Pandas 学习笔记

（利用 **pandas** 进行数据分析）

目录

NumPy 基础：数组和矢量计算

```
import numpy as np
```

NumPy 的 ndarray：一种多维数组对象

创建 ndarray

创建数组最简单的办法就是使用 array 函数。

```
data1 = range(10)
arr1 = np.array(data1)
data2 = [range(4),range(4)]
arr2 = np.array(data2)
print arr2.shape
print arr2.ndim
```

```
(2, 4)
2
```

```
print np.ones(4)
print np.ones_like(arr1)
```

```
[ 1.  1.  1.  1.]
[1 1 1 1 1 1 1 1 1 1]
```

```
print np.zeros((2,3))
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

表4-1：数组创建函数

函数	说明
array	将输入数据（列表、元组、数组或其他序列类型）转换为ndarray。要么推断出dtype，要么显式指定dtype。默认直接复制输入数据
asarray	将输入转换为ndarray，如果输入本身就是一个ndarray就不进行复制
arange	类似于内置的range，但返回的是一个ndarray而不是列表
ones、ones_like	根据指定的形状和dtype创建一个全1数组。ones_like以另一个数组为参数，并根据其形状和dtype创建一个全1数组
zeros、zeros_like	类似于ones和ones_like，只不过产生的是全0数组而已
empty、empty_like	创建新数组，只分配内存空间但不填充任何值
eye、identity	创建一个正方的N×N单位矩阵（对角线为1，其余为0）

数组和标量之间的运算

数组很重要，因为它使你不用编写循环即可对数据执行批量运算。这通常就叫做矢量化。

```
arr = np.array([[1.,2.,3.,4.],[1.,2.,3.,4.]])
print arr
```

```
[[ 1.  2.  3.  4.]
 [ 1.  2.  3.  4.]]
```

```
print arr*arr
print 1/arr
print a**0.5
```

```
[[  1.   4.   9.  16.]
 [  1.   4.   9.  16.]]
[[ 1.          0.5         0.33333333  0.25      ]
 [ 1.          0.5         0.33333333  0.25      ]]
[[ 1.          1.41421356  1.73205081  2.        ]
 [ 1.          1.41421356  1.73205081  2.        ]]
```

基本的索引和切片

```
arr = np.arange(10)
print arr[5:8]
```

```
[5 6 7]
```

跟列表最重要的区别在于，数组切片是原始数组的视图。意味着数据不会被复制，视图上的任何修改都会直接反映到源数组上：

```
arr_slice = arr[5:8]
arr_slice[1] = 100
print arr
```

```
[  0   1   2   3   4   5 100   7   8   9]
```

如果你想得到的是 ndarray 切片的一份副本而非视图，就需要显式地进行复制操作，例如 **arr[5:8].copy()**

切片索引

```
arr2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print arr2d
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
print arr2d[:2]
```

```
[[1 2 3]
 [4 5 6]]
```

```
print arr2d[:2,:1]
```

```
[[1]
 [4]]
```

注意：“只有冒号”表示选取整个轴：

```
print arr2d[:, :1]
```

```
[[1]
 [4]
 [7]]
```

布尔型索引

```
names = np.array(['bob','joe','will','bob'])
data = randn(4,5)
print names
print data
```

```
['bob' 'joe' 'will' 'bob']
[[-1.19091387  0.14642273 -0.66863316  0.44013788 -1.29671215]
 [-0.80908315  1.07817172  0.04415241 -0.51397585 -0.94635326]
 [-0.37697096  1.56597667  0.56987776  0.84733407 -0.60212763]
 [ 0.18894967 -1.51679444  0.01545517  0.16149475 -1.12305223]]
```

```
print data[names == 'bob']
```

```
[[-1.19091387  0.14642273 -0.66863316  0.44013788 -1.29671215]
 [ 0.18894967 -1.51679444  0.01545517  0.16149475 -1.12305223]]
```

通过布尔型数组设置值是一种经常用的手段。为了将 data 中的所有负值都设置为 0，我们只需：

```
data[data<0] = 0
print data
```

```
[[ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.        ]
 [ 0.          0.00512938  0.67380362  0.78094606  0.        ]
 [ 0.          0.32336722  0.25130047  0.          1.08485281]]
```

数组转置和轴对换

```
arr = np.arange(15).reshape(3,5)
print arr
print arr.T
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

np.dot 计算矩阵乘法，可以利用 np.dot 计算内积

```
print np.dot(arr, arr.T)
```

```
[[ 30   80 130]
 [ 80 255 430]
 [130 430 730]]
```

通用函数：快速的元素级数组函数

通用函数（即 ufunc）是一种对 ndarray 中的数组执行元素级运算的函数。

```
arr = np.arange(5)
print np.sqrt(arr)
print np.exp(arr)
```

```
[ 0.          1.          1.41421356  1.73205081  2.        ]
[  1.           2.71828183   7.3890561   20.08553692  54.59815003]
```

表4-3：一元ufunc

函数	说明
abs、fabs	计算整数、浮点数或复数的绝对值。对于非复数值，可以使用更快的fabs
sqrt	计算各元素的平方根。相当于arr ** 0.5
square	计算各元素的平方。相当于arr ** 2
exp	计算各元素的指数e^x
log、log10、log2、log1p	分别为自然对数（底数为e）、底数为10的log、底数为2的log、log(1 + x)
sign	计算各元素的正负号：1（正数）、0（零）、-1（负数）
ceil	计算各元素的ceiling值，即大于等于该值的最小整数
floor	计算各元素的floor值，即小于等于该值的最大整数
rint	将各元素值四舍五入到最接近的整数，保留dtype
modf	将数组的小数和整数部分以两个独立数组的形式返回
isnan	返回一个表示“哪些值是NaN（这不是一个数字）”的布尔型数组
isfinite、isinf	分别返回一个表示“哪些元素是有穷的（非inf，非NaN）”或“哪些元素是无穷的”的布尔型数组
cos、cosh、sin、sinh、tan、tanh	普通型和双曲型三角函数

函数	说明
arccos、arccosh、arcsin、arcsinh、arctan、arctanh	反三角函数
logical_not	计算各元素not x的真值。相当于-arr

表4-4：二元ufunc

函数	说明
add	将数组中对应的元素相加
subtract	从第一个数组中减去第二个数组中的元素
multiply	数组元素相乘
divide、floor_divide	除法或向下圆整除法（丢弃余数）
power	对第一个数组中的元素A，根据第二个数组中的相应元素B，计算A^B
maximum、fmax	元素级的最大值计算。fmax将忽略NaN
minimum、fmin	元素级的最小值计算。fmin将忽略NaN
mod	元素级的求模计算（除法的余数）
copysign	将第二个数组中的值的符号复制给第一个数组中的值
greater、greater_equal、less、less_equal、equal、not_equal	执行元素级的比较运算，最终产生布尔型数组。相当于中缀运算符>、>=、<、<=、==、!=
logical_and、logical_or、logical_xor	执行元素级的真值逻辑运算。相当于中缀运算符&、|、^

```
arr = np.arange(5)
print np.maximum(arr, arr*2)
```

```
[0 2 4 6 8]
```

利用数组进行数据处理

将条件逻辑表述为数组运算

numpy.where 函数是三元表达式 x if condition else y 的矢量版本。

result = np.where(cond, xarr, yarr)

np.where 的第二个和第三个参数不必是数组，它们可以是标量值。

```
arr = randn(4,4)
print arr
```

```
[[ 0.5226102   -0.62470655 -0.78233791 -0.6008009 ]
 [-1.04404238 -0.54215402  0.45638562  1.43093423]
 [ 0.76307658 -1.28710021  0.41550553  1.22019516]
 [ 0.31424298 -0.29771119  0.48784945 -1.37217858]]
```

```
arr1 = np.where(arr>0, 2, -2)
print arr1
```

```
[[ 2 -2 -2 -2]
 [-2 -2  2  2]
```

```
 [ 2 -2   2   2]
 [ 2 -2   2 -2]]
```

```
arr = np.where(arr<0, 0, arr)
print arr
```

```
[[ 0.5226102   0.          0.          0.        ]
 [ 0.          0.          0.45638562  1.43093423]
 [ 0.76307658  0.          0.41550553  1.22019516]
 [ 0.31424298  0.          0.48784945  0.        ]]
```

用 where 表述出更复杂的逻辑。

np.where(cond1 & cond2, 0, np.where(cond1, 1, np.where(cond2, 2, 3)))

数学和统计方法

```
arr = np.array([[1,2,3,4], [4,5,6,7], [6,7,8,9]])
print arr.mean()
print np.mean(arr)
print arr.sum()
```

```
5.16666666667
5.16666666667
62
```

Mean 和 sum 这类的函数可以接受一个 axis 参数（用于计算该轴向上的统计值）

```
print arr.sum(1)
print arr.sum(0)
```

```
[10 22 30]
[11 14 17 20]
```

```
print arr.cumprod(1)
```

```
[[   1    2    6   24]
 [   4   20  120  840]
 [   6   42  336 3024]]
```

表4-5：基本数组统计方法

方法	说明
sum	对数组中全部或某轴向的元素求和。零长度的数组的sum为0
mean	算术平均数。零长度的数组的mean为NaN
std、var	分别为标准差和方差，自由度可调（默认为n）
min、max	最大值和最小值
argmin、argmax	分别为最大和最小元素的索引

方法	说明
cumsum	所有元素的累计和
cumprod	所有元素的累计积

用于布尔型数组的方法

在上面这些方法中，布尔值会被强制转换为 1（True）和 0（False）。因此，sum 经常被用来对布尔型数组中的 True 值计数：

```
arr = randn(100)
print (arr > 0).sum()
```

45

另外还有两个方法 any 和 all。any 用于测试数组中是否存在一个或多个 True，而 all 检查数组中所有值是否都是 True：

```
print (arr>0).any()
print (arr>0).all()
```

True

False

排序

Numpy 数组也可以通过 sort 方法**就地排序**：

```
arr = randn(5)
print arr
arr.sort()
print arr
```

[0.12985911 0.52741643 -0.70875679 0.53760558 -0.11906363]

[-0.70875679 -0.11906363 0.12985911 0.52741643 0.53760558]

顶级方法 np.sort 返回的是数组的已排序副本，而就地排序则会修改数组本身。

唯一化以及其他的集合逻辑

针对一维 ndarray 的基本集合运算。

```
arr = np.array([1,2,2,3,4,4])
print np.unique(arr)
```

[1 2 3 4]

表4-6：数组的集合运算

方法	说明
unique(x)	计算x中的唯一元素，并返回有序结果
intersect1d(x, y)	计算x和y中的公共元素，并返回有序结果
union1d(x, y)	计算x和y的并集，并返回有序结果
in1d(x, y)	得到一个表示“x的元素是否包含于y”的布尔型数组
setdiff1d(x, y)	集合的差，即元素在x中且不在y中
setxor1d(x, y)	集合的对称差，即存在于一个数组中但不同时存在于两个数组中的元素[译注2]

线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分。通过*对两个二维数组相乘得到的是一个元素级的积，不像matlab得到的是矩阵点积。因此，numpy 提供了一个用于矩阵乘法的 dot 函数。

```
x = np.array([[1,2,3], [4,5,6]])
y = np.array([[6,23], [-1,7], [8,9]])
print x.dot(y)
print np.dot(x,y)
```

```
[[ 28  64]
 [ 67 181]]
[[ 28  64]
 [ 67 181]]
```

numpy.linalg 中有一组标准的矩阵分解运算以及诸如求逆和行列式之类的东西。

```
from numpy.linalg import inv, qr
x = randn(4,4)
mat = x.T.dot(x)
print inv(mat)
```

```
[[ 0.30159881 -0.44079435 -0.01231646 -0.04637054]
 [-0.44079435  2.0874644  -0.56048726  0.59833859]
 [-0.01231646 -0.56048726  0.61425303 -0.3809892 ]
 [-0.04637054  0.59833859 -0.3809892   0.51864183]]
```

```
print mat.dot(inv(mat))
```

```
[[  1.00000000e+00  -1.11022302e-16  -5.55111512e-17   5.55111512e-17]
 [ -6.24500451e-17   1.00000000e+00  -5.55111512e-17   5.55111512e-17]
 [  0.00000000e+00  -4.44089210e-16   1.00000000e+00   0.00000000e+00]
 [  2.77555756e-17  -4.44089210e-16   2.22044605e-16   1.00000000e+00]]
```

```
q,r = qr(mat)
print r
```

```
[[-6.30889408 -2.19516073 -2.20461763  0.48131017]
```

```
[ 0.          -0.97502881   0.47691279   2.73025888]
[ 0.           0.          -3.57698221  -3.54053026]
[ 0.           0.           0.           1.13643598]]
```

表4-7：常用的numpy.linalg函数

函数	说明
diag	以一维数组的形式返回方阵的对角线（或非对角线）元素，或将一维数组转换为方阵（非对角线元素为0）
dot	矩阵乘法
trace	计算对角线元素的和
det	计算矩阵行列式
eig	计算方阵的本征值和本征向量
inv	计算方阵的逆
pinv	计算矩阵的Moore-Penrose伪逆
qr	计算QR分解
svd	计算奇异值分解（SVD）
solve	解线性方程组Ax = b，其中A为一个方阵
lstsq	计算Ax = b的最小二乘解

随机数生成

```
from numpy import random
arr = np.arange(10)
random.shuffle(arr)
print arr
```

```
[2 7 3 9 4 8 0 6 1 5]
```

```
print random.rand(2,3)
```

```
[[ 0.76720316  0.50223528  0.11335512]
 [ 0.67951696  0.78906827  0.71987921]]
```

```
print random.randint(1,10,size=(2,3))
```

```
[[2 9 6]
 [8 6 5]]
```

```
print random.randn(2,3)
```

```
[[ 0.54188153 -1.05622626  0.05599057]
 [-1.22568593 -1.16206795  0.56627087]]
```

```
print random.normal(10,10,size=(2,3))
```

```
[[ 16.63356884  -4.65668552  11.71878629]
 [  2.39799936  16.36184207   0.83815035]]
```

表4-8：部分numpy.random函数

函数	说明
seed	确定随机数生成器的种子
permutation	返回一个序列的随机排列或返回一个随机排列的范围
shuffle	对一个序列就地随机排列
rand	产生均匀分布的样本值
randint	从给定的上下限范围内随机选取整数
randn	产生正态分布（平均值为0，标准差为1）的样本值，类似于MATLAB接口
binomial	产生二项分布的样本值
normal	产生正态（高斯）分布的样本值
beta	产生Beta分布的样本值

Pandas 入门

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

pandas 的数据结构介绍

Series

Series 是一种类似于一维数组的对象，它由一组数据以及一组与之相关的数据标签（即索引）组成。

```
obj = Series([4,7,-1,8])
print obj
```

```
0    4
1    7
2   -1
3    8
```

Series 的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据制定索引，于是会自动创建一个 0 到 N-1 的整数型索引。

```
obj = Series([4,1,-9,0],index=['a','b','c','d'])
print obj
```

```
a    4
b    1
c   -9
d    0
```

```
print obj.values
[ 4   1 -9   0]
print obj.index
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

可以通过**索引的方式选取 Series 中的单个或一组值：**

```
print obj['a']
4
print obj[['a','c']]
a    4
c   -9
```

numpy 数组运算（如根据布尔型数组进行过滤、标量乘法、应用数学函数等）都会保留索引和值之间的连接：

```
print obj[obj>0]
a    4
b    1
print obj*2
a     8
b     2
c   -18
d     0
print obj.abs()
a    4
b    1
c    9
d    0
```

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

```
sdata = {'lhq':100, 'hyj':100, 'sb':0}
obj = Series(sdata)
print obj
hyj    100
lhq    100
sb       0
index_new = ['a','sb']
sdata = {'lhq':100, 'hyj':100, 'sb':0}
obj = Series(sdata, index=index_new)
print obj
a    NaN
sb     0
```

结果为 NaN（即“非数字”，在 pandas 中，他用于表示缺失或 NA 值）。将使用缺失或 NA 表示缺失数据。Pandas 的 isnull 和 notnull 函数可用于检测缺失数据：

```
print pd.isnull(obj)
print pd.notnull(obj)
```

```
a        True
sb      False
a       False
sb       True
```

Series 最重要的一个功能是：它在算术运算中会自动对齐不同索引的数据。

```
lhq = Series({'name':'lhq', 'sex':'man', 'weigh':'70kg'})
hyj = Series({'name':'hyj', 'sex':'girl', 'smile':'nice'})
print lhq
print hyj
```

```
name        lhq
sex         man
weigh      70kg
name        hyj
sex        girl
smile      nice
```

```
print lhq + hyj
```

```
name        lhqhyj
sex        mangirl
smile          NaN
weigh          NaN
```

DataFrame

Dataframe 是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame 既有行索引也有列索引，它可以被看作由 Series 组成的字典。

创建 DataFrame

构建 DataFrame 的办法很多，最常用的一种是直接传入一个由等长列表或 numpy 数组组成的字典：

```
data = {'lhq':[100, 170, 70,], 'hyj':[100, 160, 50], }
frame = DataFrame(data)
```

```
   hyj  lhq
0  100  100
1  160  170
2   50   70
```

如果指定了列序列，则 DataFrame 的列会按照指定顺序进行排列：

```
frame = DataFrame(data, columns=['hyj', 'lhq'])
```

```
   hyj  lhq
0  100  100
1  160  170
2   50   70
```

如果传入的列在数据中找不到，就会产生 NA 值：

```
frame = DataFrame(data, columns=['hyj', 'lhq', 'sb'], index=['look', 'high',
'weigh'])
```

```
       hyj  lhq   sb
look   100  100  NaN
high   160  170  NaN
weigh   50   70  NaN
```

跟 Series 一样，values 属性会以二维 ndarray 的形式返回 DataFrame 中的数据：

```
print frame.values
```

```
[[100L 100L nan]
 [160L 170L nan]
 [50L 70L nan]]
```

访问行列

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：

```
print frame['lhq']
print frame.lhq
```

```
look     100
high     170
weigh     70
```

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：

```
print frame.ix['weigh']
```

```
hyj     50
lhq     70
sb     NaN
```

修改列

列可以通过赋值的方式进行修改：

```
frame['sb'] = 1000
```

```
        hyj  lhq   sb
look    100  100  1000
high    160  170  1000
weigh    50   70  1000
```

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，所有的空位将被填上 NA：

```
val = Series([100, 100, 100], index=['look', 'weigh', 'cc'])
frame['sbII'] = val
```

```
        hyj  lhq   sb  sbII
look    100  100  1000   100
high    160  170  1000   NaN
weigh    50   70  1000   100
```

关键字 del 用于删除列

```
del frame['sbII']
```

```
        hyj  lhq   sb
look    100  100  1000
high    160  170  1000
weigh    50   70  1000
```

通过索引方式返回的列只是相应数据的视图而已，并不是副本。因此，对返回的 Series 所做的任何就地修改全都会反映到源 DataFrame 上。通过 Series 的 copy 方法即可显式地复制列。

进行转置：

```
frame = frame.T
```

```
      look  high  weigh
hyj    100   160     50
lhq    100   170     70
sb    1000  1000   1000
```

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA/缺失值

基本功能

重新索引

Pandas 对象的一个重要方法是 reindex，其作用是创建一个适应新索引的新对象。

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

```
obj = Series([-1, 0, 1,], index=['a', 'b', 'c'])
obj1 = obj.reindex(index=['a', 'b','c', 'd'])
a    -1
b     0
c     1
d   NaN
obj2 = obj.reindex(index=['a', 'b','c', 'd'], fill_value=0)
a   -1
b    0
c    1
d    0
```

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插值处理。Method 选项即可达到此目的：

```
obj3 = obj.reindex(index=['a','b','e','f'], method='ffill')
a   -1
b    0
e    1
f    1
```

表5-4：reindex的（插值）method选项

参数	说明
ffill或pad	前向填充（或搬运）值
bfill或backfill	后向填充（或搬运）值

对于 DataFrame，reindex 可以修改（行）索引、列，或两个都修改。

重新索引行：

```
frame = DataFrame(np.arange(9).reshape(3,3), index=['a','b','c'],
columns=['lhq','hyj','sb'])
frame1 = frame.reindex(index=['a','b','c','d'])
   lhq  hyj  sb
a    0    1   2
b    3    4   5
c    6    7   8
d  NaN  NaN NaN
```

利用 columns 关键字重新索引列：

```
frame2 = frame.reindex(columns=['lhq','hyj','sb','sbII'])
   lhq  hyj  sb  sbII
a    0    1   2   NaN
b    3    4   5   NaN
c    6    7   8   NaN
```

同时对行和列进行重新索引：

```
frame3 = frame.reindex(index=['a','b','c','d'],
columns=['lhq','hyj','sb','sbII'], method='ffill')
   lhq  hyj  sb  sbII
a    0    1   2   NaN
b    3    4   5   NaN
c    6    7   8   NaN
d    6    7   8   NaN
```

表5-5：reindex函数的参数

参数	说明
index	用作索引的新序列。既可以是Index实例，也可以是其他序列型的Python数据结构。Index会被完全使用，就像没有任何复制一样
method	插值（填充）方式，具体参数请参见表5-4
fill_value	在重新索引的过程中，需要引入缺失值时使用的替代值
limit	前向或后向填充时的最大填充量
level	在MultiIndex的指定级别上匹配简单索引，否则选取其子集
copy	默认为True，无论如何都复制；如果为False，则新旧相等就不复制

丢弃指定轴上的项

Drop 方法返回的是一个在指定轴上删除了指定值的新对象：

```
obj = Series([-1, 0, 1,], index=['a', 'b', 'c'])
obj1 = obj.drop(['a'])
```

```
b    0
c    1
```

对于 DataFrame，可以删除任意轴上的索引值：

```
frame = DataFrame(np.arange(16).reshape(4,4), index=['a','b','c','d'],
columns=['lhq','hyj','sb','sbII'])
frame1 = frame.drop(['a','b'])
```

```
   lhq  hyj  sb  sbII
c    8    9  10    11
d   12   13  14    15
```

```
frame2 = frame.drop(['sb','sbII'], axis=1)
```

```
   lhq  hyj
a    0    1
b    4    5
c    8    9
d   12   13
```

索引、选取和过滤

Series 索引

```
obj = Series(np.arange(4), index=['a','b','c','d'])
print obj['b']
print obj[1]
print obj[2:4]
print obj[['a','c']]
```

```
1
1
c    2
d    3
dtype: int32
a    0
c    2
dtype: int32
```

DataFrame 索引

```
frame = DataFrame(np.arange(16).reshape(4,4), index=['a','b','c','d'],
columns=['lhq','hyj','sb','sbII'])
```

```
   lhq  hyj  sb  sbII
a    0    1   2     3
b    4    5   6     7
c    8    9  10    11
d   12   13  14    15
```

选取列：

```
print frame[['lhq','hyj']]
```

```
   lhq  hyj
a    0    1
b    4    5
c    8    9
d   12   13
```

```
print frame.ix[:,['lhq','hyj']]
```

```
   lhq  hyj
a    0    1
b    4    5
c    8    9
d   12   13
```

选取行：

```
print frame[:2]
```

```
   lhq  hyj  sb  sbII
a    0    1   2     3
b    4    5   6     7
```

布尔选行：

```
print frame[frame['lhq']<6]
```

```
   lhq  hyj  sb  sbII
a    0    1   2     3
b    4    5   6     7
```

```
print frame.ix[['a','c']]
```

```
   lhq  hyj  sb  sbll
a    0    1   2     3
c    8    9  10    11
```

选取行列子集：

```
print frame.ix[:2,['lhq','hyj']]
```

布尔索引：

```
frame[frame<5]=0
print frame
```

```
   lhq  hyj  sb  sbll
a    0    0   0     0
b    0    5   6     7
c    8    9  10    11
d   12   13  14    15
```

表5-6：DataFrame的索引选项

类型	说明
obj[val]	选取DataFrame的单个列或一组列。在一些特殊情况下会比较便利：布尔型数组（过滤行）、切片（行切片）、布尔型DataFrame（根据条件设置值）
obj.ix[val]	选取DataFrame的单个行或一组行
obj.ix[:, val]	选取单个列或列子集
obj.ix[val1, val2]	同时选取行和列
reindex方法	将一个或多个轴匹配到新索引
xs方法	根据标签选取单行或单列，并返回一个Series
icol、irow方法	根据整数位置选取单列或单行，并返回一个Series
get_value、set_value方法	根据行标签和列标签选取单个值。译注2

算术运算和数据对齐

Pandas 最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时，如果存在不同的索引对，则结果的索引就是该索引对的并集。

```
frame1 = DataFrame(np.arange(9).reshape(3,3), index=['a','b','c'],
columns=['lhq','hyj','sb'])
```

```
   lhq  hyj  sb
a    0    1   2
```

```
b     3     4    5
c     6     7    8
```

```
frame2 = DataFrame(np.arange(12).reshape(4,3), index=['a','b','c','d'],
columns=['lhq','hyj','sbII'])
```

```
    lhq   hyj   sbII
a     0     1      2
b     3     4      5
c     6     7      8
d     9    10     11
```

```
frame = frame1 + frame2
```

```
    hyj   lhq   sb   sbII
a     2     0 NaN    NaN
b     8     6 NaN    NaN
c    14    12 NaN    NaN
d   NaN   NaN NaN    NaN
```

在算术方法中填充值

```
frame = frame1.add(frame2,)
frame[pd.isnull(frame)]=0
```

```
    hyj   lhq   sb   sbII
a     2     0    0      0
b     8     6    0      0
c    14    12    0      0
d     0     0    0      0
```

表5-7：灵活的算术方法

方法	说明
add	用于加法（+）的方法
sub	用于减法（-）的方法
div	用于除法（/）的方法
mul	用于乘法（*）的方法

函数应用和映射

Numpy 的 ufuncs（元素级数组方法）也可用于操作 pandas：

```
frame = DataFrame(np.arange(15).reshape(3,5), columns=list('abcde'),
index=range(3))
```

```
    a   b   c   d   e
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
```

函数应用到各列形成的一维数组：

```
f = lambda x:x.max()-x.min()
print frame.apply(f)
```

```
a    10
b    10
c    10
d    10
e    10
```

```
def f(x):
    return Series([x.min(),x.max()],index=['max','min'])
print frame.apply(f)
```

```
       a   b   c   d   e
max    0   1   2   3   4
min   10  11  12  13  14
```

函数应用到各行形成的一维数组：

```
print frame.apply(f,axis=1)
```

```
   max  min
0    0    4
1    5    9
2   10   14
```

函数应用到元素级：

```
f = lambda x: 100*x
print frame.applymap(f)
```

```
       a     b     c     d     e
0      0   100   200   300   400
1    500   600   700   800   900
2   1000  1100  1200  1300  1400
```

排序和排名

按索引排序

对 Series 按索引排序：

```
obj = Series(range(4),index=list('dabc'))
```

```
d    0
a    1
b    2
c    3
```

```
print obj.sort_index()
```

```
d    0
a    1
b    2
c    3
```

对 DataFrame 按索引排序：

```
frame = DataFrame(np.random.random(12).reshape(3,4), columns=list('dabc'),
index=list('egf'))
```

```
          d         a         b         c
e  0.491288  0.215284  0.665717  0.653339
g  0.851490  0.992568  0.103659  0.541566
f  0.286575  0.305141  0.320389  0.342056
```

```
print frame.sort_index()
```

```
          d         a         b         c
e  0.491288  0.215284  0.665717  0.653339
f  0.286575  0.305141  0.320389  0.342056
g  0.851490  0.992568  0.103659  0.541566
```

```
print frame.sort_index(axis=1)
```

```
          a         b         c         d
e  0.215284  0.665717  0.653339  0.491288
g  0.992568  0.103659  0.541566  0.851490
f  0.305141  0.320389  0.342056  0.286575
```

默认是按升序排序的，但也可以降序排序：

```
print frame.sort_index(axis=1,ascending=False)
```

```
          d         c         b         a
e  0.491288  0.653339  0.665717  0.215284
g  0.851490  0.541566  0.103659  0.992568
f  0.286575  0.342056  0.320389  0.305141
```

按值排序

对 Series 按值排序：

```
obj = Series([4,6,np.nan,-2,np.nan,0])
print obj.order()
```

```
3   -2
5    0
0    4
1    6
2  NaN
4  NaN
```

在排序时，任何缺失值默认都会被放在最后面

对 DataFrame 按值排序：

在 DataFrame 上，可根据一个或多个列中的值进行排序。将一个或多个列的名字传递给 by 选项即可达到该目的：

```
frame = DataFrame(np.random.randint(0,10,12).reshape(3,4),
columns=list('abcd'))
   a  b  c  d
0  9  2  0  6
1  4  2  2  1
2  4  9  5  3
print frame.sort_index(by='a')
   a  b  c  d
1  4  2  2  1
2  4  9  5  3
0  9  2  0  6
print frame.sort_index(by=['a','b'])
   a  b  c  d
1  4  2  2  1
2  4  9  5  3
0  9  2  0  6
```

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。

```
frame = DataFrame([[1.4, np.nan], [7.1, -4.5],
                   [np.nan, np.nan], [0.75, -1.3]],
                   index=list('abcd'), columns=['one','two'])
     one   two
a   1.40   NaN
b   7.10 -4.5
c    NaN   NaN
d   0.75 -1.3
print frame.sum()
one      9.25
two     -5.80
```

传入 axis=1 将会按行进行求和运算：

```
print frame.sum(axis=1)
```

a 1.40
b 2.60
c NaN
d -0.55

表5-9：约简方法的选项

选项	说明
axis	约简的轴。DataFrame的行用0，列用1
skipna	排除缺失值，默认值为True
level	如果轴是层次化索引的（即MultiIndex），则根据level分组约简

```
print frame.describe()
```

describe 一次性产生多个汇总统计：

	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

表5-10：描述和汇总统计

方法	说明
count	非NA值的数量
describe	针对Series或各DataFrame列计算汇总统计
min、max	计算最小值和最大值
argmin、argmax	计算能够获取到最小值和最大值的索引位置（整数）
idxmin、idxmax	计算能够获取到最小值和最大值的索引值
quantile	计算样本的分位数（0到1）
sum	值的总和
mean	值的平均数
median	值的算术中位数（50%分位数）
mad	根据平均值计算平均绝对离差
var	样本值的方差
std	样本值的标准差

表5-10：描述和汇总统计（续）

方法	说明
skew	样本值的偏度（三阶矩）
kurt	样本值的峰度（四阶矩）
cumsum	样本值的累计和
cummin、cummax	样本值的累计最大值和累计最小值
cumprod	样本值的累计积
diff	计算一阶差分（对时间序列很有用）
pct_change	计算百分数变化

相关系数与协方差

利用 Tushare 得到股票数据，进行处理：

```
import scipy.io as sio
# matlab 文件名
matfn =
r'C:\Users\Administrator\Desktop\pythonCode\TestTushare\savedata.mat'
data = sio.loadmat(matfn)
close = data['stock_close'][0][::-1]  # 数据反转，按照时间顺序排列数据
low = data['stock_low'][0][::-1]  # 数据反转，按照时间顺序排列数据
high = data['stock_high'][0][::-1]  # 数据反转，按照时间顺序排列数据
price = DataFrame({'close':close, 'low':low, 'high':high})
returns = price.pct_change()
print returns.tail()
```

```
          close      high       low
1446 -0.015257 -0.009524  0.007112
1447 -0.007042 -0.009615 -0.011299
1448  0.034043  0.019417  0.008571
1449  0.032922  0.034014  0.032578
1450  0.029216  0.026316  0.039781
```

Series 的 corr 方法用于计算两个 Series 中重叠的、非 NA 的、按索引对齐的值的相关系数。与此类似，cov 用于计算协方差：

```
print returns['close'].corr(returns['low'])
```

```
0.641948012673
```

```
print returns['close'].cov(returns['low'])
```

```
0.000702147924951
```

```
print
returns['close'].cov(returns['low'])/(returns['close'].std()*returns['low']
.std())
```

```
0.641948012673
```

DataFrame 的 corr 和 cov 方法将以 DataFrame 的形式返回完整的相关系数或协方差矩阵：

```
print returns.cov()
```

```
          close      high       low
close  0.001130  0.000721  0.000702
high   0.000721  0.001040  0.000707
low    0.000702  0.000707  0.001058
```

```
print returns.corr()
```

```
          close      high       low
close  1.000000  0.665222  0.641948
high   0.665222  1.000000  0.673622
low    0.641948  0.673622  1.000000
```

利用 DataFrame 的 corrwith 方法，可以计算其列或行跟另一个 Series 或 DataFrame 之间的相关系数。

```
print returns.corrwith(returns['close'])
```

```
close    1.000000
high     0.665222
low      0.641948
```

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。

表5-12：NA处理方法

方法	说明
dropna	根据各标签的值中是否存在缺失数据对轴标签进行过滤，可通过阈值调节对缺失值的容忍度
fillna	用指定值或插值方法（如ffill或bfill）填充缺失数据
isnull	返回一个含有布尔值的对象，这些布尔值表示哪些值是缺失值/NA，该对象的类型与源类型一样
notnull	isnull的否定式

丢弃缺失数据

```
data = DataFrame([[1, 6.5, 3], [1, np.nan, np.nan],
                  [np.nan, np.nan, np.nan], [np.nan, 6.5, 3]])
```

```
     0    1    2
0    1  6.5    3
1    1  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5    3
```

对于 DataFrame，你可能希望丢弃全 NA 或含有 NA 的行或列。**Dropna 默认丢弃任何含有缺失值的行：**

```
print data.dropna()
```

```
   0    1  2
0  1  6.5  3
```

传入 how='all' 将只丢弃全为 NA 的那些行：

```
print data.dropna(how='all')
```

```
    0    1    2
0   1  6.5    3
1   1  NaN  NaN
3 NaN  6.5    3
```

传入 axis=1 即可，丢弃列：

```
data[4] = np.nan
print data
```

```
    0    1    2    4
0   1  6.5    3  NaN
1   1  NaN  NaN  NaN
2 NaN  NaN  NaN  NaN
3 NaN  6.5    3  NaN
```

```
print data.dropna(how='all',axis=1)
```

```
    0    1    2
0   1  6.5    3
1   1  NaN  NaN
2 NaN  NaN  NaN
3 NaN  6.5    3
```

填充缺失数据

对于大多数情况而言，fillna 方法是最主要的填补“空洞”函数。通过一个常数调用 fillna 就会将缺失值替换为那个常数值。

```
print data.fillna(0)
```

```
   0    1  2  4
0  1  6.5  3  0
1  1  0.0  0  0
2  0  0.0  0  0
3  0  6.5  3  0
```

fillna 默认会返回新对象，但也可以对现有对象进行就地修改：

```
data.fillna(0, inplace=True)
print data
```

```
   0    1  2  4
0  1  6.5  3  0
1  1  0.0  0  0
2  0  0.0  0  0
3  0  6.5  3  0
```

对 reindex 有效的那些插值方法也可用于 fillna：

```
data = DataFrame([[1, 6.5, 3], [1, np.nan, np.nan],
                  [np.nan, np.nan, np.nan], [np.nan, 6.5, 3]])
print data.fillna(method='ffill')
```

```
   0    1  2
0  1  6.5  3
1  1  6.5  3
2  1  6.5  3
3  1  6.5  3
```

表5-13：fillna函数的参数

参数	说明
value	用于填充缺失值的标量值或字典对象
method	插值方式。如果函数调用时未指定其他参数的话，默认为“ffill”

表5-13：fillna函数的参数（续）

参数	说明
axis	待填充的轴，默认axis=0
inplace	修改调用者对象而不产生副本
limit	（对于前向和后向填充）可以连续填充的最大数量

层次化索引

层次化索引是 pandas 的一项重要功能，它使你能在一个轴上拥有多个索引级别。

Series

```
data = Series(np.arange(10),index=[['a','a','a','b','b','b','c','c','d','d'],
[1,2,3,1,2,3,1,2,2,3]])
print data
```

```
a  1    0
   2    1
```

```
   3    2
b  1    3
   2    4
   3    5
c  1    6
   2    7
d  2    8
   3    9
```

层次化索引选取子集：

```
print data[['b','c']]
```

```
b  1    3
   2    4
   3    5
c  1    6
   2    7
```

可以在“内层”进行选取：

```
print data[:,2]
```

```
a    1
b    4
c    7
d    8
```

DataFrame

```
frame = DataFrame(np.arange(12).reshape(4,3),
index=[['a','a','a','b'],[1,2,1,2]],
                columns=[['lhq','lhq','hyj'],['100','90','100']])
print frame
```

```
    lhq      hyj
    100   90 100
a 1   0    1    2
  2   3    4    5
  1   6    7    8
b 2   9   10   11
```

选取列分组：

```
print frame['lhq']
```

```
     100   90
a 1    0    1
  2    3    4
```

```
   1     6    7
b 2     9   10
```

给各层指定名称：

```
frame.index.names = ['key1','key2']
frame.columns.names = ['name','look']
print frame
```

```
name        lhq       hyj
look        100   90 100
key1 key2
a    1        0    1    2
     2        3    4    5
     1        6    7    8
b    2        9   10   11
```

重新分级顺序

Swaplevel 接受两个级别编号或名称，并返回一个互换了级别的新对象。

```
print frame.swaplevel('key2','key1')
```

```
name        lhq       hyj
look        100   90 100
key2 key1
1    a        0    1    2
2    a        3    4    5
1    a        6    7    8
2    b        9   10   11
```

根据级别汇总统计

许多对 DataFrame 和 Series 的描述和汇总统计都有一个 level 选项，它用于指定在某条轴上求和的级别。

```
print frame.sum(level='key2')
```

```
name lhq       hyj
look 100   90 100
key2
1       6    8   10
2      12   14   16
```

```
print frame.sum(level='name',axis=1)
```

```
name        hyj   lhq
key2 key1
1    a        2     1
2    a        5     7
```

```
1    a       8   13
2    b      11   19
```

使用 DataFrame 的列为行索引

将 DataFrame 的一个或多个列当做行索引来用，或者希望将行索引变成列。

```
frame = DataFrame({'a':range(4), 'b':np.random.random(4), 'c':list('abcd'),
'd':[0,1,0,1]})
print frame
```

```
   a         b  c  d
0  0  0.006410  a  0
1  1  0.294273  b  1
2  2  0.506948  c  0
3  3  0.728313  d  1
```

```
print frame.set_index(['a','b'])
```

```
            c  d
a b
0 0.006410  a  0
1 0.294273  b  1
2 0.506948  c  0
3 0.728313  d  1
```

将列变为行索引，这些列会被移除，但可以保留下来：

```
print frame.set_index(['a','b'], drop=False)
```

```
            a         b  c  d
a b
0 0.006410  0  0.006410  a  0
1 0.294273  1  0.294273  b  1
2 0.506948  2  0.506948  c  0
3 0.728313  3  0.728313  d  1
```

Retset_index 的功能跟 set_index 刚好相反：

```
frame = frame.set_index(['a','b'])
print frame.reset_index()
```

```
   a         b  c  d
0  0  0.006410  a  0
1  1  0.294273  b  1
2  2  0.506948  c  0
3  3  0.728313  d  1
```

数据加载、存储与文件格式

存取 MongDB 中的数据

先在电脑上启动一个 MongDB 实例，然后用 pymongo 通过默认端口进行连接：

```
import pymongo
conn = pymongo.MongoClient('127.0.0.1', port=27017)
```

存储在 mongodb 中的文档被组织在数据库的集合中。MongDB 服务器的每个运行实例可以有多个数据库，而每个数据库又可以有多个集合。首先，先访问集合：

```
db = conn.db
collection = db.today_all
```

然后，将数据加载进来，并通过 collection.insert 逐个存入集合中：

```
for i in range(10):
    df = ts.get_today_all()
    conn.db.today_all.insert(json.loads(df.to_json(orient='records')))
```

现在，如果想从该集合中取出数据，可以用下面代码进行查询：

```
items = collection.find({"turnoverratio": { "$lte":6.34 } })
```

跟之前一样，将其转换为一个 DataFrame：

```
result = DataFrame(list(items))
```

将数据写出到文本格式

数据可以被输出为分隔符格式的文本。

利用 DataFrame 的 to_csv 方法，我们可以将数据写到一个以逗号分隔的文件中：

```
result.to_csv('sz002237.csv', encoding='utf-8')
```

还可以使用其他分隔符（由于这里直接写出到 sys.stdout，所以仅仅是打印出文本结果而已）：

```
result.to_csv(sys.stdout, sep = '|', encoding='utf-8')
```

```
|changepercent|code|high|low|name|open|settlement|time_a|time_b|trade|turnoverratio|volume
0|3.541|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:20|2016-04-13 13:35:56|10.82|6.28124|50507481
1|3.349|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:20|2016-04-13 13:35:56|10.8|6.30379|50688781
2|3.541|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:38|2016-04-13 13:36:20|10.82|6.32101|50827277
3|3.349|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:38|2016-04-13 13:36:20|10.8|6.30379|50688781
4|3.541|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:58|2016-04-13 13:36:38|10.82|6.32101|50827277
5|3.349|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:36:58|2016-04-13 13:36:38|10.8|6.30379|50688781
6|3.541|002237|11.0|10.3|恒邦股份|10.38|10.45|2016-04-13 13:37:19|2016-04-13 13:36:58|10.82|6.32101|50827277
```

如果没有设置其他选项，则会写出行和列的标签。当然，它们也都可以被禁用：

```
result.to_csv(sys.stdout, sep = '|', encoding='utf-8', index=False,
header=False)
```

读取文本格式的数据

Pandas 提供了一些用于将表格型数据读取为 DataFrame 对象的函数。其中 read_csv 和 read_table 可能会用得最多。

表6-1：pandas中的解析函数

函数	说明
read_csv	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号
read_table	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符（“\t”）

由于文件以逗号分隔，使用 read_csv 将其读入一个 DataFrame：

```
df = pd.read_csv('sz002237.csv')
print df
```

也可以用 read_table，只不过需要指定分隔符而已：

```
df = pd.read_table('sz002237.csv', sep=',')
print df
```

数据规整化：清理转换、合并、重塑

合并数据集

数据库风格的 DataFrame 合并：

数据库的合并（merge）或连接（join）运算是通过一个或多个键将行链接起来的。

```
df1 = DataFrame({'key':['b','b','a','c'], 'data1':range(4)})
```

```
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
```

```
df2 = DataFrame({'key':['b','a','d'], 'data1':range(3)})
```

```
   data1 key
0      0   b
1      1   a
2      2   d
```

```
df = pd.merge(df1, df2, on='key')
```

```
   data1_x key  data1_y
0        0   b        0
1        1   b        0
2        2   a        1
```

如果两个对象的列名不同，也可以分别进行指定：

```
df3 = DataFrame({'key1':['b','b','a','c'], 'data1':range(4)})
```

```
   data1 key1
0      0    b
1      1    b
2      2    a
3      3    c
```

```
df4 = DataFrame({'key2':['b','a','d'], 'data1':range(3)})
```

```
   data1 key2
0      0    b
1      1    a
2      2    d
```

```
df = pd.merge(df3, df4, left_on='key1', right_on='key2')
```

```
   data1_x key1  data1_y key2
0        0    b        0    b
1        1    b        0    b
2        2    a        1    a
```

默认情况下，merge 做的是“inner”连接，结果中的键是交集。其他方式还有“left”，“right”以及“outer”。外连接求取的是键的合并，组合了左连接和右连接：

```
df = pd.merge(df1, df2, how='outer')
```

```
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
```

```
4       1    a
5       2    d
```

```
df = pd.merge(df1, df2, on='key', how='left')
```

```
   data1_x key  data1_y
0        0   b        0
1        1   b        0
2        2   a        1
3        3   c      NaN
```

多对多连接产生的是行的笛卡尔积。由于左边的 DataFrame 有 2 个'b'行，右边的有 1，所以最终结果中就有 6 个'b'行。

表7-1：merge函数的参数

参数	说明
left	参与合并的左侧DataFrame
right	参与合并的右侧DataFrame
how	“inner”、“outer”、“left”、“right”其中之一。默认为“inner”

参数	说明
on	用于连接的列名。必须存在于左右两个DataFrame对象中。如果未指定，且其他连接键也未指定，则以left和right列名的交集作为连接键
left_on	左侧DataFrame中用作连接键的列
right_on	右侧DataFrame中用作连接键的列
left_index	将左侧的行索引用作其连接键
right_index	类似于left_index
sort	根据连接键对合并后的数据进行排序，默认为True。有时在处理大数据集时，禁用该选项可获得更好的性能
suffixes	字符串值元组，用于追加到重叠列名的末尾，默认为('_x', '_y')。例如，如果左右两个DataFrame对象都有“data”，则结果中就会出现“data_x”和“data_y”
copy	设置为False，可以在某些特殊情况下避免将数据复制到结果数据结构中。默认总是复制

索引上的合并

DataFrame 中的连接键位于其索引中。这时候，可以传入 left_index=True 或 right_index=True（或两个都传）以说明索引应该被用作连接键：

```
left1 = DataFrame({'key':['a','b','a','a','c'], 'value':range(5)})
```

```
  key  value
0   a      0
```

```
1   b      1
2   a      2
3   a      3
4   c      4
```

```
right1 = DataFrame({'goup_val':[3.5, 7]}, index=['a', 'b'])
```

```
   goup_val
a       3.5
b       7.0
```

```
df = pd.merge(left1, right1, left_on='key', right_index=True)
```

```
  key  value  goup_val
0   a      0       3.5
2   a      2       3.5
3   a      3       3.5
1   b      1       7.0
```

默认情况的 merge 方法是求取连接键的交集，可以通过外连接的方式得到他们的并集：

```
df = pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

```
  key  value  goup_val
0   a      0       3.5
2   a      2       3.5
3   a      3       3.5
1   b      1       7.0
4   c      4       NaN
```

同时合并双方的索引也没有问题：

```
df = pd.merge(left1, right1, left_index=True, right_index=True, how='outer')
```

```
   key  value  goup_val
0    a      0       NaN
1    b      1       NaN
2    a      2       NaN
3    a      3       NaN
4    c      4       NaN
a  NaN    NaN       3.5
b  NaN    NaN       7.0
```

轴向连接

另一种数据合并运算被称作连接(concatenation)、绑定(binding)或堆叠(stacking)。

行连接：

```
df = pd.concat([left1, right1])
```

```
   goup_val  key  value
```

```
0        NaN    a       0
1        NaN    b       1
2        NaN    a       2
3        NaN    a       3
4        NaN    c       4
a        3.5   NaN     NaN
b        7.0   NaN     NaN
```

默认情况下，concat 是在 axis=0 上工作的。

列连接：

```
left1 = DataFrame({'key':['a','b','a','a','c'], 'value':range(5)})
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   c      4
right1 = DataFrame({'goup_val':[3.5, 7]})
   goup_val
0       3.5
1       7.0
df = pd.concat([left1, right1], axis=1)
  key  value  goup_val
0   a      0       3.5
1   b      1       7.0
2   a      2       NaN
3   a      3       NaN
4   c      4       NaN
```

默认情况下，concat 都是外连接（有序并集）。传入 join=‘inner’即可得到他们的交集。

```
df = pd.concat([left1, right1], axis=1, join='inner')
  key  value  goup_val
0   a      0       3.5
1   b      1       7.0
```

创建层次化索引：

假如想要在连接轴上创建一个层次化索引。使用 keys 参数即可达到这个目的：

```
df = pd.concat([left1, right1], keys=['left','right'])
          goup_val  key  value
left  0       NaN    a      0
```

```
        1         NaN      b        1
        2         NaN      a        2
        3         NaN      a        3
        4         NaN      c        4
right 0          3.5   NaN      NaN
        1          7.0   NaN      NaN
```

沿着 axis=1 合并，则 keys 就会成为 DataFrame 的列头：

```
df = pd.concat([left1, right1], axis=1, keys=['left','right'])
```

```
  left              right
   key value goup_val
0     a      0       3.5
1     b      1       7.0
2     a      2       NaN
3     a      3       NaN
4     c      4       NaN
```

不保留连接轴上的索引，传入 ignore_index=Ture 即可：

```
left1 = DataFrame({'key':['a','b','a','a','c'], 'value':range(5)})
```

```
  key   value
0   a       0
1   b       1
2   a       2
3   a       3
4   c       4
```

```
right1 = DataFrame({'goup_val':[3.5, 7]}, index=['a', 'b'])
```

```
   goup_val
a       3.5
b       7.0
```

```
df = pd.concat([left1, right1], ignore_index=True)
```

```
   goup_val   key   value
0       NaN     a       0
1       NaN     b       1
2       NaN     a       2
3       NaN     a       3
4       NaN     c       4
5       3.5   NaN     NaN
6       7.0   NaN     NaN
```

表7-2：concat函数的参数

参数	说明
objs	参与连接的pandas对象的列表或字典。唯一必需的参数
axis	指明连接的轴向，默认为0
join	“inner”、“outer”其中之一，默认为“outer”。指明其他轴向上的索引是按交集（inner）还是并集（outer）进行合并
join_axes	指明用于其他n-1条轴的索引，不执行并集/交集运算
keys	与连接对象有关的值，用于形成连接轴向上的层次化索引。可以是任意值的列表或数组、元组数组、数组列表（如果将levels设置成多级数组的话）
levels	指定用作层次化索引各级别上的索引，如果设置了keys的话[译注3]
names	用于创建分层级别的名称，如果设置了keys和（或）levels的话
verify_integrity	检查结果对象新轴上的重复情况，如果发现则引发异常。默认（False）允许重复
ignore_index	不保留连接轴上的索引，产生一组新索引range(total_length)

合并重叠数据

对于 DataFrame，combine_first 可看作：用参数对象汇总的数据为调用者对象的缺失数据“打补丁”。

```
df1 = DataFrame({'a':[1, np.nan, 3], 'b':[np.nan, np.nan, np.nan], 'c':[4, 5, 6]})
```

```
    a   b  c
0   1 NaN  4
1 NaN NaN  5
2   3 NaN  6
```

```
df2 = DataFrame({'a':[np.nan, 2, 5], 'b':[1, np.nan, 3]})
```

```
    a   b
0 NaN   1
1   2 NaN
2   5   3
```

```
df3 = df1.combine_first(df2)
```

```
   a   b  c
0  1   1  4
1  2 NaN  5
2  3   3  6
```

是 df2 给 df1 打补丁！

重塑和轴向旋转

用于重新排列表格型数据的基础运算，这些函数称作重塑（reshape）或轴向旋转（pivot）运算。

重塑层次化索引

层次化索引为 DataFrame 数据的重排任务提供了一种具有良好一致性的方式。

- Stack：将数据的列“旋转”为行。
- unstack：将数据的行“旋转”为列。

```
data = DataFrame(np.arange(6).reshape((2,3)), index=['a', 'b'],columns=['one',
'two', 'three'])
```

```
   one  two  three
a    0    1      2
b    3    4      5
```

```
result = data.stack()
```

```
a  one      0
   two      1
   three    2
b  one      3
   two      4
   three    5
```

```
result1 = result.unstack()
```

```
   one  two  three
a    0    1      2
b    3    4      5
```

默认情况下，unstack 操作的是最内层（stack 也是如此）。传入分层级别的编号或名称即可对其他级别进行 unstack 操作：

```
result2 = result.unstack(0)
```

```
       a  b
one    0  3
two    1  4
three  2  5
```

将“长格式”旋转为“宽格式”

时间序列数据通常是以所谓的“长格式”或“堆叠格式”存储在数据库的：

```
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
collection = db.today_all
items = DataFrame(list(collection.find()))
items = items.drop(['_id'], axis=1)
```

```
items = items.sort_index(by='time_b')
print items.head(5)
```

```
   changepercent      code  high   low    name   open  settlement  \
1          3.349  002237    11  10.3  恒邦股份  10.38       10.45
0          3.541  002237    11  10.3  恒邦股份  10.38       10.45
3          3.349  002237    11  10.3  恒邦股份  10.38       10.45
2          3.541  002237    11  10.3  恒邦股份  10.38       10.45
5          3.349  002237    11  10.3  恒邦股份  10.38       10.45

                  time_a              time_b  trade  turnoverratio  \
1  2016-04-13 13:36:20  2016-04-13 13:35:56  10.80        6.30379
0  2016-04-13 13:36:20  2016-04-13 13:35:56  10.82        6.28124
3  2016-04-13 13:36:38  2016-04-13 13:36:20  10.80        6.30379
2  2016-04-13 13:36:38  2016-04-13 13:36:20  10.82        6.32101
5  2016-04-13 13:36:58  2016-04-13 13:36:38  10.80        6.30379

     volume
1  50688781
0  50507481
3  50688781
2  50827277
5  50688781
```

```
print items.head(5).set_index(['time_b'])
```

```
                     changepercent      code  high   low    name   open  \
time_b
2016-04-13 13:35:56          3.349  002237    11  10.3  恒邦股份  10.38
2016-04-13 13:35:56          3.541  002237    11  10.3  恒邦股份  10.38
2016-04-13 13:36:20          3.349  002237    11  10.3  恒邦股份  10.38
2016-04-13 13:36:20          3.541  002237    11  10.3  恒邦股份  10.38
2016-04-13 13:36:38          3.349  002237    11  10.3  恒邦股份  10.38

                     settlement              time_a  trade  turnoverratio  \
time_b
2016-04-13 13:35:56       10.45  2016-04-13 13:36:20  10.80        6.30379
2016-04-13 13:35:56       10.45  2016-04-13 13:36:20  10.82        6.28124
2016-04-13 13:36:20       10.45  2016-04-13 13:36:38  10.80        6.30379
2016-04-13 13:36:20       10.45  2016-04-13 13:36:38  10.82        6.32101
2016-04-13 13:36:38       10.45  2016-04-13 13:36:58  10.80        6.30379

                       volume
time_b
2016-04-13 13:35:56  50688781
2016-04-13 13:35:56  50507481
2016-04-13 13:36:20  50688781
2016-04-13 13:36:20  50827277
2016-04-13 13:36:38  50688781
```

```
In [116]: ldata[:10]
Out[116]:
                 date     item     value
0  1959-03-31 00:00:00  realgdp  2710.349
1  1959-03-31 00:00:00     infl     0.000
2  1959-03-31 00:00:00   unemp     5.800
3  1959-06-30 00:00:00  realgdp  2778.801
4  1959-06-30 00:00:00     infl     2.340
5  1959-06-30 00:00:00   unemp     5.100
6  1959-09-30 00:00:00  realgdp  2775.488
7  1959-09-30 00:00:00     infl     2.740
8  1959-09-30 00:00:00   unemp     5.300
9  1959-12-31 00:00:00  realgdp  2785.204
```

长格式的数据操作起来不那么轻松，可能更喜欢不同的 item 值分别形成一列，date 列中的时间则用作索引。DataFrame 的 pivot 方法完全可以实现这个转换：

```
In [117]: pivoted = ldata.pivot('date', 'item', 'value')

In [118]: pivoted.head()
Out[118]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

前两个参数分别用作行和列索引的列名，最后一个参数值则用于填充 DataFrame 的数据。

数据转换

移除重复数据

```
data = DataFrame({'k1':['one']*3+['two']*4, 'k2':[1,1,2,3,3,4,4]})
```

```
    k1  k2
0  one   1
1  one   1
2  one   2
3  two   3
4  two   3
5  two   4
6  two   4
```

Drop_duplicates 方法，用于返回一个移除了重复行的 DataFrame：

```
Data1 = data.drop_duplicates()
```

```
    k1  k2
0  one   1
2  one   2
```

```
3   two    3
5   two    4
```

默认会判断全部列，可以指定部分列进行重复项判断：

```
data2 = data.drop_duplicates(['k1'])
    k1   k2
0   one    1
3   two    3
```

Drop_duplicates 方法默认保留第一个出现的值组合，传入 take_last=True 则保留最后一个：

```
data3 = data.drop_duplicates(['k1'], take_last=True)
    k1   k2
2   one    2
6   two    4
```

替换值

用 replace 方法来进行替换功能：

```
data4 = data1.replace(1, np.nan)
    k1   k2
0   one NaN
2   one    2
3   two    3
5   two    4
```

如果希望一次性替换多个值：

```
data5 = data1.replace([1,2], np.nan)
    k1   k2
0   one NaN
2   one NaN
3   two    3
5   two    4
```

希望对不同的值进行不同的替换，则传入一个由替换关系组成的列表即可：

```
data6 = data1.replace([1,2], [np.nan, 8])
    k1   k2
0   one NaN
2   one    8
3   two    3
```

5 two 4

检测和过滤异常值

异常值的过滤或变换运算在很大程度上其实就是数组运算：

```
np.random.seed(12345)
data = DataFrame(np.random.randn(1000,4))
print data.describe()
```

```
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067684     0.067924     0.025598    -0.002298
std       0.998035     0.992106     1.006835     0.996794
min      -3.428254    -3.548824    -3.184377    -3.745356
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
75%       0.616366     0.780282     0.680391     0.654328
max       3.366626     2.653656     3.260383     3.927528
```

要选出全部含有“超过 3 或-3 的值”的行，利用布尔型 DataFrame 以及 any 方法：

```
print data[(np.abs(data)>3).any(1)]
```

```
           0         1         2         3
5  -0.539741  0.476985  3.248944 -1.021228
97 -0.774363  0.552936  0.106061  3.927528
102 -0.655054 -0.565230  3.176873  0.959533
305 -2.315555  0.457246 -0.025907 -3.399312
324  0.050188  1.951312  3.260383  0.963301
400  0.146326  0.508391 -0.196713 -3.745356
499 -0.293333 -0.242459 -3.056990  1.918403
523 -3.428254 -0.296336 -0.439938 -0.867165
586  0.275144  1.179227 -3.184377  1.369891
808 -0.362528 -3.548824  1.553205 -2.186301
900  3.366626 -2.372214  0.851010  1.332846
```

将值限制在区间-3 到 3 之间：

```
data[(np.abs(data))>3] = np.sign(data)*3
print data.describe()
```

```
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067623     0.068473     0.025153    -0.002081
std       0.995485     0.990253     1.003977     0.989736
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
```

75%	0.616366	0.780282	0.680391	0.654328
max	3.000000	2.653656	3.000000	3.000000

排列和随机采样

利用 np.random.permutation 函数可以实现对 DataFrame 的列的排列工作:

```
sampler = np.random.permutation(5)
```
```
[1 0 2 3 4]
```
```
df = DataFrame(np.arange(5*4).reshape(5,4))
```
```
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19
```
```
print df.take(sampler)
```
```
    0   1   2   3
1   4   5   6   7
0   0   1   2   3
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19
```
```
print df.take(np.random.permutation(5)[:2])
```
```
    0   1   2   3
1   4   5   6   7
2  12  13  14  15
```

指定集合的随机数生成:

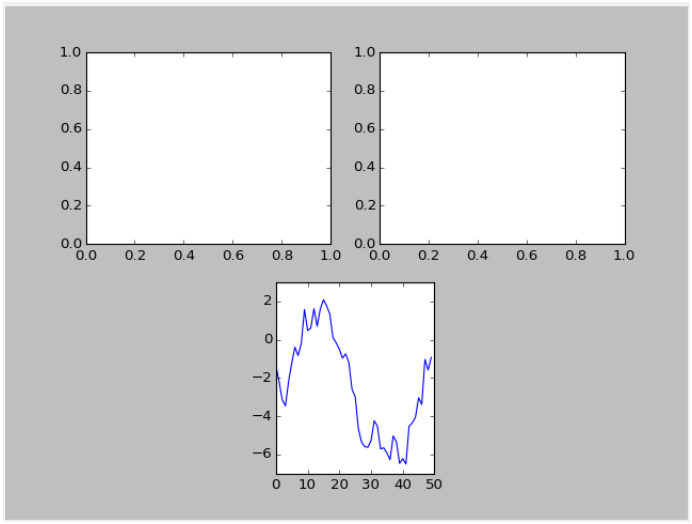
```
bag = np.array([7,4,3,2,-1])
sampler = np.random.randint(0, len(bag), size=10)
draws = bag.take(sampler)
print draws
```
```
[-1 -1  3  3  3  7  2  7 -1  4]
```

绘图和可视化

Matplotlib API 入门

Figure 和 subplot

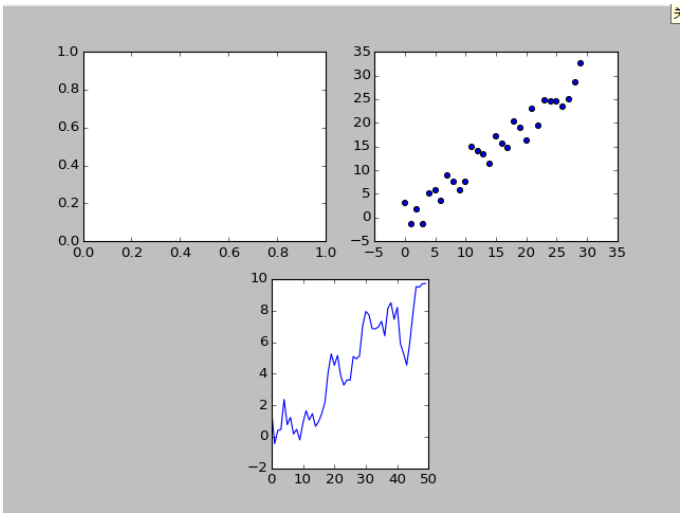
```
from matplotlib.pyplot import figure
from matplotlib.pyplot import plot
from matplotlib.pyplot import show
fig = figure()
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,3,5)
plot(np.random.randn(50).cumsum())
show()
```



这时发出一条绘图命令，matplotlib 就会在最后一个用过的 subplot 上进行绘制。

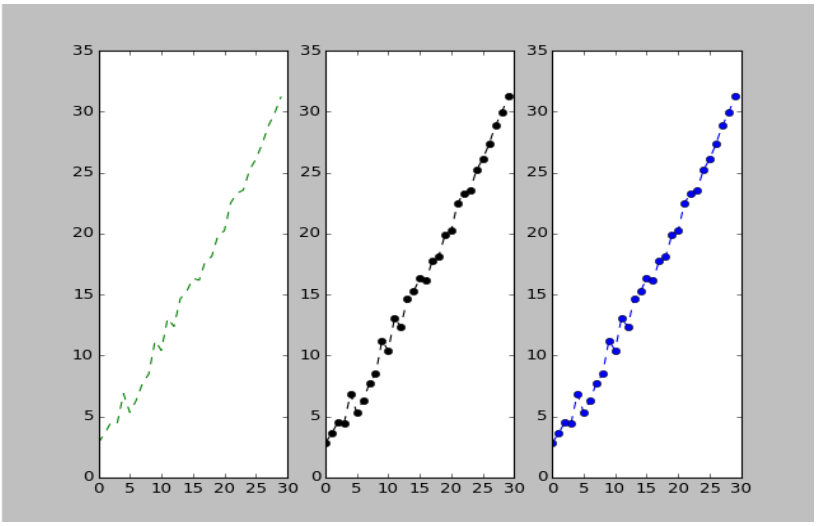
上面有 fig.add_subplot 所返回的对象是 AxesSubplot 对象，直接调用他们的实例方法就可以在其他空格子里面画图了。

```
ax2.scatter(np.arange(30), np.arange(30)+3*np.random.randn(30))
```



颜色、标记和线型

```
fig1 = figure()
ax4 = fig1.add_subplot(1,3,1)
ax5 = fig1.add_subplot(1,3,2)
ax6 = fig1.add_subplot(1,3,3)
x = np.arange(30)
y = np.arange(30)+3*np.random.rand(30)
ax4.plot(x, y, 'g--')
ax5.plot(x, y, 'ko--')
ax6.plot(x, y, 'o--')
show()
```



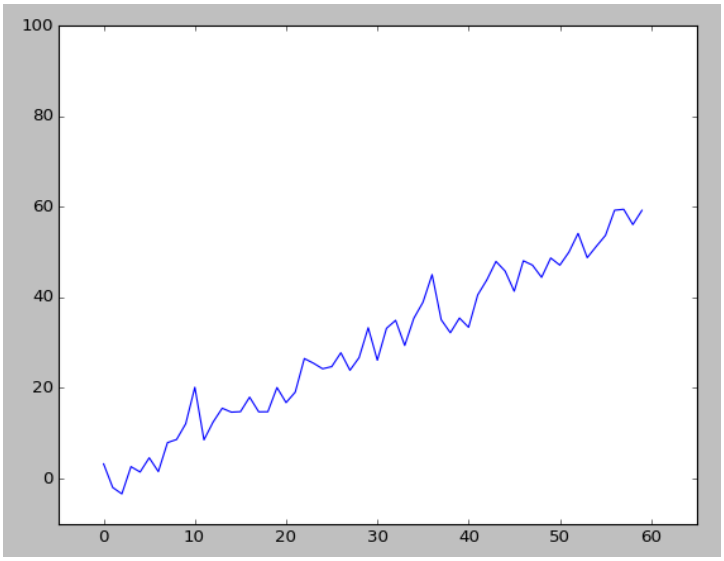
刻度、标签和图例

xlim、xticks 和 xticklabels 分别控制图标的范围、刻度位置、刻度标签。

- 调用时不带参数，则返回当前的参数值。例如 xlim（）返回当前 X 轴绘图范围。
- 调用时带参数，则设置参数值。例如 xlim（[1,10]）将 x 轴的范围设置为 0 到 10.

它们各自对应 subplot 对象上的两个方法，以 xlim 为例，就是 ax.get_xlim 和 ax.set_xlim。

```
fig3 = figure()
ax = fig3.add_subplot(1,1,1)
ax.plot(np.arange(60), np.arange(60)+3*np.random.randn(60))
ax.set_xlim([-5, 65])       # 设置x轴范围
ax.set_ylim([-10, 100])     # 设置y轴范围
```

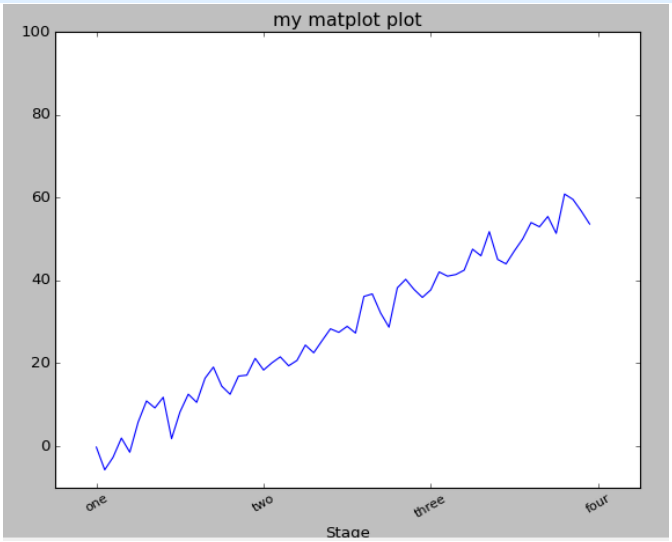


修改 x 轴的刻度，最简单的 办法就是使用 set_xticks 和 set_xticklabels，我们可通过 set_xticklabels 将任何其他的值用作标签：

```
ax.set_xticks([0,20,40,60])
ax.set_xticklabels(['one', 'two', 'three', 'four'], rotation=30,
fontsize='small')
```

set_xlabel 为 x 轴设置一个名称，并用 set_title 设置一个标题：

```
ax.set_xlabel('Stage')
ax.set_title("my matplot plot")
```



添加图例

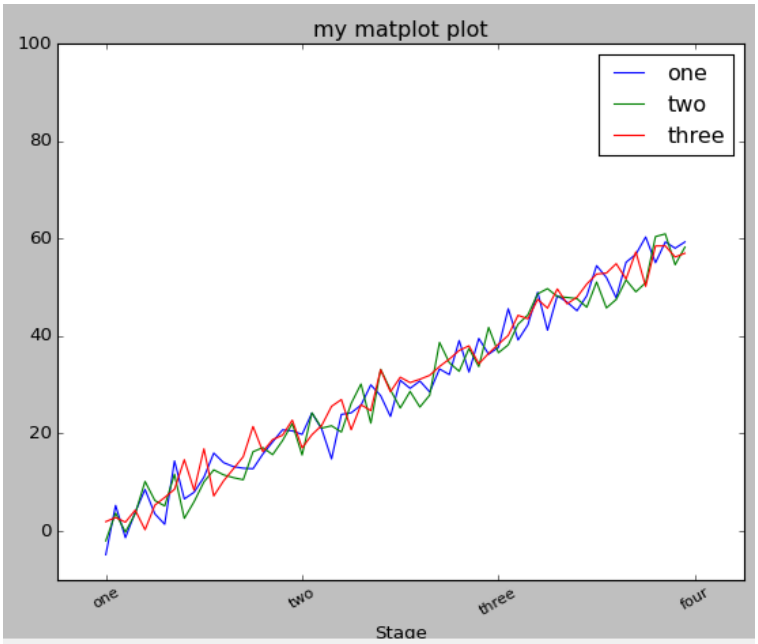
图例（legend）是另一种用于标识图标元素的重要工具。添加图例的方式有二。最简单的是在添加 subplot 的时候传入 label：

```
ax.plot(np.arange(60), np.arange(60)+3*np.random.randn(60), label='one')
```

之后，用 ax.legend()自动创建图例：

```
ax.legend(loc='best')
```

```
fig3 = figure()
ax = fig3.add_subplot(1,1,1)
ax.plot(np.arange(60), np.arange(60)+3*np.random.randn(60), label='one')
ax.plot(np.arange(60), np.arange(60)+3*np.random.randn(60), label='two')
ax.plot(np.arange(60), np.arange(60)+3*np.random.randn(60), label='three')
ax.set_xlim([-5, 65])       # 设置x轴范围
ax.set_ylim([-10, 100])     # 设置y轴范围
ax.set_xticks([0,20,40,60])
ax.set_xticklabels(['one', 'two', 'three', 'four'], rotation=30,
fontsize='small')
ax.set_xlabel('Stage')
ax.set_title("my matplot plot")
ax.legend(loc='best')
```



将图标保存到文件

利用 plt.savefig 可将当前图标保存到文件。该方法相当于 figure 对象的实例方法 savefig。

```
import matplotlib.pyplot as plt
plt.savefig('C:\Users\Administrator\Desktop\pythonCode\savepic.pdf', dpi=400,
format='pdf', bbox_inches='tight')
```

表8-2：Figure.savefig的选项

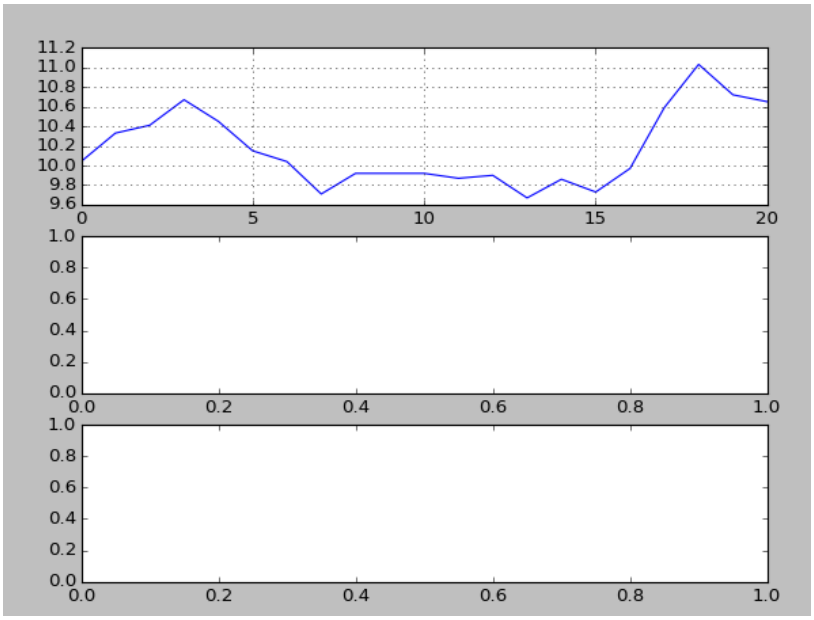
参数	说明
fname	含有文件路径的字符串或Python的文件型对象。图像格式由文件扩展名推断得出，例如，.pdf推断出PDF，.png推断出PNG
dpi	图像分辨率（每英寸点数），默认为100
facecolor、edgecolor	图像的背景色，默认为“w”（白色）
format	显式设置文件格式（“png”、“pdf”、“svg”、“ps”、“eps”……）
bbox_inches	图表需要保存的部分。如果设置为“tight”，则将尝试剪除图表周围的空白部分

Pandas 中的绘图函数

线性图

```
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
collection = db.sz002237
items = DataFrame(list(collection.find()))

fig = figure()
ax1 = fig.add_subplot(3,1,1)
ax2 = fig.add_subplot(3,1,2)
ax3 = fig.add_subplot(3,1,3)
close = items['close']
close.plot(label='close', ax=ax1, grid=True,)
```



该 Series 对象的索引会被传给 matplotlib，并用以绘制 x 轴，x 轴的刻度和界限可通过 xticks 和 xlim 选项进行调节，完整列表见表 8-3

表8-3：Series.plot方法的参数

参数	说明
label	用于图例的标签
ax	要在其上进行绘制的matplotlib subplot对象。如果没有设置，则使用当前matplotlib subplot
style	将要传给matplotlib的风格字符串（如'ko--'）
alpha	图表的填充不透明度（0到1之间）

参数	说明
kind	可以是'line'、'bar'、'barh'、'kde'
logy	在Y轴上使用对数标尺
use_index	将对象的索引用作刻度标签
rot	旋转刻度标签（0到360）
xticks	用作X轴刻度的值
yticks	用作Y轴刻度的值
xlim	X轴的界限（例如[0, 10]）
ylim	Y轴的界限
grid	显示轴网格线（默认打开）

DataFrame 的 plot 方法会在一个 subplot 中为各列绘制一条线，并自动创建图例。

```
df = items[['close', 'open', 'low', 'high']]
df.plot(title='sz002237',grid=True)
```

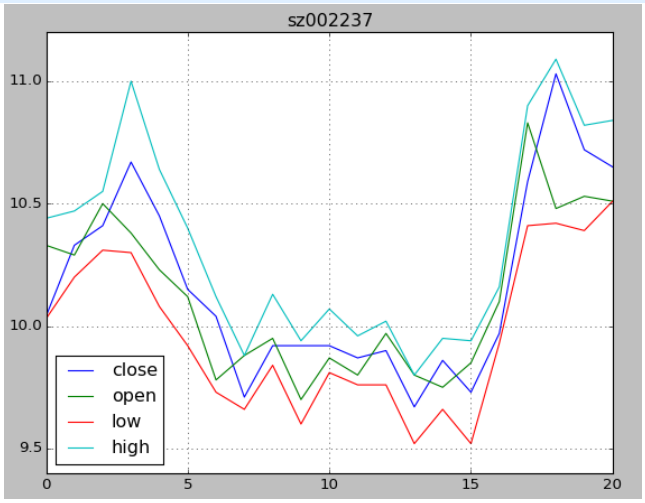


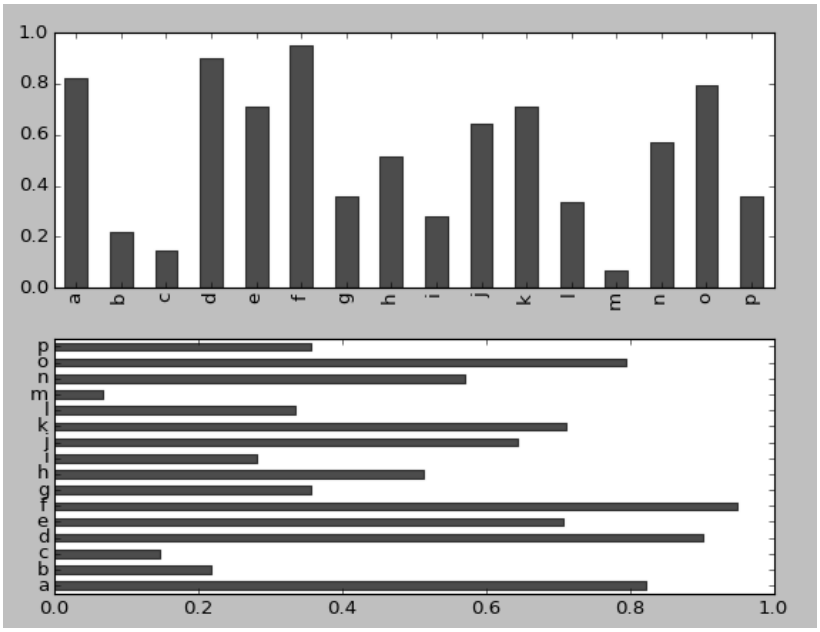
表8-4：专用于DataFrame的plot的参数

参数	说明
subplots	将各个DataFrame列绘制到单独的subplot中
sharex	如果subplots=True，则共用同一个X轴，包括刻度和界限
sharey	如果subplots=True，则共用同一个Y轴
figsize	表示图像大小的元组
title	表示图像标题的字符串
legend	添加一个subplot图例（默认为True）
sort_columns	以字母表顺序绘制各列，默认使用当前列顺序

柱状图

在生成线型吐得代码中加上 kind=‘bar’（垂直柱状图）或 kind=‘barh’即可生成柱状图。这时，Series 和 DataFrame 的索引将会被用作 x（bar）或 y（barh）刻度。

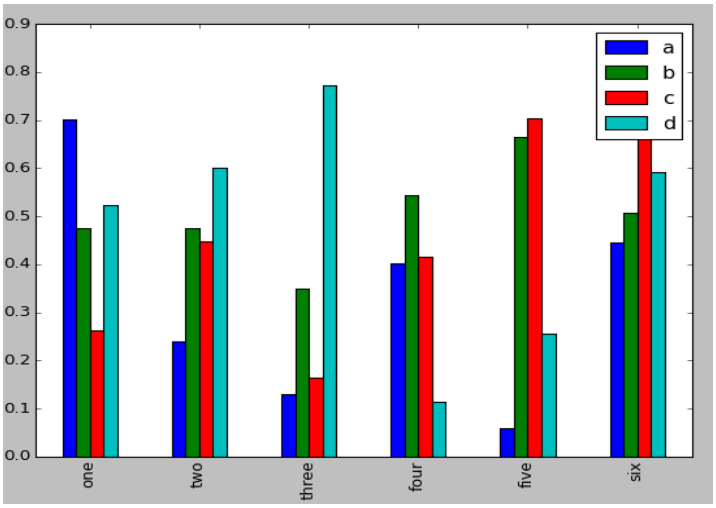
```
from matplotlib.pyplot import subplots
fig, axes = subplots(2,1)
data = Series(np.random.rand(16), index=list('abcdefghijklmnop'))
data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```



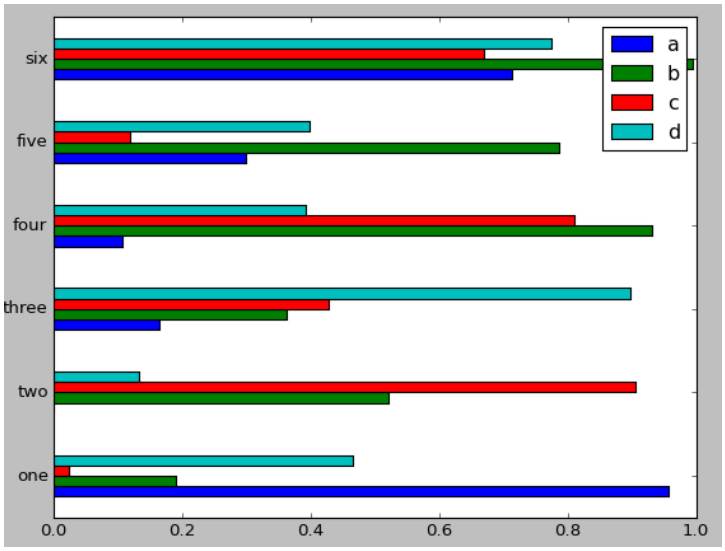
对于 DataFrame，柱状图会将每一行的值分为一组：

```
df = DataFrame(np.random.rand(6,4),
index=['one','two','three','four','five','six'],
```

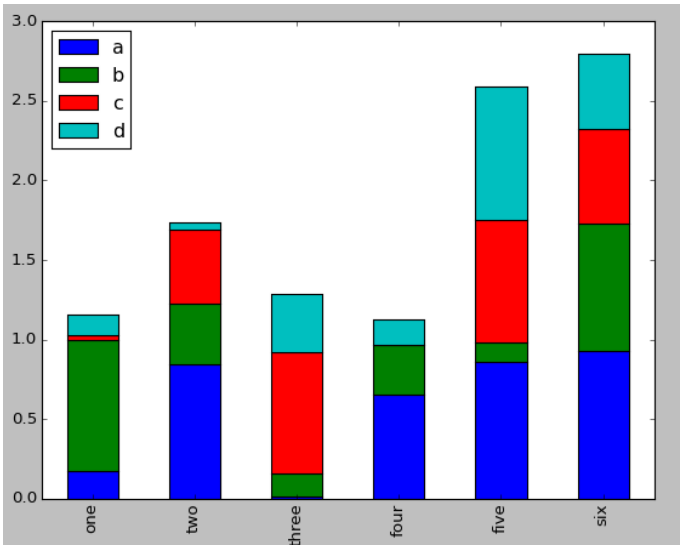
```
                columns=list('abcd'))
df.plot(kind='bar')
```




```
df.plot(kind='barh')
```

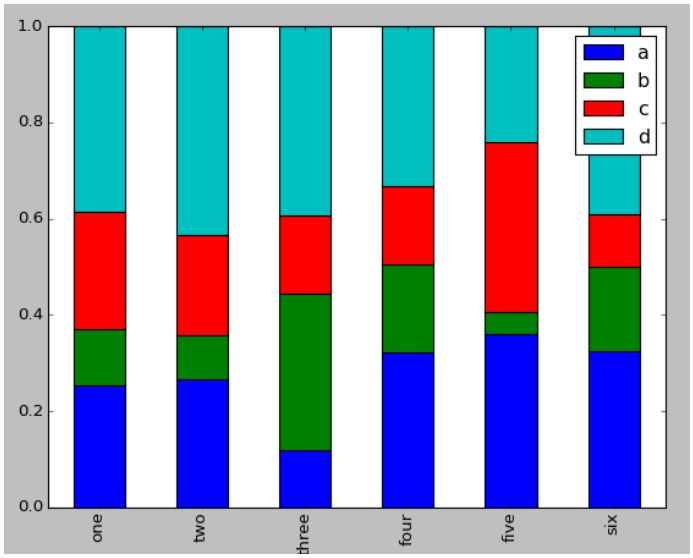


设置 stacked=True 即可为 DataFrame 生成堆积柱状图，这样每行的值会被堆积在一起：



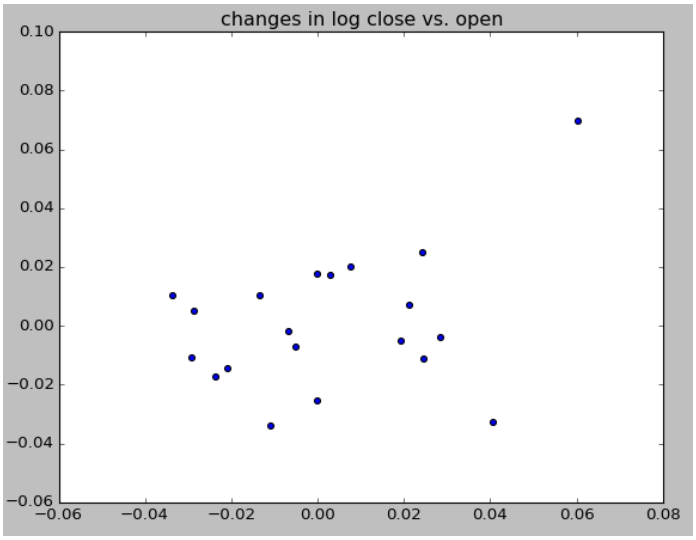
归一化，堆积柱状图：

```
df = df.div(df.sum(1), axis=0)
df.plot(kind='bar', stacked=True)
```



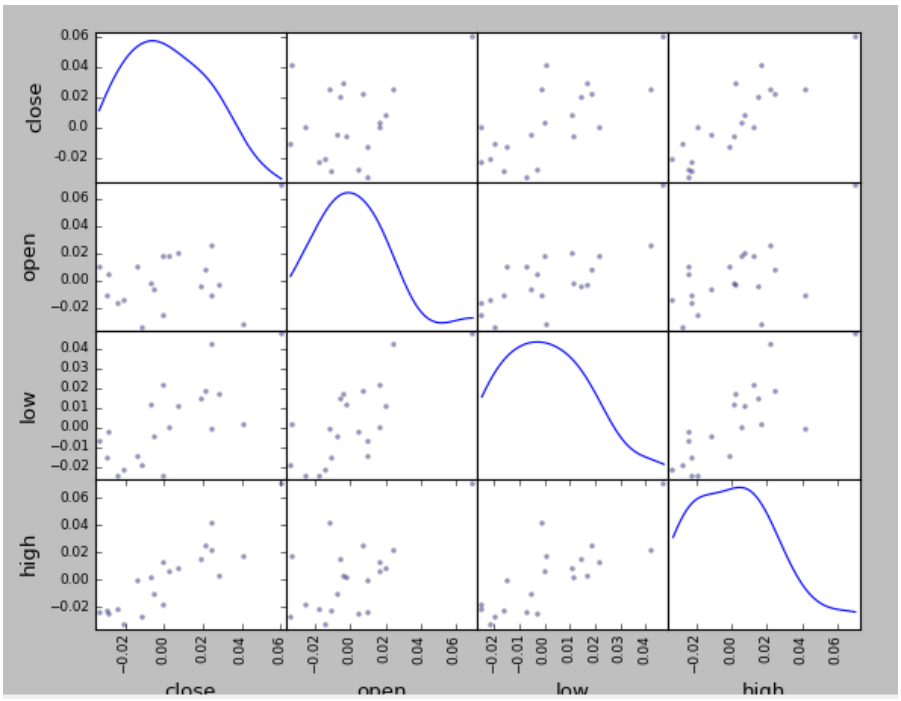
散布图

```
df = items[['close', 'open', 'low', 'high']]
trans_data = np.log(df).diff().dropna()
from matplotlib.pyplot import scatter
from matplotlib.pyplot import title
figure()
scatter(trans_data['close'], trans_data['open'])
title('changes in log %s vs. %s' % ('close', 'open'))
```



Pandas 提供了一个能从 DataFrame 创建散布图矩阵的 scatter_matrix 函数，支持在对角线上放置各变量的直方图或密度图。

```
pd.scatter_matrix(trans_data, diagonal='kde', alpha=0.3)
```



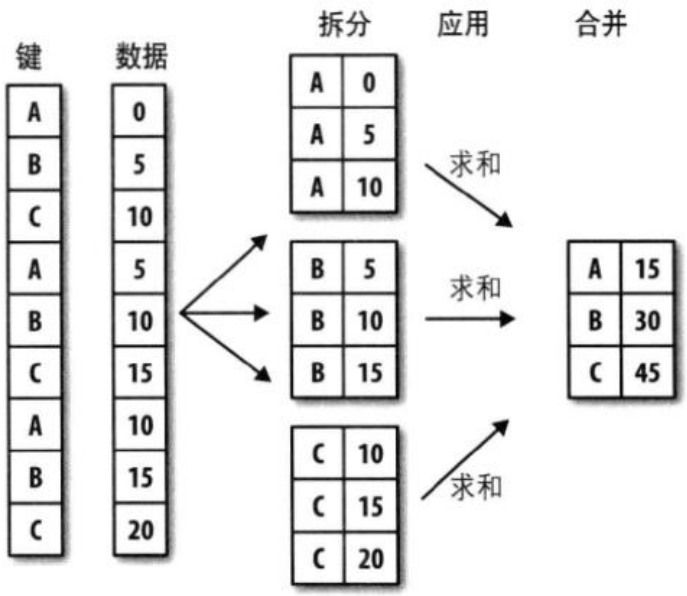
数据聚合与分组运算

对数据集进行分组并对各组应用一个函数（无论是聚合还是转换），这是数据工作中的重要环节。

GroupBy 技术

“split-apply-combine”（拆分-应用-合并）：

- 分组运算是第一阶段，pandas 对象中的数据会根据提供的一个多个键被拆分（split）为多组；
- 讲一个函数应用（apply）到各个分组并产生一个新值；
- 所有这些函数的执行结果会被合并（combine）到最终的结果对象中。




```
df = DataFrame({'key1':['a','a','b','b','a'],
                'key2':['one','two','one','two','one'],
                'data1':np.random.randn(5),
                'data2':np.random.randn(5)})
```

```
      data1       data2 key1 key2
0  0.445516 -0.732893     a  one
1  0.199184  2.091290     a  two
2  0.482823 -0.427802     b  one
3 -0.318903 -0.053562     b  two
4  0.098313 -0.908082     a  one
```

想要按 key1 进行分组，并计算 data1 列的平均值。

第一、进行分组，按可以 key1 访问 data1：

```
grouped = df['data1'].groupby(df['key1'])
```

```
<pandas.core.groupby.SeriesGroupBy object at 0x03A91290>
```

第二、进行计算，调用 GroupBy 的 mean 方法来计算分组平均值：

```
avg = grouped.mean()
```

```
key1
a      0.217605
```

b -1.221211

如果一次传入多个数组，会得到不同的结果：

```
avgs = df['data1'].groupby([df['key1'], df['key2']]).mean()
```

```
key1  key2
a     one     0.299802
      two     0.693281
b     one     0.403640
      two     0.177787
```

在执行 df.groupby('key1').mean()时，结果中没有 key2 列，这是因为 df['key2']不是数值数据，所以被从结果中排除了。默认情况下，所有数值都会被聚合，虽然有时可能会被过滤为一个子集。

按分组键分组

```
df = DataFrame({'key1':np.random.randn(5),
                'key2':np.random.randn(5),
                'data1':np.random.randn(5),
                'data2':np.random.randn(5)})
```

```
      data1     data2     key1      key2
0 -2.871995 -0.458403 -0.886256  0.217078
1  1.492347  0.051479  0.365784  0.674242
2 -0.068548  0.885587 -0.716703  1.329935
3 -0.499525 -1.068835  0.154941 -0.783925
4 -0.005732 -0.913097  0.973788 -0.634452
```

分组键值：

```
key=['one', 'two', 'one', 'one', 'two']
df1 = df.groupby(key).mean()
```

```
         data1     data2     key1      key2
one -1.146689 -0.213884 -0.482673  0.254363
two  0.743308 -0.430809  0.669786  0.019895
```

```
print dict(list(df.groupby(key)))['one']
```

```
      data1     data2     key1      key2
0 -2.871995 -0.458403 -0.886256  0.217078
2 -0.068548  0.885587 -0.716703  1.329935
3 -0.499525 -1.068835  0.154941 -0.783925
```

按 axis=0 进行分组，理解为将每个索引重新分配为分组键，进行分组。

对分组进行迭代

GroupBy 对象支持迭代，可以产生一组二元二组（由分组名和数据块组成）。

```
for name, group in df.groupby(df['key1']):
    print name
    print group
```

```
a
      data1     data2 key1 key2
0 -1.720255  0.100550    a  one
1 -0.104037  0.784891    a  two
4  0.083128  0.328684    a  one
b
      data1     data2 key1 key2
2 -2.078733 -0.800589    b  one
3  0.330629  0.841078    b  two
```

对于多重键的情况，元组的第一个元素将会是由键值组成的元组：

```
for (k1, k2), group in df.groupby([df['key1'], df['key2']]):
    print k1, k2
    print group
```

```
a one
      data1     data2 key1 key2
0  0.341317 -0.710914    a  one
4 -0.722868  1.455702    a  one
a two
      data1     data2 key1 key2
1  1.121077 -1.905531    a  two
b one
      data1     data2 key1 key2
2 -1.329791 -0.409626    b  one
b two
      data1     data2 key1 key2
3  0.016548 -0.564365    b  two
```

将这些数据片段做成一个字典：

```
pieces = dict(list( df.groupby(df['key1'])))
print pieces['a']
```

```
      data1     data2 key1 key2
0 -0.534646  0.222179    a  one
1 -0.632926  0.057627    a  two
4 -2.101187  0.149341    a  one
```

groupby 默认是在 axis=0 上进行分组的，通过设置可以在其他轴上进行分组，在通过字典或 Series 进行分组有介绍。

选取一个或一组列

如果用一个（单个字符串）或一组（字符串数组）列名对其进行索引，就能实现选取部分列进行聚合的目的：

df.groupby('key1')['data1'] -> Series

df.groupby('key1')[['data1']] ->DataFrame

跟以下代码的语法是等价的：

df ['data1'].groupby(df['key1])

df [['data1']].groupby(df['key1])

```
print df.groupby(['key1','key2'])['data1','data2'].mean()
```

```
                data1       data2
key1 key2
a    one   -0.407861 -0.053965
     two    0.061328  0.271033
b    one   -1.001777 -1.769835
     two   -0.784956  0.305674
```

通过字典或 Series 进行分组

```
df = DataFrame({'key1':np.random.randn(5),
                'key2':np.random.randn(5),
                'data1':np.random.randn(5),
                'data2':np.random.randn(5)})
mapping = {'data1':'red','data2':'red', 'key1':'green', 'key2':'green'}
print df.groupby(mapping, axis=1).mean()
```

```
       green       red
0 -0.211172  0.511038
1  0.597039 -0.860575
2  0.617000 -0.169173
3 -1.679058 -0.021179
4 -0.040688 -0.575604
```

通过函数进行分组

任何被当作分组键的函数都会在各个索引值上被调用一次，其返回值被用作分组名称。

创建函数：

```
def f(x):
    if x<10:
```

```
        return 10
    else:
        return 0
```

按索引的函数值分组：

```
print df.groupby(f).sum()
```

```
        data1       data2       key1        key2
10 -4.881017   0.412658 -0.063501 -3.330811
```

根据索引级别分组

层次化索引数据集最方便的地方在于它能够根据索引级别进行聚合。要实现该目的，通过 level 关键字传入级别编号或名称即可：

```
columns = pd.MultiIndex.from_arrays([['us','us','us','jp','jp'],
                                     [1,3,5,1,3]], names=['city', 'tenor'])
hier_df = DataFrame(np.random.randn(4,5), columns=columns)
```

```
city            us                                  jp
tenor            1           3           5           1           3
0         0.164384    0.148801    0.282967 -0.095955 -0.174523
1        -0.227091    0.771966    0.672069 -1.274180    1.401987
2        -0.049741    1.235101 -1.088575    0.938806 -0.289220
3        -1.520607 -1.097665    0.742297 -1.152529 -0.079337
```

```
count_df = hier_df.groupby(level='city', axis=1).count()
```

```
city   jp   us
0       2    3
1       2    3
2       2    3
3       2    3
```

数据聚合

对于聚合，是任何能够从数组产生标量值的数据转换过程。可以使用自己发明的聚合运算，还可以调用分组对象上已经定义好的任何方法。

自定义聚合运算

如果要使用自己的聚合函数，只需将其传入 aggregate 或 agg 方法即可：

```
df = DataFrame({'key1':['a','a','b','b','a'],
                'key2':['one','two','one','two','one'],
                'data1':np.random.randn(5),
                'data2':np.random.randn(5)})
```

```
       data1       data2 key1 key2
0   0.691044 -0.612119      a   one
1   1.794353    0.047946      a   two
```

```
2   0.051500 -2.318439      b   one
3   0.855911  0.494395      b   two
4   1.113767 -0.009171      a   one
```

```
grouped = df.groupby('key1')
def peak_to_peak(arr):
    return arr.max()-arr.min()
print grouped.agg(peak_to_peak)
```

```
          data1       data2
key1
a      1.103309    0.660065
b      0.804412    2.812834
```

表9-1：经过优化的groupby[译注4]的方法

函数名	说明
count	分组中非NA值的数量
sum	非NA值的和
mean	非NA值的平均值
median	非NA值的算术中位数
std、var	无偏（分母为n-1）标准差和方差
min、max	非NA值的最小值和最大值
prod	非NA值的积
first、last	第一个和最后一个非NA值

```
print grouped.agg('mean')
```

```
          data1       data2
key1
a     -0.917729 -0.287668
b      0.771406 -0.470491
```

面向列的多函数应用

希望对不同的列使用不同的聚合函数，或一次应用多个函数。

多个函数应用于全部列

```
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
collection = db.today_all_2
items = DataFrame(list(collection.find()))
print items.groupby('code')['trade'].size()
print  items.groupby('code')['open', 'low', 'trade'].agg(['mean',
peak_to_peak])
```

```
          open                      low                      trade
          mean peak_to_peak     mean peak_to_peak          mean peak_to_peak
```

code

000001	10.77	0	10.71	0	10.764000	0.02
000005	7.78	0	7.70	0	7.780000	0.00
000006	8.36	0	8.33	0	8.584000	0.01
000008	10.71	0	10.67	0	10.770000	0.00
000009	14.10	0	14.10	0	14.610000	0.00

不同列应用不同的函数

```
print items.groupby('code').agg({'open':['mean','max'], 'trade':['mean', peak_to_peak]})
```

	open		trade	
	mean	max	mean	peak_to_peak
code				
000001	10.77	10.77	10.764000	0.02
000005	7.78	7.78	7.780000	0.00
000006	8.36	8.36	8.584000	0.01
000008	10.71	10.71	10.770000	0.00

分组级运算和转换

聚合只是分组运算的一直，他能够将一维数组简化为标量值的函数。本节介绍 transform 和 apply 方法。

transform 方法

```
df = DataFrame({'key1':np.random.randn(5),
                'key2':np.random.randn(5),
                'data1':np.random.randn(5),
                'data2':np.random.randn(5)})
```

	data1	data2	key1	key2
0	-2.871995	-0.458403	-0.886256	0.217078
1	1.492347	0.051479	0.365784	0.674242
2	-0.068548	0.885587	-0.716703	1.329935
3	-0.499525	-1.068835	0.154941	-0.783925
4	-0.005732	-0.913097	0.973788	-0.634452

```
df1 = df.groupby(key).mean()
```

	data1	data2	key1	key2
one	-1.146689	-0.213884	-0.482673	0.254363
two	0.743308	-0.430809	0.669786	0.019895

```
df2 = df.groupby(key).transform(np.mean)
```

	data1	data2	key1	key2
0	-1.146689	-0.213884	-0.482673	0.254363
1	0.743308	-0.430809	0.669786	0.019895

2 -1.146689 -0.213884 -0.482673 0.254363
3 -1.146689 -0.213884 -0.482673 0.254363
4 0.743308 -0.430809 0.669786 0.019895

Transform 会将一个函数应用到各个分组，然后将结果放置到适当的位置上。如果各分组产生的是一个标量值，则该值就会被广播出去。注意与 agg 方法的差异！

apply 方法

apply 会将待处理的对象拆分成多个片段，然后对各片段调用传入的函数，最后尝试将各片段组合到一起。

```
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
collection = db.today_all_2
items = DataFrame(list(collection.find()))

def top(df, n=1, column='volume'):
    return df.sort_index(by=column)[-n:]
df1 = items.groupby('code').apply(top)
```

		_id	changepercent	code	high	low	\
code							
000001	23666	5715d8247921f60f309ea134	0.373	000001	10.81	10.71	
000005	23440	5715d8247921f60f309ea052	0.647	000005	7.87	7.70	
000006	22665	5715d8247921f60f309e9d4b	3.369	000006	8.75	8.33	
000008	24404	5715d8247921f60f309ea416	-0.554	000008	10.89	10.67	

……

Top 函数在 DataFrame 的各个片段上调用，然后结果由 pandas.concat 组装到一起，并以分组名称进行了标记。

如果传给 apply 的函数接受其他参数或关键字，则可以将这些内容放在函数名后面一并传入：

```
df2 = items.groupby('code').apply(top, 2)
```

		time_b	trade	turnoverratio	volume
code					
000001	21166	2016-04-19 15:02:17	10.77	0.18489	21823994
	23666	2016-04-19 15:02:39	10.77	0.18489	21823994
000005	20940	2016-04-19 15:02:17	7.78	1.95574	17870428
	23440	2016-04-19 15:02:39	7.78	1.95574	17870428
000006	20165	2016-04-19 15:02:17	8.59	3.13006	41951076
	22665	2016-04-19 15:02:39	8.59	3.13006	41951076

时间序列

日期和时间数据类型及工具

Datetime 模块

```
from datetime import datetime
now = datetime.now()
```

2016-04-20 15:11:37.067000

datetime 以毫秒形式存储日期和时间。

```
delta = datetime(2011,11,7,12,30) - datetime(2011,12,30)
print delta
```

-53 days, 12:30:00

datetime.timedelta 表示两 datetime 对象之间的时间差：

```
print datetime.now() + timedelta(days=10,hours=10)
```

2016-05-01 01:16:56.267000

表10-1：datetime模块中的数据类型

类型	说明
date	以公历形式存储日历日期（年、月、日）
time	将时间存储为时、分、秒、毫秒
datetime	存储日期和时间
timedelta	表示两个datetime值之间的差（日、秒、毫秒）

字符串和 datetime 的相互转换

利用 str、strftime 和 strptime 方法

Datetime -> 字符串

```
from datetime import datetime
now = datetime.now()
print str(now)
```

2016-04-20 15:20:54.467000

```
print now.strftime("%Y-%m-%d %H:%M:%S")
```

2016-04-20 15:20:54

字符串 -> Datetime

```
value = '2016-4-20 15:22:00'
print datetime.strptime(value, "%Y-%m-%d %H:%M:%S")
```
2016-04-20 15:22:00

datetime.strptime 是对已知格式进行日期解析的最佳方式。

利用 parse 方法

字符串 -> Datetime

对于常见格式可以使用 parse 方法

```
from dateutil.parser import parse
value = '2016-4-20 15:22:00'
print parse(value)
```
2016-04-20 15:22:00

利用 to_datetime 方法

字符串 -> Datetime

Pandas 的 to_datetime 方法可以解析日期：

```
value = ['2016-4-20 15:22:00', '2016-4-21 15:22:00']
print pd.to_datetime(value)
```
DatetimeIndex(['2016-04-20 15:22:00', '2016-04-21 15:22:00'], dtype='datetime64[ns]', freq=None, tz=None)

表10-2：datetime格式定义（兼容ISO C89）

代码	说明
%Y	4位数的年
%y	2位数的年
%m	2位数的月[01, 12]
%d	2位数的日[01, 31]

%H	时（24小时制）[00, 23]
%I	时（12小时制）[01, 12]
%M	2位数的分[00, 59]
%S	秒[00, 61]（秒60和61用于闰秒）
%w	用整数表示的星期几[0（星期天）, 6]
%U	每年的第几周[00, 53]。星期天被认为是每周的第一天，每年第一个星期天之前的那几天被认为是“第0周”
%W	每年的第几周[00, 53]。星期一被认为是每周的第一天，每年第一个星期一之前的那几天被认为是“第0周”
%z	以+HHMM或－HHMM表示的UTC时区偏移量，如果时区为naive[译注3]，则返回空字符串
%F	%Y-%m-%d简写形式，例如2012-4-18[译注4]
%D	%m/%d/%y简写形式，例如04/18/12

时间序列基础

创建时间序列

Pandas 最基本的时间序列类型就是以时间戳（通常为 datetime 对象表示）为索引的 Series：

```
from datetime import datetime
dates = [datetime(2011,1,2), datetime(2011,1,5), datetime(2011,1,7),
        datetime(2011,1,8), datetime(2011,1,10)]
ts = Series(np.random.randn(5), index=dates)
```

```
2011-01-02     0.049333
2011-01-05     1.239694
2011-01-07     0.501384
2011-01-08    -0.244129
2011-01-10     0.016669
```

这些 datetime 对象实际上是被放在一个 DatetimeIndex 中的。

跟其他 Series 一样，不同索引的时间序列之间算术运算会自动按日期补齐：

```
print ts + ts[::2]
```

```
2011-01-02     2.485904
2011-01-05          NaN
2011-01-07     0.571137
2011-01-08          NaN
2011-01-10     0.036752
```

DataFrame 的时间序列创建：

```
dates = pd.date_range('2011-10-1', periods=5, freq='W-WED')
long_df = DataFrame(np.random.randn(5,4), index=dates, columns=list('abcd'))
```

```
                   a         b         c         d
2011-10-05  0.213128  0.391661 -1.066885  0.324352
2011-10-12 -0.621748 -0.427417  1.999849 -0.341052
2011-10-19 -0.362396 -0.732818  0.810432  0.086517
2011-10-26  2.733625 -0.555588  0.709321  0.096736
2011-11-02  0.644310 -0.420109 -0.903250 -0.448338
```

索引、选取、子集构造

对于较长的时间序列，只需传入“年”或“年月”即可轻松选取数据的切片：

```
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
collection = db.sz002237
df = DataFrame(list(collection.find()))
df['date'] = pd.to_datetime(df['date']) # 将字符串日期转换为timestamp
df = df.set_index(['date'])             # 建立时间序列
print df['2016-4-12 09']
```

```
                        v_ma10      v_ma20      v_ma5     volume
date
2016-04-12 09:35:00    9917.31    9292.81   12769.9    28308.0
2016-04-12 09:40:00   11479.90    9543.18   15073.8    24444.5
2016-04-12 09:45:00   13201.00   10265.30   18102.3    21727.0
2016-04-12 09:50:00   13971.30   10556.20   18621.3    14196.1
2016-04-12 09:55:00   14189.00   10626.80   19935.8    11003.6
```

可以用不存在于该时间序列中的时间戳对其进行切片

```
print df['2016-4-12 09:30:00':'2016-4-12 9:48:00']
```

```
                     v_ma10    v_ma20    v_ma5   volume
date
2016-04-12 09:35:00   9917.31   9292.81  12769.9  28308.0
2016-04-12 09:40:00  11479.90   9543.18  15073.8  24444.5
2016-04-12 09:45:00  13201.00  10265.30  18102.3  21727.0
```

带有重复索引的时间序列

```
dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
'1/3/2000'])
dup_ts = Series(np.arange(5), index=dates)
```

```
2000-01-01    0
2000-01-02    1
2000-01-02    2
```

```
2000-01-02      3
2000-01-03      4
```

假设想要对具有非唯一时间戳的数据进行聚合。一个办法是使用 GroupBy，并传入 level=0：

```
unique = dup_ts.groupby(level=0).mean()
2000-01-01      0
2000-01-02      2
2000-01-03      4
```

日期的范围、频率以及移动

Pandas 有一套标准时间序列频率以及用于重采样、频率推断、生成固定频率日期范围的工具。

生成日期范围

Pandas.date_range 可用于生成指定长度的 DatetimeIndex：

生成指定长度

```
index = pd.date_range('2001-4-1', '2001-4-10')
DatetimeIndex(['2001-04-01', '2001-04-02', '2001-04-03', '2001-04-04',
                '2001-04-05', '2001-04-06', '2001-04-07', '2001-04-08',
                '2001-04-09', '2001-04-10'],
               dtype='datetime64[ns]', freq='D', tz=None)
```

默认情况下，date_range 会产生按天计算的时间点。起始和结束日期定义了日期索引的严格边界。

限制时间间隔

```
index = pd.date_range('2001-4-1', '2001-4-10', freq='2D')
DatetimeIndex(['2001-04-01', '2001-04-03', '2001-04-05', '2001-04-07',
                '2001-04-09', '2001-04-11'],
               dtype='datetime64[ns]', freq='2D', tz=None)
```

限制时间长度

```
index = pd.date_range('2001-4-1', periods=10, freq='3D')
DatetimeIndex(['2001-04-01', '2001-04-04', '2001-04-07', '2001-04-10',
                '2001-04-13', '2001-04-16', '2001-04-19', '2001-04-22',
                '2001-04-25', '2001-04-28'],
               dtype='datetime64[ns]', freq='3D', tz=None)
```

频率和日期偏移量

频率

基础频率通常以一个字符串别名表示，比如“M”表示每月，“H”表示每小时。对于每个基础频率，都有一个被称为日期偏移量的对象与之对应。

只需使用诸如“H”或“4H”这样的字符串别名即可。在基础频率前面放上一个整数即可创建倍数。

同理，可传入频率字符串（“2h30min”），这种字符串可以被高效地解析为等效的表达式：

```
index = pd.date_range('2001-4-1', periods=5, freq='1H40min')
DatetimeIndex(['2001-04-01 00:00:00', '2001-04-01 01:40:00',
               '2001-04-01 03:20:00', '2001-04-01 05:00:00',
               '2001-04-01 06:40:00'],
              dtype='datetime64[ns]', freq='100T', tz=None)
```

表10-4：时间序列的基础频率

别名	偏移量类型	说明
D	Day	每日历日
B	BusinessDay	每工作日
H	Hour	每小时
T或min	Minute	每分
S	Second	每秒
L或ms	Milli	每毫秒（即每千分之一秒）
U	Micro	每微秒（即每百万分之一秒）
M	MonthEnd	每月最后一个日历日

BM	BusinessMonthEnd	每月最后一个工作日
MS	MonthBegin	每月第一个日历日
BMS	BusinessMonthBegin	每月第一个工作日
W-MON、W-TUE…	Week	从指定的星期几（MON、TUE、WED、THU、FRI、SAT、SUN）开始算起，每周
WOM-1MON、WOM-2MON…	WeekOfMonth	产生每月第一、第二、第三或第四周的星期几。例如，WOM-3FRI表示每月第3个星期五
Q-JAN、Q-FEB…	QuarterEnd	对于以指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）结束的年度，每季度最后一月的最后一个日历日
BQ-JAN、BQ-FEB…	BusinessQuarterEnd	对于以指定月份结束的年度，每季度最后一月的最后一个工作日
QS-JAN、QS-FEB…	QuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个日历日
BQS-JAN、BQS-FEB…	BusinessQuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个工作日
A-JAN、A-FEB…	YearEnd	每年指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）的最后一个日历日
BA-JAN、BA-FEB…	BusinessYearEnd	每年指定月份的最后一个工作日
AS-JAN、AS-FEB…	YearBegin	每年指定月份的第一个日历日
BAS-JAN、BAS-FEB…	BusinessYearBegin	每年指定月份的第一个工作日

日期偏移（移动数据）

移动指的是沿着时间轴将数据前移或后移。Series 和 DataFrame 都有一个 shift 方法用于单纯的前移或后移操作。

对数据位移

```
ts = Series(np.random.randn(4), index=pd.date_range('2001-4-1', periods=4))
2001-04-01    0.252066
2001-04-02   -1.069974
2001-04-03   -1.614813
2001-04-04   -0.663889
ts = ts.shift(1)
2001-04-01         NaN
2001-04-02    0.252066
```

```
2001-04-03    -1.069974
2001-04-04    -1.614813
ts = ts.shift(-1)
2001-04-01     0.252066
2001-04-02    -1.069974
2001-04-03    -1.614813
2001-04-04           NaN
```

对时间戳位移

当添加频率参数时，此时的位移就是对时间戳进行位移

```
ts = ts.shift(1, freq='D')
2001-04-02    -0.558849
2001-04-03    -0.327193
2001-04-04    -0.304038
2001-04-05     1.278721
ts = ts.shift(-1, freq='D')
2001-04-01    -0.558849
2001-04-02    -0.327193
2001-04-03    -0.304038
2001-04-04     1.278721
```

时区处理

许多人选择以协调世界时（UTC，它是格林尼治标准时间（GMT）的接替者，目前已经是国际标准）来处理时间序列，时区是以 UTC 偏移量的形式表示的。

由于 pandas 包装了 pytz 的功能，只要记得时区的名称即可。时区名可以在文档中找到。

本地化和转换

时区转换用到 tz_localize、tz_convert 方法：第一步、确定本地时区；第二步、转换时区。

```
ts = Series(np.random.randn(4), index=pd.date_range('2001-4-1', periods=4))
2001-04-01    -1.626067
2001-04-02    -1.101749
2001-04-03     0.512226
2001-04-04    -0.522473
ts_utc = ts.tz_localize('UTC')
2001-04-01 00:00:00+00:00    -1.626067
2001-04-02 00:00:00+00:00    -1.101749
2001-04-03 00:00:00+00:00     0.512226
2001-04-04 00:00:00+00:00    -0.522473
```

```
ts_est = ts_utc.tz_convert('US/Eastern')
```
2001-03-31 19:00:00-05:00 -1.626067
2001-04-01 20:00:00-04:00 -1.101749
2001-04-02 20:00:00-04:00 0.512226
2001-04-03 20:00:00-04:00 -0.522473

不同时区之间的运算

如果两个不同时间序列的时区不同，将他们合并到一起，最终结果会是 UTC。由于时间戳其实是以 UTC 存储的，所以这是一个简单运算，不需要发生任何转换。

```
ts = ts_est + ts_utc
```
2001-04-01 00:00:00+00:00 2.352218
2001-04-02 00:00:00+00:00 2.249278
2001-04-03 00:00:00+00:00 4.183934
2001-04-04 00:00:00+00:00 -0.180315

Timestamp 对象时区转换

同本地化转换

```
value = '2016-4-20 15:22:00'
stamp = pd.Timestamp(value)
```
2016-04-20 15:22:00
```
stamp_utc = stamp.tz_localize('UTC')
```
2016-04-20 15:22:00+00:00
```
stamp_est = stamp_utc.tz_convert('US/Eastern')
```
2016-04-20 11:22:00-04:00
```
stamp_sh = stamp_utc.tz_convert('Asia/Shanghai')
```
2016-04-20 23:22:00+08:00

时期以及算术运算

时期（period）表示的是时间区间，比如数日、数月、数季、数年等。Period 类所表示的就是这种数据类型，其构造函数需要用到一个字符串或整数，以及表 10-4 中的频率：

```
p = pd.Period('2007', freq='A-DEC')
```

2007

Period_range 函数可用于创建规则的时间范围：

```
rng = pd.period_range('2001-4-1','2001-4-5',freq='D')
```
PeriodIndex(['2001-04-01', '2001-04-02', '2001-04-03', '2001-04-04',

'2001-04-05'],
dtype='int64', freq='D')

```
print rng.asfreq('M')
```

PeriodIndex(['2001-04', '2001-04', '2001-04', '2001-04', '2001-04'], dtype='int64', freq='M')

时期的频率转换

Period 和 PeriodIndex 对象都可以通过其 asfreq 方法被转换成别的频率。

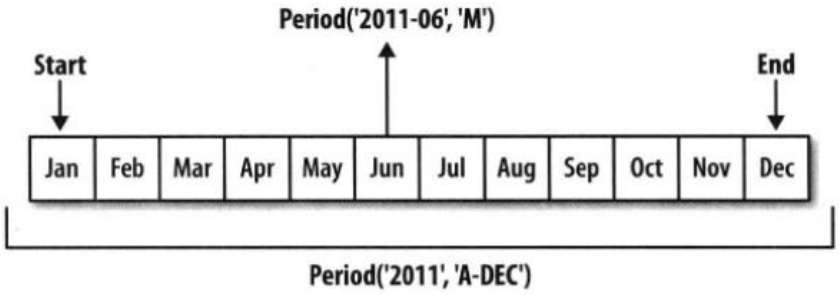
```
p = pd.Period('2007', freq='A-DEC')
print p.asfreq('M',how='end')
```

2007-12

```
print p.asfreq('M', how='start')
```

2007-01

可以将 Period('2007','A-DEC')看做一个被划分为多个月度时期的时间段中的游标。图 10-1 对此进行说明。




```
p = pd.Period('2007', freq='A-JUN')
print p.asfreq('M',how='end')
```

2007-06

```
print p.asfreq('M', how='start')
```

2006-07

按季度计算的时期频率

```
p = pd.Period('2014Q4', freq='Q-JAN')
```

2014Q4

上面是以 1 月结束的财年中，2014Q4 是从 11 月到 1 月。图 10-2 进行说明：


```
print p.asfreq('M', 'start')
```
2013-11
```
print p.asfreq('D', 'end')
```
2014-01-31

Period 之间的算术运算会非常简单。例如，要获取该季度倒数第二个工作日下午 4 点的时间戳：

```
p4pm = (p.asfreq('B','end')-1).asfreq('T','start')+60*4
print p4pm.to_timestamp()
```
2014-01-30 04:00:00

Timestamp 与 Period 相互转换

将 Timestamp -> Period，可以对 Series 和 DataFrame 使用 to_period 方法，直接将时间戳索引转换为时期戳索引：

```
df = DataFrame(np.random.randn(4,4), pd.date_range('2012-4-1 00:00:00',
freq='S',periods=4))
```
```
                            0         1         2         3
2012-04-01 00:00:00 -0.045163  1.747858  0.798775 -0.728527
2012-04-01 00:00:01  0.875872  0.623478  0.028988  1.025606
2012-04-01 00:00:02  0.677464 -2.162518 -0.504438  0.530255
2012-04-01 00:00:03  0.436050 -0.447054  1.106134 -0.645568
```
```
df = df.to_period('min')
```
```
                            0         1         2         3
2012-04-01 00:00 -0.045163  1.747858  0.798775 -0.728527
2012-04-01 00:00  0.875872  0.623478  0.028988  1.025606
2012-04-01 00:00  0.677464 -2.162518 -0.504438  0.530255
2012-04-01 00:00  0.436050 -0.447054  1.106134 -0.645568
```

将 Period ->Timestamp，可以对 Series 和 DataFrame 使用 to_timestamp 方法，直接将时期戳索引转换为时间戳索引：

```
df = df.to_timestamp(freq='s',how='end')
```
```
                            0         1         2         3
2012-04-01 00:00:59 -0.735270 -0.816308 -1.092244 -0.720311
```

2012-04-01 00:00:59 -0.742635 -1.571055 -1.476913 -0.238203
2012-04-01 00:00:59 -0.821683 0.440819 1.024575 0.122602
2012-04-01 00:00:59 -0.025346 2.090951 -0.994120 -0.254794

重采样及频率转换

重采样指的是将时间序列从一个频率转换到另一个频率的处理过程。

```
df = DataFrame(np.random.randn(100,4), pd.date_range('2016-4-1', freq='D',
periods=100))
df = df.resample('M', how='mean')
```

	0	1	2	3
2016-04-30	0.001021	0.494716	0.093167	0.022736
2016-05-31	-0.061791	-0.028323	-0.052431	0.036357
2016-06-30	0.001247	-0.055189	0.003550	0.150103
2016-07-31	0.603814	0.073968	-0.109508	-0.331184

表10-5：resample方法的参数

参数	说明
freq	表示重采样频率的字符串或DateOffset，例如'M'、'5min'或Second(15)
how='mean'	用于产生聚合值的函数名或数组函数，例如'mean'、'ohlc'、np.max等。默认为'mean'。其他常用的值有：'first'、'last'、'median'、'ohlc'、'max'、'min'
axis=0	重采样的轴，默认为axis=0
fill_method=None	升采样时如何插值，比如'ffill'或'bfill'。默认不插值
closed='right'	在降采样中，各时间段的哪一端是闭合（即包含）的，'right'或'left'。默认为'right'
label='right'	在降采样中，如何设置聚合值的标签，'right'或'left'（面元的右边界或左边界）。例如，9:30到9:35之间的这5分钟会被标记为9:30或9:35。默认为'right'（本例中就是9:35）

参数	说明
loffset=None	面元标签的时间校正值，比如'-1s' / Second(-1)用于将聚合标签调早1秒
limit=None	在前向或后向填充时，允许填充的最大时期数
kind=None	聚合到时期（'period'）或时间戳（'timestamp'），默认聚合到时间序列的索引类型
convention=None	当重采样时期时，将低频率转换到高频率所采用的约定（'start'或'end'）。默认为'end'

降采样

将数据聚合到规整的低频率是一件非常普通的时间序列处理任务。

在用 resample 对数据进行降采样时，需要考虑两样东西：

- 各区间哪边是闭合的。
- 如何标记各个聚合区间，用区间的开头还是末尾。

```
df = DataFrame(np.arange(1,13), pd.date_range('2016-4-1', freq='T',
periods=12))
```

```
                     0
2016-04-01 00:00:00  1
2016-04-01 00:01:00  2
2016-04-01 00:02:00  3
2016-04-01 00:03:00  4
2016-04-01 00:04:00  5
2016-04-01 00:05:00  6
2016-04-01 00:06:00  7
2016-04-01 00:07:00  8
2016-04-01 00:08:00  9
2016-04-01 00:09:00  10
2016-04-01 00:10:00  11
2016-04-01 00:11:00  12
```

```
df1 = df.resample('5min',how='sum',label='right',closed='right')
```

```
                     0
2016-04-01 00:00:00  1
2016-04-01 00:05:00  20
2016-04-01 00:10:00  45
2016-04-01 00:15:00  12
```

```
df2 = df.resample('5min',how='sum',label='right',closed='left')
```

```
                     0
2016-04-01 00:05:00  15
2016-04-01 00:10:00  40
2016-04-01 00:15:00  23
```

确定哪个区间闭合用的是 closed 参数，确定区间的开头还是末尾标记用是的是 label 参数。

图 10-3 说明了“1 分钟”数据被转换为“5 分钟”数据的处理过程。

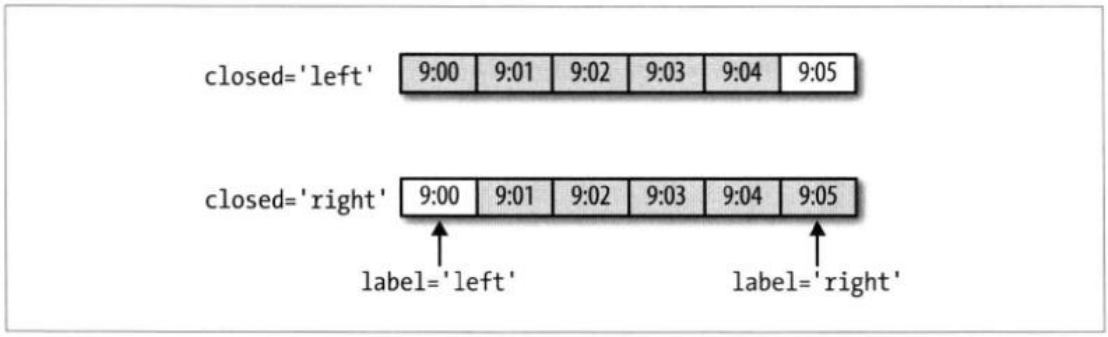


图10-3：各种closed、label约定的“5分钟”重采样演示

OHLC 重采样

金融领域中有一种无所不在的时间序列聚合方式，即计算各面元的四个值：第一个值（open，开盘）、最后一个值（close，收盘）、最大值（high，最高）以及最小值（low，最低）。传入 how='ohlc'即可得到一个含有这四种聚合值的 DataFrame。

```
df3 = df.resample('5min',how='ohlc',label='right',closed='left')
```

	open	high	low	close
2016-04-01 00:05:00	1	5	1	5
2016-04-01 00:10:00	6	10	6	10
2016-04-01 00:15:00	11	12	11	12

对金融数据的降采样

若金融数据为 tick 级别，如：

```
df = DataFrame(np.arange(1000000), pd.date_range('2016-4-1 00:00:00',
freq='500ms', periods=1000000))
```

方法一：采样 Timestamp->Period 手工处理，或者 ohlc：

```
df = df.to_period('min')                    # 将 Timestamp -> period
stock_open = df.groupby(level=0).first()    # 按需要的时间进行聚合
stock_close = df.groupby(level=0).last()
stock_low = df.groupby(level=0).min()
stock_high = df.groupby(level=0).max()
stock = pd.concat([stock_open, stock_close, stock_low, stock_high], axis=1)
stock.columns = ['open', 'close', 'low', 'high']
stock.index = stock.index.asfreq('min')+1   # label 选用选取末尾
stock = stock.to_timestamp('S')             # 将 period -> Timestamp
print stock.head()
```

	open	close	low	high
2016-04-01 00:01:00	0	119	0	119
2016-04-01 00:02:00	120	239	120	239

```
2016-04-01 00:03:00    240     359   240     359
2016-04-01 00:04:00    360     479   360     479
2016-04-01 00:05:00    480     599   480     599
```

方法二：直接采用 resample 方法，处理数据，对于金融数据，选用 close=left，label=right：

```
df = DataFrame(np.arange(1000000), pd.date_range('2016-4-1 00:00:00',
freq='500ms', periods=1000000))
stock = df.resample('min',how='ohlc',label='right',closed='left')
print stock.head()
```

```
                     open high   low close
2016-04-01 00:01:00     0   119     0   119
2016-04-01 00:02:00   120   239   120   239
2016-04-01 00:03:00   240   359   240   359
2016-04-01 00:04:00   360   479   360   479
2016-04-01 00:05:00   480   599   480   599
```

同理可以从 tick->3min、5min、10min、15min；

测试发现用 resample 方法速度快，可以直接进行聚合到 3min、5min、10min、15min，不需要从 1min 到 3min、5min、10min、15min 的中间转换。

升采样和插值

在将数据从低频率转换到高频率时，默认会引入缺失值：

```
df = DataFrame(np.arange(3), pd.date_range('2016-4-1 00:00:00', freq='10D',
periods=3))
```

```
            0
2016-04-01  0
2016-04-11  1
2016-04-21  2
```

```
df1 = df.resample('3D')
```

```
              0
2016-04-01    0
2016-04-04  NaN
2016-04-07  NaN
2016-04-10  NaN
2016-04-13  NaN
2016-04-16  NaN
2016-04-19  NaN
```

Resample 的填充和插值方式跟 fillna 和 reindex 的一样：

```
df2 = df.resample('3D', fill_method='ffill')
```

```
             0
2016-04-01   0
2016-04-04   0
2016-04-07   0
2016-04-10   0
2016-04-13   1
2016-04-16   1
2016-04-19   1
```

通过时期进行重采样

```
frame = DataFrame(np.random.randn(24,4),
index=pd.period_range('2000-1','2001-12',freq='M'))
                 0         1         2         3
2000-01 -0.441679  0.519802  1.274544  1.276596
2000-02 -0.311124 -1.337438 -1.863059 -1.310897
2000-03 -3.034698 -0.008900  0.432500  1.222504
2000-04  0.661234  1.239868  0.770950  0.706017
2000-05 -0.661363  1.294814  0.197158  0.644761
anuual_frame = frame.resample('A-DEC', how='mean')
              0         1         2         3
2000 -0.017582  0.341246 -0.046359  0.132039
2001 -0.229836 -0.406371 -0.481831  0.073032
```

时间序列绘图

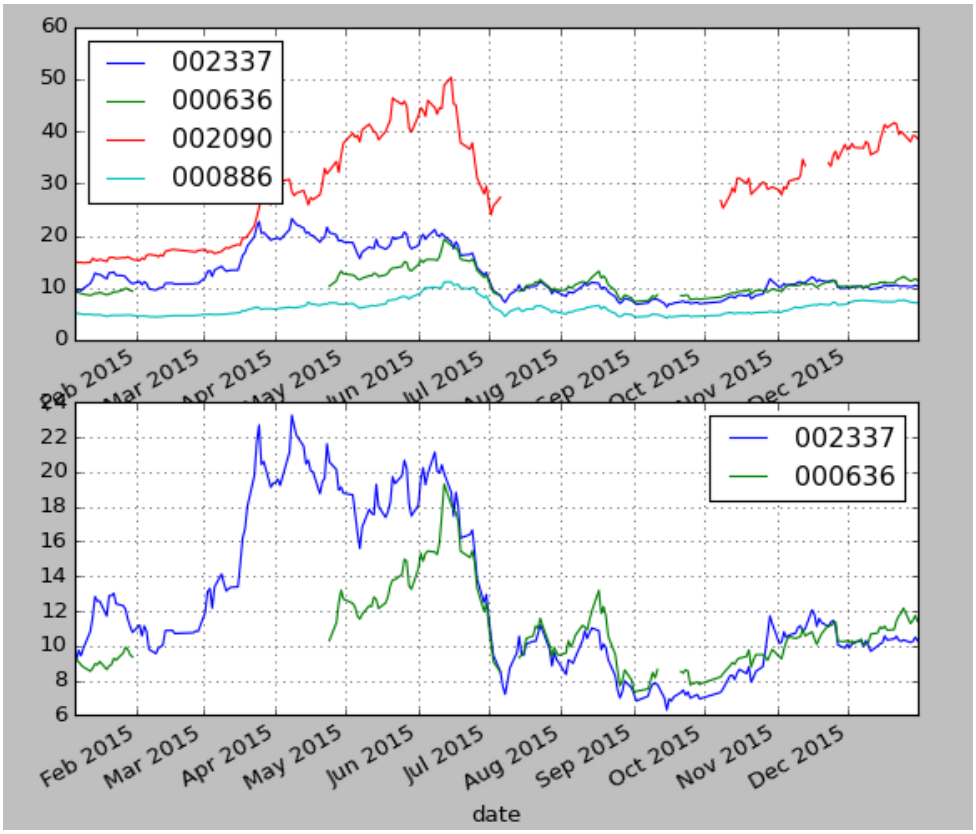
Pandas 时间序列的绘图能力在日期格式化方面比 matplotlib 原生的要好。

时间序列 plot

```
stocklist = ['002337','000636','002090','000886']
conn = pymongo.MongoClient('127.0.0.1', port=27017)
db = conn.db
first_list =1
for stockcode in stocklist:
    collection = db['sz' + stockcode]
    stockdocs = DataFrame(list(collection.find()))
    def str_date(value):
        return datetime.strptime(value, "%Y-%m-%d %H:%M:%S")
    stockdocs['date'] = DataFrame(stockdocs['date_str']).applymap(str_date) #
将字符串转换为时间戳
    stockdocs = stockdocs.set_index(['date'])     # 将时间戳作为索引
    if first_list == 1:
        close_stock = stockdocs['close']
```

```
        first_list =0
    else:
        close_stock = pd.concat([close_stock, stockdocs['close']], axis=1)
close_stock.columns = stocklist                # 定义列索引

from matplotlib.pyplot import figure
fig = figure()
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)
close_stock['2015'].plot(ax=ax1, grid=True)     # 画出 2015 年的数据
close_stock[['002337','000636']].ix['2015'].plot(ax=ax2, grid=True)
from matplotlib.pyplot import show
show()
```

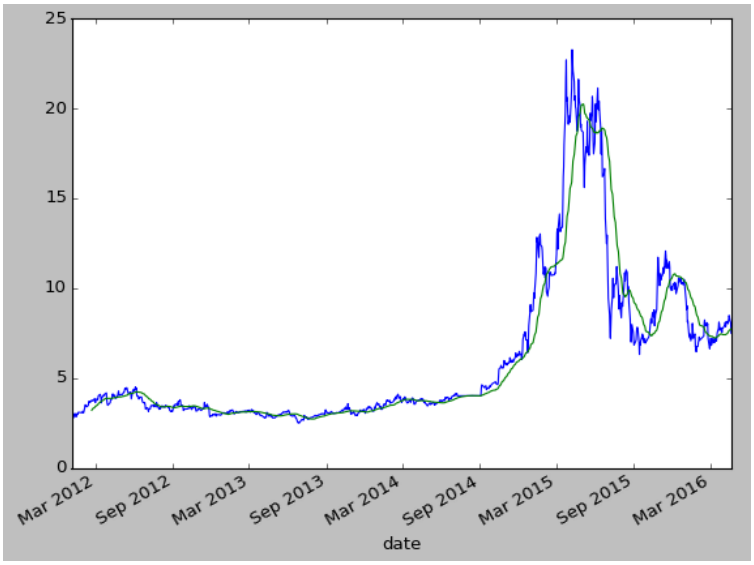


移动窗口函数

在移动窗口（可以带有指数衰减权数）上计算的各种统计函数也是一类常见于时间序列的数组变换。

```
close_stock.fillna(method='ffill', inplace=True)    # 填充缺失值（由于停盘有缺失）
close_stock['002337MA30'] = pd.rolling_mean(close_stock['002337'], 30)
```

```
close_stock['002337'].plot()
close_stock['002337MA30'].plot()
```




```
close_stock[['002337','002337MA30']].ix['2015'].plot(grid=True)
```

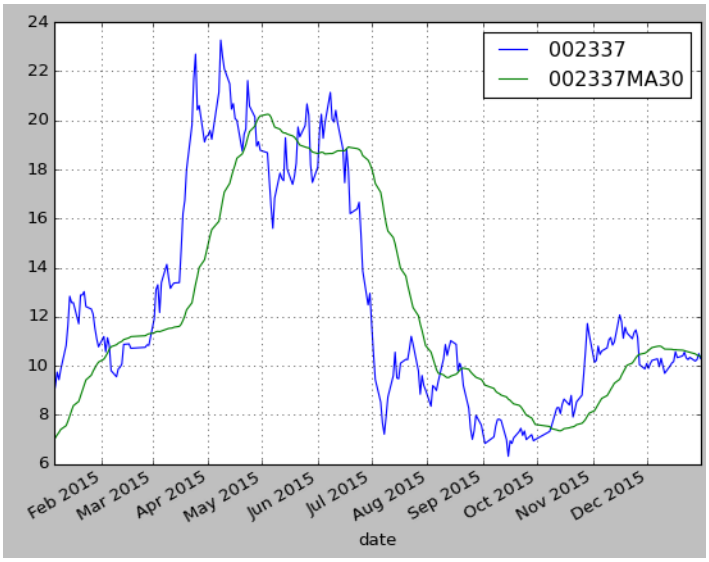


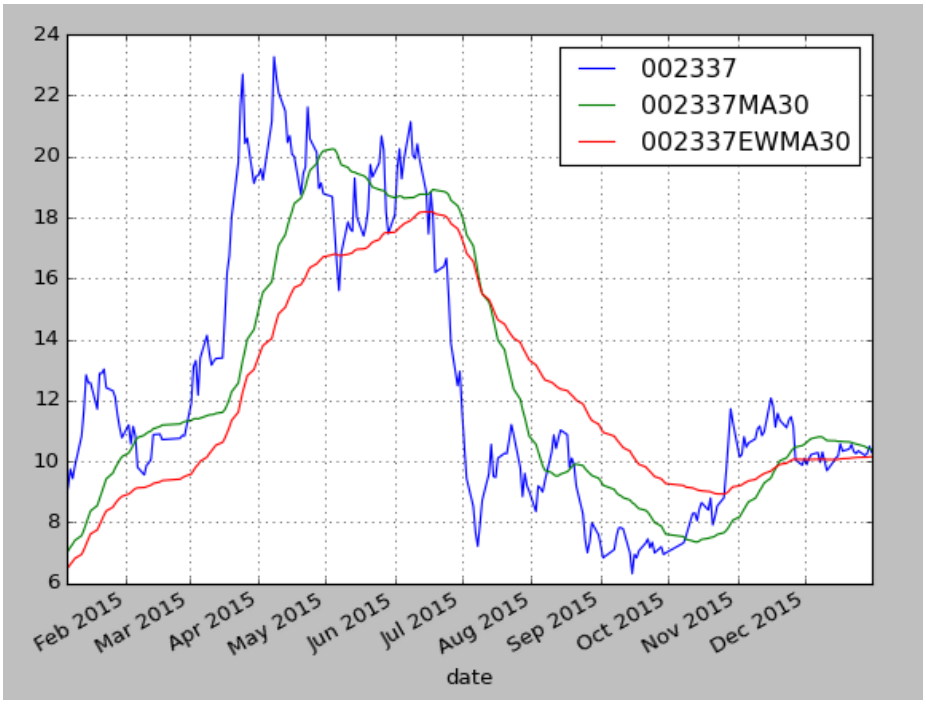
表10-6：移动窗口和指数加权函数

函数	说明
rolling_count	返回各窗口非NA观测值的数量
rolling_sum	移动窗口的和
rolling_mean	移动窗口的平均值
rolling_median	移动窗口的中位数
rolling_var、rolling_std	移动窗口的方差和标准差。分母为n－1
rolling_skew、olling_kurt	移动窗口的偏度（三阶矩）和峰度（四阶矩）
rolling_min、rolling_max	移动窗口的最小值和最大值
rolling_quantile	移动窗口指定百分位数/样本分位数位置的值
rolling_corr、rolling_cov	移动窗口的相关系数和协方差
rolling_apply	对移动窗口应用普通数组函数
ewma	指数加权移动平均
ewmvar、ewmstd	指数加权移动方差和标准差
ewmcorr、ewmcov	指数加权移动相关系数和协方差

指数加权函数

另一种使用固定大小窗口及相等权数观测值的办法是，定义一个衰减因子常量，以便使近期的观测值拥有更大的权数。

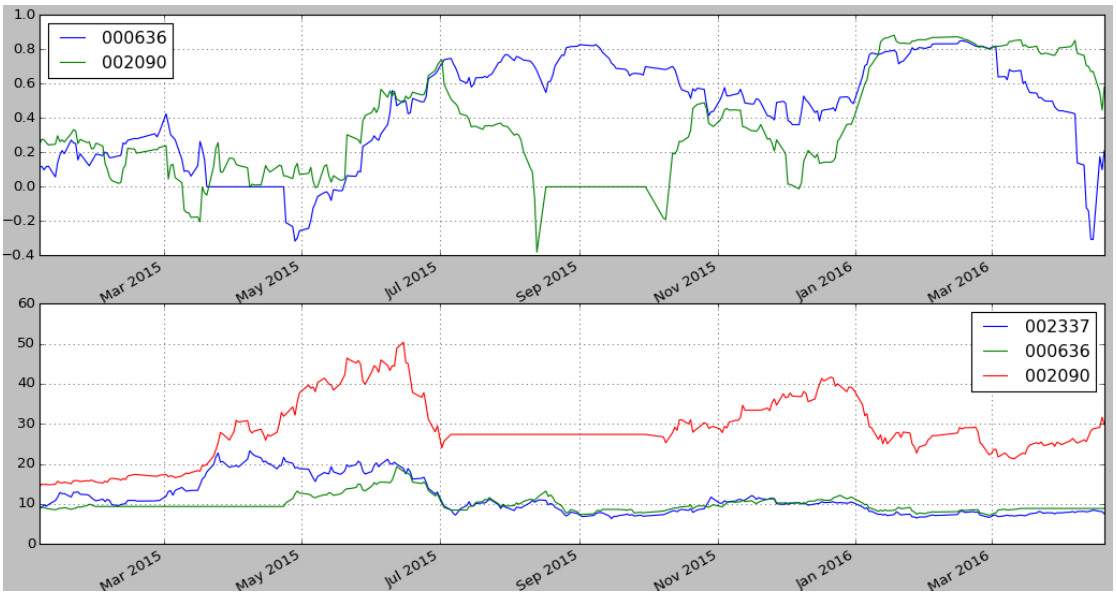
```
close_stock['002337EWMA30'] = pd.ewma(close_stock['002337'], 30)
close_stock[['002337','002337MA30','002337EWMA30']].ix['2015'].plot(grid=Tr
ue)
```



二元移动窗口函数

有些统计运算（如相关系数和协方差）需要在两个时间序列上执行。

```
sz002337 = close_stock['002337']
sz002337_rets = sz002337 / sz002337.shift(1) - 1    # 计算单只票的回报率
returns = close_stock.pct_change()                  # 计算所有票的回报率
corr = pd.rolling_corr(returns[['000636','002090']], returns['002337'], 30) #
计算移动窗口的相关系数
fig = figure()
ax1 = fig.add_subplot(2,1,1)
ax2 = fig.add_subplot(2,1,2)
corr['2015':'2016'].plot(ax=ax1, grid=True)         # 绘制相关系数
close_stock[['002337','000636','002090']].ix['2015':'2016'].plot(ax=ax2,
grid=True) #绘制股价，验证相关性
```



金融和经济数据应用

Pandas 可以在算术运算中自动补齐数据。在实际工作中，这不仅能带来极大的自由度，而且还能提高工作效率。

频率不同的时间序列的运算

频率转换和重对齐的两大主要工具是 resample 和 reindex 方法。Resample 用于将数据转换到固定频率，而 reindex 则用于使数据符合一个新索引。

```
ts1 = DataFrame(np.arange(3), index=pd.date_range('2016-4-1', periods=3,
freq='W-WED'))
print ts1
```

```
            0
2016-04-06  0
2016-04-13  1
2016-04-20  2
```

```
print ts1.resample('B')
```

```
             0
2016-04-06   0
2016-04-07 NaN
2016-04-08 NaN
2016-04-11 NaN
2016-04-12 NaN
2016-04-13   1
2016-04-14 NaN
2016-04-15 NaN
2016-04-18 NaN
2016-04-19 NaN
2016-04-20   2
```

在实际工作中，将较低频率的数据升采样到较高的规整频率是一种不错的解决方案，但是对于更一般化的不规整时间序列可能就不太合适。

```
ts2 = DataFrame(np.arange(3), index=pd.date_range('2016-4-4', periods=3,
freq='W-FRI'))
print ts2
```

```
            0
2016-04-08  0
2016-04-15  1
2016-04-22  2
```

如果要将 ts2 中‘最当前’的值（即前向填充）加到 ts1 上。一个方法是将两者重采样为规则频率后再相加，但是如果想维持 ts1 中的日期索引，则 reindex 会是一种更好的方案：

```
print ts2.reindex(ts1.index, method='ffill')
```

```
             0
2016-04-06 NaN
2016-04-13   0
2016-04-20   1
```

```
print ts1 + ts2.reindex(ts1.index, method='ffill')
```

```
             0
2016-04-06 NaN
2016-04-13   1
2016-04-20   3
```

时间数据选取

假设一个很长的盘中市场数据时间序列，希望抽取其中每天特定时间的价格数据，

```
ts = DataFrame(np.arange(200), index=pd.date_range('2016-4-22 09:00:00',
periods=200, freq='min'))
```

```
                     0
2016-04-22 09:00:00  0
2016-04-22 09:01:00  1
2016-04-22 09:02:00  2
2016-04-22 09:03:00  3
2016-04-22 09:04:00  4
2016-04-22 09:05:00  5
2016-04-22 09:06:00  6
…..
```

利用 python 的 datetime.time 对象进行索引即可抽出这些时间点上的值，实际上操作用到了实例方法 at_time：

```
from  datetime import time
print ts.at_time(time(10,0))
```

```
                     0
2016-04-22 10:00:00  60
```

还有一个 between_time 方法，它用于选取两个 time 对象之间的值：

```
print ts.between_time(time(10,0), time(10,8))
```

```
                     0
2016-04-22 10:00:00  60
2016-04-22 10:01:00  61
2016-04-22 10:02:00  62
2016-04-22 10:03:00  63
2016-04-22 10:04:00  64
2016-04-22 10:05:00  65
2016-04-22 10:06:00  66
2016-04-22 10:07:00  67
2016-04-22 10:08:00  68
```

收益指数和累计收益

收益指数

```
index = pd.date_range('2001-1-29', periods=40, freq='D')
price = DataFrame(np.arange(1,41), index=index)
returns = price.pct_change()
ret_index = (1 + returns).cumprod()
ret_index.ix[0] = 1
print returns
```

```
                   0
2001-01-29       NaN
2001-01-30  1.000000
2001-01-31  0.500000
2001-02-01  0.333333
2001-02-02  0.250000
2001-02-03  0.200000
....
```

```
print ret_index
```

```
             0
2001-01-29   1
2001-01-30   2
2001-01-31   3
2001-02-01   4
2001-02-02   5
2001-02-03   6
.....
```

Returns 为每日收益；

Ret_index 即为每日收益指数；

累计收益

计算每月的累计收益，只需要对每日收益指数进行降采样：

```
print ret_index.resample('M', how='last')
```

```
             0
2001-01-31   3
2001-02-28  31
2001-03-31  40
```

```
m_returns = ret_index.resample('M', how='last').pct_change()
m_first = ret_index.resample('M', how='first')
m_last = ret_index.resample('M', how='last')
m_returns.values[0] = (m_last.values[0] - m_first.values[0]) /
m_first.values[0]
print m_returns
```

	0
2001-01-31	2.000000
2001-02-28	9.333333
2001-03-31	0.290323

m_returns 为每月累计收益；

分组变换

行业分类

随机生成 1000 个股票代码，创建一个含有 2 列的 DataFrame 来承载假想数据：

```
import random; random.seed(10)
import string
N = 1000
def rands(n):
    choices = string.ascii_uppercase
    return ''.join([random.choice(choices) for _ in xrange(n)])
tickers = np.array([rands(5) for _ in xrange(N)])
M = 500
df = DataFrame({'volume':np.random.randn(M)/200 + 0.03,
                'value':np.random.randn(M)/200 + 0.08},
               index = tickers[:M])
```

为股票创建一个行业分类，只选用两个行业：

```
ind_name = np.array(['finance', 'tech'])
sampler = np.random.randint(0, len(ind_name), M)
industries = Series(ind_name[sampler], index=tickers[:M], name='industry')
```

用 groupy 分组键聚合技术，根据行业分类进行分组并执行分组聚合和变换：

```
by_industry = df.groupby(industries)
print by_industry.mean()
```

industry	value	volume
finance	0.080146	0.030434
tech	0.080213	0.029623

行业内标准化处理

```
# 行业内标准化处理
def zscore(group):
```

```
    # print group.mean()
    return (group - group.mean()) / group.std()
df_stand = df.groupby(industries).apply(zscore)
print df_stand
```

```
            value       volume
OLPFV -0.496828 -0.374068
VQENI -0.221098 -0.921791
GYZBW -0.638921 -0.177768
PJHRL -0.074188 -0.233968
RRDTZ    0.168312 -0.378706
……
```