#### **字符串**

字符串是以单引号'或双引号"括起来的任意文本，比如'abc'，"xyz"等等。请注意，''或""本身只是一种表示方式，不是字符串的一部分，因此，字符串'abc'只有a，b，c这3个字符。如果'本身也是一个字符，那就可以用""括起来，比如"I'm OK"包含的字符是I，'，m，空格，O，K这6个字符。

如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

'I\'m \"OK\"!'

表示的字符串内容是：

I'm "OK"!

转义字符\可以转义很多字符，比如\n表示换行，\t表示制表符，字符\本身也要转义，所以\\表示的字符就是\，可以在Python的交互式命令行用print()打印字符串看看：

>>> **print**('I\'m ok.')

I'm ok.

>>> print('I\'m learning\nPython.')

I'm learning

Python.

>>> print('\\\n\\')

\

\

如果字符串里面有很多字符都需要转义，就需要加很多\，为了简化，Python还允许用r''表示''内部的字符串默认不转义，可以自己试试：

>>> print('\\\t\\')

\ \>>> print(r'\\\t\\')

\\\t\\

如果字符串内部有很多换行，用\n写在一行里不好阅读，为了简化，Python允许用'''...'''的格式表示多行内容，可以自己试试：

>>> print('''line1... line2... line3''')

line1

line2

line3

上面是在交互式命令行内输入，注意在输入多行内容时，提示符由>>>变为...，提示你可以接着上一行输入。如果写成程序，就是：

print('''line1

line2

line3''')

多行字符串'''...'''还可以在前面加上r使用，请自行测试。

#### **空值**

空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

### **变量**

在Python中，等号=是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，例如：

a = 123 *# a是整数***print**(a)

a = 'ABC' *# a变为字符串***print**(a)

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言

### 常量

所谓常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量：

PI = 3.14159265359

但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

最后解释一下整数的除法为什么也是精确的。在Python中，有两种除法，一种除法是/：

>>> 10 / 33.3333333333333335

/除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数：

>>> 9 / 33.0

还有一种除法是//，称为地板除，两个整数的除法仍然是整数：

>>> 10 *// 3*

3

### 字符编码

搞清楚了ASCII、Unicode和UTF-8的关系，我们就可以总结一下现在计算机系统通用的字符编码工作方式：

在计算机内存中，统一使用Unicode编码，当需要保存到硬盘或者需要传输的时候，就转换为UTF-8编码。

对于单个字符的编码，Python提供了ord()函数获取字符的整数表示，chr()函数把编码转换为对应的字符：

>>> ord('A')65>>> ord('中')20013>>> chr(66)'B'>>> chr(25991)'文'

Python对bytes类型的数据用带b前缀的单引号或双引号表示：

x = b'ABC'

要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但bytes的每个字符都只占用一个字节。

以Unicode表示的str通过encode()方法可以编码为指定的bytes，例如：

>>> 'ABC'.encode('ascii')

b'ABC'

>>> '中文'.encode('utf-8')

b'\xe4\xb8\xad\xe6\x96\x87'

>>> '中文'.encode('ascii')

Traceback (most recent call last):

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

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range(128)

纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错。

在bytes中，无法显示为ASCII字符的字节，用\x##显示。

反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是bytes。要把bytes变为str，就需要用decode()方法：

>>> b'ABC'.decode('ascii')'ABC'>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')'中文'

### list

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

>>> classmates[-1]'Tracy'

以此类推，可以获取倒数第2个、倒数第3个

要删除list末尾的元素，用pop()方法：

>>> classmates.pop()'Adam'>>> classmates

['Michael', 'Jack', 'Bob', 'Tracy']

要删除指定位置的元素，用pop(i)方法，其中i是索引位置：

>>> classmates.pop(1)'Jack'>>> classmates

['Michael', 'Bob', 'Tracy']

### tuple

tuple和list非常类似，但是tuple一旦初始化就不能修改，比如同样是列出同学的名字：

>>> classmates = ('Michael', 'Bob', 'Tracy')

现在，classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple。

要定义一个只有1个元素的tuple，如果你这么定义：

>>> t = (1)>>> t1

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的tuple定义时必须加一个逗号,，来消除歧义：

>>> t = (1,)>>> t

(1,)

Python在显示只有1个元素的tuple时，也会加一个逗号,，以免你误解成数学计算意义上的括号。

### 条件判断

elif是else if的缩写，完全可以有多个elif，所以if语句的完整形式就是：

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

else:

<执行4>

#### 函数的参数

默认参数必须指向不变对象！

### \_\_slots\_\_

Python允许在定义class的时候，定义一个特殊的\_\_slots\_\_变量，来限制该class实例能添加的属性：

**class Student(object):**

\_\_slots\_\_ = ('name', 'age') *# 用tuple定义允许绑定的属性名称*

使用\_\_slots\_\_要注意，\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的。除非在子类中也定义\_\_slots\_\_，这样，子类实例允许定义的属性就是自身的\_\_slots\_\_加上父类的\_\_slots\_\_

#### @property

**class Student(object):**

**def** **get\_score**(self):

**return** self.\_score

**def** **set\_score**(self, value):

**if** **not** isinstance(value, int):

**raise** ValueError('score must be an integer!')

**if** value < 0 **or** value > 100:

**raise** ValueError('score must between 0 ~ 100!')

self.\_score = value

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

**class Student(object):**

@property

**def** **birth**(self):

**return** self.\_birth

@birth.setter

**def** **birth**(self, value):

self.\_birth = value

@property

**def** **age**(self):

**return** 2015 - self.\_birth

上面的birth是可读写属性，而age就是一个****只读****属性，因为age可以根据birth和当前时间计算出来。

#### 多重继承

Python自带的很多库也使用了MixIn。举个例子，Python自带了TCPServer和UDPServer这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由ForkingMixIn和ThreadingMixIn提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

**class MyTCPServer(TCPServer, ForkingMixIn):**

**pass**

编写一个多线程模式的UDP服务，定义如下：

**class MyUDPServer(UDPServer, ThreadingMixIn):**

**pass**

如果你打算搞一个更先进的协程模型，可以编写一个CoroutineMixIn：

**class MyTCPServer(TCPServer, CoroutineMixIn):**

**pass**

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用MixIn的设计。

在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。

#### 文件读写

为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

**try**:

f = open('/path/to/file', 'r')

print(f.read())**finally**:

**if** f:

f.close()

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

**with** open('/path/to/file', 'r') **as** f:

print(f.read())

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。

#### 序列化

把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization，marshalling，flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

Python提供了pickle模块来实现序列化。

#### 多线程

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：\_thread和threading，\_thread是低级模块，threading是高级模块，对\_thread进行了封装。绝大多数情况下，我们只需要使用threading这个高级模块。

多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。

Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

#### ThreadLocal

全局变量local\_school就是一个ThreadLocal对象，每个Thread对它都可以读写student属性，但互不影响。你可以把local\_school看成全局变量，但每个属性如local\_school.student都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，ThreadLocal内部会处理。

可以理解为全局变量local\_school是一个dict，不但可以用local\_school.student，还可以绑定其他变量，如local\_school.teacher等等。

ThreadLocal最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

### 小结

一个ThreadLocal变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。ThreadLocal解决了参数在一个线程中各个函数之间互相传递的问题。

#### 进程 vs. 线程

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用fork调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

### 线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

### 计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

### 异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单线程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

多任务的两种方式：多进程和多线程。如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。 多进程模式最大的优点就是稳定性高，缺点是创建代价大 过多的线程切换消耗资源。 计算密集型任务：使用像C语言，实现效率较高 IO密集型任务:使用脚本语言 例如python开发效率最高

异步IO，Python语言，单线程的异步编程模型称为协程

#### 分布式进程

Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。 注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

#### 正则表达式

Python提供re模块，包含所有正则表达式的功能。由于Python的字符串本身也用\转义，所以要特别注意：

s = 'ABC\\-001' *# Python的字符串# 对应的正则表达式字符串变成：# 'ABC\-001'*

因此我们强烈建议使用Python的r前缀，就不用考虑转义的问题了：

s = r'ABC\-001' *# Python的字符串# 对应的正则表达式字符串不变：# 'ABC\-001'*

#### 模块

datetime模块还包含一个datetime类，通过from datetime import datetime导入的才是datetime这个类。

collections是Python内建的一个集合模块，提供了许多有用的集合类 namedtuple: namedtuple是一个函数，它用来创建一个自定义的tuple对象，并且规定了tuple元素的个数，并可以用属性而不是索引来引用tuple的某个元素 deque：  
deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈。可以再首或者尾进行删除和插入 defaultdict:  
使用dict时，如果引用的Key不存在，就会抛出KeyError。如果希望key不存在时，返回一个默认值，就可以用defaultdict OrderedDict 可以保证key的顺序

Base64是一种用64个字符来表示任意二进制数据的方法。 Base64：二进制编码方法，使二进制能够转换为字符串 Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据，二进制的3个字节到文本的4个字节

摘要算法就是通过摘要函数f()对任意长度的数据data计算出固定长度的摘要digest，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算f(data)很容易，但通过digest反推data却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

由于常用口令的MD5值很容易被计算出来，所以，要确保存储的用户口令不是那些已经被计算出来的常用口令的MD5，这一方法通过对原始口令加一个复杂字符串来实现，俗称“加盐”：

**def** **calc\_md5**(password):

**return** get\_md5(password + 'the-Salt')

经过Salt处理的MD5口令，只要Salt不被黑客知道，即使用户输入简单口令，也很难通过MD5反推明文口令。

摘要算法在很多地方都有广泛的应用。要注意摘要算法不是加密算法，不能用于加密（因为无法通过摘要反推明文），只能用于防篡改，但是它的单向计算特性决定了可以在不存储明文口令的情况下验证用户口令。

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。 无限迭代器： 打印出：1,2,3，。。。。 natuals = itertools.count(1) for n in natuals: print(n)

打印出ABC.....一直循环 cs = itertools.cycle('ABC') # 注意字符串也是序列的一种 for c in cs: print(c)

打印三个A  
ns = itertools.repeat('A', 3) for n in ns: print(n)  
无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器 groupby()把迭代器中相邻的重复元素挑出来放在一起

#### contextlib

并不是只有open()函数返回的fp对象才能使用with语句。实际上，任何对象，只要正确实现了上下文管理，就可以用于with语句。

实现上下文管理是通过\_\_enter\_\_和\_\_exit\_\_这两个方法实现的。

@contextmanager这个decorator接受一个generator，用yield语句把with ... as var把变量输出出去，然后，with语句就可以正常地工作了

closing（）它的作用就是把任意对象变为上下文对象，并支持with语句

#### XML

操作XML有两种方法：DOM和SAX。 DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。 解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据

#### virtualenv

如果每个应用需要各自拥有一套“独立”的Python运行环境，virtualenv就是用来为一个应用创建一套“隔离”的Python运行环境。

#### 电子邮件

MUA:邮件用户代理 MTA:邮件传输代理 MDA:邮件投递代理

一封电子邮件的旅程就是： 发件人 -> MUA -> MTA -> MTA -> 若干个MTA -> MDA <- MUA <- 收件人 如果编写程序来发送和接收邮件，本质上就是： 编写MUA把邮件发到MTA； 编写MUA从MDA上收邮件。 发邮件时，MUA和MTA使用的协议就是SMTP，后面的MTA到另一个MTA也是用SMTP协议。 收邮件时，MUA和MDA使用的协议有两种：POP3和IMAP。

#### 协程

最大的优势就是协程极高的执行效率。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销，和多线程比，线程数量越多，协程的性能优势就越明显。

第二大优势就是不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

因为协程是一个线程执行，那怎么利用多核CPU呢？最简单的方法是多进程+协程，既充分利用多核，又充分发挥协程的高效率，可获得极高的性能。

Python对协程的支持是通过generator实现的。

在generator中，我们不但可以通过for循环来迭代，还可以不断调用next()函数获取由yield语句返回的下一个值。

但是Python的yield不但可以返回一个值，它还可以接收调用者发出的参数。

#### async/await

用asyncio提供的@asyncio.coroutine可以把一个generator标记为coroutine类型，然后在coroutine内部用yield from调用另一个coroutine实现异步操作。

为了简化并更好地标识异步IO，从Python 3.5开始引入了新的语法async和await，可以让coroutine的代码更简洁易读。

请注意，async和await是针对coroutine的新语法，要使用新的语法，只需要做两步简单的替换：

1. 把@asyncio.coroutine替换为async；
2. 把yield from替换为await。

Python从3.5版本开始为asyncio提供了async和await的新语法；

注意新语法只能用在Python 3.5以及后续版本，如果使用3.4版本，则仍需使用上一节的方案。

#### MVVM

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择[uikit](http://getuikit.com/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

编写后端Python代码不但很简单，而且非常容易测试，上面的API：api\_create\_blog()本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串,显然这种方式完全不具备可维护性。所以有第二种模板方式,ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的[MVVM](http://en.wikipedia.org/wiki/Model_View_ViewModel" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示<script>

**var** blog = {

name: 'hello',

summary: 'this is summary',

content: 'this is content...'

};</script>

View是纯HTML：

<form action="/api/blogs" method="post">

<input name="name">

<input name="summary">

<textarea name="content"></textarea>

<button type="submit">OK</button></form>

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择[Vue](http://vuejs.org/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)这个简单易用的MVVM框架来实现创建Blog的页面

#### watchdog

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测www目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序pymonitor.py，让它启动wsgiapp.py，并时刻监控www目录下的代码改动，有改动时，先把当前wsgiapp.py进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库watchdog可以利用操作系统的API来监控目录文件的变化，并发送通知。

#### 部署Web App

很多做开发的同学把部署这件事情看成是运维同学的工作，这种看法是完全错误的。首先，最近流行[DevOps](http://zh.wikipedia.org/wiki/DevOps" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)理念，就是说，开发和运维要变成一个整体。其次，运维的难度，其实跟开发质量有很大的关系。代码写得垃圾，运维再好也架不住天天挂掉。最后，DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务？那就得在开发时考虑到这一点。

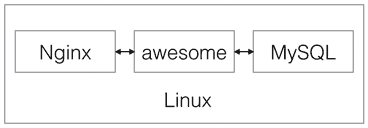
要部署到Linux，首先得有一台Linux服务器。要在公网上体验的同学，可以在Amazon的[AWS](http://aws.amazon.com/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)申请一台EC2虚拟机（免费使用1年），或者使用国内的一些云服务器，一般都提供Ubuntu Server的镜像。想在本地部署的同学，请安装虚拟机，推荐使用[VirtualBox](https://www.virtualbox.org/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)。

我们选择的Linux服务器版本是[Ubuntu Server 14.04 LTS](http://www.ubuntu.com/download/server" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)，原因是apt太简单了。如果你准备使用其他Linux版本，也没有问题。

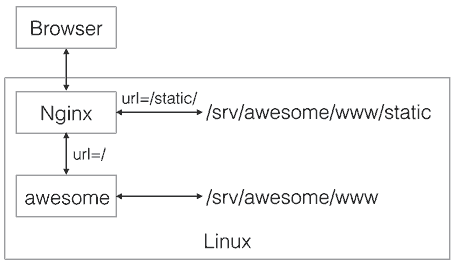
Linux安装完成后，请确保ssh服务正在运行，否则，需要通过apt安装

### 部署方式

利用Python自带的asyncio，我们已经编写了一个异步高性能服务器。但是，我们还需要一个高性能的Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。这个模型如下：



Nginx负责分发请求：



在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把www作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本

Nginx可以作为服务进程直接启动，但app.py还不行，所以，[Supervisor](http://supervisord.org/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

Nginx：高性能Web服务器+负责反向代理；

Supervisor：监控服务进程的工具；

MySQL：数据库服务。

### 部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。[Fabric](http://www.fabfile.org/" \t "https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/_blank)就是一个自动化部署工具。由于Fabric是用Python 2.x开发的，所以，部署脚本要用Python 2.7来编写，本机还必须安装Python 2.7版本。

### 问题Q&A

1、安装tensorflow可能会导致pip错误，恰好前一段时间刚安装了tensorflow，解决方法为

conda install -c anaconda html5lib 或者

conda install pip

1. image\_to\_string raise tesseracterror

安装OCR模块的时候没有选择安装中文语言包

3、windows下使用PyCharm编程，import的类找不到

需要导入正确的python库setting——> Project Interpreter——>下拉框选择python.exe所在的路径。

4、