ini文件处理

作为配置文件，ini文件格式的很流行。

```
[DEFAULT]
a = test

[mysql]
default-character-set=utf8

[mysqld]
datadir =/dbserver/data
port = 33060
character-set-server=utf8
sql_mode=NO_ENGINE_SUBSTITUTION,STRICT_TRANS_TABLES
```

中括号里面的部分称为section，译作节、区、段。
每一个section内，都是key=value形成的键值对，key称为option选项。

注意这里的DEFAULT是缺省section的名字，必须大写。

configparser

configparser模块的ConfigParser类就是用来操作。

可以将section当做key，section存储着键值对组成的字典，可以把ini配置文件当做一个嵌套的字典。
默认使用的是有序字典。

`read(filenames, encoding=None)`
读取ini文件，可以是单个文件，也可以是文件列表。可以指定文件编码。

`sections()` 返回section列表。缺省section不包括在内。
`add_section(section_name)` 增加一个section。
`has_section(section_name)` 判断section是否存在

`options(section)` 返回section的所有option，会追加缺省section的option
`has_option(section, option)` 判断section是否存在这个option

`get(section, option, *, raw=False, vars=None[, fallback])`
从指定的段的选项上取值，如果找到返回，如果没有找到就去找DEFAULT段有没有。

`getint(section, option, *, raw=False, vars=None[, fallback])`
`getfloat(section, option, *, raw=False, vars=None[, fallback])`
`getboolean(section, option, *, raw=False, vars=None[, fallback])`
上面3个方法和get一样，返回指定类型数据。

`items(raw=False, vars=None)`
`items(section, raw=False, vars=None)`
没有section，则返回所有section名字及其对象；如果指定section，则返回这个指定的section的键值对组成二元组。

`set(section, option, value)`
section存在的情况下，写入option=value，要求option、value必须是字符串。

`remove_section(section)`
移除section及其所有option
`remove_option(section, option)`
移除section下的option。

`write(fileobject, space_around_delimiters=True)`
将当前config的所有内容写入fileobject中，一般open函数使用w模式。

```
from configparser import ConfigParser

filename = 'test.ini'
newfilename = 'mysql.ini'

cfg = ConfigParser()
read_ok = cfg.read(filename)
print(read_ok)
print(cfg.sections())  #
print(cfg.has_section('client'))

for k,v in cfg.items(): # 未指定section
    print(k, type(k))
    print(v, type(v))
    print(cfg.items(k))
    print()

print('-'*30)

for k,v in cfg.items("mysqld"): # 指定section
    print(k, type(k))
    print(v, type(v))
    print('~~~~~~')
print('-'*30)

# 取值
tmp = cfg.get('mysqld', 'port')
print(type(tmp), tmp)
tmp = cfg.get('mysqld', 'a')
print(type(tmp), tmp)
# print(cfg.get('mysqld','magedu'))
print(cfg.get('mysqld', 'magedu', fallback='python')) # 缺省值
# 按照类型
tmp = cfg.getint('mysqld', 'port')
print(type(tmp), tmp)

if cfg.has_section('test'):
    cfg.remove_section('test')

cfg.add_section('test')
cfg.set('test', 'test1', '1')
cfg.set('test', 'test2', '2')
#cfg.set('test', 'test2', 2)

with open(newfilename, 'w') as f:
    cfg.write(f)

print(cfg.getint('test', 'test2'))
```

```
cfg.remove_option('test', 'test2')

# 字典操作更简单
cfg['test']['x'] = '100'  # key不存在
cfg['test2'] = {'test2': '1000'}  # section不存在

print('x' in cfg['test'])
print('x' in cfg['test2'])

# 修改后需再次写入
with open(newfilename, 'w') as f:
    cfg.write(f)
```

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

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 = 'c'
l = list('123')
d = {'a':127, '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))
```

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程序之间可以都用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、msgpack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 1999年ES3 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/

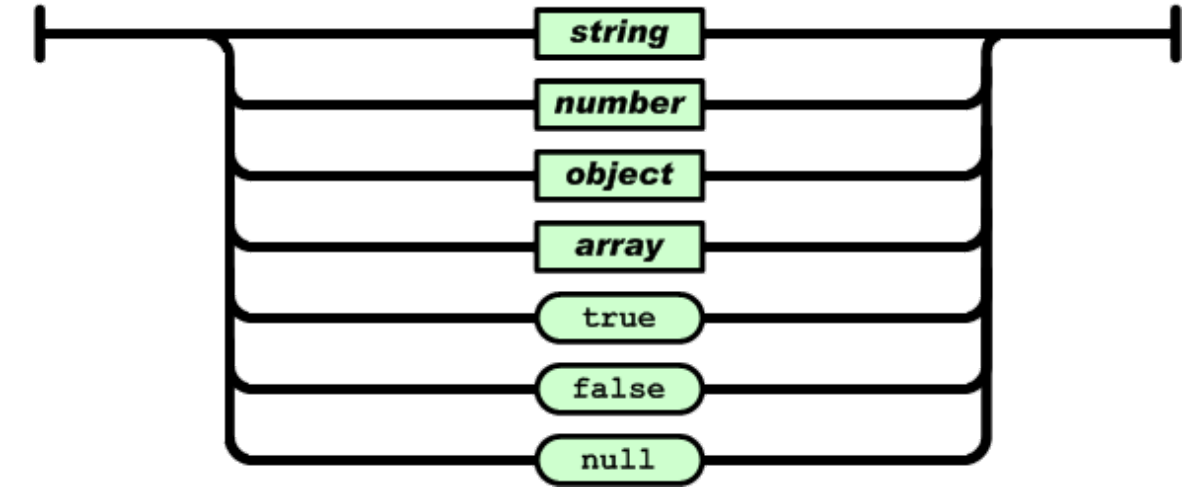
https://www.json.org/json-zh.html

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

value



字符串

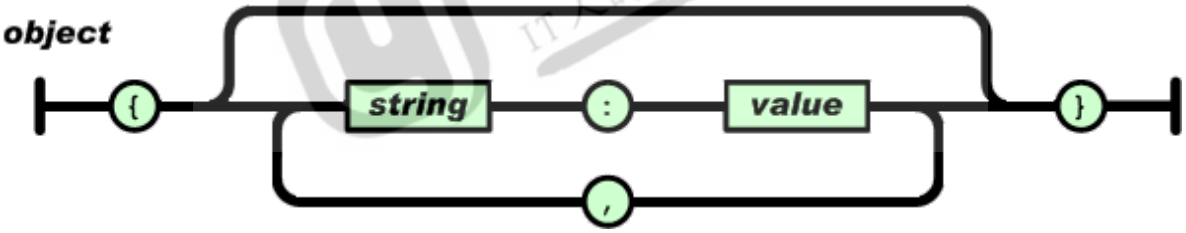
由**双引号**包围起来的任意字符的组合，可以有转义字符。

数值

有正负，有整数、浮点数。

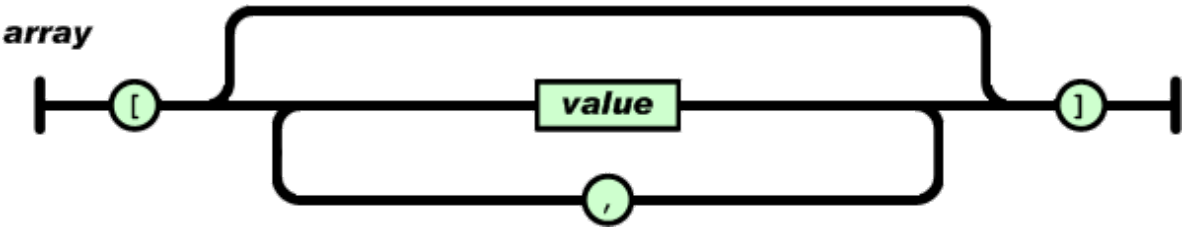
对象

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'), 'class':['python']}
j = json.dumps(d)
print(j, type(j)) # 请注意引号、括号的变化，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。