第一章：

1.1 IO编程

1.1.1 打开文件

open(name[.mode[.buffering]])

name:强制参数

mode 和 buffering(缓冲区) 参数可选，默认：读模式 ，默认：无缓冲区

例子: >>> f = open('./mydata1.txt')

1.1.2 文件模式

open函数中的mode参数

‘r’ 读模式

‘w’ 写模式

‘a’ 追加模式

‘b’ 二进制模式（可添加到其他模式中使用）

‘+’ 读/写模式（可添加到其他模式中使用）

1.1.3 文件缓冲区

open函数中第三个参数可选参数buffering控制文件的缓冲

如果参数是0，I/O操作是无缓冲区，直接将数据写到硬盘上

如果参数是1，I/O操作是有缓冲区，数据写到内存里，只使用flush函数或close函数才将数据更新到硬盘

如果参数大于1,则表示缓冲区的大小， -1代表默认缓冲区的大小

1.1.4 文件读取

文件读取：按字节读取 按行读取 ----》 read(), readlines()

read()则可以一次性将文件内容全部读到内存中，最后返回的是str类型对象。

例子：>>> f = open('./mydata1.txt')

>>> f.read()

"{'title': ['<title>新浪首页</title>']}\n"

最后一步：close()，可以关闭对文件的引用

保证程序的健壮性，我们需要使用try…finally来实现

>>> try:

... f = open('./mydata1.txt','r')

... print(f.read())

... finally:

... if f:

... f.close()

...

{'title': ['<title>新浪首页</title>']}

使用with语句来替代try….finally代码块 和close()

>>> with open('./mydata1.txt') as f:

... print(f.read())

...

{'title': ['<title>新浪首页</title>']}

注意：read()一次将文件内容读到内存，但是如果文件过大，将会出现内存不足，可以反复调用read(size)方法

1.1.5 文件写入

>>> f = open('./mydata1.txt','w')

>>> f.write('qiye') #写入文件

4

>>> f.close() #关闭文件

使用write()方法的时候，操作系统不是立即将数据写入文件中，而是先写入内存缓存起来，等到空闲时候在写入文件中，

最后使用close()方法就将数据完整地写入文件中。

推荐使用with语句：

>>> with open('./mydata1.txt','w') as f:

... f.write('111')

...

3

1.2.1 操作文件和目录

1）获得当前python脚本工作的目录路径：os.getcwd()

例子：>>> import os

>>> os.getcwd()

'/home/pythonproject'

2）返回指定目录下的所有文件和目录：os.listdir()

例子：>>> import os

>>> os.listdir()

3）删除一个文件：os.remove(filepath):

例子：>>> import os

>>> os.remove('./mydata1.txt')

4）删除多个空目录：os.removedirs(dir)

5) 检验给出的路径是否是一个文件：os.path.isfile(filepath)

例子：>>> import os

>>> os.path.isfile('./mydata1.txt')

False

6）检验给出的路径是否是一个目录：os.path.isdir(filepath)

例子：>>> import os

>>> os.path.isdir('./mydata1.txt')

False

7）判断是否是绝对路径： os.path.isabs()

例子：>>> import os

>>> os.path.isabs('./mydata1.txt')

False

8) 检验路径是否真的存在：os.path.exists()

例子：>>> import os

>>> os.path.exists('mydata1.txt')

False

9) 分离一个路径的目录和文件名：os.path.split():

例子：>>> import os

>>> os.path.split('./mydata1.txt')

('.', 'mydata1.txt') #返回元组

10) 分离扩展名： os.path.splitext()

例子：>>> import os

>>> os.path.splitext('./mydata1.txt')

('./mydata1', '.txt')

11）获取路径名：os.path.dirname(filepath)

例子：>>> import os

>>> os.path.dirname('./mydata1.txt')

'.'

12)获取文件名： os.path.basename(filepath)

例子：>>> import os

>>> os.path.basename('./mydata1.txt')

'mydata1.txt'

13）读取和设置环境变量：os.getenv() 与 os.putenv()

例子：>>> import os

>>> os.getenv("PATH")

'/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games'

例子：os.putenv() #在大多版本的unix，Windows中有效

14) 给出当前平台使用的行终止符：os.linesep.

例子：>>> os.linesep

'\n'

15） 指出你正在使用的平台：os.name

例子：>>> os.name

'posix'

17）重命名文件或者目录：os.rename(old,new)

例子：>>> import os

>>> os.rename('./mydata.json','./mydata2.json')

18）创建多级目:os.makedirs(“”)

例子：>>> import os

>>> os.makedirs('./test/test')

19）创建单个目录：os.mkdir(“test”)

例子：>>> import os

>>> os.mkdir('test1')

20）获取文件属性:os.stat(file)

例子：>>> import os

>>> os.stat('./mydata2.json')

os.stat\_result(st\_mode=33188, st\_ino=1049898, st\_dev=64769, st\_nlink=1, st\_uid=0, st\_gid=0, st\_size=0, st\_atime=1510499453, st\_mtime=1510499453, st\_ctime=1512488305)

21) 修改文件权限与时间：os.chmod(file)

例子：>>> import os

>>> import stat

>>> os.chmod('./mydata2.json',stat.S\_IXGRP)

21）获取文件大小：os.path.getsize(filename)

例子：>>> import os

>>> os.path.getsize('./mydata2.json')

0

22)复制文件夹:shutil.copytree(“olddir”,”newdir”) #newdir必须不存在

例子: >>> import shutil

>>> shutil.copytree("./test","./test3")

'./test3'

23)复制文件：shutil.copyfile(“oldfile”,”newfile”) #

例子：>>> import shutil

>>> shutil.copyfile("./mydata2.json","./mydata4.txt")

'./mydata4.txt'

24)移动文件：shutil.move(“oldps”,”newpos”)

例子: >>> import shutil

>>> shutil.move("./mydata2.json","./mydata2.txt")

'./mydata2.txt'

25）删除目录：os.rmdir(“dir”) #只能删除空目录

shutil.rmtree(“dir”) #空目录，有内容的目录可以删

1.3.3 序列化操作

1. 把内存中的变量编程可存储或可传输的过程，就是序列化

序列化的数据可以做什么：1.写入内存 2.网络传输

2. 把变量内容从序列化的对象重新读取到内存，称为反序列化

3. 模块： cPickle 和 pickle (前者效率高)

例子：>>> try:

... import cPickle as pickle

... except ImportError:

... import pickle

主要使用方法： dumps 或 dump

dumps 方法可以将任意对象序列化成一个str,将这个str写入进行保存。

例子：>>> import pickle

>>> d = dict(url='index.html')

>>> pickle.dumps(d)

b'\x80\x03}q\x00X\x03\x00\x00\x00urlq\x01X\n\x00\x00\x00index.htmlq\x02s.'

>>>

dump方法，可将序列化后的对象直接写入文件中

例子：>>> import pickle

>>> d = dict(url='index.html')

b'\x80\x03}q\x00X\x03\x00\x00\x00urlq\x01X\n\x00\x00\x00index.htmlq\x02s.'

>>> f = open("./dump.txt",'wb')

>>> pickle.dump(d,f)

>>> f.close()

反序列化： loads 和 load ， 把序列化的文件从磁盘上读取为一个str,使用loads将str反序列化对象，load使用方法将文件直接反序列化为对象。

例子：>>> f = open("./dump.txt",'rb')

>>> d = pickle.load(f)

>>> f.close()

>>> d

{'url': 'index.html'}

1.4 进程 和 线程

1.4.1 多进程

多线程： 1. os模块的fork 2.multiprocessing (前者适用于Unix、Linux，后者扩平台）

1.os模块中的fork方式实现多线程

#!/usr/bin/python

import os

if \_\_name\_\_ == '\_\_main\_\_':

print ('current process (%s) start...'% (os.getpid()))

pid = os.fork()

if pid < 0:

print('error in fork')

elif pid == 0:

print('I am child prcess (%s) and my parent process is (%s)'%(os.getpid(),os.getppid()))

else:

print(' I (%s) created child prcess(%s)'%(os.getpid(),pid))

2. 使用multiprocessing模块创建多进程 （可以创建线程池）

import os

from multiprocessing import Process

def run\_proc(name):

print ('Child process %s (%s) Running...' % (name,os.getpid()))

if \_\_name\_\_ == '\_\_main\_\_':

print('Parent process %s.' % os.getpid())

for i in range(5):

p = Process(target=run\_proc,args=(str(i),))

print('Process will start.')

p.start()

p.join()

print('Process end.')

3. multiprocessing 模块提供了一个Pool类来代表进程池对象

Pool 可以提供指定数量的进程供用户使用，默认大小是CPU核数。

过程：当有新的请求提交到Pool中时， 如果池还没有满，那么就创建一个新的进程用来执行该请求；

但如果池中的进程数已经达到规定最大值，那么该请求就会等待，直到池中有进程结束，才会创建新的进程来处理它。

例子：from multiprocessing import Pool

import os, time, random

def run\_task(name):

print('Task %s (pid = %s) is running...'%(name,os.getpid()))

time.sleep(random.random()\*3)

print('Task %s end.' % name)

if \_\_name\_\_ == '\_\_main\_\_':

print('Current process %s.' % os.getpid())

p = Pool(processes=3)

for i in range(5):

p.apply\_async(run\_task,args=(i,))

print('Wating for all subprocesses done..')

p.close()

p.join()

print('All subprocesses done.')

三个线程 五个任务

注意： join()方法会等待所有进程执行完毕，调用join()之前必须先调用close()，调用close()之后，就不能继续添加新process

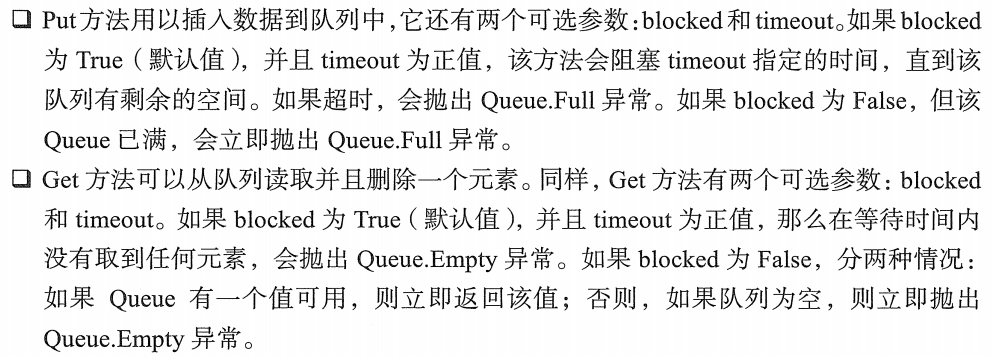
4. 进程间通信

Queue 和 Pipe两种方式

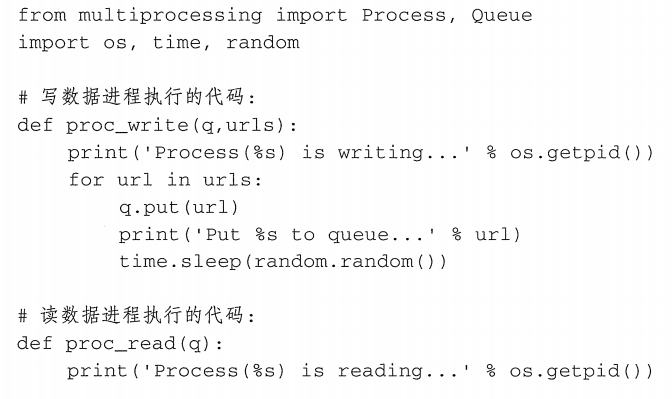
Pipe常用来在两个进程间通信

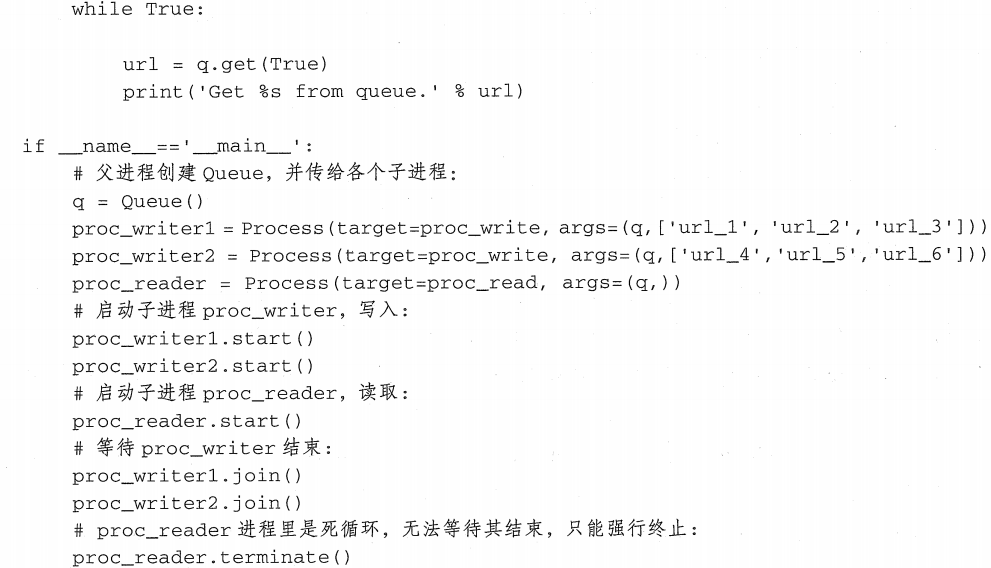
Queue用来多进程间通信

4.1 Queue 通信方式（Queue是多线程安全队列，可以使用Queue实现多线程之间的数据传递）

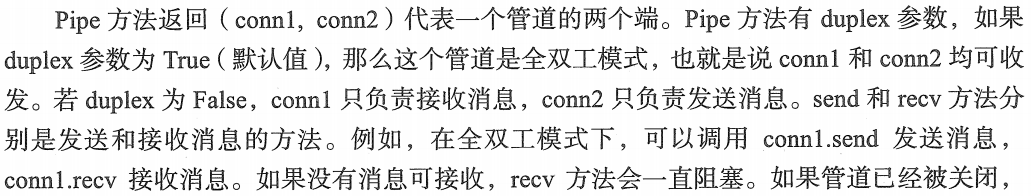


例子：三个进程（两个往Queue写入数据， 一个子进程从Queue读数据）





Pipe 常用来在两个进程间进行通信，两个进程分别位于管道的两端





#!/usr/bin/python

import multiprocessing

import random

import time,os

def proc\_send(pipe,urls):

for url in urls:

print("Process(%s) send:%s" % (os.getpid(),url))

pipe.send(url)

time.sleep(1)

def proc\_recv(pipe):

while True:

print("Process (%s) rev:%s" %(os.getpid(),pipe.recv()))

time.sleep(2)

if \_\_name\_\_=="\_\_main\_\_":

pipe = multiprocessing.Pipe()

p1 = multiprocessing.Process(target=proc\_send,args=(pipe[0],['url\_'+str(i) for i in range(10)]))

p2 = multiprocessing.Process(target=proc\_recv,args=(pipe[1],))

p1.start()

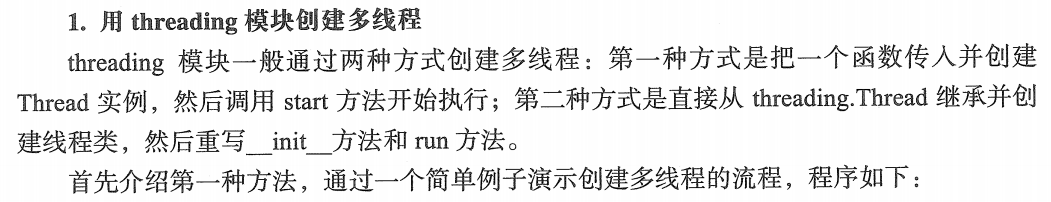
p2.start()

p1.join()

p2.join()

4.2 多线程

python的标准库提供了两个模块:thread 和 threading



import random

import time, threading

def thread\_run(urls):

print("Current %s is running.."%threading.current\_thread().name)

for url in urls:

print("%s -->>> %s" % (threading.current\_thread().name,url))

time.sleep(random.random())

print("%s ended." % threading.current\_thread().name)

print("%s is running...." % threading.current\_thread().name)

t1 = threading.Thread(target=thread\_run,name="Thread\_1",args=(['url\_1','url\_2','url\_3'],))

t2 = threading.Thread(target=thread\_run,name="Thread\_2",args=(['url\_4','url\_5','url\_6'],))

t1.start()

t2.start()

t1.join()

t2.join()

print("%s is running...." % threading.current\_thread().name)

第二种方式：从threading.Thread继承创建线程类。

import random

import threading

import time

class myThread(threading.Thread):

def \_\_init\_\_(self,name,urls):

threading.Thread.\_\_init\_\_(self,name=name)

self.urls = urls

def run(self):

print("Current %s is runniing..." % threading.current\_thread().name)

for url in self.urls:

print("%s---->>> %s" % (threading.current\_thread().name,url))

time.sleep(random.random())

print("%s ended." % threading.current\_thread().name)

print("%s is running..."% threading.current\_thread().name)

t1 = myThread(name="Thread\_1",urls =["url\_1","url\_2","url\_3"])

t2 = myThread(name="Thread\_2",urls =["url\_4","url\_5","url\_6"])

t1.start()

t2.start()

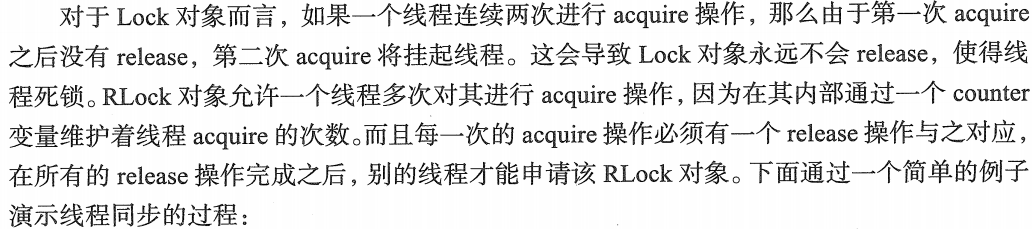
t1.join()

t2.join()

print("%s ended." % threading.current\_thread().name)

2.线程同步

Thread对象的Lock 和 RLock可以实现简单的线程同步，这两个对象都有acquire 和 release.每次只允许一个线程操作的数据，可以放在acquire 和 release方法之间。



import threading

mylock = threading.RLock()

num = 0

class myThread(threading.Thread):

def \_\_init\_\_(self,name):

threading.Thread.\_\_init\_\_(self,name=name)

def run(self):

global num

while True:

mylock.acquire()

print("%s locked, number: %d"%(threading.current\_thread().name,num))

if num >= 4:

mylock.release()

print("%s release, number:%d"%(threading.current\_thread().name,num))

break

num +=1

print("%s released, number:%d"% (threading.current\_thread().name,num))

mylock.release()

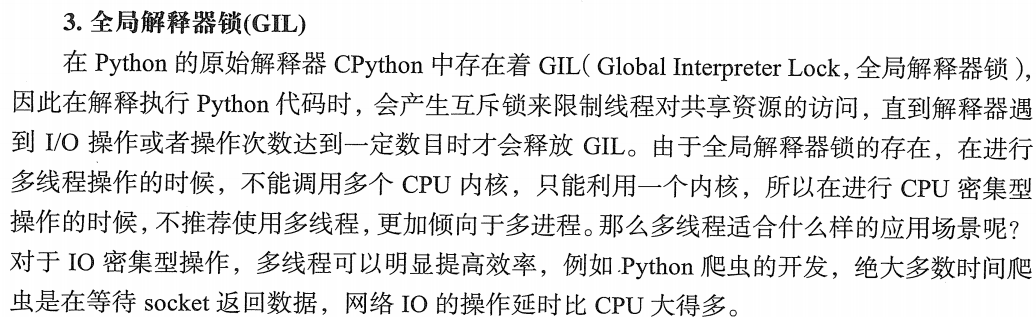
if \_\_name\_\_ == '\_\_main\_\_':

thread1 = myThread("Thread\_1")

thread2 = myThread("Thread\_2")

thread1.start()

thread2.start()



1.4.3 协程

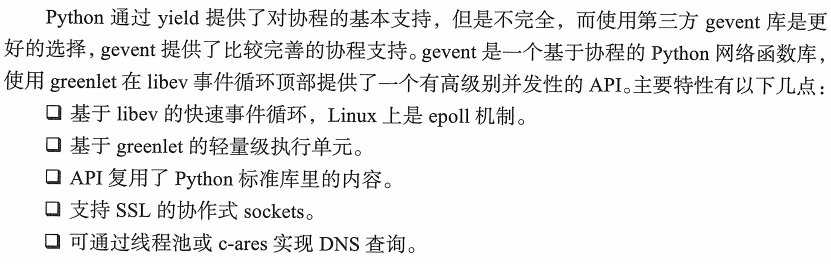
1. 是一种用户级的轻量级线程。

2. 协程拥有自己的寄存器上下文和栈。

3. 协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。

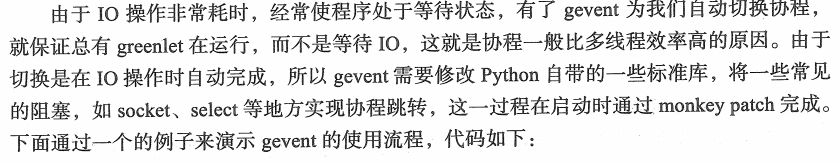
4.在并发编程中，协程与线程类似，每个协程一个执行单元，有自己的本地数据，与其他协程共享全局数据和其他资源。

5. 协程需要自己来编写调度逻辑，对于cpu来说，协程其实是单线程，所有cpu不用考虑怎么调度，切换上下文，这就省去了cpu的切换开销，所以协程在一定程度上又好于多线程。









from gevent import monkey; mokey.patch\_all()

import gevent

import urllib2

def run\_task(url):

print("Visit-->"% url)

try:

response = urllib2.urlopen(url)

data = response.read()

print("%d bytes received from %s." %(len(data),url))

except Exception as e:

print(e)

if \_\_name\_\_ == "\_\_main\_\_":

urls = ['https://github.com/','https://www.python.org/','http://www.cnblogs.com/']

greenlets = [gevent.spawn(run\_task,url) for url in urls]

gevent.joinall(greenlets)

备注： 主要用了gevent中的spawn方法和joinall方法。 spawn方法可以看作用来形成协程。joinall方法就是添加这些协程任务，并且启动运行。

gevent 中还提供了对池的支持，对拥有动态数量的greenlet需要进行并发管理，就可以使用池，这在处理大量的网络和IO操作时是非常需要的。

接下来，使用gevent中pool对象。

from gevent import monkey

monkey.patch\_all()

from gevent.pool import Pool

def run\_task(url):

print("Visit --> %s" % url)

try:

response = urllib2.urlopen(url)

data = response.read()

print("%d bytes received from %s." % (len(data),url))

except Exception as e:

print(e)

return ("url:%s-->finish"%url)

if \_\_name\_\_ == '\_\_main\_\_':

pool = Pool(2)

urls = ['https://github.com/','https://www.python.org/','http://wwwcnblogs.com/']

results = pool.map(run\_task,urls)

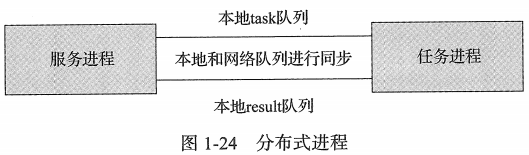
print(results)

1.4.4 分布式进程

分布式进程指的是Process进程分布到多台机器上，充分利用多台机器的性能完成复杂的任务。

multiprocess模块其中managers子模块还支持把多进程分布到多台机器上。

可以写一个服务进程作为调度者，将任务分布到其他多个进程中，依靠网络通信进行管理。



创建分布式进程需要分为六步骤：

1）建立队列Queue,用来进行进程间的通信。

服务进程创建任务队列task\_queue,用来作为传递任务给任务进程的通道；

服务进程创建结果队列 result\_queue,作为任务进程完成任务后回复服务进程的通道。

在分布式多进程环境下，必须通过由Queuemanager获得的Queue接口来添加任务。

2） 把第一步建立的队列在网络上注册，暴露给其他进程（主机），注册后获得网络队列，相当于本地队列的映像。

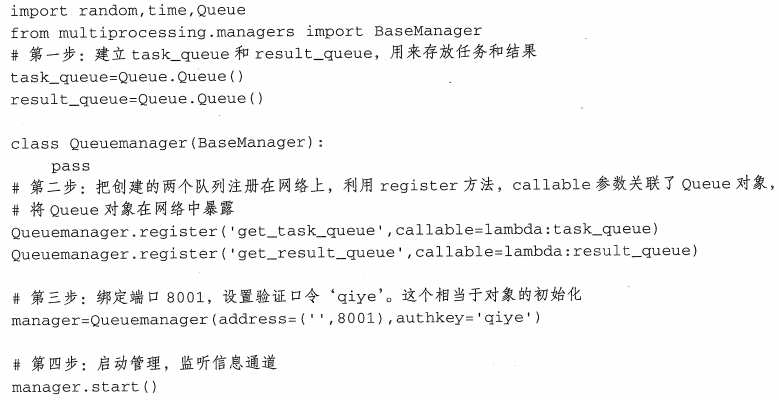
3） 建立对象（Queuemanager(BaseManager）实例manager，绑定端口和验证口令。

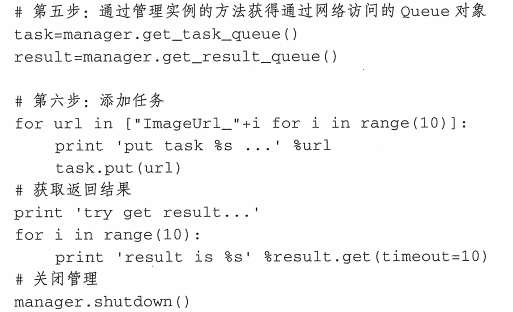
4）启动第三步中建立的实例，即启动管理manager，监管信息通道。

5）通过管理实例的方法获得通过网络访问的Queue对象，即再把网络队列实体化可以使用的本地队列。

6）创建任务到“本地”队列中，自动上传任务到网络队列中，分配给任务进程进行处理。

实例：





创建任务进程的步骤：

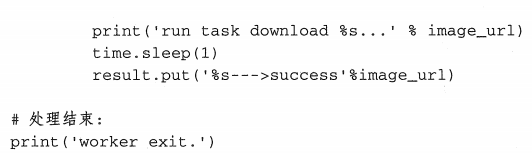
1）使用QueueManager注册用于获取Queue的方法名称，任务进程只能通过名称来在网络上获取Queue.

2) 连接服务器，端口和验证口令注意保持与服务进程中完全一致。

3）从网络上获取Queue,进行本地化。

4）从task队列获取任务，并把结果写入result队列。





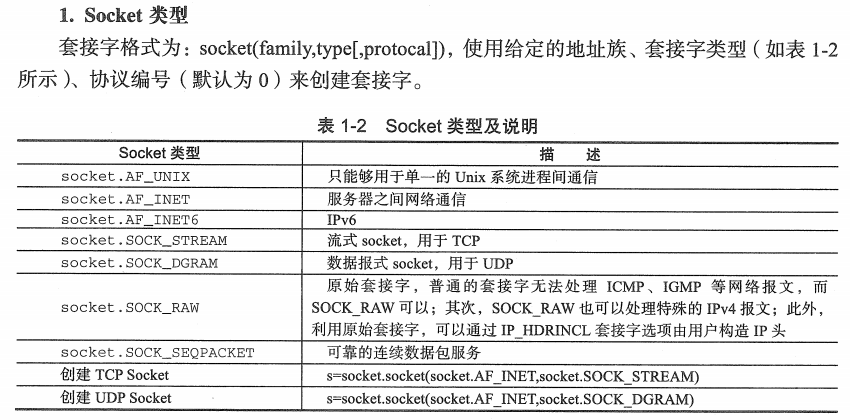
1.5 网络编程

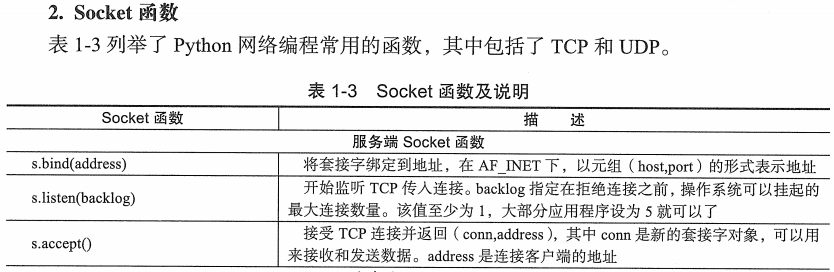
Python 提供了两个基本的Socket模块：

1） Socket提供了标准的BSD Socket API

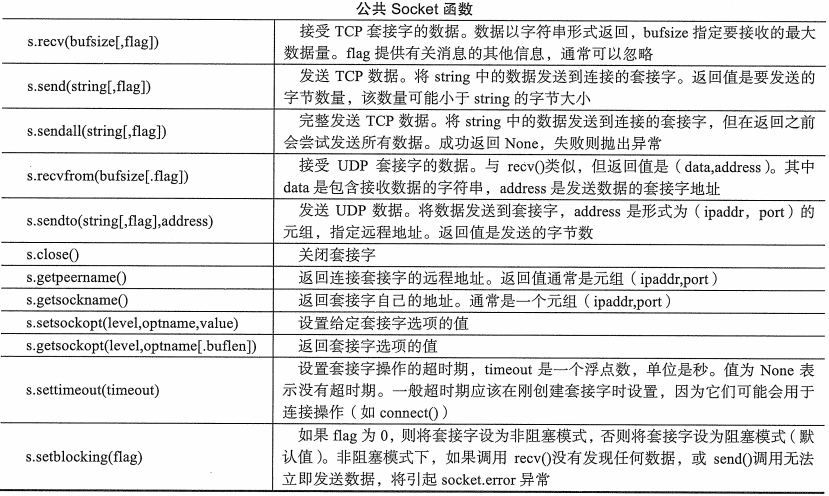
2） SocketServer,提供了服务器中心类，可以简化网络服务器开发。

1. Socket类型









1.5.1 TCP编程

创建和运行TCP服务器端一般五步骤：

1） 创建Socket,绑定Socket到本地IP与端口

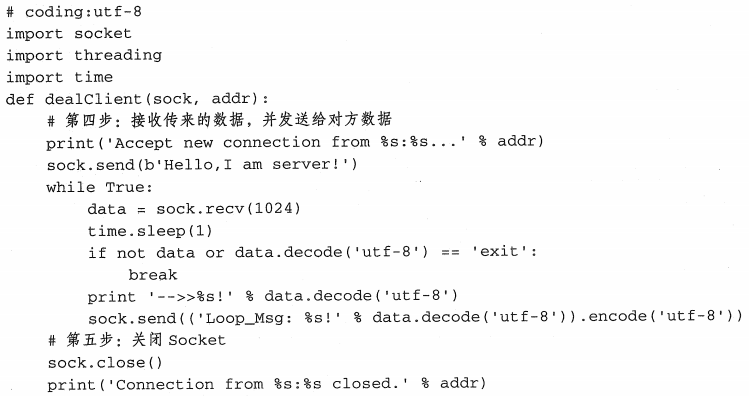
2）开始监听连接

3）进入循环，不断接收客户端的连接请求

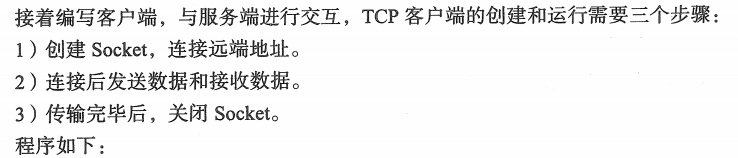
4）接收传来的数据，并发送给对方数据

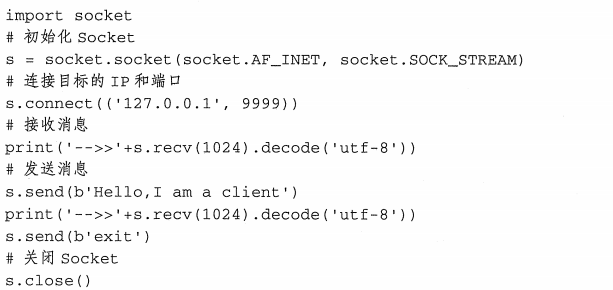
5）传输完毕后，关闭Socket

例子：







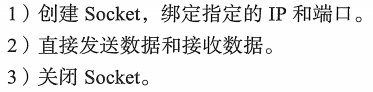


1.5.2 UDP编程

UDP则是面向无连接的协议

不需要建立连接，只需要知道对方的IP地址和端口号

服务器创建和运行只需三个步骤：





第二章： Web前端基础

2.1 W3C标准

网页主要由三部分组成： 结构，表现， 行为

对应的标准也分三方面：

结构化标准语言（XHTML,XML）

表现标准语言（CSS）

行为标准（对象模型（W3C,DOM），ECMAScript）

2.1.1 HTML

1. HTML的基本结构

<html></html>

<head></head> HTML文件头标记

<title></title> <meta></meta> <style></style>

2. 文档设置标记

格式标记 和 文本标记

<br> 强制换行标记

<p> 换段落标记

<center>: 居中对齐标记

<pre> :预格式化标记

<li> :列表项目标记 <ul> :无序列表标记 <ol>: 有序列表标记

<dl><dt><dd>:定义型列表

<br>:水平分割线标记

<div>: 分区显示标记

文本标记：

<hn>:标题标记， 共6个级别，n的范围1~6， h1最大 h6最小

<font>:字体设置标记

<b>: 粗体标记

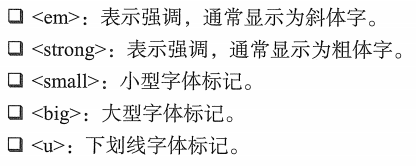
<i>:斜体字标记

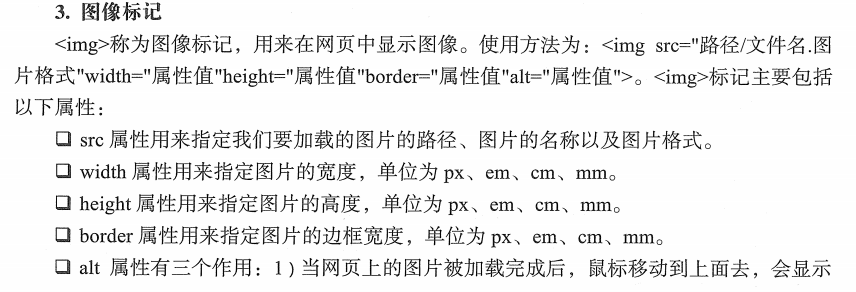
<sub>:文字下标字体标记

<sup>:文字上标字体标记

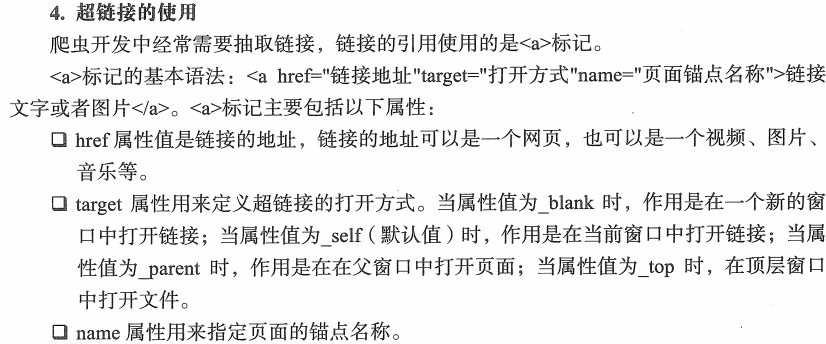
<tt>:打印字体标记

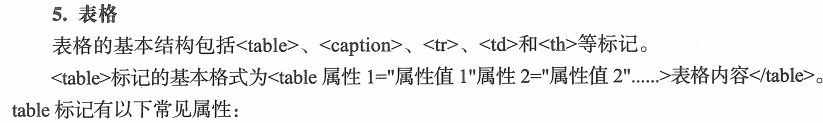
<cite>:引用方式的字体

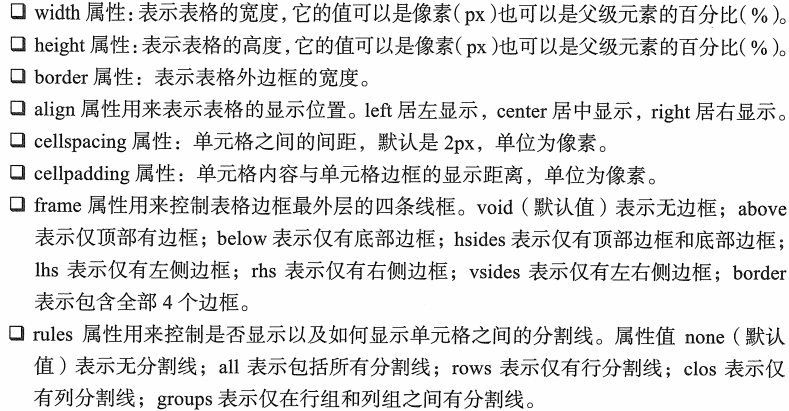


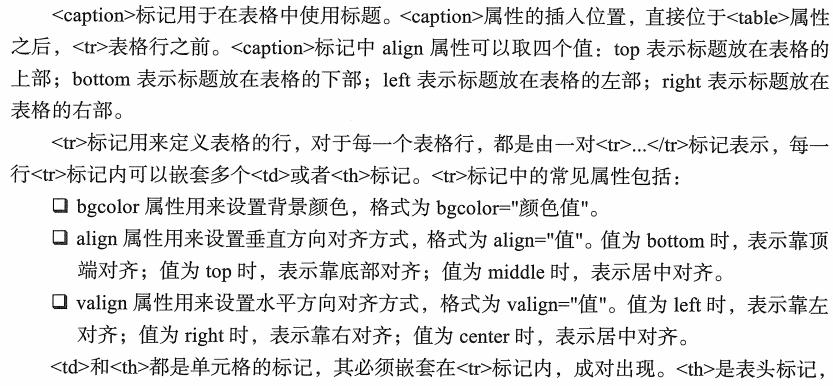


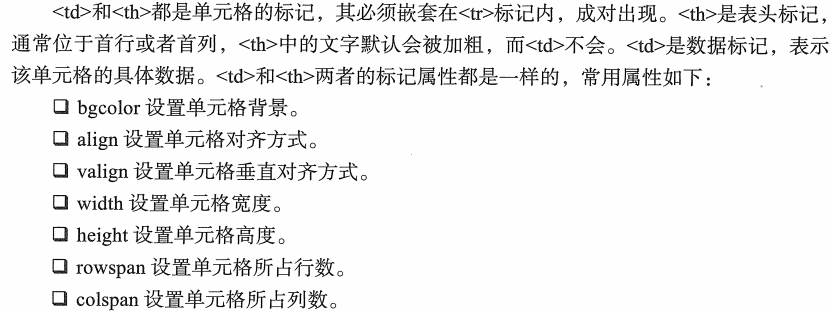








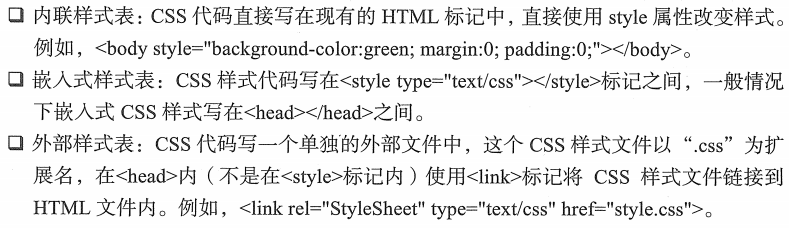




2.1.2 CSS

指层叠样式表

一般有三种：



CSS规则由两部分构成：选择器 属性

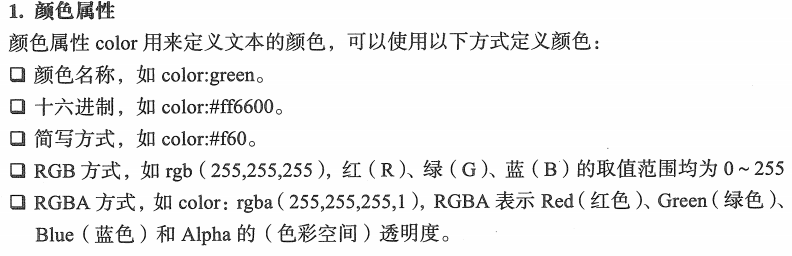
选择器：

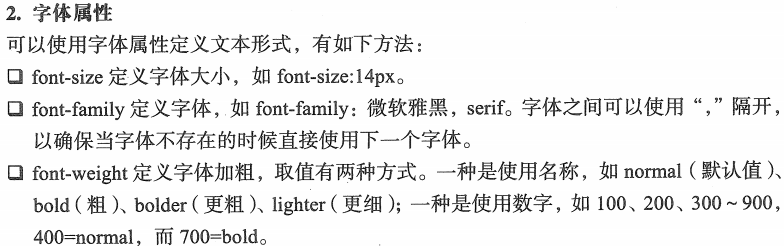
HTML标记定义：

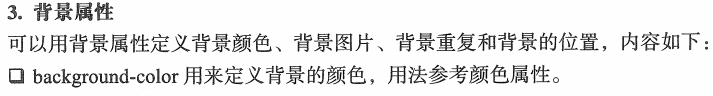
ID选择器定义：

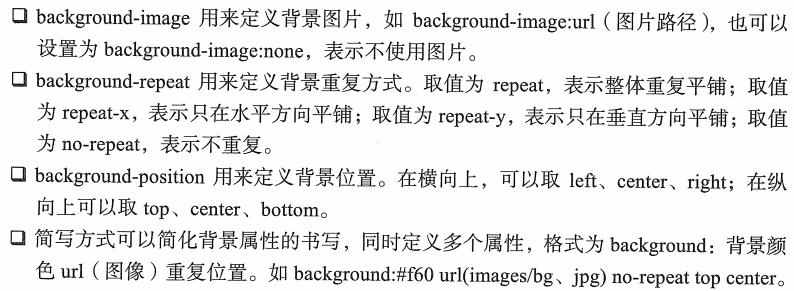
class选择器定义：

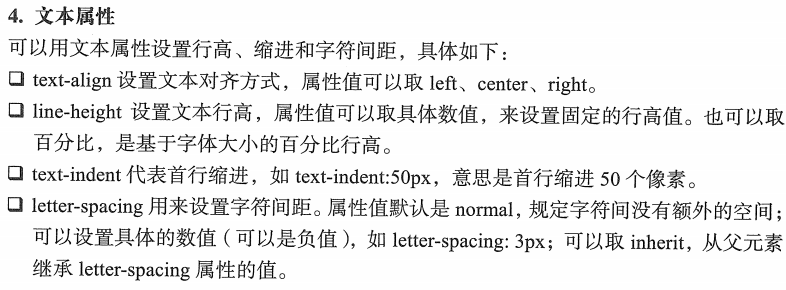
属性：

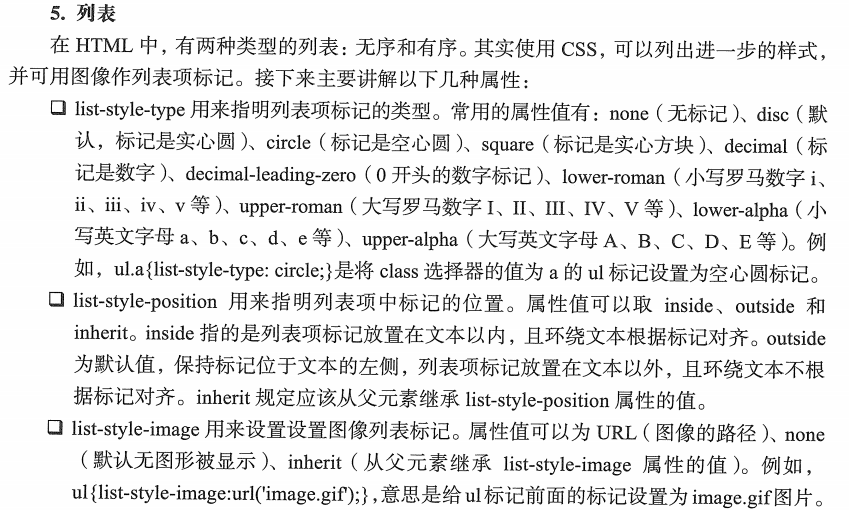






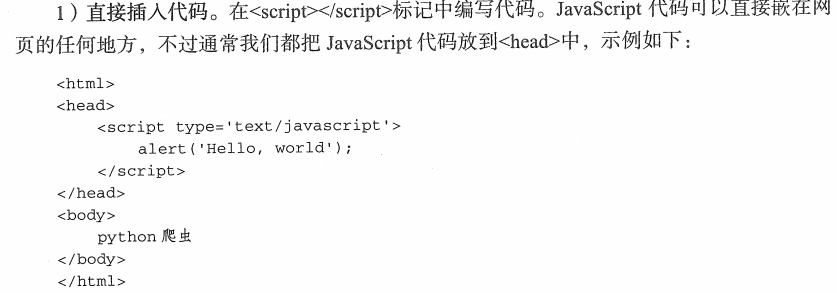


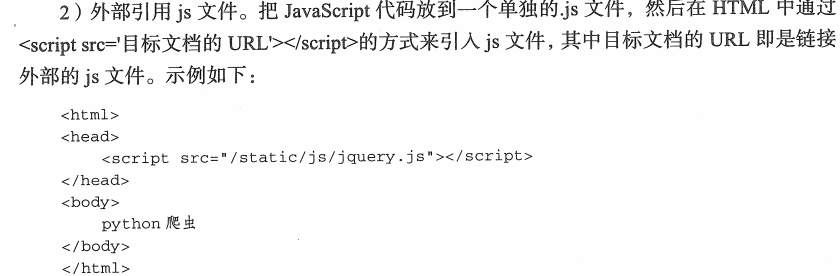




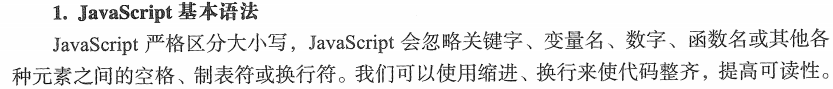
2.1.3 JavaScript

1.如何使用JavaScript –》直接插入代码 和 外部引用js文件





JavaScript基础语法：



2. 数据类型和变量

Number类型

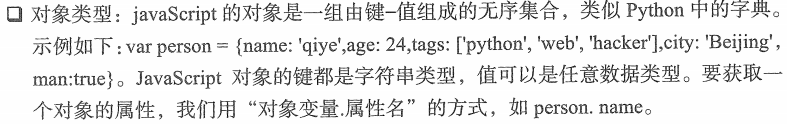
字符串类型：

布尔值类型

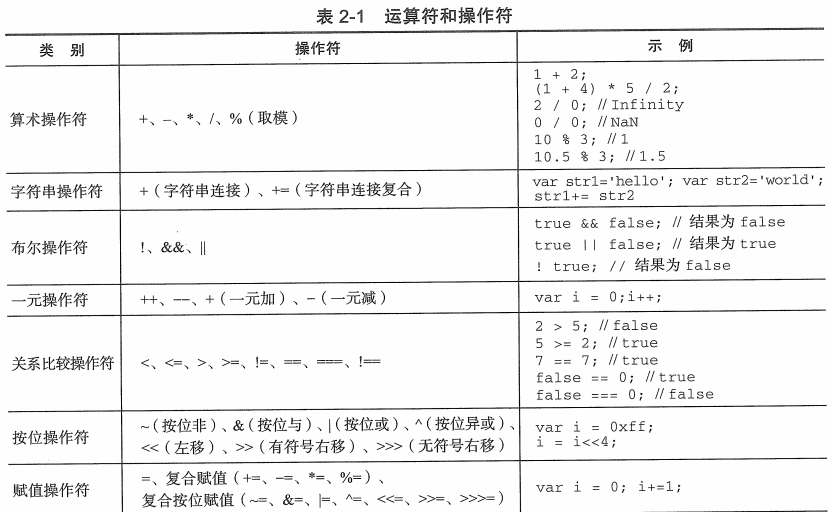
数组类型：

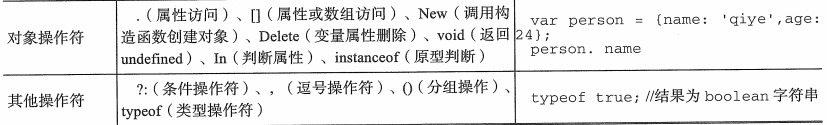






3. 运算符和操作符



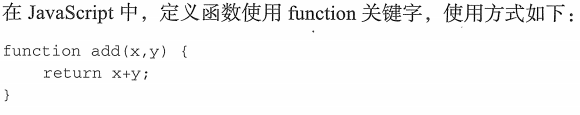


4. 条件判断

if(){..}else{..}

5. 循环

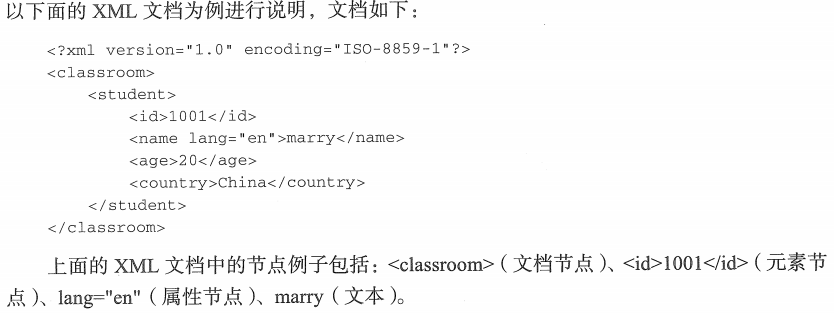
for循环

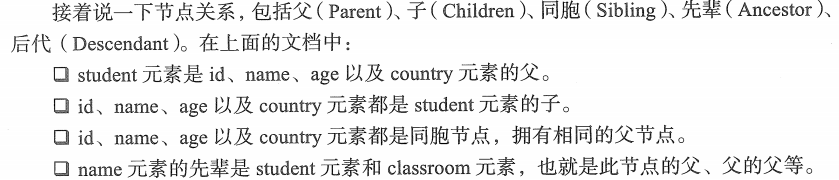
6. 函数  
 

2.1.4 Xpath

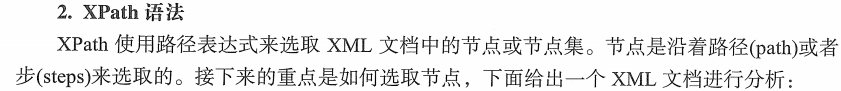
1. XPath 节点

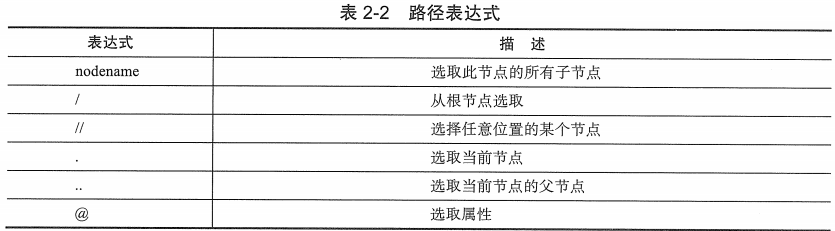


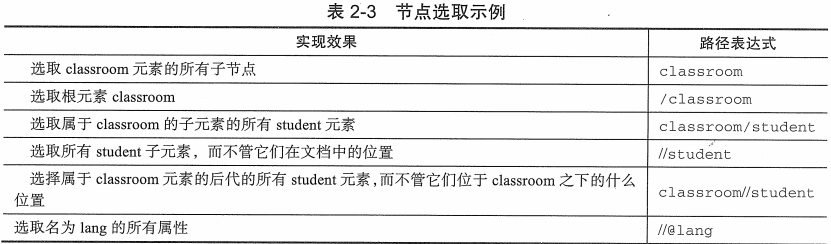


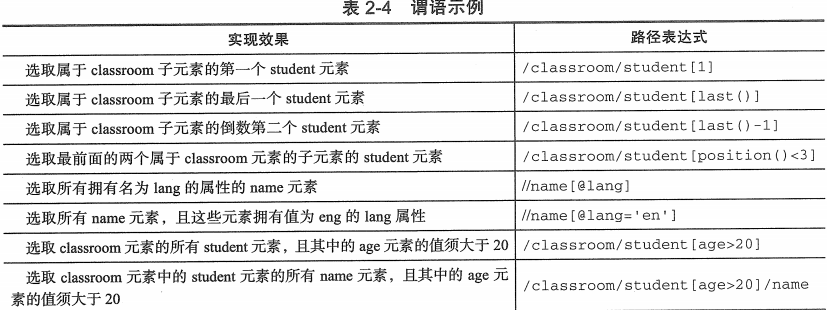


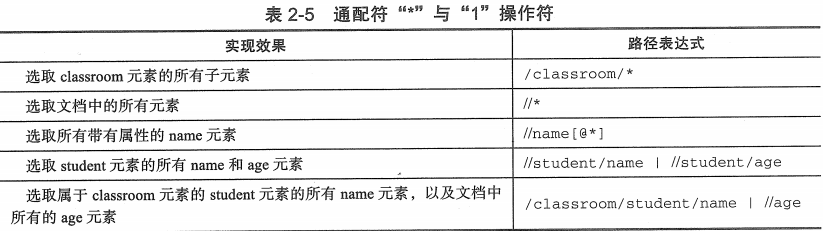












3. XPath 轴

轴定义了所选节点与当前节点之间的树关系。

相对路径：

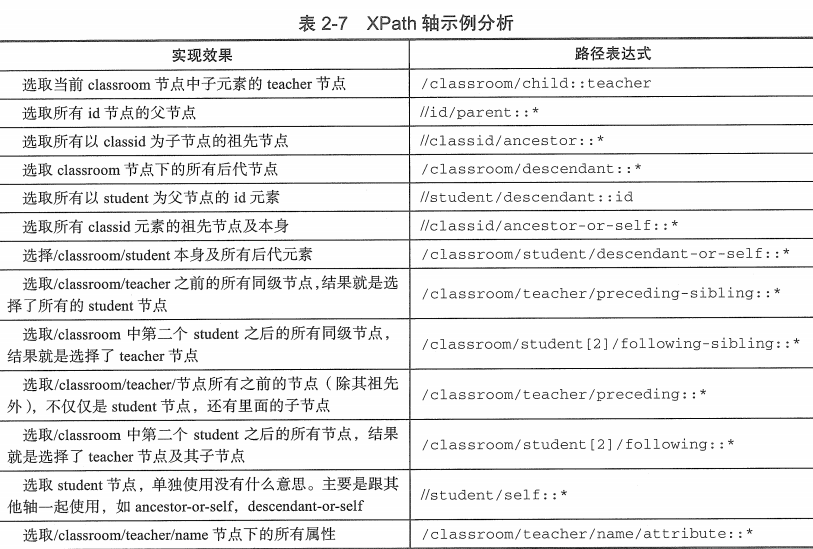
绝对路径：

步（step）: 轴（axis）, 节点测试（node-test）,零个或者更多谓语，用来更深入地提炼所选的节点集。

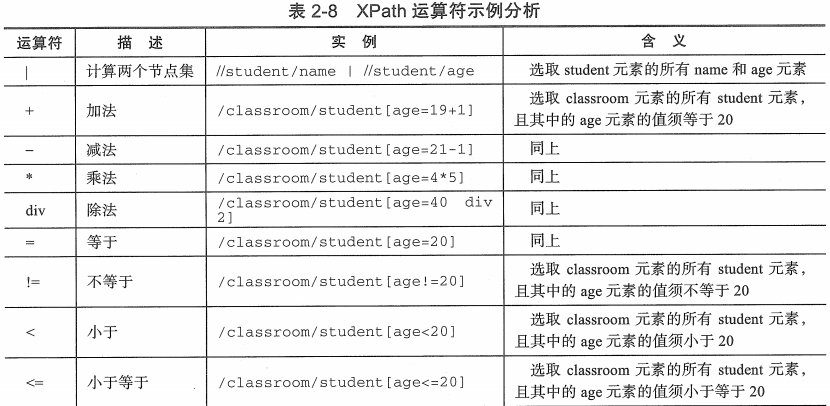
步的语法为：轴名称::节点测试[谓语]

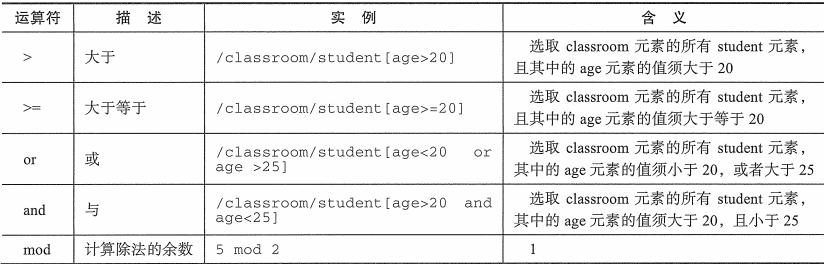






4. XPath 运算符





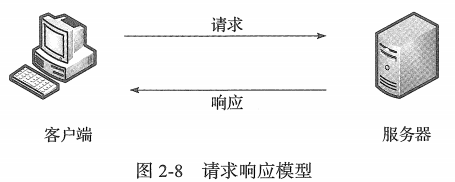
2.1.5 JSON

JSON是JavaScript对象表示法，用于存储和交换文本信息、

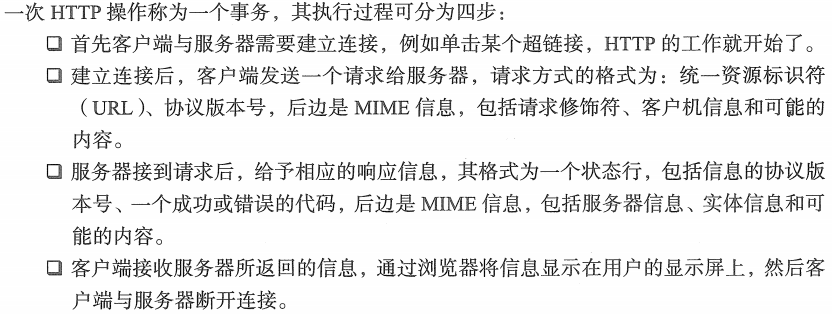
2.2 HTTP标准

HTTP协议（hyerText Transfer Protocol）

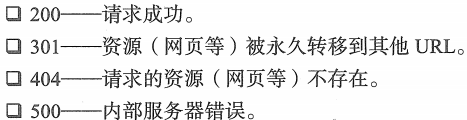
2.2.1 HTTP请求过程

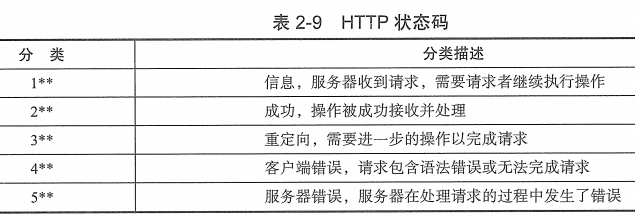






2.2.2 HTTP状态码含义





第三章： 初始网络爬虫

3.1

3.1.6 重定向

3.2.2 httplib/urllib实现



3.2.3 更人性化的Requests

下载连接：<https://github.com/kennethreitz/requests/releases>

1. 以GET请求为例，最简单的形式如下：

import requests

r = requests.get(‘http://www.baidu.com’)

` 

<http://zzk.cnblogs.com/s/blogpost?Keyload=blog%3Aqiyeboy&pageindex=1&AspxAutoDetectCookieSupport=1>

“?” 后面还有参数

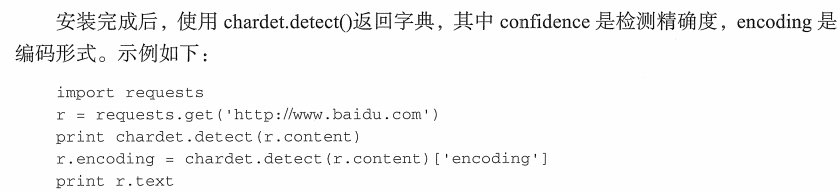
>>> import requests

>>> payload = {'Keyload':'blog:qiyeboy','pageindex':1}

>>> r = requests.get('http://zzk.cnblogs.com/s/blogpost',params=payload)



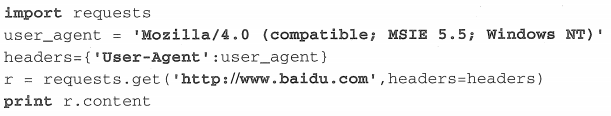


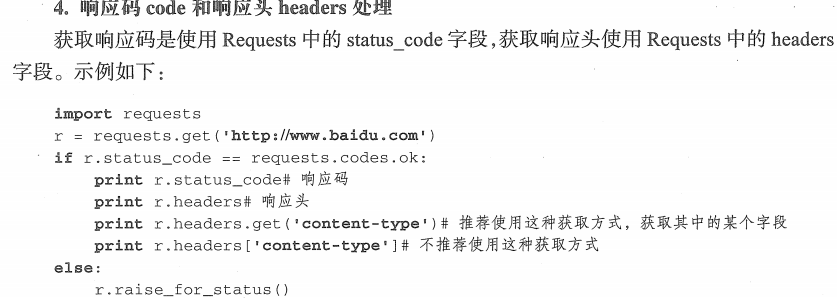


流模式：



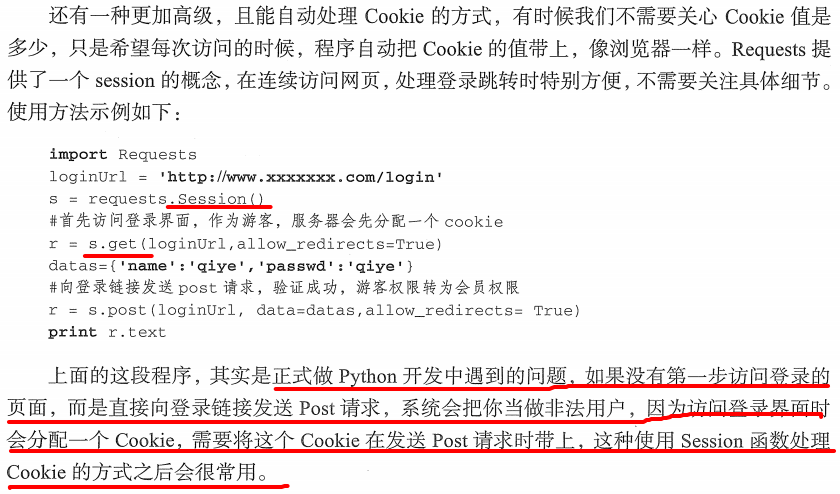
3. 请求头headers处理

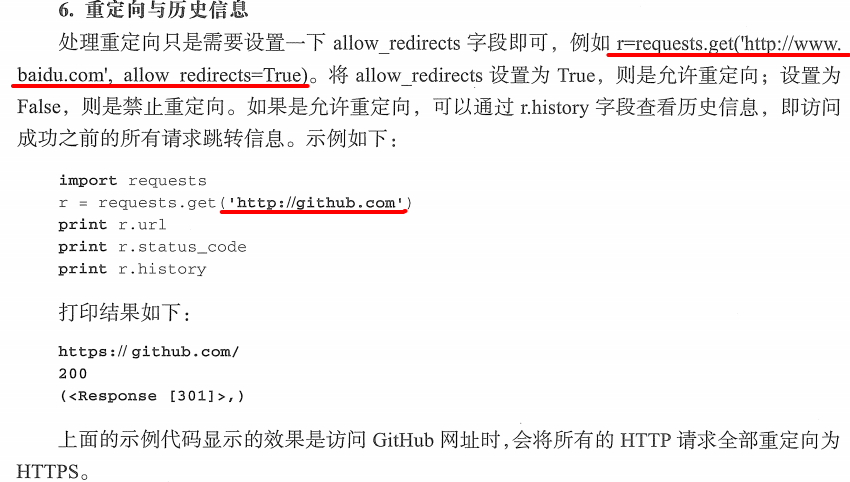




5 











第四章： HTML解析大法

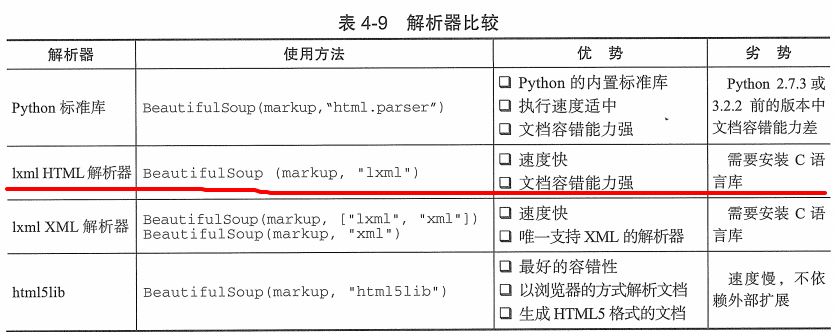
分别为Firebug工具， 正则表达式， beautiful soup

4.3 强大的BeautifulSoup

安装方式：

1）apt-get install Python-bs4

2)easy\_install beautifulsoup4



4.3 BeautifulSoup的使用

1. 快速开始

导入： from bs4 import BeautifulSoup

2.使用

1) 第一种方式：soup = BeautifulSoup(html\_str,’lxml’,from\_encoding=’utf-8’)

2）第二种方式：soup = BeautifuleSoup(open(‘index.html’))

打印soup对象的内容，格式化输出：

print soup.prettify()

第五章： 数据存储（无数据库版）

主要将HTML正文存储为：JSON 和 CSV