Python的数据的持久性

Python对象持久性的模块

1. pickle/cPickle
2. copy_reg
3. shelve
4. marshal
5. anydbm
6. dbm
7. gdbm
8. sqllite3

什么是序列化

维基百科上说:在数据储存与传送的部分是指将一个对象存储至一个储存媒介，例如档案或是记忆体缓冲等，或者透过网络传送资料时进行编码的过程，可以是字节或是XML等格式。而字节的或XML编码格式可以还原完全相等的对象。这程序被应用在不同应用程序之间传送对象，以及服务器将对象储存到档案或数据库。相反的过程又称为反序列化。

序列化的优点：

1. 一个简单和坚固的方法使对象持续.
2. 一个发型远程过程调用的方法, 例如在SOAP内的.
3. 一个分发对象的方法.

Python序列化的方式:

Python的序列化一个结构可以采用Pickle(cPickle),Json, Xml等方式.

Pickle

Pickle与cPickle

cPickle是C语言实现的Pickle,比Pickle处理起来高效.但cPickle不支持Pickler()与Unpicker()子类,因为cPickle全部是函数组成.

多数情况下,使用cPickle就可以 解决问题了.

Pickle与marshal的区别

1. Pickle可以跟踪一个对象是否被序列化, 如果已经被序列化,再后面引用的时候,不再序列化同一个对象.
2. Pickle支持自定义的类的序列化,但必须在类定义的相同环境下进行使用.
3. marshal是不稳定序列化.

Pickle数据格式

Pickle进行的序列化数据,一般不会被别的程序正确反序列化的.

Pickle默认的情况下,使用ASCII进行序列化,效率没有二进制是形式高

三种不同序列化协议:

1. 0,普通ascii的序列化,向前兼容老版本的Python.
2. 1,旧的二进制序列化方式,兼容老版本.
3. 2,新的二进制序列化方式,Python 2.3 开始, 不兼容.

用法

在序列化一个一对显示, 第一次时,先使用`dump()`方法,在反序列化时使用`load()`.

相关方法

- `pickle.dump(obj, file[, protocol])`: 将obj序列化写入一个文件中,`file`: 一个拥有 write() 方法的对象，并且这个 write() 方法能接收一个字符串作为参数。这个对象可以是一个以写模式打开的文件对象或者一个 `StringIO` 对象，或者其他自定义的满足条件的对象。`protocol`: 这是一个可选的参数，默认为 0 ， 如果设置为 1 或 True，则以高压缩的二进制格式保存持久化后的对象，否则以ASCII格式保存。
- `pickle.load(file)` : 从file中读取一个字符串,将其翻序列化为相应的结构,这个函数等同于`Unpickler(file).load()`

`file`，对应于上面 dump 方法中的 file 参数。这个 file 必须是一个拥有一个能接收一个整数为参数的 read() 方法以及一个不接收任何参数的 readline() 方法，并且这两个方法的返回值都应该是字符串。这可以是一个打开为读的文件对象、StringIO 对象或其他任何满足条件的对象。

```
import pickle
# 也可以这样:
# import cPickle as pickle

obj = {"a": 1, "b": 2, "c": 3}

# 将 obj 持久化保存到文件 tmp.txt 中
pickle.dump(obj, open("tmp.txt", "w"))

# do something else ...

# 从 tmp.txt 中读取并恢复 obj 对象
obj2 = pickle.load(open("tmp.txt", "r"))

print obj2
```

过实际应用中，我们可能还会有一些改进，比如用 cPickle 来代替 pickle，前者是后者的一个 C 语言实现版本，拥有更快的速度，另外，有时在 dump 时也会将第三个参数设为 True 以提高压缩比。

- `pickle.dumps(obj[, protocol])`: 返回一个被序列化的字符串,代替写入文件.
- `pickle.loads(string)`: 返回反序列化的对象.

```
In [11]: t
Out[11]: {'a': {'a': 'a'}, 'alist': [1, 2, 43, 'ds'], 'atuple': (3, 2,
'sd')}
In [12]: p = pickle.dump
pickle.dump   pickle.dumps
In [12]: p = pickle.dumps(t)
In [13]: p
Out[13]: "
(dp0\nS'a'\np1\n(dp2\ng1\ng1\nssS'alist'\np3\n(lp4\nI1\naI2\naI43\naS'ds'
\np5\nasS'atuple'\np6\n(I3\nI2\nS'sd'\np7\ntp8\ns."
In [14]: p1 = pickle.dumps(t,2)
In [15]: p1
Out[15]:
'\x80\x02}q\x00(U\x01aq\x01}q\x02h\x01h\x01sU\x05alistq\x03]q\x04(K\x01K\
x02K+U\x02dsq\x05eU\x06atupleq\x06K\x03K\x02U\x02sdq\x07\x87q\x08u.'

In [16]: fp = pickle.loads(p)

In [17]: fp
Out[17]: {'a': {'a': 'a'}, 'alist': [1, 2, 43, 'ds'], 'atuple': (3, 2,
'sd')}

In [18]: fp1 = pickle.loads(p1)

In [19]: fp1
Out[19]: {'a': {'a': 'a'}, 'alist': [1, 2, 43, 'ds'], 'atuple': (3, 2,
'sd')}
```

`PickleError`, `PicklingError` dumps , `UnpicklingError`

- `class pickle.Pickler(file[, protocol])`: *dump(obj)*Write a pickled representation of obj to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the protocol argument passed to the constructor.

*clear_memo()*Clears the pickler's "memo". The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

- `class pickle.Unpickler(file)`:*load()*Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

This method automatically determines whether the data stream was written in binary mode or not.

*noload()*This is just like load() except that it doesn't actually create any objects. This is useful primarily for finding what's called "persistent ids" that may be referenced in a pickle data stream. See section The pickle protocol below for more details.

哪些数据被序列化

1. `None`, `True`, and `False`
2. `integers`, `long integers`, `floating point numbers`, `complex numbers`
3. `normal` and `Unicode strings`
4. `tuples`, `lists`, `sets`, and `dictionaries` containing only picklable objects
5. `functions` defined at the top level of a module
6. built-in functions defined at the top level of a module
7. classes that are defined at the top level of a module
8. instances of such classes whose `__dict__` or the result of calling `__getstate__()` is picklable

JSON

什么是**JSON?**

- JSON 指的是 JavaScript 对象表示法（JavaScript Object Notation）
- JSON 是轻量级的文本数据交换格式
- JSON 独立于语言 *
- JSON 具有自我描述性，更易理解

JSON 使用 JavaScript 语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。

JSON的数据格式

JS下的格式:

- 数字（整数或浮点数）
- 字符串（在双引号中）
- 逻辑值（true 或 false）
- 数组（在方括号中）
- 对象（在花括号中）
- null

Python类型与**JSON**类型比较:

Python	**JSON**
dict	object
list, tuple	array
str, unicode	string
int, long, float	number
True	true
False	false
	null

Python处理**JSON**的标准库里是**json**模块

常用的方式

Python数据类型**encode**为**JSON**

```
In [1]: import json

In [2]:

In [2]: import json

In [3]: json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
Out[3]: '["foo", {"bar": ["baz", null, 1.0, 2]}]'

In [4]: print json.dumps("\"foo\bar")
"\"foo\bar"

In [5]: print json.dumps(u'\u1234')
"\u1234"

In [7]: print json.dumps('\\')
"\\"

In [8]: print json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True)
{"a": 0, "b": 0, "c": 0}

In [9]: from StringIO import StringIO

In [10]: io = StringIO()

In [11]: json.du
json.dump   json.dumps

In [11]: json.dump(['streaming API', 'a'], io)

In [12]: io.getvalue()
Out[12]: '["streaming API", "a"]'

## 紧凑的编码
In [13]: json.dumps([1,2,3,{'4':5, '6':7}], separators=(',',':'))
Out[13]: '[1,2,3,{"4":5,"6":7}]'

## 格式化打印


In [17]: print json.dumps({'4': 5, '6': 7, 'person':{'name':'user',
```

```
'age':10}}, sort_keys=True, indent=4, separators=(',',':'))

{
    "4":5,
    "6":7,
    "person":{
        "age":10,
        "name":"user"
    }
}
```

JSON decode为Python数据类型

```
In [20]: json1=json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])

In [21]: json1
Out[21]: '["foo", {"bar": ["baz", null, 1.0, 2]}]'

In [22]: print type(json1)
<type 'str'>

In [23]: json.loads(json1)
Out[23]: [u'foo', {u'bar': [u'baz', None, 1.0, 2]}]

In [24]: json.loads("\\"foo\\bar")
  File "<ipython-input-24-916e4d87d281>", line 1
    json.loads("\\"foo\\bar")
                      ^
SyntaxError: invalid syntax


In [25]: json.loads('"\\"foo\\bar"')
Out[25]: u'"foo\x08ar'

In [26]: json.loads('["straing api","d"]')
Out[26]: [u'straing api', u'd']

In [32]: io = StringIO('["sfsf","a"]')
In [33]: json.load(io)
Out[33]: [u'sfsf', u'a']
```

json模块的用法

类似于Pickle模块

- `json.dump(obj, fp, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)` :

  Serialize *obj* as a JSON formatted stream to *fp* (a `.write()`-supporting file-like object) using this conversion table.

- `json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, encoding="utf-8", default=None, sort_keys=False, **kw)`:

  Serialize *obj* to a JSON formatted `str` using this conversion table.
  If *ensure_ascii* is `False`, the result may contain non-ASCII characters and the return value may be a `unicode` instance.

- `json.load(fp[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])`:

  Deserialize fp (a .read()-supporting file-like object containing a JSON document) to a Python object using this conversion table.

- `json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, **kw]]]]]]]])`:

  Deserialize *s* (a `str` or `unicode` instance containing a JSON document) to a Python object using this conversion table.