IO编程

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用Python内置的`open()`函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，`open()`函数就会抛出一个`IOError`的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'/Users/michael/notfound.txt'
```

如果文件打开成功，接下来，调用`read()`方法可以一次读取文件的全部内容，Python把内容读到内存，用一个`str`对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用`close()`方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生`IOError`，一旦出错，后面的`f.close()`就不会调用。所以，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。

调用`read()`会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用`read(size)`方法，每次最多读取size个字节的内容。另外，调用`readline()`可以每次读取一行内容，调用`readlines()`一次读取所有内容并按行返回`list`。因此，要根据需要决定怎么调用。

如果文件很小，`read()`一次性读取最方便；如果不能确定文件大小，反复调用`read(size)`比较保险；如果是配置文件，调用`readlines()`最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像`open()`函数返回的这种有个`read()`方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个`read()`方法就行。

`StringIO`就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用`'rb'`模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

字符编码

要读取非UTF-8编码的文本文件，需要给`open()`函数传入`encoding`参数，例如，读取GBK编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'r',
encoding='gbk')
>>> f.read()
'测试'
```

遇到有些编码不规范的文件，你可能会遇到`UnicodeDecodeError`，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，`open()`函数还接收一个`errors`参数，表示如果遇到编码错误后如何处理。最简单的方式是直接忽略：

```
>>> f = open('/Users/michael/gbk.txt', 'r',
encoding='gbk', errors='ignore')
```

写文件

写文件和读文件是一样的，唯一区别是调用`open()`函数时，传入标识符`'w'`或者`'wb'`表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用`write()`来写入文件，但是务必要调用`f.close()`来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用`close()`方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用`close()`的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用`with`语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请给`open()`函数传入`encoding`参数，将字符串自动转换成指定编码。

细心的童鞋会发现，以`'w'`模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入`'a'`以追加（append）模式写入。

所有模式的定义及含义可以参考Python的官方文档。

练习

请将本地一个文本文件读为一个str并打印出来：

```
# -*- coding: utf-8 -*-
fpath = r'C:\Windows\system.ini'

with open(fpath, 'r') as f:
    s = f.read()
    print(s)
```

小结

- 在Python中，文件读写是通过`open()`函数打开的文件对象完成的。使用`with`语句操作文件IO是个好习惯。

`StringIO`和`BytesIO`

`StringIO`

很多时候，数据读写不一定是文件，也可以在内存中读写。

`StringIO`顾名思义就是在内存中读写str。

要把str写入`StringIO`，我们需要先创建一个`StringIO`，然后，像文件一样写入即可：

```
>>> from io import `StringIO`
>>> f = `StringIO`()
>>> f.write('hello')
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue())
hello world!
```

`getvalue()`方法用于获得写入后的str。

要读取`StringIO`，可以用一个str初始化`StringIO`，然后，像读文件一样读取：

```
>>> from io import `StringIO`
>>> f = `StringIO`('Hello!\nHi!\nGoodbye!')
>>> while True:
...     s = f.readline()
...     if s == '':
...         break
...     print(s.strip())
...
Hello!
Hi!
Goodbye!
```

`BytesIO`

`StringIO`操作的只能是str，如果要操作二进制数据，就需要使用`BytesIO`。

`BytesIO`实现了在内存中读写bytes，我们创建一个`BytesIO`，然后写入一些bytes：

```
>>> from io import `BytesIO`
>>> f = `BytesIO`()
>>> f.write('中文'.encode('utf-8'))
6
>>> print(f.getvalue())
b'\xe4\xb8\xad\xe6\x96\x87'
```

请注意，写入的不是str，而是经过UTF-8编码的bytes。

和`StringIO`类似，可以用一个bytes初始化`BytesIO`，然后，像读文件一样读取：

```
>>> from io import `BytesIO`
>>> f = `BytesIO`(b'\xe4\xb8\xad\xe6\x96\x87')
>>> f.read()
b'\xe4\xb8\xad\xe6\x96\x87'
```

小结

`StringIO`和`BytesIO`是在内存中操作str和bytes的方法，使得和读写文件具有一致的接口。

操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如`dir`、`cp`等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的`os`模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用`os`模块的基本功能：

```
>>> import os
>>> os.name # 操作系统类型
'posix'
```

如果是`posix`，说明系统是`Linux`、`Unix`或`Mac OS X`，如果是`nt`，就是`Windows`系统。

要获取详细的系统信息，可以调用`uname()`函数：

```
>>> os.uname()
posix.uname_result(sysname='Darwin',
nodename='MichaelMacPro.local', release='14.3.0',
version='Darwin Kernel Version 14.3.0: Mon Mar 23 11:59:05
PDT 2015; root:xnu-2782.20.48~5/RELEASE_X86_64',
machine='x86_64')
```

注意`uname()`函数在Windows上不提供，也就是说，`os`模块的某些函数是跟操作系统相关的。

环境变量

在操作系统中定义的环境变量，全部保存在`os.environ`这个变量中，可以直接查看：

```
>>> os.environ
environ({'VERSIONER_PYTHON_PREFER_32_BIT': 'no',
'TERM_PROGRAM_VERSION': '326', 'LOGNAME': 'michael',
'USER': 'michael', 'PATH':
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin
:/usr/local/mysql/bin', ...})
```

要获取某个环境变量的值，可以调用`os.environ.get('key')`：

```
>>> os.environ.get('PATH')
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin
:/usr/local/mysql/bin'
>>> os.environ.get('x', 'default')
'default'
```

操作文件和目录

操作文件和目录的函数一部分放在`os`模块中，一部分放在`os.path`模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

```
# 查看当前目录的绝对路径:
>>> os.path.abspath('.')
'/Users/michael'
# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:
>>> os.path.join('/Users/michael', 'testdir')
'/Users/michael/testdir'
# 然后创建一个目录:
>>> os.mkdir('/Users/michael/testdir')
# 删掉一个目录:
>>> os.rmdir('/Users/michael/testdir')
```

把两个路径合成一个时，不要直接拼字符串，而要通过`os.path.join()`函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，`os.path.join()`返回这样的字符串：

```
part-1/part-2
```

而Windows下会返回这样的字符串：

```
part-1\part-2
```

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过`os.path.split()`函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

`os.path.splitext()`可以直接让你得到文件扩展名，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个`test.txt`文件：

```
# 对文件重命名:
>>> os.rename('test.txt', 'test.py')
# 删掉文件:
>>> os.remove('test.py')
```

但是复制文件的函数居然在`os`模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是`shutil`模块提供了`copyfile()`的函数，你还可以在`shutil`模块中找到很多实用函数，它们可以看做是`os`模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash',
'.vim', 'Applications', 'Desktop', ...]
```

要列出所有的`.py`文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and
os.path.splitext(x)[1]=='.py']
['apis.py', 'config.py', 'models.py', 'pymonitor.py',
'test_db.py', 'urls.py', 'wsgiapp.py']
```

是不是非常简洁？

小结

- Python的`os`模块封装了操作系统的目录和文件操作，要注意这些函数有的在`os`模块中，有的在`os.path`模块中。

练习

1. 利用`os`模块编写一个能实现`dir -l`输出的程序。
2. 编写一个程序，能在当前目录以及当前目录的所有子目录下查找文件名包含指定字符串的文件，并打印出相对路径。

序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个dict：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把`name`改成`'Bill'`，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的`'Bill'`存储到磁盘上，下次重新运行程序，变量又被初始化为`'Bob'`。

我们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。

首先，我们尝试把一个对象序列化并写入文件：

```
>>> import pickle
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x
00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\
x00Bobq\x04u.'
```

`pickle.dumps()`方法把任意对象序列化成一个`bytes`，然后，就可以把这个`bytes`写入文件。或者用另一个方法`pickle.dump()`直接把对象序列化后写入一个file-like Object：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的`dump.txt`文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个`bytes`，然后用`pickle.loads()`方法反序列化出对象，也可以直接用`pickle.load()`方法从一个`file-like Object`中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	PYTHON类型
{}	dict
[]	list
"string"	str
1234.56	int或float
true/false	True/False
null	None

Python内置的`json`模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`dumps()`方法返回一个`str`，内容就是标准的JSON。类似的，`dump()`方法可以直接把JSON写入一个`file-like Object`。

要把JSON反序列化为Python对象，用`loads()`或者对应的`load()`方法，前者把JSON的字符串反序列化，后者从`file-like Object`中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的`str`与JSON的字符串之间转换。

JSON进阶

Python的`dict`对象可以直接序列化为JSON的`{}`，不过，很多时候，我们更喜欢用`class`表示对象，比如定义`Student`类，然后序列化：

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个`TypeError`：

```
Traceback (most recent call last):
  ...
TypeError: <__main__.Student object at 0x10603cc50> is not
JSON serializable
```

错误的原因是`Student`对象不是一个可序列化为JSON的对象。

如果连`class`的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看`dumps()`方法的参数列表，可以发现，除了第一个必须的`obj`参数外，`dumps()`方法还提供了一大堆的可选参数：

https://docs.python.org/3/library/json.html#json.dumps

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把`Student`类实例序列化为JSON，是因为默认情况下，`dumps()`方法不知道如何将`Student`实例变为一个JSON的`{}`对象。

可选参数`default`就是把任意一个对象变成一个可序列为JSON的对象，我们只需要为`Student`专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }
```

这样，`Student`实例首先被`student2dict()`函数转换成`dict`，然后再被顺利序列化为JSON：

```
>>> print(json.dumps(s, default=student2dict))
{"age": 20, "name": "Bob", "score": 88}
```

不过，下次如果遇到一个`Teacher`类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意`class`的实例变为`dict`：

```
print(json.dumps(s, default=lambda obj: obj.__dict__))
```

因为通常`class`的实例都有一个`__dict__`属性，它就是一个`dict`，用来存储实例变量。也有少数例外，比如定义了`__slots__`的class。

同样的道理，如果我们要把JSON反序列化为一个`Student`对象实例，`loads()`方法首先转换出一个`dict`对象，然后，我们传入的`object_hook`函数负责把`dict`转换为`Student`实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])
```

运行结果如下：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> print(json.loads(json_str, object_hook=dict2student))
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的`Student`实例对象。

练习

对中文进行JSON序列化时，`json.dumps()`提供了一个`ensure_ascii`参数，观察该参数对结果的影响：

```
# -*- coding: utf-8 -*-

import json
obj = dict(name='小明', age=20)
s = json.dumps(obj, ensure_ascii=True)
```

```
print(s)
```

小结

- Python语言特定的序列化模块是`pickle`，但如果要把序列化搞得更通用、更符合Web标准，就可以使用`json`模块。

`json`模块的`dumps()`和`loads()`函数是定义得非常好的接口的典范。当我们使用时，只需要传入一个必须的参数。但是，当默认的序列化或反序列机制不满足我们的要求时，我们又可以传入更多的参数来定制序列化或反序列化的规则，既做到了接口简单易用，又做到了充分的扩展性和灵活性。