序列化和反序列化

为什么要序列化

内存中的字典、列表、集合以及各种对象，如何保存到一个文件中？
如果是自己定义的类的实例，如何保存到一个文件中？
如何从文件中读取数据，并让它们在内存中再次恢复成自己对应的类的实例？

要设计一套**协议**，按照某种规则，把内存中数据保存到文件中。文件是一个字节序列，所以必须把数据转换成字节序列，输出到文件。这就是序列化。反之，从文件的字节序列恢复到内存并且还是原来的类型，就是反序列化。

定义

serialization 序列化
将内存中对象存储下来，把它变成一个个字节。-> 二进制

deserialization 反序列化
将文件的一个个字节恢复成内存中对象。<- 二进制

序列化保存到文件就是持久化。
可以将数据序列化后持久化，或者网络传输；也可以将从文件中或者网络接收到的字节序列反序列化。

Python 提供了pickle 库。

pickle库

Python中的序列化、反序列化模块。

函数	说明
dumps	对象序列化为bytes对象
dump	对象序列化到文件对象，就是存入文件
loads	从bytes对象反序列化
load	对象反序列化，从文件读取数据

```
import pickle

filename = 'o:/ser'

# 序列化后看到什么
i = 99
c = 'b'
l = list('123')
d = {'a':1, 'b':'abc', 'c':[1,2,3]}
```

```
# 序列化
with open(filename, 'wb') as f:
    pickle.dump(i, f)
    pickle.dump(c, f)
    pickle.dump(l, f)
    pickle.dump(d, f)

# 反序列化
with open(filename, 'rb') as f:
    print(f.read(), f.seek(0))
    for i in range(4):
        x = pickle.load(f)
        print(x, type(x))
```

```
import pickle

# 对象序列化
class AAA:
    tttt = 'ABC'
    def show(self):
        print('abc')

a1 = AAA() # 创建AAA类的对象

# 序列化
ser = pickle.dumps(a1)
print('ser={}'.format(ser)) # AAA

# 反序列化
a2 = pickle.loads(ser)
print(a2.tttt)
a2.show()
```


上面的例子中，故意使用了连续的AAA、ABC、abc等字符串，就是为了在二进制文件中能容易的发现它们。

上例中，其实就保存了一个类名，因为所有的其他东西都是类定义的东西，是不变的，所以只序列化一个AAA类名。反序列化的时候找到类就可以恢复一个对象。

```
import pickle

# 对象序列化
class AAA:
    def __init__(self):
        self.aaaa = 'abc'

a1 = AAA() # 创建AAA类的对象

# 序列化
ser = pickle.dumps(a1)
print('ser={}'.format(ser)) # AAA aaaa abc
```

```
# 反序列化
a2 = pickle.loads(ser)
print(a2, type(a2))
print(a2.aaaa)
print(id(a1), id(a2))
```

可以看出这回除了必须保存的AAA，还序列化了aaaa和abc，因为这是每一个对象自己的属性，每一个对象不一样的，所以这些数据需要序列化。

序列化、反序列化实验

定义类AAA，并序列化到文件

```
import pickle

# 实验
# 对象序列化
class AAA:
    def __init__(self):
        self.aaaa = 'abc'

a1 = AAA() # 创建AAA类的对象

# 序列化
ser = pickle.dumps(a1)
print('ser={}'.format(ser)) # AAA aaaa abc
print(len(ser))

filename = 'o:/ser'

with open(filename, 'wb') as f:
    pickle.dump(a1, f)
```


将产生的序列化文件ser发送到其他节点上。

增加一个x.py文件，内容如下。最后执行这个脚本 `$ python x.py`

```
import pickle

with open('ser','rb') as f:
    a = pickle.load(f) # 异常
```

会抛出异常`AttributeError: Can't get attribute 'AAA' on <module '__main__' from 't.py'>`。
这个异常实际上是找不到类AAA的定义，增加类定义即可解决。
反序列化的时候要找到AAA类的定义，才能成功。否则就会抛出异常。
可以这样理解：反序列化的时候，类是模子，二进制序列就是铁水

```
import pickle

class AAA:
    def show(self):
        print('xyz')

with open('ser', 'rb') as f:
    a = pickle.load(f)
    print(a)
    a.show()
    print(a.aaaa)
```

这里定义了类AAA，并且上面的代码也能成功的执行。

注意：这里的AAA定义和原来完全不同了。
因此，序列化、反序列化必须保证使用同一套类的定义，否则会带来不可预料的结果。

序列化应用

一般来说，本地序列化的情况，应用较少。大多数场景都应用在网络传输中。
将数据序列化后通过网络传输到远程节点，远程服务器上的服务将接收到的数据反序列化后，就可以使用了。
但是，要注意一点，远程接收端，反序列化时必须有对应的数据类型，否则就会报错。尤其是自定义类，必须远程得有一致的定义。

现在，大多数项目，都不是单机的，也不是单服务的，需要多个程序之间配合。需要通过网络将数据传送到其他节点上去，这就需要大量的序列化、反序列化过程。

但是，问题是，Python程序之间还可以都是用pickle解决序列化、反序列化，如果是跨平台、跨语言、跨协议pickle就不太适合了，就需要公共的协议。例如XML、Json、Protocol Buffer等。

不同的协议，效率不同、学习曲线不同，适用不同场景，要根据不同的情况分析选型。

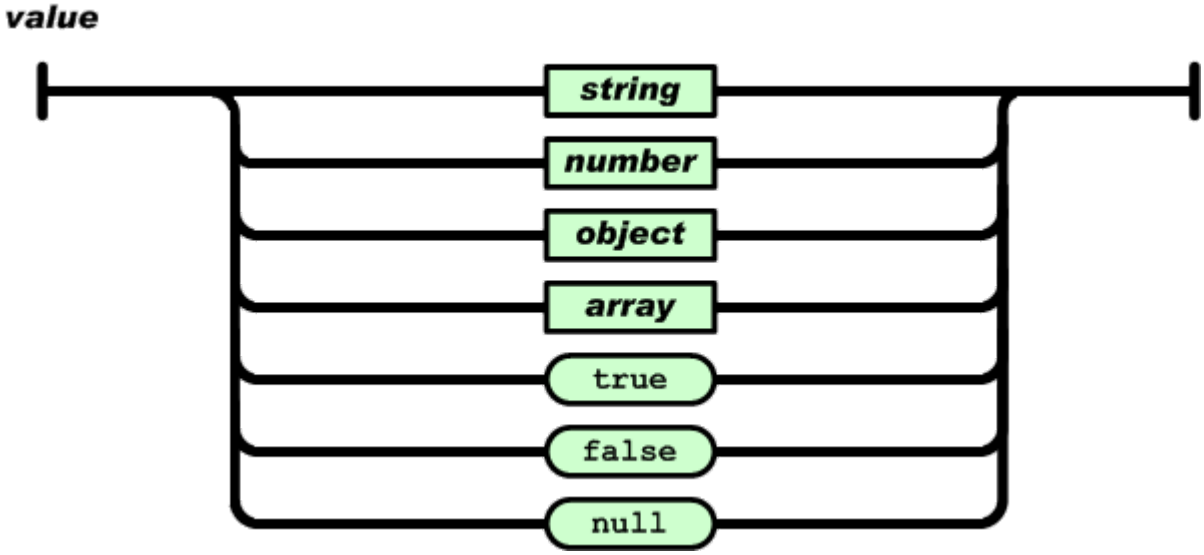
Json

JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式。它基于 ECMAScript (w3c组织制定的JS规范)的一个子集，采用完全独立于编程语言的文本格式来存储和表示数据。

http://json.org/

Json的数据类型

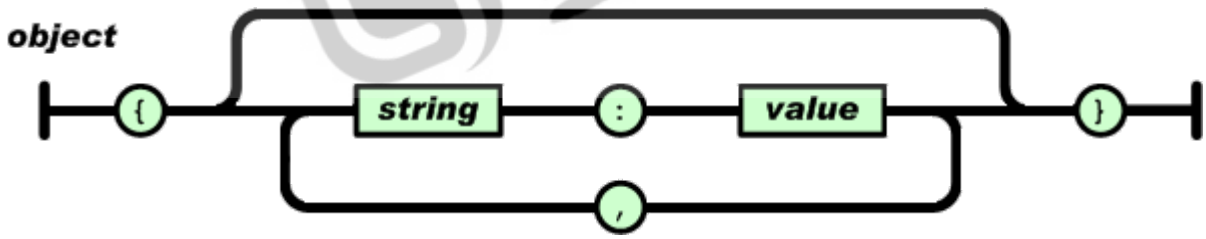
值
双引号引起来的字符串，数值，true和false，null，对象，数组，这些都是值



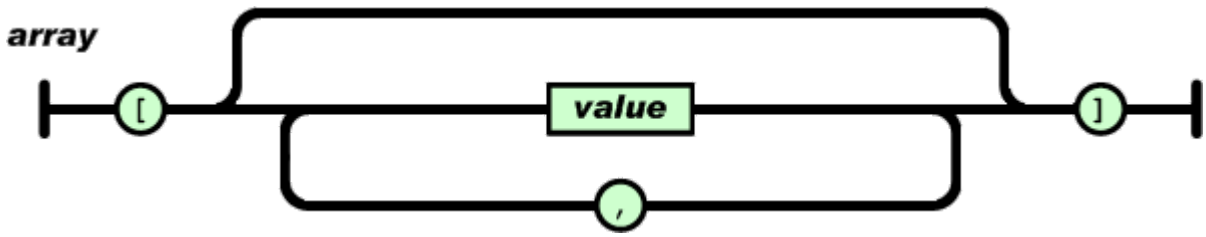
字符串
由双引号包围起来的任意字符的组合，可以有转义字符。

数值
有正负，有整数、浮点数。

对象
无序的键值对的集合
格式: {key1:value1, ... ,keyn:valulen}
key必须是一个字符串，需要双引号包围这个字符串。
value可以是任意合法的值。



数组
有序的值的集合
格式：[val1,...,valn]



实例

```
{
  "person": [
    {
```

```
      "name": "tom",
      "age": 18
    },
    {
      "name": "jerry",
      "age": 16
    }
  ],
  "total": 2
}
```

json模块

Python 与 Json

Python支持少量内建数据类型到Json类型的转换。

Python类型	Json类型
True	true
False	false
None	null
str	string
int	integer
float	float
list	array
dict	object

常用方法

Python类型	Json类型
dumps	json编码
dump	json编码并存入文件
loads	json解码
load	json解码，从文件读取数据

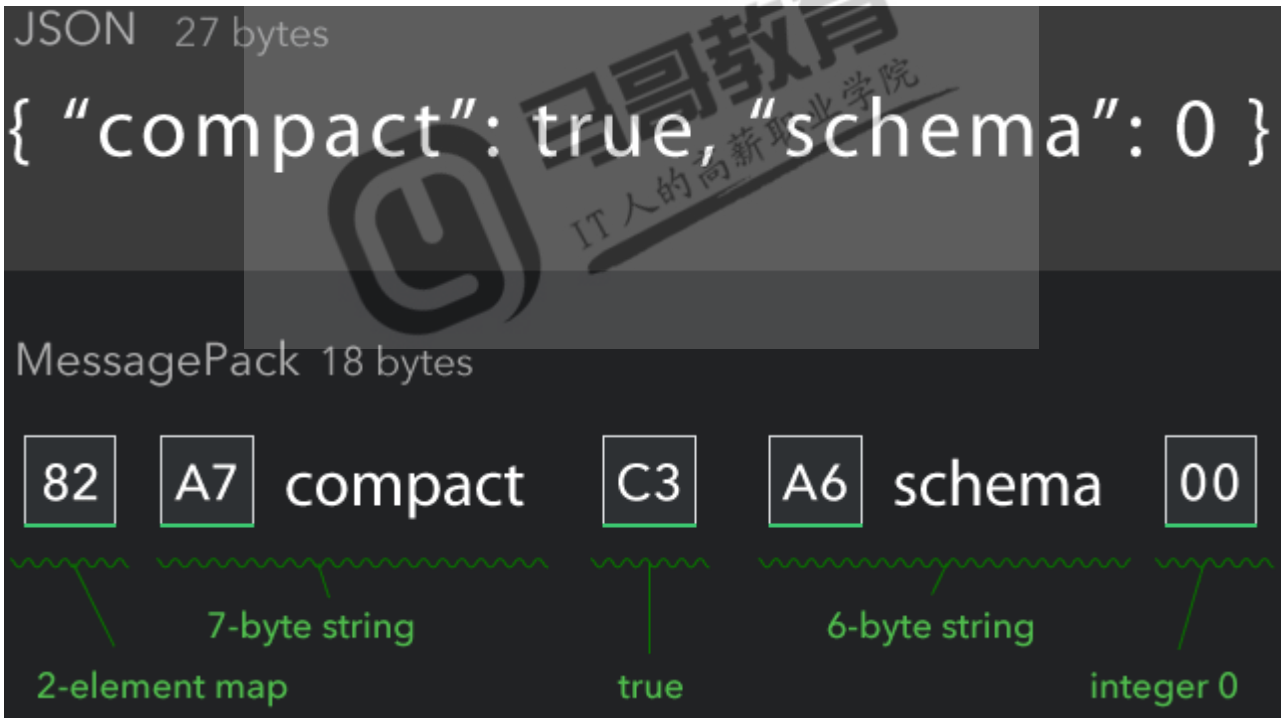
```python
import json
d = {'name':'Tom', 'age':20, 'interest':['music', 'movie']}
j = json.dumps(d)
print(j, type(j)) # 请注意引号的变化，注意数据类型的变化

d1 = json.loads(j)
print(d1)
print(id(d), id(d1))
```

一般json编码的数据很少落地，数据都是通过网络传输。传输的时候，要考虑压缩它。
本质上来说它就是个文本，就是个字符串。
json很简单，几乎编程语言都支持json，所以应用范围十分广泛。

MessagePack

MessagePack是一个基于二进制高效的对象序列化类库，可用于跨语言通信。
它可以像JSON那样，在许多种语言之间交换结构对象。
但是它比JSON更快速也更轻巧。
支持Python、Ruby、Java、C/C++等众多语言。宣称比Google Protocol Buffers还要快4倍。
兼容 json和pickle。




```
# 72 bytes
{"person":[{"name":"tom","age":18},{"name":"jerry","age":16}],"total":2}

# 48 bytes
# 82 a6 70 65 72 73 6f 6e 92 82 a4 6e 61 6d 65 a3 74 6f 6d a3 61 67 65 12 82 a4 6e 61 6d 65 a5
6a 65 72 72 79 a3 61 67 65 10 a5 74 6f 74 61 6c 02
```

可以看出，大大的节约了空间。

安装

```
$ pip install msgpack
```

常用方法

packb 序列化对象。提供了dumps来兼容pickle和json。
unpackb 反序列化对象。提供了loads来兼容。

pack 序列化对象保存到文件对象。提供了dump来兼容。
unpack 反序列化对象保存到文件对象。提供了load来兼容。

```
import pickle
import msgpack
import json

d = {'person': [{'name': 'tom', 'age': 18}, {'name': 'jerry', 'age': 16}], 'total': 2}

# json
data = json.dumps(d)
print(type(data), len(data), data)
print(data.replace(' ', ''))
print(len(data.replace(' ', ''))) # 72 bytes 注意这样替换的压缩是不对的

# pickle
data = pickle.dumps(d)
print(type(data), len(data), data) # 101 bytes

# msgpack
data = msgpack.dumps(d)
print(type(data), len(data), data) # 48 bytes

print('-' * 30)

u = msgpack.unpackb(data)
print(type(u), u)
print(msgpack.loads(data))

u = msgpack.loads(data, encoding='utf-8')
print(type(u), u)
```

MessagePack简单易用，高效压缩，支持语言丰富。
所以，用它序列化也是一种很好的选择。Python很多大名鼎鼎的库都是用了msgpack。