snip error-checking] */
    f = PyFrame_New(tstate, co, globals, locals); /* <----- new frame */
    if (f == NULL)
        return NULL;
    fastlocals = f->f_localsplus;
    freevars = f->f_localsplus + co->co_nlocals;
    /* [snip 150 lines of argument parsing] */
```

```

23 if (co->co_flags & CO_GENERATOR) { /* <---- if the "it's a generator" flag is set */
24     /* [snip] */
25
26     /* Create a new generator that owns the ready to run frame
27      * and return that as the value. */
28     return PyGen_New(f); /* <---- new generator object */
29 }
30
31 retval = PyEval_EvalFrameEx(f,0); /* <---- otherwise, run the frame */
32
33 fail: /* Jump here from prelude on failure */
34     /* [snip some cleanup] */
35     return retval;           /* <---- return the value from running the frame */
36 }

```

大部分情况下是 C 函数 `function_call` 在调用 `PyEval_EvalCodeEx`。所有 Python 对象被调用的时候都会调用 `function_call`。每当一个可调用的 Python 对象被调用，都会写入一个 `code` 对象用来创建栈帧。

`funcobject.c`

C

```

static PyObject *
function_call(PyObject *func)
{
    static PyObject *result;
    PyObject *argdefs;
    PyObject *kwtuple = NULL;
    PyObject **d, **k;
    Py_ssize_t nk, nd;
    /* [snip 30 lines of argument parsing] */

    result = PyEval_EvalCodeEx(
        PyFunction_GET_CODE(func),
        PyFunction_GET_GLOBALS(func), (PyObject *)NULL,
        &PyTuple_GET_ITEM(arg, 0), PyTuple_GET_SIZE(arg),
        k, nk, d, nd,
        PyFunction_GET_KW_DEFAULTS(func),
        PyFunction_GET_CLOSURE(func));
}

return result;
}

```

上面说大部分情况下是 `function_call` 在调用 `PyEval_EvalCodeEx`。而 `PyEval_EvalCodeEx` 也可以被 entry point 调用，比如 `pythonrun.c` 中的 `run_pyc_file` 以及 `import.c` 中的 `exec_code_in_module`。这些函数很相似，区别只是在于它们取得 `code` 对象的方式（通过编译还是通过读文件）和运行的环境（比如命名空间不同）。

我们回到 `PyEval_EvalFrameEx`。这个函数大约有2400行代码，占了 `ceval.c` 文件的大部分；其中1500行代码是一个庞大的 switch 声明。我的那篇《[1,500 line switch statement powering your Python](#)》提到的就是它。`PyEval_EvalFrameEx` 占用了一个单独的栈帧，并且会运行直到它返回。

在本系列文章的[第3篇](#)我们介绍了字节码。对解释器来讲，字节码是一序列字节指令。我们回到本篇

开头的例子，下面列出了这段代码和函数中 code 对象的详细拆解。

Python

```
PY3 >>> def foo():
PY3 ...     x = 1
1 PY3 >>> def foo():
2 PY3 ...     x = 1
3 PY3 ...     def bar(y):
4 PY3 ...         z = y + 2
5 PY3 ...         return z
6 PY3 ...     return bar(x)
7 PY3 ...
8 PY3 >>> dis.dis(foo)
9  2      0 LOAD_CONST           1 (1)
10   3 STORE_FAST             0 (x)
11
12  3      6 LOAD_CONST           2 (<code object bar at 0x107b548a0, file "<stdin>", line 3>)
13      9 LOAD_CONST           3 ('foo.<locals>.bar')
14      12 MAKE_FUNCTION        0
15      15 STORE_FAST            1 (bar)
16
17  6      18 LOAD_FAST             1 (bar)
18      21 LOAD_FAST             0 (x)
19      24 CALL_FUNCTION         1 (1 positional, 0 keyword pair)
20      27 RETURN_VALUE
21
22 PY3 >>> bar_code_obj = foo.__code__.co_consts[2]
23 PY3 >>> dis.dis(bar_code_obj)
24  4      0 LOAD_FAST             0 (y)
25      3 LOAD_CONST           1 (2)
26      6 BINARY_ADD
27      7 STORE_FAST            1 (z)
28
29  5      10 LOAD_FAST             1 (z)
30      13 RETURN_VALUE
```

PyEval_EvalFrameEx 从字节码中的第一个字节开始执行，在本例中，从 foo 函数字节码中的 LOAD_CONST 字节开始，并且去找那个庞大 switch 中对应的 case。当执行完 switch 中找到的操作指令，栈帧将移动到下一个相关操作码继续整个程序。在一些地方，操作码会中止循环通过 goto 跳出switch，通过下面 RETURN_VALUE 示例。

为使你更好地理解它如何工作，我在下面列出了 PyEval_EvalFrameEx 函数中的一段摘录。不过像往常一样，我更希望你去阅读 CPython 的完整代码。

ceval.c

C

```
/* Interpreter main loop */
*/
1 /* Interpreter main loop */
2
3 PyObject *
4 PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
5 {
6     PyObject **stack_pointer; /* Next free slot in value stack */
```

```

7  unsigned char *next_instr;
8  int opcode;      /* Current opcode */
9  int oparg;       /* Current opcode argument, if any */
10 enum why_code why; /* Reason for block stack unwind */
11 PyObject **fastlocals, **freevars;
12 PyObject *retval = NULL;      /* Return value */
13 PyThreadState *tstate = PyThreadState_GET();
14 PyCodeObject *co;
15
16 unsigned char *first_instr;
17 PyObject *names;
18 PyObject *consts;
19
20 #define TARGET(op)
21     case op:
22 #define DISPATCH() continue
23 #define FAST_DISPATCH() goto fast_next_opcode
24
25 #define NEXTOP()    (*next_instr++)
26
27 /* Stack manipulation macros */
28 #define STACK_LEVEL() ((int)(stack_pointer - f->f_valuestack))
29 #define TOP()        (stack_pointer[-1])
30 #define PUSH(v)     (*stack_pointer++ = (v))
31 #define POP()       (*--stack_pointer)
32
33 /* Start of code */
34
35 /* push frame */
36 tstate->frame = f;
37
38 co = f->f_code;
39 names = co->co_names;
40 consts = co->co_consts;
41 fastlocals = f->f_localsplus;
42 freevars = f->f_localsplus + co->co_nlocals;
43 first_instr = (unsigned char*) PyBytes_AS_STRING(co->co_code);
44 next_instr = first_instr + f->f_lasti + 1;
45 stack_pointer = f->f_stacktop;
46 f->f_stacktop = NULL; /* remains NULL unless yield suspends frame */
47 f->f_executing = 1;
48
49 why = WHY_NOT;
50
51 for (;;) {
52
53     fast_next_opcode:
54         f->f_lasti = INSTR_OFFSET();
55
56     /* Extract opcode and argument */
57
58     opcode = NEXTOP();
59     oparg = 0;
60     if (HAS_ARG(opcode))
61         oparg = NEXTARG();
62
63     /* Main switch on opcode */
64
65     switch (opcode) {
66         /* [snip 1,500 lines of switch, about 100 cases] */
67         /* [three cases shown below] */
68
69     TARGET(BINARY_ADD) {

```

```

70     PyObject *right = POP();
71     PyObject *left = TOP();
72     PyObject *sum;
73     sum = PyNumber_Add(left, right);
74     Py_DECREF(left);
75     Py_DECREF(right);
76     SET_TOP(sum);
77     if (sum == NULL)
78         goto error;
79     DISPATCH();
80 }
81
82 TARGET(LOAD_CONST) {
83     PyObject *value = GETITEM(consts, oparg);
84     Py_INCREF(value);
85     PUSH(value);
86     FAST_DISPATCH();
87 }
88
89 TARGET(RETURN_VALUE) {
90     retval = POP();
91     why = WHY_RETURN;
92     goto fast_block_end;
93 }
94
95 /* end switch */
96
97 /* complex block cleanup code here */
98
99 } /* main loop */
100
101 /* pop frame */
102 exit_eval_frame:
103     f->f_executing = 0;
104     tstate->frame = f->f_back;
105
106     return retval;
107 }
```

[在第三篇中](#)，我们看到 BINARY_ADD 没有参数，从上面的 dis 的第四行输出来看并没有命令行参数。这有点奇怪，我们原本希望看到一个带有两个参数的二进制函数。现在通过看解释器的情况，我们便知道发生了什么：这两个参数在栈帧的数据栈栈顶。下面我给出了 bar 函数执行时候数据栈的情况。

Python

data: [] <-- the data stack starts out empty	
--	--

```

1 data: [] <-- the data stack starts out empty
2 4 0 LOAD_FAST 0 (y)
3   data: [1] <-- y has the value 1 when bar is called
4 3 LOAD_CONST      1 (2)
5   data: [1, 2]
6 6 BINARY_ADD
7   data: [3] <-- BINARY_ADD adds the top two things on the stack
8     and pushes the answer onto the stack

```

打赏支持我翻译更多好文章，谢谢！

[打赏译者](#)

打赏支持我翻译更多好文章，谢谢！

任选一种支付方式

WeChat Scan Payment



Transfer to _v7_ [**龙]

伯乐在线-酒钱

¥1.28

支付宝
ALIPAY



v7

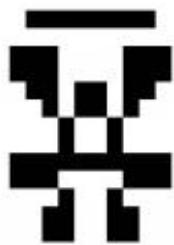
¥1.28

赞赏你发在伯乐在线的文章

用支付宝扫一扫付款

1 赞 6 收藏 [1 评论](#)

关于作者： [v7](#)



微博：[@_v7_](#) [个人主页](#) · [我的文章](#) · [17](#) ·

与 Python 无缝集成：基本特殊方法 1

原文出处：[young_ipython \(@小乌龟coolboy 译\)](#)

有许多特殊方法允许类与Python紧密结合，标准库参考将其称之为基本，基础或本质可能是更好的术语。这些特殊方法构成了创建与其他Python特性无缝集成的类的基础。

例如，对于给定对象的值，我们需要字符串表示。基类、对象都有默认的`__repr__()`和`__str__()`用于提供对象的字符串表示。遗憾的是，这些默认表示不提供信息。我们总是想要覆盖这些默认定义中的一个或两个。我们可以看看`__format__()`，这个更复杂但目的和前面是一样的。

我们也可以看看其他转换，特别是`__hash__()`、`__bool__()`、和`__bytes__()`。这些方法将一个对象转换成数字、true/false值或字符串的字节。例如，当我们实现`__bool__()`时，可以在if语句中使用对象，如下：if `someobject`:

然后，我们可以看看实现比较操作符的特殊方法`__lt__()`、`__le__()`、`__eq__()`、`__ne__()`、`__gt__()`和`__ge__()`。

这些基本的特殊方法在类中定义中几乎总是需要的。

最后我们来看看`__new__()`和`__del__()`，因为这些方法的用例相当复杂。每当我们需要其他基本特殊方法时，是不需要这些的。

我们会详细看看如何添加这些特殊方法来扩大一个简单的类定义。我们需要看一下两个从对象继承的默认行为，这样我们可以了解需要什么样的覆盖以及何时真正需要。

`__repr__()` 和 `__str__()` 方法

对于一个对象，Python有两种字符串表示方法。这些都和内置函数`__repr__()`、`__str__()`、`__print__()`以及`string.format()`方法紧密结合。

- `str()`方法表示的对象通常是适用于人理解的，由对象的`__str__()`方法创建。
- `repr()`方法表示的对象通常是适用于解释器理解的，可能是完整的Python表达式来重建对象。文档中是这样说的：对于许多类型，这个函数试图返回一个字符串，将该字符串传递给`eval()`会重新生成对象。这是由对象的`__repr__()`方法创建的。
- `print()`函数会使用`str()`准备对象用于打印。
- 字符串的`format()`方法也可以访问这些方法。当我们使用`{!r}`或`{!s}`格式，我们分别需要`__repr__()`或`__str__()`。

首先让我们看看默认实现。

下面是一个简单的类层次结构：

```
class Card:  
    insure = False  
  
1 class Card:  
2 insure = False  
3 def __init__(self, rank, suit):  
4     self.suit = suit  
5     self.rank = rank  
6     self.hard, self.soft = self._points()  
7  
8 class NumberCard(Card):  
9     def _points(self):  
10        return int(self.rank), int(self.rank)
```

我们已经定义了带有四个属性的两个简单类。

以下是一个与其中一个类对象的交互：

Python

```
>>> x=NumberCard( '2',  
1 |  
1 >>> x=NumberCard( '2', '♣')  
2 >>> str(x)  
3 '<__main__.NumberCard object at 0x1013ea610>'  
4 >>> repr(x)  
5 '<__main__.NumberCard object at 0x1013ea610>'  
6 >>> print(x)  
7 <__main__.NumberCard object at 0x1013ea610>
```

从这个输出知道默认的`__str__()`和`__repr__()`实现不是很丰富。

当我们需要覆盖`__str__()`和`__repr__()`时我们考虑下面两个广泛的设计用例：

- 非集合对象：一个不包含其他对象集合的“简单”对象，通常不涉及非常复杂格式的集合。
- 集合对象：一个包含一组更复杂格式的对象。

1. 非集合`__str__()`和`__repr__()`

正如我们之前看到的，`__str__()`和`__repr__()`的输出不是很丰富。我们几乎总是需要覆盖它们。以下是当没有包含集合的时候覆盖`__str__()`和`__repr__()`的方法。这些方法从属于之前定义的Card类：

Python

```
def __repr__(self):  
    return "  
  
1 def __repr__(self):  
2     return "{__class__.__name__}(suit={suit!r}, rank={rank!r})".format(  
3         __class__=self.__class__, **self.__dict__)  
4 def __str__(self):  
5     return "{rank}{suit}".format(**self.__dict__)
```

这两个方法依赖于传递内部对象的实例变量`__dict__()`给`format()`函数。对于对象使用`__slots__`是不适合的；通常，这些是不可变的对象。在格式说明符中使用名称会使得格式化更显式。当然也使得格式模板更长了。在`__repr__()`中，我们传递内部`__dict__`加上对象的`__class__`作为关键字参数值给

`format()`函数。

模板字符串使用两种格式说明符：

- `{__class__.__name__}`模板也可以写成`{__class__.__name__!s}`从而当只提供简单字符串的类名时变得更显式。
- `{suit!r}`和`{rank!r}`模板都使用`!r`格式说明符生成属性值的`repr()`方法。

在`__str__()`中，我们只有传递内部`__dict__`对象。格式化使用隐式的`{!s}`格式说明符来生成属性值的`str()`方法。

2. 集合`__str__()`和`__repr__()`

当包含一个集合时，我们需要格式化集合中的每个项目以及整个容器。以下是带有`__str__()`和`__repr__()`方法的简单集合：

Python

```
class Hand:  
    def __init__(self,  
1     class Hand:  
2     def __init__(self, dealer_card, *cards):  
3     self.dealer_card = dealer_card  
4     self.cards = list(cards)  
5     def __str__(self):  
6     return ", ".join(map(str, self.cards))  
7     def __repr__(self):  
8     return "{__class__.__name__}({dealer_card!r}, {__dict__})".format(  
9     __class__=self.__class__, __dict__=", ".join(  
10    map(repr, self.cards)), **self.__dict__)
```

`__str__()`方法是一个简单的设计，如下：

- 映射`str()`到集合中的每一项。这将在生成的每个字符串值上创建一个迭代器。
- 使用`”, “.join()`合并所有项的字符串到一个长字符串中。

`__repr__()`方法是一个多部分的设计，如下：

- 映射`repr()`到集合中的每一项。这将在生成的每个字符串值上创建一个迭代器。
- 使用`”, “.join()`合并所有项的字符串。
- 创建一组带有`__class__`的关键字、集合字符串和来自`__dict__`的各种属性。我们已经命名集合字符串为`_card_str`，与现有的属性不冲突。
- 使用`”{__class__.__name__}({dealer_card!r}, {__dict__})”`.format()将类名与项目值的长字符串结合。我们使用`!r`进行格式化以确保属性也使用了`repr()`转换。

在某些情况下，这可以使一些事情变得稍微简单些。位置参数的格式化可以一定程度上缩短模板字符串的长度。

`__format__()` 方法

和内置函数format()一样，string.format()使用__format__()方法。这两个接口都是用来从给定对象得到像样的字符串的方式。

以下是将参数提供给__format__()的两种方法：

- someobject.__format__(“”): 发生在应用程序使用format(someobject)或等价的”{0}”.format(someobject)的时候。在这些情况下，提供了长度为零的字符串说明符。这会产生一个默认格式。
- someobject.__format__(specification): 发生在应用程序使用format(someobject, specification)或等价的”{0:specification}”.format(someobject)的时候。

请注意，类似于”{0!r}”.format()或”{0!s}”.format() 的方法没有使用__format__()方法。它们直接使用了__repr__()或__str__()。

带有””说明符的合理响应是返回str(self)。它对各种对象的字符串提供了显式的一致性表示。

格式说明符必须都是文本，且在格式字符串的”.”之后。当我们写”{0:06.4f}”，06.4f是适用于第0项参数列表的格式说明符。

Python标准库文档中的6.1.3.1节定义了一个复杂的数值说明符作为九个部分字符串，这是格式说明符的一种迷你语言。有如下语法：

Python

```
[[fill][align][sign][#[0][width]
[.][precision][type]]]
```

1 [[fill][align][sign][#[0][width][.][precision][type]]]

我们可以用正则表达式解析这些标准说明符，如下面代码片段所示：

Python

```
re.compile(
r"(?P<fill_align>.\?[\
1 re.compile(
2 r"(?P<fill_align>.\?[\<\>=\^\"])?"
3 "(?P<sign>[-+ ])?"
4 "(?P<alt>#)?"
5 "(?P<padding>0)?"
6 "(?P<width>\d*)"
7 "(?P<comma>,\.)?"
8 "(?P<precision>.\d*)?"
9 "(?P<type>[bcdeEfFgGnosxX%])?" )
```

这个正则将说明符拆分成八组。第一组和原说明符一样有fill和alignment字段。我们可以使用这些得出我们已定义类的格式化数值数据。

然而，Python的格式说明符迷你语言可能不适用于我们的类定义。因此，我们需要定义我们自己的说明符迷你语言并在类的__format__方法中执行。如果我们定义数值类型，我们应该坚持预定义的迷你语言。然而，对于其他类型则没有理由再坚持预定义的语言。

作为一个示例，这里有个微不足道的语言，使用字符%r和%s分别给我们展示牌值和花色。在结果字符串中%%字符变成%。所有其他字符是重复的。

我们可以通过格式化扩展我们的Card类，如下面代码片段所示：

Python

```
def __format__(self, format_spec):  
    if format_spec == "":  
        return str(self)  
    rs = format_spec.replace("%r", self.rank).replace("%s", self.suit)  
    rs = rs.replace("%%", "%")  
    return rs
```

这个定义会检查格式说明符。如果没有说明符，则使用str()函数。如果提供了一个说明符，会合拢牌值、花色和任何%字符格式说明符，将其转化为输出字符串。

这允许我们像下面这样格式化Card：

Python

```
print( "Dealer Has {0:%r of %s}" .format(hand.dealer_card))
```

格式说明符(" %r of %s")被作为format的参数传递给我们的__format__()方法。使用这个，我们能够提供一个一致的接口来表示我们已经定义的类的对象。

或者，我们可以定义如下：

Python

```
default_format = "some specification"  
1 default_format = "some specification"  
2 def __str__(self):  
3     return self.__format__(self.default_format)  
4 def __format__(self, format_spec):  
5     if format_spec == "":  
6         format_spec = self.default_format  
7     # process the format specification.
```

这个的优势在于把所有字符串放置到__format__()方法，而不是分开到的__format__()和__str__()。劣势在于，我们不总是需要实现__format__(), 但我们几乎总是需要实现__str__()。

1. 嵌套格式化说明符

string.format()方法可以处理嵌套的{}实例来执行简单的关键字置换到格式说明符中。这个置换完成，会创建最终格式字符串并传递给类的__format__()方法。这种嵌套置换通过参数化通用说明符简化了某些相对复杂的数值格式。

下面的例子，我们可以在format参数中很容易的修改width：

Python

```
width = 6
for hand, count in
1 width = 6
2 for hand, count in statistics.items():
3 print( "{hand} {count:{width}d}".format(hand=hand, count=count, width=width))
```

我们定义了一个通用的格式，”{hand:%r%s } {count:{width}d}”，这需要一个width参数让它变成适用的格式说明符。

为format()方法提供width=参数的值被用于替代{width}嵌套说明符。一旦被替换，最终格式会作为一个整体提供给__format__()方法。

2. 集合与委托格式说明符

格式化一个包括集合的复杂对象，有两个格式化问题：如何格式化整体对象以及如何格式化集合中的项目。当我们看到Hand，例如，我们看到我们有单独的Card类集合。我们需要Hand委托格式化细节给单独的Card实例。

下面是一个适用于Hand的__format__()方法：

Python

```
def __format__(self,
format_specification):
1 def __format__(self, format_specification):
2 if format_specification == "":
3 return str(self)
4 return ", ".join("{0:{fs}}".format(c, fs=format_specification) for c in self.cards)
```

format_specification参数将用于每个Hand集合里面的Card实例。格式说明符”{0:{fs}}”使用嵌套格式说明符技术将format_specification字符串置入到应用于Card实例的创建。使用这种方法我们可以格式化Hand对象、player_hand，如下所示：

Python

```
"Player:
{hand:%r%s}".format(ha
1 "Player: {hand:%r%s}".format(hand=player_hand)
```

这将应用%r%s格式说明符到Hand对象中的Card实例。

1 赞 收藏 评论

Python模块学习： re 正则表达式

原文地址：[DarkBull](#)

今天学习了Python中有关正则表达式的知识。关于正则表达式的语法，不作过多解释，网上有许多学习的资料。这里主要介绍Python中常用的正则表达式处理函数。

re.match

re.match 尝试从字符串的开始匹配一个模式，如：下面的例子匹配第一个单词。

Python

```
import re
1 import re
2
3 text = "JGood is a handsome boy, he is cool, clever, and so on..."
4 m = re.match(r"/w+/s", text)
5 if m:
6     print m.group(0), '/n', m.group(1)
7 else:
8     print 'not match'
```

re.match的函数原型为：re.match(pattern, string, flags)

第一个参数是正则表达式，这里为”(/w+)/s”，如果匹配成功，则返回一个Match，否则返回一个None；

第二个参数表示要匹配的字符串；

第三个参数是标致位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

re.search

re.search函数会在字符串内查找模式匹配,只到找到第一个匹配然后返回，如果字符串没有匹配，则返回None。

Python

```
import re
1 import re
2
3 text = "JGood is a handsome boy, he is cool, clever, and so on..."
4 m = re.search(r'/shan(ds)ome/s', text)
5 if m:
6     print m.group(0), m.group(1)
7 else:
8     print 'not search'
```

re.search的函数原型为： re.search(pattern, string, flags)

每个参数的含意与re.match一样。

re.match与re.search的区别： re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

re.sub

re.sub用于替换字符串中的匹配项。下面一个例子将字符串中的空格‘ ‘替换成‘ ‘：

Python

```
import re  
1 import re  
2  
3 text = "JGood is a handsome boy, he is cool, clever, and so on..."  
4 print re.sub(r's+', '-', text)
```

re.sub的函数原型为： re.sub(pattern, repl, string, count)

其中第二个参数是替换后的字符串；本例中为‘ -’

第四个参数指替换个数。默认为0，表示每个匹配项都替换。

re.sub还允许使用函数对匹配项的替换进行复杂的处理。如： re.sub(r's', lambda m: '[' + m.group(0) + ']', text, 0)；将字符串中的空格‘ ‘替换为‘ [] ’。

re.split

可以使用re.split来分割字符串，如： re.split(r's+', text)；将字符串按空格分割成一个单词列表。

re.findall

re.findall可以获取字符串中所有匹配的字符串。如： re.findall(r'w*oo/w*', text)；获取字符串中，包含‘ oo ’的所有单词。

re.compile

可以把正则表达式编译成一个正则表达式对象。可以把那些经常使用的正则表达式编译成正则表达式对象，这样可以提高一定的效率。下面是一个正则表达式对象的一个例子：

Python

```
import re  
1 import re
```

```
2
3 text = "JGood is a handsome boy, he is cool, clever, and so on..."
4 regex = re.compile(r'w*oo>w*')
5 print regex.findall(text) #查找所有包含'oo'的单词
6 print regex.sub(lambda m: '[' + m.group(0) + ']', text) #将字符串中含有'oo'的单词用[]括起来。
```

更详细的内容，可以参考[Python手册](#)。

1 赞 2 收藏 [1 评论](#)

Python模块学习： hashlib hash加密

原文地址：[DarkBull](#)

不积跬步，无以至千里；不积小流，何以成江海。

每天学习一个Python模块，一个月可以了解30个模块，一年可以……

今天看了一个Python中的hashlib及hmac模块，这两个模块用来hash加密。说到加密，首先要了解加密的基本知识：hash加密，对称加密，不对称加密，数字签名等等。相关的知识可以在msdn上查阅：<http://msdn.microsoft.com/zh-cn/library/92f9ye3s.aspx>

使用Python中的hashlib来进行hash加密是非学简单的，下面是一段简单代码：

Python

```
import hashlib  
md5 = hashlib.md5() #  
  
1 import hashlib  
2 md5 = hashlib.md5() #创建一个MD5加密对象  
3 md5.update("JGood is a handsome boy") #更新要加密的数据  
4 print md5.digest() #加密后的结果（二进制）  
5 print md5.hexdigest() #加密后的结果，用十六进制字符串表示。  
6 print 'block_size:', md5.block_size  
7 print 'digest_size:', md5.digest_size
```

非常的简单，其实如果说加密一个字符串，根本不用写上面这么多代码，一条语句就可以了：

Python

```
print '-' * 25, '更简洁的语法'  
print '-' * 25  
1 print '-' * 25, '更简洁的语法', '-' * 25  
2 print hashlib.new("md5", "JGood is a handsome boy").hexdigest()
```

hashlib模块还支持其他的hash加密算法，如：sha1, SHA224等等，要了解更多的知识，查一下Python手册。

Python hash VS .NET hash

我出身是一个.NET程序员，以前也写过一些.NET下的hash加密类，在.NET中可以使用以下代码来实现MD5加密算法，其实也不是很难：

Python

```
/// <summary>  
/// 按指定加密算法,对字  
1 /// <summary>  
2 /// 按指定加密算法,对字符串进行加密  
3 /// </summary>
```

```

4 /// <param name="hashName">加密算法名称</param>
5 /// <param name="data">要加密数据</param>
6 /// <returns>加密后的数据</returns>
7 private static string Encrypt(string hashName, string data)
8 {
9     byte[] btData = System.Text.Encoding.ASCII.GetBytes(data);
10    //创建一个 HashAlgorithm派生类的实例
11    HashAlgorithm hasher = HashAlgorithm.Create(hashName);
12    //使用hash加密
13    byte[] hashedData = hasher.ComputeHash(btData);
14    StringBuilder result = new StringBuilder();
15    foreach (byte b in hashedData)
16    {
17        result.Append(b.ToString("x2")); //转换成16进制字符串保存
18    }
19    return result.ToString();
20 }

```

附上用.NET编写的常用hash加密算法代码：

Python

```

/// &lt;summary&gt;
/// HashEncryptor类：提
1 /// &lt;summary&gt;
2 /// HashEncryptor类：提供各种hash加密算法的实现
3 /// &lt;/summary&gt;
4 /// &lt;example&gt;
5 /// 使用MD5加密
6 /// string data = "JGood";
7 /// string encryptedData = HashEncryptor.MD5(data);
8 /// &lt;/example&gt;
9 /// &lt;remarks&gt;
10 /// HashEncryptor提供的加密算法都是不可逆的。
11 /// &lt;/remarks&gt;
12 public sealed class HashEncryptor
13 {
14     /// &lt;summary&gt;
15     /// 私有构造函数，该类不能被实例化
16     /// &lt;/summary&gt;
17     private HashEncryptor()
18     {
19     }
20     /// &lt;summary&gt;
21     /// 使用MD5加密算法加密字符串
22     /// &lt;/summary&gt;
23     /// <param name="data">要加密的字符串</param>
24     /// <returns>加密后的字符串</returns>
25     public static string MD5(string data)
26     {
27         return Encrypt("MD5", data);
28     }
29     /// &lt;summary&gt;
30     /// 使用SHA1加密算法加密字符串
31     /// &lt;/summary&gt;
32     /// <param name="data">要加密的字符串</param>
33     /// <returns>加密后的字符串</returns>
34     public static string SHA1(string data)
35     {
36         return Encrypt("SHA1", data);
37     }

```

```
38 /// <summary>
39 /// 使用SHA 256位加密算法加密字符串
40 /// </summary>
41 /// <param name="data">要加密的字符串</param>
42 /// <returns>加密后的字符串</returns>
43 public static string SHA256(string data)
44 {
45     return Encrypt("SHA256", data);
46 }
47 /// <summary>
48 /// 使用SHA 384位加密算法加密字符串
49 /// </summary>
50 /// <param name="data">要加密的字符串</param>
51 /// <returns>加密后的字符串</returns>
52 public static string SHA384(string data)
53 {
54     return Encrypt("SHA384", data);
55 }
56 /// <summary>
57 /// 使用SHA 512位加密算法加密字符串
58 /// </summary>
59 /// <param name="data">要加密的字符串</param>
60 /// <returns>加密后的字符串</returns>
61 public static string SHA512(string data)
62 {
63     return Encrypt("SHA512", data);
64 }
65 /// <summary>
66 /// 按指定加密算法,对字符串进行加密
67 /// </summary>
68 /// <param name="hashName">加密算法名称</param>
69 /// <param name="data">要加密数据</param>
70 /// <returns>加密后的数据</returns>
71 private static string Encrypt(string hashName, string data)
72 {
73     byte[] btData = System.Text.Encoding.ASCII.GetBytes(data);
74     //创建一个 HashAlgorithm派生类的实例
75     HashAlgorithm hasher = HashAlgorithm.Create(hashName);
76     //使用hash加密
77     byte[] hashedData = hasher.ComputeHash(btData);
78     StringBuilder result = new StringBuilder();
79     foreach (byte b in hashedData)
80     {
81         result.Append(b.ToString("x2")); //转换成16进制保存
82     }
83     return result.ToString();
84 }
85 }
```

1 赞 1 收藏 [评论](#)

Python模块学习：struct 数据格式转换

原文出处：[Darkbull](#)

Python是一门非常简洁的语言，对于数据类型的表示，不像其他语言预定义了许多类型（如：在C#中，光整型就定义了8种），它只定义了六种基本类型：字符串，整数，浮点数，元组，列表，字典。通过这六种数据类型，我们可以完成大部分工作。但当Python需要通过网络与其他的平台进行交互的时候，必须考虑到将这些数据类型与其他平台或语言之间的类型进行互相转换问题。打个比方：C++写的客户端发送一个int型(4字节)变量的数据到Python写的服务器，Python接收到表示这个整数的4个字节数据，怎么解析成Python认识的整数呢？Python的标准模块struct就用来解决这个问题。

struct模块的内容不多，也不是太难，下面对其中最常用的方法进行介绍：

struct.pack

struct.pack用于将Python的值根据格式符，转换为字符串（因为Python中没有字节(Byte)类型，可以把这里的字符串理解为字节流，或字节数组）。其函数原型为：struct.pack(fmt, v1, v2, ...)，参数fmt是格式字符串，关于格式字符串的相关信息在[下面](#)有所介绍。v1, v2, ...表示要转换的python值。下面的例子将两个整数转换为字符串（字节流）：

Python

```
import struct
1 import struct
2
3 a = 20
4 b = 400
5 str = struct.pack("ii", a, b) #转换后的str虽然是字符串类型，但相当于其他语言中的字节流（字节数组），可以在网络上
6 传输
7 print 'length:', len(str)
8 print str
9 print repr(str)
10
11 #---- result
12 #length: 8
13 # ----这里是乱码
14 #'x14/x00/x00/x00/x90/x01/x00/x00'
```

格式符”i”表示转换为int，”ii”表示有两个int变量。进行转换后的结果长度为8个字节（int类型占用4个字节，两个int为8个字节），可以看到输出的结果是乱码，因为结果是二进制数据，所以显示为乱码。可以使用python的内置函数repr来获取可识别的字符串，其中十六进制的0x00000014, 0x00001009分别表示20和400。

struct.unpack

struct.unpack做的工作刚好与struct.pack相反，用于将字节流转换成python数据类型。它的函数原型为：struct.unpack(fmt, string)，该函数返回一个元组。下面是一个简单的例子：

Python

```
str = struct.pack("ii", 20, 400)
1 str = struct.pack("ii", 20, 400)
2 a1, a2 = struct.unpack("ii", str)
3 print 'a1:', a1
4 print 'a2:', a2
5
6 #---- result:
7 #a1: 20
8 #a2: 400
```

struct.calcsize

struct.calcsize用于计算格式字符串所对应的结果的长度，如：struct.calcsize('ii')，返回8。因为两个int类型所占用的长度是8个字节。

struct.pack_into, struct.unpack_from

这两个函数在Python手册中有所介绍，但没有给出如何使用的例子。其实它们在实际应用中用的并不多。Google了很久，才找到一个例子，贴出来共享一下：

Python

```
import struct
from ctypes import
1 import struct
2 from ctypes import create_string_buffer
3
4 buf = create_string_buffer(12)
5 print repr(buf.raw)
6
7 struct.pack_into("iii", buf, 0, 1, 2, -1)
8 print repr(buf.raw)
9
10 print struct.unpack_from('iii', buf, 0)
11
12 #---- result
13 #'x00/x00/x00/x00/x00/x00/x00/x00/x00/x00/x00/x00'
14 #'x01/x00/x00/x02/x00/x00/xff/xff/xff'
15 #(1, 2, -1)
```

关于格式字符串

在Python手册中，给出了C语言中常用类型与Python类型对应的格式符：

格式符	C语言类型	Python类型	注
x	pad byte	no value	
c	char	string of length 1	

格式符	C语言类型	Python类型	注
B	unsigned char	integer	
?	Bool	bool	
h	short	integer	
H	unsigned short	integer	
i	int	integer	
I	unsigned int	integer or long	
l	long	integer	
L	unsigned long	long	
q	long long	long	
Q	unsigned long long	long	
f	float	float	
d	double	float	
s	char[]	string	
p	char[]	string	
P	void *	long	

具体内容请参考Python手册 [struct](#) 模块

1 赞 1 收藏 [评论](#)

Python模块学习：glob 文件路径查找

原文地址：[Darkbull](#)

glob模块是最简单的模块之一，内容非常少。用它可以查找符合特定规则的文件路径名。跟使用windows下的文件搜索差不多。查找文件只用到三个匹配符：“*”，“？”，“[]”。”*”匹配0个或多个字符；”？”匹配单个字符；”[]”匹配指定范围内的字符，如：[0-9]匹配数字。

glob.glob

返回所有匹配的文件路径列表。它只有一个参数 pathname，定义了文件路径匹配规则，这里可以是绝对路径，也可以是相对路径。下面是使用glob.glob的例子：

Python

```
import glob  
1 import glob  
2  
3 #获取指定目录下的所有图片  
4 print glob.glob(r"E:/Picture/*/*.jpg")  
5  
6 #获取上级目录的所有.py文件  
7 print glob.glob(r'../*.py') #相对路径
```

glob.iglob

获取一个可编历对象，使用它可以逐个获取匹配的文件路径名。与glob.glob()的区别是：glob.glob同时获取所有的匹配路径，而glob.iglob一次只获取一个匹配路径。这有点类似于.NET中操作数据库用到的DataSet与DataReader。下面是一个简单的例子：

Python

```
import glob  
1 import glob  
2  
3 #父目录中的.py文件  
4 f = glob.iglob(r'../*.py')  
5  
6 print f #<generator object iglob at 0x00B9FF80>  
7  
8 for py in f:  
9     print py
```

It's so easy, is't it?

1 赞 3 收藏 [评论](#)

Python模块学习：time 日期时间处理

原文地址：[Darkbull](#)

在应用程序的开发过程中，难免要跟日期、时间处理打交道。如：记录一个复杂算法的执行时间；网络通信中数据包的延迟等等。Python中提供了time, datetime calendar等模块来处理时间日期，今天对time模块中最常用的几个函数作一个介绍。

time.time

time.time()函数返回从1970年1月1日以来的秒数，这是一个浮点数。

time.sleep

可以通过调用time.sleep来挂起当前的进程。time.sleep接收一个浮点型参数，表示进程挂起的时间。

time.clock

在windows操作系统上，time.clock() 返回第一次调用该方法到现在的秒数，其精确度高于1微秒。可以使用该函数来记录程序执行的时间。下面是一个简单的例子：

Python

```
import time
1 import time
2
3 print time.clock() #1
4 time.sleep(2)
5 print time.clock() #2
6 time.sleep(3)
7 print time.clock() #3
8
9 #---- result
10 #3.91111160776e-06
11 #1.99919151736
12 #4.99922364435
```

time.gmtime

该函数原型为：time.gmtime([sec])，可选的参数sec表示从1970-1-1以来的秒数。其默认值为time.time()，函数返回time.struct_time类型的对象。（struct_time是在time模块中定义的表示时间的对象），下面是一个简单的例子：

Python

```
import time
```

```
import time
1
2 print time.gmtime() #获取当前时间的struct_time对象
3 print time.gmtime(time.time() - 24 * 60 * 60) #获取昨天这个时间的struct_time对象
4
5 #---- result
6 #time.struct_time(tm_year=2009, tm_mon=6, tm_mday=23, tm_hour=15, tm_min=16, tm_sec=3, tm_wday=1,
7 tm_yday=174, tm_isdst=0)
8 #time.struct_time(tm_year=2009, tm_mon=6, tm_mday=22, tm_hour=15, tm_min=16, tm_sec=3, tm_wday=0,
tm_yday=173, tm_isdst=0)
```

time.localtime

time.localtime与time.gmtime非常类似，也返回一个struct_time对象，可以把它看作是gmtime()的本地化版本。

time.mktime

time.mktime执行与gmtime(), localtime()相反的操作，它接收struct_time对象作为参数，返回用秒数来表示时间的浮点数。例如：

Python

```
import time
1 import time
2
3 #下面两个函数返回相同（或相近）的结果
4 print time.mktime(time.localtime())
5 print time.time()
```

time.strftime

time.strftime将日期转换为字符串表示，它的函数原型为：time.strftime(format[, t])。参数format是格式字符串（格式字符串的知识可以参考：[time.strftime](#)），可选的参数t是一个struct_time对象。下面的例子将struct_time对象转换为字符串表示：

Python

```
import time
1 import time
2
3 print time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime())
4 print time.strftime('Weekday: %w; Day of the year: %j')
5
6 #---- result
7 #2009-06-23 15:30:53
8 #Weekday: 2; Day of the year: 174
```

time.strptime

按指定格式解析一个表示时间的字符串，返回struct_time对象。该函数原型为：time.strptime(string, format)，两个参数都是字符串，下面是一个简单的例子，演示将一个字符串解析为一个struct_time对象：

Python

```
import time
1 import time
2 print time.strptime('2009-06-23 15:30:53', '%Y-%m-%d %H:%M:%S')
3
4 #---- result
5 #time.struct_time(tm_year=2009, tm_mon=6, tm_mday=23, tm_hour=15, tm_min=30, tm_sec=53, tm_wday=1,
6 tm_yday=174, tm_isdst=-1)
```

以上介绍的方法是time模块中最常用的几个方法，在Python手册中还介绍了其他的方法和属性，如：time.timezone, time.tzname ...感兴趣的朋友可以参考Python手册 [time](#) 模块。

1 赞 3 收藏 [评论](#)