# pandas数据处理

## 1. 删除重复元素

使用duplicated()函数检测重复的行，返回元素为布尔类型的Series对象，每个元素对应一行，如果该行不是第一次出现，则元素为True

data.duplicated(subset=None, keep='first')

- keep参数：指定保留哪一重复的行数据

In [1]:

```
import pandas as pd
import numpy as np
from pandas import DataFrame, Series
```

In [2]:

```
data_list = [{"name": "jack", "age": 12}, {"name": "jemy", "age": 23}, {"name": "jer
```

In [3]:

```
df = DataFrame(data_list)
```

In [4]:

```
df
```

Out[4]:

| | age | name |
|---|---|---|
| **0** | 12 | jack |
| **1** | 23 | jemy |
| **2** | 12 | jeny |
| **3** | 23 | jemy |

In [5]:

```
# 检查重复的行，如果有多行重复，则keep="last"表示保留最后一行，因为要保留最后一行，所以这里的第二
df.duplicated(keep="last")
```

Out[5]:

```
0    False
1     True
2    False
3    False
dtype: bool
```

In [6]:

```
# 去除重复的行
df[~df.duplicated(keep="last")]
```

Out[6]:

| | age | name |
|---|---|---|
| **0** | 12 | jack |
| **2** | 12 | jeny |
| **3** | 23 | jemy |

In [10]:

```
# 也可以使用drop_duplicates()函数删除重复的行, 相当于drop_duplicates()做了上面检测和去除重复
df.drop_duplicates(keep="last")
```

Out[10]:

| | age | name |
|---|---|---|
| **0** | 12 | jack |
| **2** | 12 | jeny |
| **3** | 23 | jemy |

# 2. 对数据做映射

对数据做映射指的是把某些异常的数据根据设定的规则转换为我们想要的数据, 这种数据映射在日常工作中还是经常需要用到的。

## 1. replace()函数：替换元素

df.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad')

其中to_replace和value都可以是单值或者列表, 其中to_replace还可以是字典形式。

注意：DataFrame中的replace, 无法使用method和limit参数

In [11]:

```
df = pd.DataFrame(
            {
                '名称':['产品1','产品2','产品3','产品4','产品5','产品6','产品7','产品8'],
                '数量':['A','0.7','0.8','0.4','0.7','B','0.76','0.28'],
                '金额':['0','0.48','0.33','C','0.74','0','0','0.22'],
                '合计':['D','0.37','0.28','E','0.57','F','0','0.06'],
            }
        )
```

In [12]:

```
df
```

Out[12]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | A | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | C | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [14]:

```
# 把所有的A替换成数字0.2
df.replace(to_replace="A", value=0.2)
```

Out[14]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | 0.2 | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | C | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [16]:

```
# 把所有的A和B替换成0.2
df.replace(to_replace=["A", "B"], value=0.2)
```

Out[16]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | 0.2 | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | C | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | 0.4 | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [17]:

```
# 把所有的A和B替换成0.2和0.4
df.replace(to_replace=["A", "B"], value=[0.2, 0.4])
```

Out[17]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | 0.2 | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | C | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | 0.4 | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [18]:

```
# 把所有的A和B替换成0.2和0.4，除了可以使用上面的2个列表的形式，也可以使用语义更明确的字典的形式
df.replace(to_replace={"A": 0.2, "B": 0.4})
```

Out[18]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | 0.2 | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | C | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | 0.4 | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [21]:

```
# 单纯地替换某一列的值，比如把金额的C替换成0，替换列值的时候一般结合 to_replace 参数一起使用
df["金额"].replace("C", 0, inplace=True)
```

In [22]:

```
df
```

Out[22]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | A | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | 0 | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [23]:

```
# 可以使用正则表达式进行替换，这种也是经常使用的
df.replace('[A-Z]', 0.99, regex=True)
```

Out[23]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | 0.99 | 0 | 0.99 |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | 0 | 0.99 |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | 0.99 | 0 | 0.99 |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [30]:

```
# 替换数据的部分内容，比如说把名称这一列的中文 “产品” 替换成英文 “product”
# 值得注意的是，此时不能再使用inplace参数，但是可以df['名称'] = df['名称'].str.replace('产品
df['名称'].str.replace('产品', 'product')
```

Out[30]:

```
0    product1
1    product2
2    product3
3    product4
4    product5
5    product6
6    product7
7    product8
Name: 名称, dtype: object
```

除了使用replace方法可以达到修改数据的效果, 也可以使用原生的 = 直接进行修改赋值操作

In [32]:

```
df
```

Out[32]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | A | 0 | D |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | 0 | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [46]:

```
# 数量等于A的那一行的合计变成"changed"
df.loc[df['数量'] == 'A', '合计'] = "changed"
```

In [47]:

```
df
```

Out[47]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | A | 0 | changed |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | 0 | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

In [50]:

```
df.loc[df['合计'].str.contains('change'), '数量'] = 'new_A'
```

In [51]:

```
df
```

Out[51]:

| | 名称 | 数量 | 金额 | 合计 |
|---|---|---|---|---|
| **0** | 产品1 | new_A | 0 | changed |
| **1** | 产品2 | 0.7 | 0.48 | 0.37 |
| **2** | 产品3 | 0.8 | 0.33 | 0.28 |
| **3** | 产品4 | 0.4 | 0 | E |
| **4** | 产品5 | 0.7 | 0.74 | 0.57 |
| **5** | 产品6 | B | 0 | F |
| **6** | 产品7 | 0.76 | 0 | 0 |
| **7** | 产品8 | 0.28 | 0.22 | 0.06 |

## 2. map、apply、applymap

新建一列, map函数并不是df的方法，而是series的方法, 所以map经常对某一列进行映射。

- map() 可以映射新一列数据
- map() 中参数可以是字典, 可以使用lambd表达式, 也可以使用自定义的方法

并不是任何形式的函数都可以作为map的参数。只有当一个函数具有一个参数且有返回值，那么该函数才可以作为map的参数。

注意: map()中不能使用sum之类的函数，for循环.

map、apply、applymap三者区别:

- map() 方法是pandas.series.map()方法, 对DF中的元素级别的操作, 可以对df的某列或某多列
- apply(func) 是DF的属性, 对DF中的行数据或列数据应用func操作.
- applymap(func) 也是DF的属性, 对整个DF所有元素应用func操作.

In [110]:

```
toward_dict = {1: '东', 2: '南', 3: '西', 4: '北'}
df = pd.DataFrame({'house' : list('AABCEFG'),
                   'price' : [100, 90, '', 50, 120, 150, 200],
                   'toward' : ['1','1','2','3','','3','2']})
```

In [111]:

```
df
```

Out[111]:

| | house | price | toward |
|---|---|---|---|
| **0** | A | 100 | 1 |
| **1** | A | 90 | 1 |
| **2** | B | | 2 |
| **3** | C | 50 | 3 |
| **4** | E | 120 | |
| **5** | F | 150 | 3 |
| **6** | G | 200 | 2 |

In [112]:

```
df["朝向"] = df["toward"].map(toward_dict)
```

In [113]:

```
# 发现朝向是NaN, 因为字典的key是字符串类型, 但是toward却是数字类型, 最简单的方式是改一下映射字典
df
```

Out[113]:

| | house | price | toward | 朝向 |
|---|---|---|---|---|
| **0** | A | 100 | 1 | NaN |
| **1** | A | 90 | 1 | NaN |
| **2** | B | | 2 | NaN |
| **3** | C | 50 | 3 | NaN |
| **4** | E | 120 | | NaN |
| **5** | F | 150 | 3 | NaN |
| **6** | G | 200 | 2 | NaN |

In [114]:

```
df.toward = df.toward.map(lambda x: np.nan if x == '' else x).map(int,na_action='igr
df["朝向"] = df["toward"].map(toward_dict)
```

In [115]:

```
df
```

Out[115]:

| | house | price | toward | 朝向 |
|---|---|---|---|---|
| **0** | A | 100 | 1.0 | 东 |
| **1** | A | 90 | 1.0 | 东 |
| **2** | B | | 2.0 | 南 |
| **3** | C | 50 | 3.0 | 西 |
| **4** | E | 120 | NaN | NaN |
| **5** | F | 150 | 3.0 | 西 |
| **6** | G | 200 | 2.0 | 南 |

map自定义方法, 比如说把price加20%

In [83]:

```
def func(price):
    if price == "":
        return 0
    else:
        return price + price * 0.2
```

In [84]:

```
df["price"].map(func)
```

Out[84]:

```
0    120.0
1    108.0
2      0.0
3     60.0
4    144.0
5    180.0
6    240.0
Name: price, dtype: float64
```

In [120]:

```
df = DataFrame(np.random.randint(1, 23, size=(3,3)), columns=["A", "B", "C"], index=
```

df.apply(func, axis=0, broadcast=None, raw=False, reduce=None, result_type=None, args=(), **kwds)

In [121]:

```
df.apply(sum)
```

Out[121]:

```
A    11
B    21
C    31
dtype: int64
```

In [122]:

```
df.apply(sum, axis=1)
```

Out[122]:

```
a    23
b     7
c    33
dtype: int64
```

df.applymap(func)

In [124]:

```
df.applymap(lambda x: x + 1)
```

Out[124]:

| | A | B | C |
|---|---|---|---|
| **a** | 5 | 5 | 16 |
| **b** | 4 | 3 | 3 |
| **c** | 5 | 16 | 15 |

# 3. 使用聚合操作对数据异常值检测和过滤

In [86]:

```
df = DataFrame(data=np.random.random(size=(1000, 3)), columns=["A", "B", "C"])
```

In [87]:

```
# 对df应用筛选条件,去除标准差太大的数据:假设过滤条件为 C列数据大于两倍的C列标准差
df["C"] > df["C"].std()
```

Out[87]:

```
0       True
1       True
2       True
3       True
4       True
5       True
6       True
7      False
8       True
9       True
10      True
11     False
12     False
13     False
14      True
15      True
16     False
17     False
18      True
19      True
20      True
21     False
22      True
23      True
24     False
25      True
26     False
27      True
28      True
29     False
       ...
970     True
971     True
972     True
973    False
974     True
975     True
976     True
977    False
978     True
979     True
980     True
981     True
982     True
983     True
984    False
985    False
986     True
987    False
988     True
989     True
990     True
991    False
992     True
```

```
993     True
994    False
995     True
996    False
997     True
998     True
999     True
Name: C, Length: 1000, dtype: bool
```

In [88]:

```
df.loc[df["C"] < df["C"].std()]
```

Out[88]:

| | A | B | C |
|---|---|---|---|
| 7 | 0.670165 | 0.173967 | 0.153213 |
| 11 | 0.875892 | 0.074858 | 0.170779 |
| 12 | 0.422831 | 0.999585 | 0.155590 |
| 13 | 0.923034 | 0.655788 | 0.289979 |
| 16 | 0.777833 | 0.702853 | 0.061747 |
| 17 | 0.330928 | 0.803797 | 0.026978 |
| 21 | 0.816011 | 0.137062 | 0.061944 |
| 24 | 0.827712 | 0.802288 | 0.122861 |
| 26 | 0.996989 | 0.056052 | 0.162501 |
| 29 | 0.369555 | 0.659806 | 0.154624 |
| 32 | 0.824508 | 0.394953 | 0.086610 |
| 36 | 0.236297 | 0.716234 | 0.026979 |
| 42 | 0.853171 | 0.416299 | 0.286832 |
| 54 | 0.612565 | 0.740495 | 0.182254 |
| 58 | 0.164190 | 0.192465 | 0.142194 |
| 72 | 0.951508 | 0.980927 | 0.201791 |
| 79 | 0.733014 | 0.769344 | 0.124624 |
| 89 | 0.824745 | 0.304000 | 0.079809 |
| 93 | 0.275154 | 0.208849 | 0.264426 |
| 97 | 0.725948 | 0.536183 | 0.295275 |
| 100 | 0.977804 | 0.028849 | 0.118457 |
| 101 | 0.524023 | 0.936114 | 0.051828 |
| 104 | 0.154243 | 0.774049 | 0.169918 |
| 106 | 0.168693 | 0.591628 | 0.183313 |
| 109 | 0.103750 | 0.940425 | 0.063484 |
| 110 | 0.924050 | 0.516224 | 0.120030 |
| 115 | 0.877215 | 0.918822 | 0.137266 |
| 117 | 0.757711 | 0.042508 | 0.001940 |
| 119 | 0.233300 | 0.777975 | 0.279408 |
| 120 | 0.426409 | 0.657528 | 0.010253 |
| ... | ... | ... | ... |
| 912 | 0.435801 | 0.607922 | 0.104728 |
| 913 | 0.553991 | 0.683365 | 0.039408 |

| | A | B | C |
|---|---|---|---|
| **914** | 0.444234 | 0.243227 | 0.091394 |
| **915** | 0.890045 | 0.565060 | 0.215358 |
| **917** | 0.945927 | 0.751375 | 0.193650 |
| **922** | 0.999544 | 0.754941 | 0.065104 |
| **923** | 0.091876 | 0.053344 | 0.118102 |
| **926** | 0.441234 | 0.740102 | 0.215893 |
| **929** | 0.307097 | 0.096509 | 0.213567 |
| **930** | 0.216505 | 0.387210 | 0.228709 |
| **934** | 0.010232 | 0.314137 | 0.194262 |
| **936** | 0.664546 | 0.356042 | 0.191392 |
| **938** | 0.474916 | 0.657910 | 0.286798 |
| **940** | 0.868810 | 0.982862 | 0.250421 |
| **943** | 0.506047 | 0.221709 | 0.102487 |
| **947** | 0.219540 | 0.326411 | 0.149314 |
| **948** | 0.141129 | 0.648130 | 0.144553 |
| **949** | 0.704114 | 0.529434 | 0.060076 |
| **951** | 0.329526 | 0.903956 | 0.012533 |
| **953** | 0.577225 | 0.910598 | 0.226619 |
| **963** | 0.694188 | 0.522610 | 0.258912 |
| **964** | 0.393889 | 0.304543 | 0.046838 |
| **973** | 0.982085 | 0.962332 | 0.263448 |
| **977** | 0.298311 | 0.838295 | 0.144072 |
| **984** | 0.086280 | 0.446442 | 0.280912 |
| **985** | 0.027780 | 0.297484 | 0.219346 |
| **987** | 0.030477 | 0.801756 | 0.026076 |
| **991** | 0.519357 | 0.701984 | 0.165398 |
| **994** | 0.065642 | 0.279668 | 0.252231 |
| **996** | 0.887302 | 0.470982 | 0.267601 |

306 rows × 3 columns

# 4. 索引重排序

take()函数接受一个索引列表，用数字表示,使得df根据列表中索引的顺序进行排序，take的axis的轴向比较特殊，和drop是一类的，并且第一个参数列表必须是隐式索引。

In [89]:

```
df = DataFrame(data_list)
```

In [90]:

```
df
```

Out[90]:

| | age | name |
|---|---|---|
| 0 | 12 | jack |
| 1 | 23 | jemy |
| 2 | 12 | jeny |
| 3 | 23 | jemy |

In [91]:

```
df.take([3, 1, 0], axis=0)
```

Out[91]:

| | age | name |
|---|---|---|
| 3 | 23 | jemy |
| 1 | 23 | jemy |
| 0 | 12 | jack |

可以借助np.random.permutation()函数随机排序, np.random.permutation(x)可以生成x个从0-(x-1)的随机数列

In [92]:

```
np.random.permutation(10)
```

Out[92]:

```
array([7, 4, 0, 3, 5, 6, 1, 9, 2, 8])
```

In [95]:

```
df.take(np.random.permutation(2), axis=1)
```

Out[95]:

| | name | age |
|---|---|---|
| 0 | jack | 12 |
| 1 | jemy | 23 |
| 2 | jeny | 12 |
| 3 | jemy | 23 |

# 5. 分组聚合

df.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False, **kwargs)

数据分类处理：

1. 分组：先把数据分为几组
2. 用函数处理：为不同组的数据应用不同的函数以转换数据
3. 合并：把不同组得到的结果合并起来

数据分类处理的核心：

- groupby()函数
- groups属性查看分组情况
- eg: df.groupby(by='item').groups

In [96]:

```
data = {'key1': {0: 'a', 1: 'a', 2: 'b', 3: 'b', 4: 'a'},
 'key2': {0: 'one', 1: 'two', 2: 'one', 3: 'two', 4: 'one'},
 'data1': {0: 0.027953391617551962,
  1: 0.29174347663752437,
  2: -0.6707023485497721,
  3: 0.1940925164374532,
  4: -0.4855059362792516},
 'data2': {0: 2.2576340085211277,
  1: 0.6322052569996676,
  2: -1.306279167754625,
  3: -0.3248704978785065,
  4: 1.1890947845152855}}
df = DataFrame(data)
```

In [97]:

```
df
```

Out[97]:

| | key1 | key2 | data1 | data2 |
|---|---|---|---|---|
| **0** | a | one | 0.027953 | 2.257634 |
| **1** | a | two | 0.291743 | 0.632205 |
| **2** | b | one | -0.670702 | -1.306279 |
| **3** | b | two | 0.194093 | -0.324870 |
| **4** | a | one | -0.485506 | 1.189095 |

In [104]:

```
# 以1个key作为分组参考
df.groupby(by="key1")["data1"].mean()
```

Out[104]:

```
key1
a   -0.055270
b   -0.238305
Name: data1, dtype: float64
```

In [99]:

```
# 以2个key作为分组参考
df.groupby(by=["key1", "key2"])["data1"].sum()
```

Out[99]:

```
key1  key2
a     one    -0.457553
      two     0.291743
b     one    -0.670702
      two     0.194093
Name: data1, dtype: float64
```

In [100]:

```
df.groupby(by="key1").size()
```

Out[100]:

```
key1
a    3
b    2
dtype: int64
```

对分组进行迭代： for name, group in df.groupby('key1') 或 for (k1, k2), group in df.groupby(["k1", "k2"]):

In [101]:

```
df.groupby(by=["key1", "key2"])["data1"].sum().to_dict()
```

Out[101]:

```
{('a', 'one'): -0.45755254466169965,
 ('a', 'two'): 0.29174347663752437,
 ('b', 'one'): -0.6707023485497721,
 ('b', 'two'): 0.1940925164374532}
```

有时我们需要对组内不同列采取不同的操作, 这里需要用到agg函数

In [105]:

```
df.groupby(by="key1").agg({"data1": "sum", "data2": "std"})
```

Out[105]:

| key1 | data1 | data2 |
|---|---|---|
| a | -0.165809 | 0.826027 |
| b | -0.476610 | 0.693961 |

使用groupby分组后，也可以使用transform和apply提供自定义函数实现更多的运算

df.groupby('item')['price'].sum() <==> df.groupby('item')['price'].apply(sum)

transform和apply都会进行运算，在transform或者apply中传入函数即可

transform和apply也可以传入一个lambda表达式

In [106]:

```
df
```

Out[106]:

| | key1 | key2 | data1 | data2 |
|---|---|---|---|---|
| **0** | a | one | 0.027953 | 2.257634 |
| **1** | a | two | 0.291743 | 0.632205 |
| **2** | b | one | -0.670702 | -1.306279 |
| **3** | b | two | 0.194093 | -0.324870 |
| **4** | a | one | -0.485506 | 1.189095 |

In [107]:

```
# 自定义均值函数
def func(group_data):
    s = 0
    for i in group_data:
        s += i
    return s / group_data.size
```

In [108]:

```
df.groupby(by="key1")["data1"].apply(func)
```

Out[108]:

```
key1
a   -0.055270
b   -0.238305
Name: data1, dtype: float64
```

# 6. 数据加载

pandas提供了一些用于将表格型数据读取为DataFrame对象的函数，期中read_csv和read_table这两个使用最多，此外read_sql用的也比较多。 常用的用于读取数据的read系列, 调用者是pandas库：

1. read_csv
2. read_excel
3. read_json
4. read_table
5. read_sql

常用的用于保存数据的to系列, 调用者是DataFrame对象：

1. to_csv
2. to_excel
3. to_json
4. to_sql
5. to_dict

具体这些用到什么再去查就ok。

In [128]:

```
import pymongo
#连接数据库
client = pymongo.MongoClient('localhost',27017)
db = client['etcd']
table = db['service']
#加载数据
df = DataFrame(list(table.find()))
```

值得说明的是, 上述内容只是数据分析中的一小部分基础性的内容, 根本就谈不上是什么中级, pandas库远比上述强大得多, 学习的过程不是一蹴而就, 先掌握基础, 搭好大体的知识框架, 然后在工作或日常的学习过程中慢慢地添砖加瓦。要是一下子想要全部掌握, 可能是比较枯燥的过程, 很多人分不清主次甚至连基础都无法掌握。

In [ ]:

```

```