上篇介绍了 NumPy，本篇介绍 pandas。

目录

- pandas 入门
- pandas 的数据结构介绍
- 基本功能
- 汇总和计算描述统计
- 处理缺失数据
- 层次化索引

pandas 入门

Pandas 是基于 Numpy 构建的，让以 NumPy 为中心的应用变的更加简单。

pandas 的数据结构介绍

1、Series

由一组数据（各种 NumPy 数据类型）和一组索引组成：

```
In [3]: obj = Series([4, 7, -5, 3])

In [4]: obj

Out[4]: 0    4
        1    7
        2   -5
        3    3
        dtype: int64
```

Values 和 index 属性：

```
In [5]: obj.values

Out[5]: array([ 4,  7, -5,  3])

In [6]: obj.index

Out[6]: RangeIndex(start=0, stop=4, step=1)
```

给所创建的 Series 带有一个可以对各个数据点进行标记的索引：

```
In [13]: obj2 = Series([4, 7, -5, 3], index = ['d', 'b', 'a', 'c'])

In [14]: obj2
Out[14]: d    4
         b    7
         a   -5
         c    3
         dtype: int64

In [15]: obj2.index
Out[15]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

与普通 NumPy 数组相比，可以通过索引的方式选取 Series 中的单个或一组值：

```
In [16]: obj2['a']
Out[16]: -5

In [17]: obj2[['c', 'a', 'd']]
Out[17]: c    3
         a   -5
         d    4
         dtype: int64
```

可将 Series 看成是一个定长的有序字典，它是索引值到数据值的一个映射（它可以用在许多原本需要字典参数的函数中）。

如果数据被存放在一个 python 字典中，可以直接通过这个字典来创建 Series：

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False

In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = Series(sdata)

In [28]: obj3
Out[28]: Ohio      35000
         Oregon    16000
         Texas     71000
         Utah       5000
         dtype: int64

In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = Series(sdata, index = states)

In [31]: obj4
Out[31]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64
```

如果只传入一个字典，则结果 Series 中的索引就是原字典的键（有序排列），上面的 states。

Series 最重要的一个功能是在算数运算中自动对齐不同索引的数据：

```
In [35]: obj3
Out[35]: Ohio      35000
         Oregon    16000
         Texas     71000
         Utah       5000
         dtype: int64

In [36]: obj4
Out[36]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64

In [37]: obj3 + obj4
Out[37]: California         NaN
         Ohio           70000.0
         Oregon         32000.0
         Texas         142000.0
         Utah               NaN
         dtype: float64
```

Series 对象本身及其索引都有一个 name 属性：

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]: state
         California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         Name: population, dtype: float64
```

Series 的索引可以通过赋值的方式就地修改：

```
In [41]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [42]: obj
Out[42]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```

2、DataFrame

是一个表格型的数据结构。既有行索引也有列索引。DataFrame 中面向行和面向列的操作基本是平衡的。DataFrame 中的数据是以一个或多个二维块存放的。用层次化索引，将其表示为更高维度的数据。

构建 DataFrame：直接传入一个由等长列表或 NumPy 数组组成的字典。

```
In [43]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada']}

In [44]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
                 'year': [2000, 2001, 2002, 2001, 2002],
                 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}

In [45]: frame = DataFrame(data)

In [ ]:

In [46]: frame
Out[46]:
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

会自动加上索引，但指定列序列，则按指定顺序进行排列：

```
In [48]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[48]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9

和 Series 一样，如果传入的列在数据中找不到，就会产生 NA 值：

```
In [52]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                            index=['one', 'two', 'three', 'four', 'five'])

In [53]: frame2
Out[53]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

```
In [54]: frame2.columns
Out[54]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

通过赋值的方式进行修改：

```
In [60]: frame2['debt'] = 16.5

In [61]: frame2
```

Out[61]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5

通过类似字典标记的方式或属性的方式，可以将 DataFrame 的列获取为一个 Series：

```
In [55]: frame2['state']
Out[55]: one        Ohio
         two        Ohio
         three      Ohio
         four     Nevada
         five     Nevada
         Name: state, dtype: object

In [56]: frame2.year
Out[56]: one      2000
         two      2001
         three    2002
         four     2001
         five     2002
         Name: year, dtype: int64
```

行也可以通过位置或名称的方式进行获取，比如用索引字段 ix。

将列表或数组赋值给某个列时，其长度必须跟 DataFrame 的长度相匹配。如果赋值的是一个 Series，就会精确匹配 DataFrame 的索引，所有的空位都将被填上缺失值：

```
In [65]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [66]: frame2['debt'] = val

In [67]: frame2
Out[67]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

给不存在的列赋值会创建出一个新列，关键字 del 用于删除列：

```
In [68]: frame2['eastern'] = frame2.state == 'Ohio'

In [69]: frame2
Out[69]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

```
In [70]: del frame2['eastern']

In [71]: frame2.columns
Out[71]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

通过索引方式返回的列是相应数据的视图，并不是副本，对返回的 Series 做的任何修改都会反映到源 DataFrame 上，通过 series 的 copy 方法即可显式地复制列。

另一种常见的数据形式是嵌套字典，如果将它传给 DataFrame，解释为——外层字典的键作为列，内层键作为行索引。

```
In [72]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
                'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

In [73]: frame3 = DataFrame(pop)

In [74]: frame3
Out[74]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

对结果进行转置：

```
In [75]: frame3.T
Out[75]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

指定索引按序列：

```
In [77]: DataFrame(pop, index=[2001, 2002, 2003])
```

Out[77]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

由 Series 组成的字典差不多也是一样的用法：

```
In [80]: pdata = {'Ohio': frame3['Ohio'][:-1],
                  'Nevada': frame3['Nevada'][:2]}
```

```
In [81]: DataFrame(pdata)
```

Out[81]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7

设置了 DataFrame 的 index 和 columns 的 name 属性，这些信息也会被显示，values 属性以二维 ndarray 的形式返回 DataFrame 中的数据：

```
In [82]: frame3.index.name = 'year'; frame3.columns.name = 'state'
```

```
In [83]: frame3
```

Out[83]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [84]: frame3.values
Out[84]: array([[nan, 1.5],
                [2.4, 1.7],
                [2.9, 3.6]])
```

如果 DataFrame 各列的数据类型不同，值数组的数据类型就会选用能兼容所有列的数据类型（如 dtype = object）。

3、索引对象

pandas 的索引对象，管理轴标签和其他元数据（如轴名称等）。

构建 Series 或 DataFrame 时，所用到的任何数组或其他序列的标签都会被转换成一个 Index，且 Index 对象是不可修改的：

```
In [86]: obj = Series(range(3), index=['a', 'b', 'c'])

In [87]: index = obj.index

In [88]: index
Out[88]: Index(['a', 'b', 'c'], dtype='object')

In [89]: index[1:]
Out[89]: Index(['b', 'c'], dtype='object')

In [90]: obj
Out[90]: a    0
         b    1
         c    2
         dtype: int64

In [91]: index[1] = 'd'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-91-a452e55ce13b> in <module>()
```

Index 的功能类似一个固定大小的集合：

```
In [96]: frame3
Out[96]:
```

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [97]: 'Ohio' in frame3.columns
Out[97]: True

In [98]: 2003 in frame3.index
Out[98]: False
```

基本功能

1、重新索引

方法 reindex：创建一个适应新索引的新对象。

```
In [100]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [101]: obj
Out[101]: d    4.5
          b    7.2
          a   -5.3
          c    3.6
          dtype: float64

In [102]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [103]: obj2
Out[103]: a   -5.3
          b    7.2
          c    3.6
          d    4.5
          e    NaN
          dtype: float64
```

调用该 Series 的 reindex 将会根据新索引进行重排。如果某个索引值当前不存在，就引入缺失值。

```
In [105]: obj = obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)

In [106]: obj

Out[106]: a   -5.3
          b    7.2
          c    3.6
          d    4.5
          e    0.0
          dtype: float64
```

对于时间序列这样的有序数据，重新索引时可能需要做一些差值处理：

```
In [107]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0,2,4])

In [108]: obj3.reindex(range(6), method='ffill')

Out[108]: 0      blue
          1      blue
          2    purple
          3    purple
          4    yellow
          5    yellow
          dtype: object
```

对于 DataFrame ，reindex 可以修改行、列索引，或两个都修改。如果仅传入一列，则会重新索引行：

```
In [111]: frame = DataFrame(np.arange(9).reshape((3,3)), index=['a', 'c', 'd'],
                            columns=['Ohio', 'Texas', 'California'])

In [112]: frame

Out[112]:
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
In [113]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [114]: frame2

Out[114]:
```

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

使用 columns 关键字可重新索引列：

```
In [115]: states = ['Texas', 'Utah', 'California']
```

```
In [116]: frame.reindex(columns=states)
```

Out[116]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

同时对行、列进行索引：

```
In [127]: frame.reindex(index=['1', '2', '3', '4'], method='ffill',
                        columns=states.sort())
```

Out[127]:

	Ohio	Texas	California
1	NaN	NaN	NaN
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN

ix 标签索引功能：

```
In [128]: frame.ix[['a', 'b', 'c', 'd'], states]
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  """Entry point for launching an IPython kernel.
```

Out[128]:

	California	Texas	Utah
a	2.0	1.0	NaN
b	NaN	NaN	NaN
c	5.0	4.0	NaN
d	8.0	7.0	NaN

丢弃制定轴上的项

drop 方法返回的是一个在指定轴上删除了指定值的新对象：

```
In [129]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [130]: new_obj = obj.drop('c')
```

```
In [131]: new_obj
```

```
Out[131]: a    0.0
          b    1.0
          d    3.0
          e    4.0
          dtype: float64
```

对于 DataFrame，可以删除任意轴上的索引值：

```
In [136]: data = DataFrame(np.arange(16).reshape((4,4)),
                           index=['Ohio', 'Colorado', 'Utah', 'New York'],
                           columns=['one', 'two', 'three', 'four'])

In [137]: data
Out[137]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [138]: data.drop(['Colorado', 'Ohio'])
Out[138]:
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
In [141]: data.drop(['two', 'four'], axis=1)
Out[141]:
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

```
In [142]: data.drop('two', axis=1)
Out[142]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

2、索引、选取和过滤

Series 索引的工作方式类似于 NumPy 数组的索引，但 Series 的索引值不只是整数：

```
In [143]: obj = Series(np.arange(4.), index=['a', 'b', 'c','d'])

In [144]: obj
Out[144]: a    0.0
          b    1.0
          c    2.0
          d    3.0
          dtype: float64

In [145]: obj['b']
Out[145]: 1.0

In [146]: obj[1]
Out[146]: 1.0

In [147]: obj[2:4]
Out[147]: c    2.0
          d    3.0
          dtype: float64

In [148]: obj[[1,3]]
Out[148]: b    1.0
          d    3.0
          dtype: float64

In [149]: obj[obj<2]
Out[149]: a    0.0
          b    1.0
          dtype: float64
```

利用标签的切片运算，其包含闭区间（与普通 python 切片运算不同）：

```
In [151]: obj['b':'c']
Out[151]: b    1.0
          c    2.0
          dtype: float64

In [152]: obj['b':'c'] = 5

In [153]: obj
Out[153]: a    0.0
          b    5.0
          c    5.0
          d    3.0
          dtype: float64
```

对 DataFrame 进行索引就是获取一个列：

```
In [154]: data = DataFrame(np.arange(16).reshape((4,4)),
                           index=['Ohio', 'Colorado', 'Utah', 'New York'],
                           columns=['one', 'two', 'three', 'four'])

In [155]: data
Out[155]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
In [156]: data['two']
Out[156]: Ohio         1
          Colorado     5
          Utah         9
          New York    13
          Name: two, dtype: int64
```

或多个列：

```
In [158]: data[['three','one']]
Out[158]:
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

这种索引方式的特殊情况：通过切片或布尔型数组选取行。

```
In [159]: data[:2]
Out[159]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [160]: data[data['three'] > 5]
Out[160]:
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

另一种用法是通过布尔型 DataFrame 进行索引（在语法上更像 ndarray）：

```
In [161]: data < 5
```

Out[161]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
In [162]: data[data < 5] = 0
In [163]: data
```

Out[163]:

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

专门的索引字段 ix，是一种重新索引的简单手段：

```
In [170]: data.ix[2]
Out[170]: one      8
          two      9
          three    10
          four     11
          Name: Utah, dtype: int64

In [171]: data.ix[:'Utah', 'two']
Out[171]: Ohio        0
          Colorado    5
          Utah        9
          Name: two, dtype: int64

In [172]: data.ix[data.three > 5, :3]
```

Out[172]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

3、算术运算和数据对齐

pandas 最重要的一个功能是对不同索引的对象进行算术运算。

对不同的索引对，取并集：

```
In [174]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [175]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [176]: s1
Out[176]: a    7.3
          c   -2.5
          d    3.4
          e    1.5
          dtype: float64

In [177]: s2
Out[177]: a   -2.1
          c    3.6
          e   -1.5
          f    4.0
          g    3.1
          dtype: float64

In [178]: s1 + s2
Out[178]: a    5.2
          c    1.1
          d    NaN
          e    0.0
          f    NaN
          g    NaN
          dtype: float64
```

自动的数据对齐操作在不重叠的索引出引入了 NA 值，即一方有的索引，另一方没有，运算后该处索引的值为缺失值。

对 DataFrame，对齐操作会同时发生在行和列上。

4、在算术方法中填充值

对运算后的 NA 值处填充一个特殊值（比如 0）：

```
In [183]: df1 = DataFrame(np.arange(12.).reshape((3,4)), columns=list('abcd'))

In [184]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [185]: df1
Out[185]:
```

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

```
In [186]: df2
Out[186]:
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

```
In [187]: df1.add(df2, fill_value=0)
Out[187]:
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

否则 e 列都是 NaN 值。

类似，在对 Series 和 DataFrame 重新索引时，也可以指定一个填充值：

```
In [188]: df1.reindex(columns=df2.columns, fill_value=0)
```

Out[188]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

用这几个特定字的，叫算术方法：add/ sub/ div/ mul ，即：加 / 减 / 除 / 乘。

5、DataFrame 和 Series 之间的运算

计算一个二维数组与其某行之间的差：

```
In [189]: arr = np.arange(12.).reshape((3, 4))

In [190]: arr
Out[190]: array([[ 0.,  1.,  2.,  3.],
                 [ 4.,  5.,  6.,  7.],
                 [ 8.,  9., 10., 11.]])

In [191]: arr[0]
Out[191]: array([0., 1., 2., 3.])

In [192]: arr - arr[0]
Out[192]: array([[0., 0., 0., 0.],
                 [4., 4., 4., 4.],
                 [8., 8., 8., 8.]])
```

这个就叫做广播，下面的每行都做这个运算了。

默认情况下，DataFrame 和 Series 之间的算术运算会将 Series 的索引匹配到 DataFrame 的列，然后沿着行一直向下广播：

```
In [193]: frame = DataFrame(np.arange(12.).reshape((4,3)), columns=list('bde'),
                            index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [194]: series = frame.ix[0]
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-deprecated
  """Entry point for launching an IPython kernel.

In [195]: frame
Out[195]:
```

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [196]: series
Out[196]: b    0.0
          d    1.0
          e    2.0
          Name: Utah, dtype: float64
```

得到

```
In [197]: frame - series
```

Out[197]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

做加法 frame+series2，找不到的值就并集为 NaN。

如果你希望匹配行，且在列上广播，则必须使用算术运算方法：

```
In [198]: series3 = frame['d']

In [199]: frame
```

Out[199]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

```
In [200]: series3
```

Out[200]:

```
Utah      1.0
Ohio      4.0
Texas     7.0
Oregon    10.0
Name: d, dtype: float64
```

```
In [201]: frame.sub(series3, axis=0)
```

Out[201]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

传入的轴号就是希望匹配的轴。

6、函数的应用和映射

NumPy 的 ufuncs 可用于操作 pandas 对象，以 abs 为例：

```
In [202]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
                            index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [203]: frame
Out[203]:
```

	b	d	e
Utah	0.500638	-0.817804	-0.289553
Ohio	0.808509	1.869038	-0.693006
Texas	0.403443	0.091379	0.842242
Oregon	-0.015062	1.006599	-0.694294

```
In [204]: np.abs(frame)
Out[204]:
```

	b	d	e
Utah	0.500638	0.817804	0.289553
Ohio	0.808509	1.869038	0.693006
Texas	0.403443	0.091379	0.842242
Oregon	0.015062	1.006599	0.694294

DataFrame 的 apply 方法：将函数应用到各列或行所形成的一维数组上：

```
In [205]: f = lambda x: x.max() - x.min()

In [206]: frame.apply(f)
Out[206]: b    0.823571
          d    2.686841
          e    1.536537
          dtype: float64

In [207]: frame.apply(f, axis=1)
Out[207]: Utah      1.318442
          Ohio      2.562043
          Texas     0.750863
          Oregon    1.700893
          dtype: float64
```

许多最为常见的数据统计功能都被封装为 DataFrame 的方法，无需使用 apply 方法。

除标量值外，传递给 apply 的函数还可以返回由多个值组成的 Series：

```
In [208]: def f(x):
              return Series([x.min(), x.max()], index=['min', 'max'])

In [209]: frame.apply(f)
Out[209]:
```

	b	d	e
min	-0.015062	-0.817804	-0.694294
max	0.808509	1.869038	0.842242

用 applymap 得到 frame 中各个浮点值的格式化字符串：

```
In [210]: format = lambda x: '%.2f' %x

In [211]: frame.applymap(format)
Out[211]:
```

	b	d	e
Utah	0.50	-0.82	-0.29
Ohio	0.81	1.87	-0.69
Texas	0.40	0.09	0.84
Oregon	-0.02	1.01	-0.69

Series 有一个用于应用元素级函数的 map 方法：

```
In [212]: frame['e'].map(format)
Out[212]: Utah      -0.29
          Ohio      -0.69
          Texas      0.84
          Oregon    -0.69
          Name: e, dtype: object
```

7、排序和排名

sort_index 方法：返回一个已排序的新对象

```
In [213]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [214]: obj.sort_index()
Out[214]: a    1
          b    2
          c    3
          d    0
          dtype: int64
```

对于 DataFrame，可以根据任意一个轴上的索引进行排序：

```
In [215]: frame = DataFrame(np.arange(8).reshape((2,4)), index=['three', 'one'],
                            columns=['d', 'a', 'b', 'c'])

In [216]: frame
Out[216]:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

In [217]: frame.sort_index()
Out[217]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [218]: frame.sort_index(axis=1)
Out[218]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

指定了 axis=1，是对列进行排序。

默认按升序，降序用 ascending=False：

```
In [219]: frame.sort_index(axis=1, ascending=False)
Out[219]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

对 Series 进行排序，可用方法 sort_values()：

```
In [12]: obj = pd.Series([4, 7, -3, 2])

In [13]: obj.sort_values()
Out[13]: 2   -3
         3    2
         0    4
         1    7
         dtype: int64
```

在排序时，任何缺失值默认都会被放到 Series 末尾。

在 DataFrame 上，用 by 根据列的值进行排序：

```
In [16]: frame = DataFrame({'b': [4,7,-3,2], 'a': [0,1,0,1]})

In [17]: frame
Out[17]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [18]: frame.sort_index(by='b')
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: by argument to sort_index is deprecate
d, please use .sort_values(by=...)
  """Entry point for launching an IPython kernel.
Out[18]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7

In [19]: frame.sort_values(by='b')
Out[19]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

根据多个列：

```
In [20]: frame.sort_values(by=['a', 'b'])
Out[20]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

rank 方法：默认情况下，rank 是通过“为各组分配一个平均排名”的方式破坏平级关系的。

```
In [21]: obj = Series([7, -5, 7, 4, 2, 0 , 4])

In [22]: obj.rank()
Out[22]: 0    6.5
         1    1.0
         2    6.5
         3    4.5
         4    3.0
         5    2.0
         6    4.5
         dtype: float64
```

根据值在原数据中出现的顺序给出排名：

```
In [23]: obj.rank(method='first')
Out[23]: 0    6.0
         1    1.0
         2    7.0
         3    4.0
         4    3.0
         5    2.0
         6    5.0
         dtype: float64
```

按降序进行排名：

```
In [24]: obj.rank(ascending=False, method='max')
Out[24]: 0    2.0
         1    7.0
         2    2.0
         3    4.0
         4    5.0
         5    6.0
         6    4.0
         dtype: float64
```

DataFrame 可以在行或列上计算排名：

```
In [25]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0,1,0,1],
                            'c': [-2, 5, 8, -2.5]})
In [26]: frame
Out[26]:
```

	a	b	c
0	0	4.3	-2.0
1	1	7.0	5.0
2	0	-3.0	8.0
3	1	2.0	-2.5

```
In [27]: frame.rank(axis=1)
Out[27]:
```

	a	b	c
0	2.0	3.0	1.0
1	1.0	3.0	2.0
2	2.0	1.0	3.0
3	2.0	3.0	1.0

8、带有重复值的轴索引

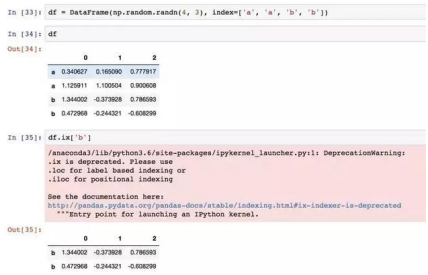
虽然许多 pandas 函数都要求标签唯一（如 reindex），但这不是强制性的。

带有重复索引的 Series：

```
In [28]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [29]: obj

Out[29]: a    0
         a    1
         b    2
         b    3
         c    4
         dtype: int64

In [30]: obj.index.is_unique

Out[30]: False

In [31]: obj['a']

Out[31]: a    0
         a    1
         dtype: int64

In [32]: obj['c']

Out[32]: 4
```

索引的 is_unique 属性可以判断它的值是否唯一。

带有重复索引的 DataFrame：



在 Pandas 中，DataFrame.ix[i] 和 DataFrame.iloc[i] 都可以选取 DataFrame 中第 i 行的数据，那么这两个命令的区别在哪里呢？

ix 可以通过行号和行标签进行索引，而 iloc 只能通过行号索引，即 ix 可以看做是 loc 和 iloc 的综合。

解析：
loc 在index的标签上进行索引,范围包括start和end.
iloc 在index的位置上进行索引,不包括end.
ix 先在index的标签上索引，索引不到就在index的位置上索引(如果index非全整数),不包括end.

汇总和计算描述统计

pandas 对象拥有一组常用的数学和统计方法：用于从 Series 中提取单个值，或从 DataFrame 的行或列中提取一个 Series。

跟 Numpy 数组方法相比，它们都是基于没有缺失数据的假设而构建的。

```
In [36]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],
                         [np.nan, np.nan], [0.75, -1.3]],
                        index=['a', 'b', 'c', 'd'],
                        columns=['one', 'two'])

In [37]: df
```

Out[37]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

```
In [38]: df.sum()
Out[38]: one    9.25
         two   -5.80
         dtype: float64
```

传入 axis=1 将会按行进行求和运算：

```
In [39]: df.sum(axis=1)
Out[39]: a    1.40
         b    2.60
         c    0.00
         d   -0.55
         dtype: float64
```

NA 值会自动被排除，如 1.40+NaN=1.40, NaN+NaN=0.00。

通过 skipna 选项可以禁用该功能：（得到 1.40+NaN=NaN, NaN+NaN=NaN）

```
In [40]: df.sum(axis=1, skipna=False)
Out[40]: a     NaN
         b    2.60
         c     NaN
         d   -0.55
         dtype: float64
```

返回间接统计（输出了值所在的行名）：

```
In [43]: df.idxmax()
Out[43]: one    b
         two    d
         dtype: object
```

累计型的（样本值的累计和）：

```
In [45]: df.cumsum()
```

Out[45]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

一次性产生多个汇总统计：

```
In [46]: df.describe()
```

Out[46]:

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

对于非数值型数据，describe 会产生另外一种汇总统计：

```
In [47]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [48]: obj.describe()
Out[48]: count      16
         unique      3
         top         a
         freq        8
         dtype: object
```

1、相关系数与协方差

Series 和 DataFrame：

- corr 方法： 相关系数
- cov 方法：协方差

DataFrame 的 corrwith 方法：计算其列或行跟另一个 Series 或 DataFrame 之间的相关系数。传入一个 DataFrame 计算按列名配对的相关系数，传入 axis=1 即可按行进行计算。

2、唯一值、值计数以及成员资格

从一维 Series 的值中抽取信息。

unique 函数：得到 Series 中的唯一值数组

```
In [69]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])

In [70]: uniques = obj.unique()

In [71]: uniques
Out[71]: array(['c', 'a', 'd', 'b'], dtype=object)

In [72]: uniques.sort()

In [73]: uniques
Out[73]: array(['a', 'b', 'c', 'd'], dtype=object)
```

value_counts：用于计算一个 Series 中各值出现的频率：

```
In [74]: obj.value_counts()
Out[74]: c    3
         a    3
         b    2
         d    1
         dtype: int64

In [76]: pd.value_counts(obj.values, sort=False)
Out[76]: a    3
         b    2
         d    1
         c    3
         dtype: int64

In [78]: obj.values
Out[78]: array(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'], dtype=object)
```

Series 按降序排列。value_counts 是一个顶级 pandas 方法，可用于任何数组或序列。

isin：用于判断矢量化集合的成员资格，可用于选取 Series 或 DataFrame 列中数据的子集：

```
In [79]: mask = obj.isin(['b', 'c'])

In [80]: mask
Out[80]: 0     True
         1    False
         2    False
         3    False
         4    False
         5     True
         6     True
         7     True
         8     True
         dtype: bool

In [81]: obj[mask]
Out[81]: 0    c
         5    b
         6    b
         7    c
         8    c
         dtype: object
```

处理缺失数据

pandas 的设计目标之一就是让缺失数据的处理任务尽量轻松。

pandas 使用浮点值 NaN(Not a Number) 表示浮点和非浮点数组中的缺失数据。它只是一个便于被检测出来的标记而已。

python 内置的 None 值也会被当做 NA 处理（如 string_data[0]=None）。

1、滤掉缺失数据

对于一个 Series, dropna 返回一个仅含非空数据和索引值的 Series：

```
In [89]: from numpy import nan as NA

In [90]: data = Series([1, NA, 3.5, NA, 7])

In [92]: data.dropna()
Out[92]: 0    1.0
         2    3.5
         4    7.0
         dtype: float64
```

通过布尔型索引也可以达到这个目的：

```
In [93]: data[data.notnull()]

Out[93]: 0    1.0
         2    3.5
         4    7.0
         dtype: float64
```

对于 DataFrame 对象，dropna 默认丢弃任何含有缺失值的行：

```
In [95]: data = DataFrame([[1.,6.5,3.], [1.,NA,NA],
                           [NA,NA,NA],[NA,6.5,3.]])

In [96]: cleaned = data.dropna()

In [97]: data

Out[97]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [98]: cleaned

Out[98]:
```

	0	1	2
0	1.0	6.5	3.0

丢弃全为 NA 的那些行，axis=1 则丢弃列：

```
In [99]: data.dropna(how='all')
```

Out[99]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

```
In [100]: data[4] = NA
```

```
In [101]: data
```

Out[101]:

	0	1	2	4
0	1.0	6.5	3.0	NaN
1	1.0	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN
3	NaN	6.5	3.0	NaN

```
In [102]: data.dropna(axis=1, how='all')
```

Out[102]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

只想留下一部分参数，用 thresh 参数：

```
In [103]: df = DataFrame(np.random.randn(7,3))
```

```
In [104]: df.loc[:4,1] = NA; df.loc[:2,2] = NA
```

```
In [105]: df
```

Out[105]:

	0	1	2
0	-1.004434	NaN	NaN
1	0.140507	NaN	NaN
2	1.177915	NaN	NaN
3	0.660015	NaN	-0.662396
4	0.142568	NaN	-0.446689
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

```
In [106]: df.dropna(thresh=3)
```

Out[106]:

	0	1	2
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

thresh=3：保留至少 3 个非空值的行，即一行中有 3 个值是非空的就保留.

2、填充缺失数据

fillna 方法：通过一个常数调用 fillna 就会将缺失值替换为那个常数值。

```
In [107]: df.fillna(0)
Out[107]:
```

	0	1	2
0	-1.004434	0.000000	0.000000
1	0.140507	0.000000	0.000000
2	1.177915	0.000000	0.000000
3	0.660015	0.000000	-0.662396
4	0.142568	0.000000	-0.446689
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

```
In [108]: df
Out[108]:
```

	0	1	2
0	-1.004434	NaN	NaN
1	0.140507	NaN	NaN
2	1.177915	NaN	NaN
3	0.660015	NaN	-0.662396
4	0.142568	NaN	-0.446689
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

通过一个字典调用 fillna，可以实现对不同的列填充不同的值：

```
In [109]: df.fillna({1:0.5, 3:-1})
Out[109]:
```

	0	1	2
0	-1.004434	0.500000	NaN
1	0.140507	0.500000	NaN
2	1.177915	0.500000	NaN
3	0.660015	0.500000	-0.662396
4	0.142568	0.500000	-0.446689
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

fillna 默认会返回新对象（副本），但也可以对现有对象进行就地修改：

```
In [110]: _ = df.fillna(0, inplace=True)  #修改调用者对象而不产生副本
In [111]: df
Out[111]:
```

	0	1	2
0	-1.004434	0.000000	0.000000
1	0.140507	0.000000	0.000000
2	1.177915	0.000000	0.000000
3	0.660015	0.000000	-0.662396
4	0.142568	0.000000	-0.446689
5	1.317111	-0.047906	-0.897771
6	-2.065662	0.034912	1.070042

插值方法（对 reindx 有效的也可用于 fillna）：

```
In [112]: df = DataFrame(np.random.randn(6,3))

In [113]: df.loc[2:, 1] = NA;  df.loc[4:, 2] = NA

In [114]: df
Out[114]:
```

	0	1	2
0	0.833757	-0.063065	-0.010618
1	-1.718235	-0.013298	0.281971
2	0.416726	NaN	0.290094
3	-0.658968	NaN	0.346584
4	0.974747	NaN	NaN
5	0.513581	NaN	NaN

```
In [115]: df.fillna(method='ffill')
Out[115]:
```

	0	1	2
0	0.833757	-0.063065	-0.010618
1	-1.718235	-0.013298	0.281971
2	0.416726	-0.013298	0.290094
3	-0.658968	-0.013298	0.346584
4	0.974747	-0.013298	0.346584
5	0.513581	-0.013298	0.346584

```
In [116]: df.fillna(method='ffill', limit=2)
Out[116]:
```

	0	1	2
0	0.833757	-0.063065	-0.010618
1	-1.718235	-0.013298	0.281971
2	0.416726	-0.013298	0.290094
3	-0.658968	-0.013298	0.346584
4	0.974747	NaN	0.346584
5	0.513581	NaN	0.346584

你可以用 fillna 实现许多别的功能，比如传入 Series 的平均值或中位数：

```
In [117]: data = Series([1., NA, 3.5, NA, 7])

In [118]: data.fillna(data.mean())

Out[118]: 0    1.000000
          1    3.833333
          2    3.500000
          3    3.833333
          4    7.000000
          dtype: float64
```

层次化索引

在一个轴上用多个（2 个以上）索引级别，即以低维度形式处理高维度数据。

MultiIndex 索引的 Series 的格式化输出形式：

```
In [119]: data = Series(np.random.randn(10),
                        index=[['a','a','a','b','b','b','c','c','d','d'],
                               [1,2,3,1,2,3,1,2,2,3]])

In [120]: data

Out[120]: a  1   -1.339425
          2   -0.454056
          3    1.826734
       b  1   -0.462856
          2   -0.372202
          3   -1.337323
       c  1   -1.368236
          2   -0.133705
       d  2   -0.918471
          3    0.981914
       dtype: float64

In [121]: data.index

Out[121]: MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
                     labels=[[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 2]])
```

选取数据子集：

```
In [122]: data['b']

Out[122]: 1   -0.462856
          2   -0.372202
          3   -1.337323
          dtype: float64

In [124]: data['b':'c']

Out[124]: b  1   -0.462856
             2   -0.372202
             3   -1.337323
          c  1   -1.368236
             2   -0.133705
          dtype: float64

In [126]: data.loc[['b','d']]

Out[126]: b  1   -0.462856
             2   -0.372202
             3   -1.337323
          d  2   -0.918471
             3    0.981914
          dtype: float64
```

在“内层”中进行选取：

```
In [127]: data[:, 2]  #'a','b','c','d'中label为 2 的

Out[127]: a   -0.454056
          b   -0.372202
          c   -0.133705
          d   -0.918471
          dtype: float64
```

层次化索引在**数据重塑**和**基于分组的操作**中很重要。比如说，上面的数据可以通过其 unstack 方法被重新安排到一个 DataFrame 中，它的逆运算是 stack：

```
In [128]: data.unstack()

Out[128]:
```

	1	2	3
a	-1.339425	-0.454056	1.826734
b	-0.462856	-0.372202	-1.337323
c	-1.368236	-0.133705	NaN
d	NaN	-0.918471	0.981914

```
In [129]: data.unstack().stack()

Out[129]: a  1   -1.339425
             2   -0.454056
             3    1.826734
          b  1   -0.462856
             2   -0.372202
             3   -1.337323
          c  1   -1.368236
             2   -0.133705
          d  2   -0.918471
             3    0.981914
          dtype: float64
```

对于一个 DataFrame，每条轴都可以有分层索引：

```
In [132]: frame = DataFrame(np.arange(12).reshape((4,3)),
                            index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                            columns=[['Ohio', 'Ohio', 'Colorado'],
                                     ['Green', 'Red', 'Green']])
```

```
In [133]: frame
```

Out[133]:

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

各层都可以有名字（可以是字符串，也可以是别的 Python 对象）。

注意不要将**索引名称**跟**轴标签**混为一谈。

```
In [135]: frame.index.names = ['Key1','Key2']
```

```
In [136]: frame.columns.names = ['state', 'color']
```

```
In [137]: frame
```

Out[137]:

state		Ohio		Colorado
color		Green	Red	Green
Key1	Key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

```
In [138]: frame['Ohio']
```

Out[138]:

color		Green	Red
Key1	Key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

有了分部的列索引，可以轻松选取列分组。

可以单独创建 MultiIndex 然后复用。上面的 DataFrame 中的分级列可以这样创建：

```
In [143]: pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                                    names=['State', 'color'])
Out[143]: MultiIndex(levels=[['Colorado', 'Ohio'], ['Green', 'Red']],
                     labels=[[1, 1, 0], [0, 1, 0]],
                     names=['State', 'color'])
```

1、重排分级顺序

重新调整某条轴上各级别的顺序，或根据指定级别上的值对数据进行排序。

swaplevel：接受两个级别编号或名称，返回一个互换了级别的新对象，数据不发生改变：

```
In [145]: frame.swaplevel('Key1','Key2')
```

Out[145]:

	state	Ohio		Colorado
	color	Green	Red	Green
Key2	Key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

sortlevel：根据单个级别中的值对数据进行排序（得到的最终结果是有序的）

```
In [146]: frame.sortlevel(1)
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: sortlevel is deprecated, use sort_index(level= ...)
  """Entry point for launching an IPython kernel.
```

Out[146]:

	state	Ohio		Colorado
	color	Green	Red	Green
Key1	Key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

```
In [147]: frame.sort_index(level=1)
```

Out[147]:

	state	Ohio		Colorado
	color	Green	Red	Green
Key1	Key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

2、根据级别汇总统计

level 选项：用于指定在某条轴上求和的级别。

如下所示，分别根据行或列上的级别来对行、对列进行求和：

```
In [149]: frame.sum(level='Key2')
```

Out[149]:

state	Ohio		Colorado
color	Green	Red	Green
Key2			
1	6	8	10
2	12	14	16

```
In [150]: frame.sum(level='color', axis=1)
```

Out[150]:

	color	Green	Red
Key1	Key2		
a	1	2	1
	2	8	4
b	1	14	7
	2	20	10

3、使用 DataFrame 的列

将 DataFrame 的一个或多个列当做行索引来用，或将行索引变成 DataFrame 的列：

```
In [151]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
                             'c':['one', 'one', 'one', 'two', 'two', 'two', 'two'],
                             'd':[0,1,2,0,1,2,3]})
In [152]: frame
Out[152]:
```

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

```
In [153]: frame2 = frame.set_index(['c', 'd'])
In [154]: frame2
Out[154]:
```

		a	b
c	d		
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

set_index 函数：将其一个或多个列转换为行索引，并创建一个新的 DataFrame。

默认情况下，那些列会从 DataFrame 中移除，也可以将其保留下来：

```
In [155]: frame.set_index(['c', 'd'], drop=False)
Out[155]:
```

		a	b	c	d
c	d				
one	0	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	0	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

reset_index：将层次化索引的级别转移到列里面（和 set_index 相反）

```
In [156]: frame2.reset_index()
```

Out[156]:

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

不足之处，欢迎指正。