目录

[Python CGI编程 1](#_Toc521093262)

[Python3 MySQL 数据库连接 4](#_Toc521093263)

[Python3 多线程 8](#_Toc521093264)

[Python3 XML解析 9](#_Toc521093265)

[1.SAX (simple API for XML ) 9](#_Toc521093266)

[2.DOM(Document Object Model) 10](#_Toc521093267)

[Python3 JSON 数据解析 11](#_Toc521093268)

[Python3 日期和时间 11](#_Toc521093269)

# Python CGI编程

CGI 目前由NCSA维护，NCSA定义CGI如下：

CGI(Common Gateway Interface),通用网关接口,它是一段程序,运行在服务器上如：HTTP服务器，提供同客户端HTML页面的接口。CGI程序可以是Python脚本，PERL脚本，SHELL脚本，C或者C++程序等。

说白了，CGI的作用就是将脚本程序运行，然后返回其结果。若没有CGI程序，或其他的服务器端程序，则服务器一般只是将一个文件原模原样进行传输。

Web服务器支持及配置

设置好CGI目录：

ScriptAlias /cgi-bin/ /var/www/cgi-bin/

所有的HTTP服务器执行CGI程序都保存在一个预先配置的目录。这个目录被称为CGI目录，并按照惯例，它被命名为/var/www/cgi-bin目录。

如果你想指定其他运行CGI脚本的目录，可以修改httpd.conf配置文件，如下所示：

<Directory "/var/www/cgi-bin">

AllowOverride None

Options +ExecCGI

Order allow,deny

Allow from all

</Directory>

在 AddHandler 中添加 .py 后缀，这样我们就可以访问 .py 结尾的 python 脚本文件：

AddHandler cgi-script .cgi .pl .py

HTTP头部

HTTP头部的格式如下：

HTTP 字段名: 字段内容

以下表格介绍了CGI程序中HTTP头部经常使用的信息：

|  |  |
| --- | --- |
| 头 | 描述 |
| Content-type: | 请求的与实体对应的MIME信息。例如: Content-type:text/html |
| Expires: Date | 响应过期的日期和时间 |
| Location: URL | 用来重定向接收方到非请求URL的位置来完成请求或标识新的资源 |
| Last-modified: Date | 请求资源的最后修改时间 |
| Content-length: N | 请求的内容长度 |
| Set-Cookie: String | 设置Http Cookie |

CGI环境变量

所有的CGI程序都接收以下的环境变量，可以理解为这些就是能够在脚本中访问的CGI程序的全局常量。这些变量可以通过os.environ字典来查看：

|  |  |
| --- | --- |
| 变量名 | 描述 |
| CONTENT\_TYPE | 这个环境变量的值指示所传递来的信息的MIME类型。目前，环境变量CONTENT\_TYPE一般都是：application/x-www-form-urlencoded,他表示数据来自于HTML表单。 |
| CONTENT\_LENGTH | 如果服务器与CGI程序信息的传递方式是POST，这个环境变量即使从标准输入STDIN中可以读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。 |
| HTTP\_COOKIE | 客户机内的 COOKIE 内容。 |
| HTTP\_USER\_AGENT | 提供包含了版本数或其他专有数据的客户浏览器信息。 |
| PATH\_INFO | 这个环境变量的值表示紧接在CGI程序名之后的其他路径信息。它常常作为CGI程序的参数出现。 |
| QUERY\_STRING | 如果服务器与CGI程序信息的传递方式是GET，这个环境变量的值即使所传递的信息。这个信息经跟在CGI程序名的后面，两者中间用一个问号'?'分隔。 |
| REMOTE\_ADDR | 这个环境变量的值是发送请求的客户机的IP地址，例如上面的192.168.1.67。这个值总是存在的。而且它是Web客户机需要提供给Web服务器的唯一标识，可以在CGI程序中用它来区分不同的Web客户机。 |
| REMOTE\_HOST | 这个环境变量的值包含发送CGI请求的客户机的主机名。如果不支持你想查询，则无需定义此环境变量。 |
| REQUEST\_METHOD | 提供脚本被调用的方法。对于使用 HTTP/1.0 协议的脚本，仅 GET 和 POST 有意义。 |
| SCRIPT\_FILENAME | CGI脚本的完整路径 |
| SCRIPT\_NAME | CGI脚本的的名称 |
| SERVER\_NAME | 这是你的 WEB 服务器的主机名、别名或IP地址。 |
| SERVER\_SOFTWARE | 这个环境变量的值包含了调用CGI程序的HTTP服务器的名称和版本号。例如，上面的值为Apache/2.2.14(Unix) |

使用GET方法传输数据

GET方法发送编码后的用户信息到服务端，数据信息包含在请求页面的URL上，以"?"号分割, 如下所示：

http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2

有关 GET 请求的其他一些注释：

* GET 请求可被缓存
* GET 请求保留在浏览器历史记录中
* GET 请求可被收藏为书签
* GET 请求不应在处理敏感数据时使用
* GET 请求有长度限制
* GET 请求只应当用于取回数据

使用POST方法传递数据

使用POST方法向服务器传递数据是更安全可靠的，像一些敏感信息如用户密码等需要使用POST传输数据。

处理GET和POST传输的数据的Python代码机制相同，均为使用cgi.FieldStorage()创建的实例的成员getvalue(name)来进行数据获取。getvalue()函数同样可以获取checkbox、radio等众多组件的选取结果。一般python脚本使用print()函数将需要返回的页面的html文本输出即可。

CGI中使用Cookie

http cookie的发送是通过http头部来实现的，它早于html文体传递，头部set-cookie的语法如下：

Set-cookie:name=name;expires=date;path=path;domain=domain;secure

* name=name: 需要设置cookie的值(name不能使用";"和","号),有多个name值时用 ";" 分隔，例如：name1=name1;name2=name2;name3=name3。
* expires=date: cookie的有效期限,格式： expires="Wdy,DD-Mon-YYYY HH:MM:SS"
* path=path: 设置cookie支持的路径,如果path是一个路径，则cookie对这个目录下的所有文件及子目录生效，例如： path="/cgi-bin/"，如果path是一个文件，则cookie指对这个文件生效，例如：path="/cgi-bin/cookie.cgi"。
* domain=domain: 对cookie生效的域名，例如：domain="www.runoob.com"
* secure: 如果给出此标志，表示cookie只能通过SSL协议的https服务器来传递。
* cookie的接收是通过设置环境变量HTTP\_COOKIE来实现的，CGI程序可以通过检索该变量获取cookie信息。

检索Cookie信息

Cookie信息检索页非常简单，Cookie信息存储在CGI的环境变量HTTP\_COOKIE中，存储格式如下：

key1=value1;key2=value2;key3=value3....

服务器端调用用户本地Cookie的脚本代码片段举例：

import os

import http.cookies

#此处省略部分代码

if 'HTTP\_COOKIE' in os.environ:

cookie\_string=os.environ.get('HTTP\_COOKIE')

c=cookie.SimpleCookie()

c.load(cookie\_string)

try:

data=c['name'].value

print ("cookie data: "+data+"<br>")

except KeyError:

print ("cookie 没有设置或者已过期<br>")

文件上传

HTML设置上传文件的表单需要设置 enctype 属性为 multipart/form-data。

服务器端文件处理的脚本代码片段举例：

import cgi, os

import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# 获取文件名

fileitem = form['filename']

# 检测文件是否上传

if fileitem.filename:

# 设置文件路径

fn = os.path.basename(fileitem.filename)

open('/tmp/' + fn, 'wb').write(fileitem.file.read())

message = '文件 "' + fn + '" 上传成功'

else:

message = '文件没有上传'

如果你使用的系统是Unix/Linux，你必须替换文件分隔符，在window下只需要使用open()语句即可：

fn = os.path.basename(fileitem.filename.replace("\\", "/" ))

# Python3 MySQL 数据库连接

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2中则使用mysqldb。PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。

以下使用方法均为例子：

数据库连接

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用 cursor() 方法创建一个游标对象

cursor cursor = db.cursor()

# 使用 execute() 方法执行 SQL 查询

cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取单条数据.

data = cursor.fetchone()

print ("Database version : %s " % data)

# 关闭数据库连接

db.close()

执行以上脚本输出结果如下：

Database version : 5.5.20-log

创建数据库表

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用 cursor() 方法创建一个游标对象

cursor cursor = db.cursor()

# 使用 execute() 方法执行 SQL，如果表存在则删除

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 使用预处理语句创建表

sql = """CREATE TABLE EMPLOYEE (

FIRST\_NAME CHAR(20) NOT NULL,

LAST\_NAME CHAR(20),

AGE INT,

SEX CHAR(1),

INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接

db.close()

数据库插入操作

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标

cursor = db.cursor()

# SQL 插入语句

sql = """INSERT INTO EMPLOYEE(FIRST\_NAME,

LAST\_NAME, AGE, SEX, INCOME)

VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

try:

# 执行sql语句

cursor.execute(sql)

# 提交到数据库执行

db.commit()

except:

# 如果发生错误则回滚

db.rollback()

# 关闭数据库连接

db.close()

以上例子也可以写成如下形式：

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标

cursor = db.cursor()

# SQL 插入语句

sql = "INSERT INTO EMPLOYEE(FIRST\_NAME, \

LAST\_NAME, AGE, SEX, INCOME) \

VALUES ('%s', '%s', '%d', '%c', '%d' )" % \

('Mac', 'Mohan', 20, 'M', 2000)

try:

# 执行sql语句

cursor.execute(sql)

# 执行sql语句

db.commit()

except:

# 发生错误时回滚

db.rollback()

# 关闭数据库连接

db.close()

数据库查询操作

Python查询Mysql使用 fetchone() 方法获取单条数据, 使用fetchall() 方法获取多条数据。

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标

cursor = db.cursor()

# SQL 查询语句

sql = "SELECT \* FROM EMPLOYEE \ WHERE INCOME > '%d'" % (1000) try: # 执行SQL语句 cursor.execute(sql) # 获取所有记录列表 results = cursor.fetchall() for row in results: fname = row[0] lname = row[1] age = row[2] sex = row[3] income = row[4] # 打印结果 print ("fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \ (fname, lname, age, sex, income )) except: print ("Error: unable to fetch data") # 关闭数据库连接 db.close()

以上脚本执行结果如下：

fname=Mac, lname=Mohan, age=20, sex=M, income=2000

数据库更新操作

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标

cursor = db.cursor()

# SQL 更新语句

sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')

try:

# 执行SQL语句

cursor.execute(sql)

# 提交到数据库执行

db.commit()

except:

# 发生错误时回滚

db.rollback()

# 关闭数据库连接

db.close()

删除操作

#!/usr/bin/python3

import pymysql

# 打开数据库连接

db = pymysql.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标

cursor = db.cursor()

# SQL 删除语句

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

# 执行SQL语句

cursor.execute(sql)

# 提交修改

db.commit()

except:

# 发生错误时回滚

db.rollback()

# 关闭连接

db.close()

执行事务

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

* 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
* 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
* 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
* 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

实例(Python 3.0+)：

# SQL删除记录语句

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

# 执行SQL语句

cursor.execute(sql)

# 向数据库提交

db.commit()

except:

# 发生错误时回滚

db.rollback()

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

错误处理

DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常:

|  |  |
| --- | --- |
| 异常 | 描述 |
| Warning | 当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。 |
| Error | 警告以外所有其他错误类。必须是 StandardError 的子类。 |
| InterfaceError | 当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是Error的子类。 |
| DatabaseError | 和数据库有关的错误发生时触发。 必须是Error的子类。 |
| DataError | 当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是DatabaseError的子类。 |
| OperationalError | 指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。 必须是DatabaseError的子类。 |
| IntegrityError | 完整性相关的错误，例如外键检查失败等。必须是DatabaseError子类。 |
| InternalError | 数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是DatabaseError子类。 |
| ProgrammingError | 程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、 参数数量错误等等。必须是DatabaseError的子类。 |
| NotSupportedError | 不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上 使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。 必须是DatabaseError的子类。 |

# Python3 多线程

Python3 线程中常用的两个模块为：

* \_thread
* threading(推荐使用)

thread 模块已被废弃。用户可以使用 threading 模块代替。所以，在 Python3 中不能再使用"thread" 模块。为了兼容性，Python3 将 thread 重命名为 "\_thread"。

Python中使用线程有两种方式：函数或者用类来包装线程对象。

①函数式：调用 \_thread 模块中的start\_new\_thread()函数来产生新线程。语法如下:

\_thread.start\_new\_thread ( function, args[, kwargs] )

参数说明:

* function - 线程函数。
* args - 传递给线程函数的参数,他必须是个tuple类型。
* kwargs - 可选参数。

②线程对象类：我们可以通过直接从 threading.Thread 继承创建一个新的子类，并实例化后调用 start() 方法启动新线程，即它调用了线程的 run() 方法。

线程模块

Python3 通过两个标准库 \_thread 和 threading 提供对线程的支持。

\_thread 提供了低级别的、原始的线程以及一个简单的锁，它相比于 threading 模块的功能还是比较有限的。

threading 模块除了包含 \_thread 模块中的所有方法外，还提供的其他方法：

* threading.currentThread(): 返回当前的线程变量。
* threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
* threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。

除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法:

* run(): 用以表示线程活动的方法。
* start():启动线程活动。
* join([time]): 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
* isAlive(): 返回线程是否活动的。
* getName(): 返回线程名。
* setName(): 设置线程名。

线程同步

使用 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 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。

线程安全队列（ queue）

Python 的 queue 模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列 PriorityQueue。

这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

Queue 类中的常用方法:

* Queue.qsize() 返回队列的大小
* Queue.empty() 如果队列为空，返回True,反之False
* Queue.full() 如果队列满了，返回True,反之False
* Queue.full 与 maxsize 大小对应
* Queue.get([block[, timeout]])获取队列，timeout等待时间
* Queue.get\_nowait() 相当Queue.get(False)
* Queue.put(item) 写入队列，timeout等待时间
* Queue.put\_nowait(item) 相当Queue.put(item, False)
* Queue.task\_done() 在完成一项工作之后，Queue.task\_done()函数向任务已经完成的队列发送一个信号
* Queue.join() 实际上意味着等到队列为空，再执行别的操作

# Python3 XML解析

常见的XML编程接口有DOM和SAX，这两种接口处理XML文件的方式不同，当然使用场合也不同。python有三种方法解析XML，SAX，DOM，以及ElementTree:

## 1.SAX (simple API for XML )

python 标准库包含SAX解析器，SAX用事件驱动模型，通过在解析XML的过程中触发一个个的事件并调用用户定义的回调函数来处理XML文件。

利用SAX解析XML文档牵涉到两个部分:解析器和事件处理器。解析器负责读取XML文档,并向事件处理器发送事件,如元素开始跟元素结束事件;而事件处理器则负责对事件作出相应,对传递的XML数据进行处理。

在python中使用sax方式处理xml要先引入xml.sax中的make\_parser函数来构建解析器，还有xml.sax.handler中的ContentHandler用于构建事件处理器。

make\_parser方法

以下方法创建一个新的解析器对象并返回。

xml.sax.make\_parser( [parser\_list] )

参数说明:

* parser\_list - 可选参数，解析器列表

parser方法

以下方法为解析器对象成员函数用以设定事件处理器和解析xml文档：

parse( xmlfile, contenthandler[, errorhandler])

参数说明:

* xmlfile - xml文件名
* contenthandler - 必须是一个ContentHandler的对象
* errorhandler - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

parseString方法

parseString方法创建一个XML解析器并解析xml字符串：

xml.sax.parseString(xmlstring, contenthandler[, errorhandler])

参数说明:

* xmlstring - xml字符串
* contenthandler - 必须是一个ContentHandler的对象
* errorhandler - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象

ContentHandler类方法介绍

该类用于派生构建自定义的事件处理器。

characters(content)方法

调用时机：

从行开始，遇到标签之前，存在字符，content的值为这些字符串。

从一个标签，遇到下一个标签之前， 存在字符，content的值为这些字符串。

从一个标签，遇到行结束符之前，存在字符，content的值为这些字符串。

标签可以是开始标签，也可以是结束标签。

startDocument()方法

文档启动的时候调用。

endDocument()方法

解析器到达文档结尾时调用。

startElement(name, attrs)方法

遇到XML开始标签时调用，name是标签的名字，attrs是标签的属性值字典。

endElement(name)方法

遇到XML结束标签时调用。

完整的 SAX API 文档请查阅[Python SAX APIs](http://docs.python.org/library/xml.sax.html" \t "_blank)

## 2.DOM(Document Object Model)

将XML数据在内存中解析成一个树，通过对树的操作来操作XML。

文件对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展置标语言的标准编程接口。

一个 DOM 的解析器在解析一个 XML 文档时，一次性读取整个文档，把文档中所有元素保存在内存中的一个树结构里，之后你可以利用DOM 提供的不同的函数来读取或修改文档的内容和结构，也可以把修改过的内容写入xml文件。

python中用xml.dom.minidom来解析xml文件

完整的 DOM API 文档请查阅[Python DOM APIs](http://docs.python.org/library/xml.dom.html" \t "_blank)。

# Python3 JSON 数据解析

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于ECMAScript的一个子集。

Python3 中可以使用 json 模块来对 JSON 数据进行编解码，它包含了两个函数：

* json.dumps(): 对数据进行编码。
* json.loads(): 对数据进行解码。

在json的编解码过程中，python 的原始类型与json类型会相互转换，通过输出的结果可以看出，简单类型通过编码后跟其原始的repr()输出结果非常相似。

具体的转化对照如下：

Python 编码为 JSON 类型转换对应表：

|  |  |
| --- | --- |
| Python | JSON |
| dict | object |
| list, tuple | array |
| str | string |
| int, float, int- & float-derived Enums | number |
| True | true |
| False | false |
| None | null |

JSON 解码为 Python 类型转换对应表：

|  |  |
| --- | --- |
| JSON | Python |
| object | dict |
| array | list |
| string | str |
| number (int) | int |
| number (real) | float |
| true | True |
| false | False |
| null | None |

# Python3 日期和时间

Python 提供了一个 time 和 calendar 模块可以用于格式化日期和时间。时间间隔是以秒为单位的浮点小数。每个时间戳都以自从1970年1月1日午夜（历元）经过了多长时间来表示。

时间元组

很多Python函数用一个元组封装起来的9组数字处理时间，上述也就是struct\_time元组。这种结构具有如下属性：

|  |  |  |
| --- | --- | --- |
| 序号 | 属性 | 值 |
| 0 | tm\_year | 2008 |
| 1 | tm\_mon | 1 到 12 |
| 2 | tm\_mday | 1 到 31 |
| 3 | tm\_hour | 0 到 23 |
| 4 | tm\_min | 0 到 59 |
| 5 | tm\_sec | 0 到 61 (60或61 是闰秒) |
| 6 | tm\_wday | 0到6 (0是周一) |
| 7 | tm\_yday | 一年中的第几天，1 到 366 |
| 8 | tm\_isdst | 是否为夏令时，值有：1(夏令时)、0(不是夏令时)、-1(未知)，默认 -1 |

格式化日期

我们可以使用 time 模块的 strftime 方法来格式化日期：

time.strftime(format[, t])

python中时间日期格式化符号：

* %y 两位数的年份表示（00-99）
* %Y 四位数的年份表示（000-9999）
* %m 月份（01-12）
* %d 月内中的一天（0-31）
* %H 24小时制小时数（0-23）
* %I 12小时制小时数（01-12）
* %M 分钟数（00=59）
* %S 秒（00-59）
* %a 本地简化星期名称
* %A 本地完整星期名称
* %b 本地简化的月份名称
* %B 本地完整的月份名称
* %c 本地相应的日期表示和时间表示
* %j 年内的一天（001-366）
* %p 本地A.M.或P.M.的等价符
* %U 一年中的星期数（00-53）星期天为星期的开始
* %w 星期（0-6），星期天为星期的开始
* %W 一年中的星期数（00-53）星期一为星期的开始
* %x 本地相应的日期表示
* %X 本地相应的时间表示
* %Z 当前时区的名称
* %% %号本身

Time 模块

|  |  |
| --- | --- |
| 序号 | 函数及描述 |
| 1 | time.altzone 返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。对夏令时启用地区才能使用。 |
| 2 | time.asctime([tupletime]) 接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日 周二18时07分14秒）的24个字符的字符串。 |
| 3 | [time.clock()](http://www.runoob.com/python3/python3-att-time-clock.html) 用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。 |
| 4 | time.ctime([secs]) 作用相当于asctime(localtime(secs))，未给参数相当于asctime() |
| 5 | time.gmtime([secs]) 接收时间辍（1970纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组t。注：t.tm\_isdst始终为0 |
| 6 | time.localtime([secs] 接收时间辍（1970纪元后经过的浮点秒数）并返回当地时间下的时间元组t（t.tm\_isdst可取0或1，取决于当地当时是不是夏令时）。 |
| 7 | [time.mktime(tupletime)](http://www.runoob.com/python3/python3-att-time-mktime.html) 接受时间元组并返回时间辍（1970纪元后经过的浮点秒数）。 |
| 8 | time.sleep(secs) 推迟调用线程的运行，secs指秒数。 |
| 9 | time.strftime(fmt[,tupletime]) 接收以时间元组，并返回以可读字符串表示的当地时间，格式由fmt决定。 |
| 10 | time.strptime(str,fmt='%a %b %d %H:%M:%S %Y') 根据fmt的格式把一个时间字符串解析为时间元组。 |
| 11 | time.time( ) 返回当前时间的时间戳（1970纪元后经过的浮点秒数）。 |
| 12 | [time.tzset()](http://www.runoob.com/python3/python3-att-time-tzset.html) 根据环境变量TZ重新初始化时间相关设置。 |

Time模块包含了以下2个非常重要的属性：

|  |  |
| --- | --- |
| 序号 | 属性及描述 |
| 1 | time.timezone 属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。 |
| 2 | time.tzname 属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。 |

日历（Calendar）模块

|  |  |
| --- | --- |
| 序号 | 函数及描述 |
| 1 | calendar.calendar(year,w=2,l=1,c=6) 返回一个多行字符串格式的year年年历，3个月一行，间隔距离为c。 每日宽度间隔为w字符。每行长度为21\* W+18+2\* C。l是每星期行数。 |
| 2 | calendar.firstweekday( ) 返回当前每周起始日期的设置。默认情况下，首次载入caendar模块时返回0，即星期一。 |
| 3 | calendar.isleap(year) 是闰年返回True，否则为false。 |
| 4 | calendar.leapdays(y1,y2) 返回在Y1，Y2两年之间的闰年总数。 |
| 5 | calendar.month(year,month,w=2,l=1) 返回一个多行字符串格式的year年month月日历，两行标题，一周一行。每日宽度间隔为w字符。每行的长度为7\* w+6。l是每星期的行数。 |
| 6 | calendar.monthcalendar(year,month) 返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。Year年month月外的日期都设为0;范围内的日子都由该月第几日表示，从1开始。 |
| 7 | calendar.monthrange(year,month) 返回两个整数。第一个是该月的星期几的日期码，第二个是该月的日期码。日从0（星期一）到6（星期日）;月从1到12。 |
| 8 | calendar.prcal(year,w=2,l=1,c=6) 相当于 print calendar.calendar(year,w,l,c). |
| 9 | calendar.prmonth(year,month,w=2,l=1) 相当于 print calendar.calendar（year，w，l，c）。 |
| 10 | calendar.setfirstweekday(weekday) 设置每周的起始日期码。0（星期一）到6（星期日）。 |
| 11 | calendar.timegm(tupletime) 和time.gmtime相反：接受一个时间元组形式，返回该时刻的时间辍（1970纪元后经过的浮点秒数）。 |
| 12 | calendar.weekday(year,month,day) 返回给定日期的日期码。0（星期一）到6（星期日）。月份为 1（一月） 到 12（12月）。 |