

pandas入门

pandas是本书后续内容的首选库。它含有使数据分析工作变得更快更简单的高级数据结构和操作工具。pandas是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。

先介绍一点背景。我是在2008年早期还在AQR（一家定量投资管理公司）任职期间开始着手构建pandas的。那时候，没有任何一个单独的工具能够满足我工作上的全部需求：

- 具备按轴自动或显式数据对齐功能的数据结构。这可以防止许多由于数据未对齐以及来自不同数据源（索引方式不同）的数据而导致的常见错误。
- 集成时间序列功能。
- 既能处理时间序列数据也能处理非时间序列数据的数据结构。
- 数学运算和约简（比如对某个轴求和）可以根据不同的元数据（轴编号）执行。
- 灵活处理缺失数据。
- 合并及其他出现在常见数据库（例如基于SQL的）中的关系型运算。

我希望能够在一个地方完成所有这些事情，最好是一种能进行通用软件开发的语言。Python是一门不错的候选语言，但那时候它还没有一组能完全提供上述功能的数据结构和工具。

在过去的4年中，pandas逐渐成长为一个非常大的库，它所能解决的数据处理问题已经比我期望的要多得多了。但随着其范围的扩大，它也逐渐背离了我最初所期望的简洁性和易用性。我希望你在读完本书之后，也能像我一样认为它是一个不可或缺的工具。

在本书后续部分中，我将使用下面这样的pandas引入约定：



```
In [1]: from pandas import Series, DataFrame
```

```
In [2]: import pandas as pd
```

因此，只要你在代码中看到pd.，就得想到这是pandas。因为Series和DataFrame用的次数非常多，所以将其引入本地命名空间中会更方便。

pandas的数据结构介绍

要使用pandas，你首先就得熟悉它的两个主要数据结构：Series和DataFrame。虽然它们并不能解决所有问题，但它们为大多数应用提供了一种可靠的、易于使用的基础。

Series

Series是一种类似于一维数组的对象，它由一组数据（各种NumPy数据类型）以及一组与之相关的数据标签（即索引）组成。仅由一组数据即可产生最简单的Series：

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

```
In [5]: obj
```

```
Out[5]:
```

0	4
1	7
2	-5
3	3

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指定索引，于是会自动创建一个0到N-1（N为数据的长度）的整数型索引。你可以通过Series的values和index属性获取其数组表示形式和索引对象：

```
In [6]: obj.values
```

```
Out[6]: array([ 4,  7, -5,  3])
```

```
In [7]: obj.index
```

```
Out[7]: Int64Index([0, 1, 2, 3])
```

通常，我们希望所创建的Series带有一个可以对各个数据点进行标记的索引：

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [9]: obj2
```

```
Out[9]:
```

d	4
b	7
a	-5
c	3



```
In [10]: obj2.index  
Out[10]: Index([d, b, a, c], dtype=object)
```

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

```
In [11]: obj2['a']  
Out[11]: -5  
  
In [12]: obj2['d'] = 6  
  
In [13]: obj2[['c', 'a', 'd']]  
Out[13]:  
c    3  
a   -5  
d    6
```

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链接：

```
In [14]: obj2  
Out[14]:  
d    6  
b    7  
a   -5  
c    3  
  
In [15]: obj2[obj2 > 0]           In [16]: obj2 * 2            In [17]: np.exp(obj2)  
Out[15]:                           Out[16]:                           Out[17]:  
d    6                            d    12                            d    403.428793  
b    7                            b    14                            b    1096.633158  
c    3                            a   -10                            a    0.006738  
                                c     6                            c    20.085537
```

还可以将Series看成是一个定长的有序字典，因为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：

```
In [18]: 'b' in obj2  
Out[18]: True  
  
In [19]: 'e' in obj2  
Out[19]: False
```

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：

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}  
  
In [21]: obj3 = Series(sdata)  
  
In [22]: obj3  
Out[22]:  
Ohio      35000  
Oregon    16000
```



```
Texas      71000  
Utah       5000
```

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）。

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
In [24]: obj4 = Series(sdata, index=states)
```

```
In [25]: obj4  
Out[25]:  
California      NaN  
Ohio            35000  
Oregon          16000  
Texas           71000
```

在这个例子中，sdata中跟states索引相匹配的那3个值会被找出来并放到相应的位置上，但由于“California”所对应的sdata值找不到，所以其结果就为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值）。我将使用缺失（missing）或NA表示缺失数据。pandas的isnull和notnull函数可用于检测缺失数据：

```
In [26]: pd.isnull(obj4)      In [27]: pd.notnull(obj4)  
Out[26]:  
Out[27]:  
California    True      California    False  
Ohio          False     Ohio          True  
Oregon        False     Oregon        True  
Texas         False     Texas         True
```

Series也有类似的实例方法：

```
In [28]: obj4.isnull()  
Out[28]:  
California    True  
Ohio          False  
Oregon        False  
Texas         False
```

我将在本章详细讲解如何处理缺失数据。

对于许多应用而言，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。

```
In [29]: obj3      In [30]: obj4  
Out[29]:  
Out[30]:  
Ohio      35000    California      NaN  
Oregon    16000    Ohio          35000  
Texas     71000    Oregon         16000  
Utah      5000     Texas          71000
```

```
In [31]: obj3 + obj4  
Out[31]:
```



```
California      NaN
Ohio          70000
Oregon        32000
Texas         142000
Utah          NaN
```

数据对齐功能将在一个单独的主题中讲解。

Series对象本身及其索引都有一个name属性，该属性跟pandas其他的关键功能关系非常密切：

```
In [32]: obj4.name = 'population'
In [33]: obj4.index.name = 'state'
In [34]: obj4
Out[34]:
state
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
Name: population
```

Series的索引可以通过赋值的方式就地修改：

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff   -5
Ryan     3
```

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，它可以被看做由Series组成的字典（共用同一个索引）。跟其他类似的数据结构相比（如R的data.frame），DataFrame中面向行和面向列的操作基本上是平衡的。其实，DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或别的一维数据结构）。有关DataFrame内部的技术细节远远超出了本书所讨论的范围。

注意： 虽然DataFrame是以二维结构保存数据的，但你仍然可以轻松地将其表示为更高维度的数据（层次化索引的表格型结构，这是pandas中许多高级数据处理功能的关键要素，我们稍后再来讨论这个问题）。



构建DataFrame的办法有很多，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 = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

结果DataFrame会自动加上索引（跟Series一样），且全部列会被有序排列：

```
In [38]: frame
Out[38]:
   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
```

如果指定了列序列，则DataFrame的列就会按照指定顺序进行排列：

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
   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
```

跟Series一样，如果传入的列在数据中找不到，就会产生NA值：

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                           ...:                  index=['one', 'two', 'three', 'four', 'five'])
In [41]: frame2
Out[41]:
      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 [42]: frame2.columns
Out[42]: Index(['year', 'state', 'pop', 'debt'], dtype=object)
```

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 [43]: frame2['state']          In [44]: frame2.year
Out[43]:                         Out[44]:
one           Ohio                  one       2000
```



```
two      Ohio          two    2001
three    Ohio          three   2002
four     Nevada        four    2001
five     Nevada        five    2002
Name: state         Name: year
```

注意，返回的Series拥有原DataFrame相同的索引，且其name属性也已经被相应地设置了。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（稍后将对此进行详细讲解）：

```
In [45]: frame2.ix['three']
Out[45]:
year    2002
state   Ohio
pop     3.6
debt    NaN
Name: three
```

列可以通过赋值的方式进行修改。例如，我们可以给那个空的“debt”列赋上一个标量值或一组值：

```
In [46]: frame2['debt'] = 16.5

In [47]: frame2
Out[47]:
       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

In [48]: frame2['debt'] = np.arange(5.)

In [49]: frame2
Out[49]:
       year  state  pop  debt
one    2000   Ohio  1.5      0
two    2001   Ohio  1.7      1
three  2002   Ohio  3.6      2
four   2001  Nevada 2.4      3
five   2002  Nevada 2.9      4
```

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

In [51]: frame2['debt'] = val
In [52]: frame2
Out[52]:
```



```
      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
```

为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 [53]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [54]: frame2
```

```
Out[54]:
```

```
      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 [55]: del frame2['eastern']
```

```
In [56]: frame2.columns
```

```
Out[56]: Index([year, state, pop, debt], dtype=object)
```

警告：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。

另一种常见的数据形式是嵌套字典（也就是字典的字典）：

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
.....: 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

如果将它传给DataFrame，它就会被解释为：外层字典的键作为列，内层键则作为行索引：

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3
```

```
Out[59]:
```

```
      Nevada  Ohio
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

当然，你也可以对该结果进行转置：

```
In [60]: frame3.T
```

```
Out[60]:
```



```
      2000  2001  2002  
Nevada   NaN   2.4   2.9  
Ohio     1.5   1.7   3.6
```

内层字典的键会被合并、排序以形成最终的索引。如果显式指定了索引，则不会这样：

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])  
Out[61]:  
      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 [62]: pdata = {'Ohio': frame3['Ohio'][::-1],  
.....:           'Nevada': frame3['Nevada'][:2]}  
  
In [63]: DataFrame(pdata)  
Out[63]:  
      Nevada  Ohio  
2000      NaN   1.5  
2001      2.4   1.7
```

表5-1列出了DataFrame构造函数所能接受的各种数据。

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



如果设置了DataFrame的index和columns的name属性，则这些信息也会被显示出来：

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
   state  Nevada  Ohio
   year
2000      NaN    1.5
2001      2.4    1.7
2002      2.9    3.6
```

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

```
In [66]: frame3.values
Out[66]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

如果DataFrame各列的数据类型不同，则值数组的数据类型就会选用能兼容所有列的数据类型：

```
In [67]: frame2.values
Out[67]:
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, Nevada, 2.9, -1.7]], dtype=object)
```

索引对象

pandas的索引对象负责管理轴标签和其他元数据（比如轴名称等）。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：

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])

In [69]: index = obj.index

In [70]: index
Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Index对象是不可修改的（immutable），因此用户不能对其进行修改：

```
In [72]: index[1] = 'd'
-----
```



```

Exception                                Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'

/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302     def __setitem__(self, key, value):
    303         """Disable the setting of values."""
--> 304         raise Exception(str(self.__class__) + ' object is immutable')
    305
    306     def __getitem__(self, key):

Exception: <class 'pandas.core.index.Index'> object is immutable

```

不可修改性非常重要，因为这样才能使Index对象在多个数据结构之间安全共享：

```

In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True

```

表5-2列出了pandas库中内置的Index类。由于开发人员的不懈努力，Index甚至可以被继承从而实现特别的轴索引功能。

注意： 虽然大部分用户都不需要知道太多关于Index对象的细节，但它们确实是pandas数据模型的重要组成部分。

表5-2：pandas中主要的Index对象

类	说明
Index	最泛化的Index对象，将轴标签表示为一个由Python对象组成的NumPy数组
Int64Index	针对整数的特殊Index
MultiIndex	“层次化”索引对象，表示单个轴上的多层索引。可以看做由元组组成的数组
DatetimeIndex	存储纳秒级时间戳（用NumPy的datetime64类型表示）
PeriodIndex	针对Period数据（时间间隔）的特殊Index

除了长得像数组，Index的功能也类似一个固定大小的集合：

```

In [76]: frame3
Out[76]:
state  Nevada  Ohio
year
2000      NaN   1.5

```



```
2001      2.4  1.7  
2002      2.9  3.6
```

```
In [77]: 'Ohio' in frame3.columns  
Out[77]: True
```

```
In [78]: 2003 in frame3.index  
Out[78]: False
```

每个索引都有一些方法和属性，它们可用于设置逻辑并回答有关该索引所包含的数据的常见问题。表5-3列出了这些函数。

表5-3：Index的方法和属性

方法	说明
append	连接另一个Index对象，产生一个新的Index
diff	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
isin	计算一个指示各值是否都包含在参数集合中的布尔型数组
delete	删除索引i处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引i处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is_unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

基本功能

本节中，我将介绍操作Series和DataFrame中的数据的基本手段。后续章节将更加深入地挖掘pandas在数据分析和处理方面的功能。本书不是pandas库的详尽文档，主要关注的是最重要的功能，那些不大常用的内容（也就是那些更深奥的内容）就交给你自己去摸索吧。

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。以之前的一个简单示例来说：

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
```



```
In [80]: obj  
Out[80]:  
d    4.5  
b    7.2  
a   -5.3  
c    3.6
```

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

```
In [82]: obj2  
Out[82]:  
a   -5.3  
b    7.2  
c    3.6  
d    4.5  
e    NaN
```

```
In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
```

```
Out[83]:  
a   -5.3  
b    7.2  
c    3.6  
d    4.5  
e    0.0
```

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，例如，使用ffill可以实现前向值填充：

```
In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

```
In [85]: obj3.reindex(range(6), method='ffill')  
Out[85]:  
0    blue  
1    blue  
2  purple  
3  purple  
4  yellow  
5  yellow
```

表5-4列出了可用的method选项。其实我们有时需要比前向和后向填充更为精准的插值方式。

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。如果仅传入一个序列，则会重新索引行：

```
In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
...:                         columns=['Ohio', 'Texas', 'California'])
```

```
In [87]: frame
Out[87]:
   Ohio  Texas  California
a      0      1          2
c      3      4          5
d      6      7          8
```

```
In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

```
In [89]: frame2
Out[89]:
   Ohio  Texas  California
a      0      1          2
b     NaN     NaN          NaN
c      3      4          5
d      6      7          8
```

使用columns关键字即可重新索引列：

```
In [90]: states = ['Texas', 'Utah', 'California']
```

```
In [91]: frame.reindex(columns=states)
Out[91]:
   Texas  Utah  California
a      1    NaN          2
c      4    NaN          5
d      7    NaN          8
```

也可以同时对行和列进行重新索引，而插值则只能按行应用（即轴0）：

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
...:                      columns=states)
Out[92]:
   Texas  Utah  California
a      1    NaN          2
b      1    NaN          2
c      4    NaN          5
d      7    NaN          8
```

利用ix的标签索引功能，重新索引任务可以变得更简洁：

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
Out[93]:
   Texas  Utah  California
a      1    NaN          2
b     NaN     NaN          NaN
c      4    NaN          5
d      7    NaN          8
```



表5-5列出了reindex函数的各参数及说明。

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

丢弃某条轴上的一个或多个项很简单，只要有一个索引数组或列表即可。由于需要执行一些数据整理和集合逻辑，所以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 [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [95]: new_obj = obj.drop('c')
```

```
In [96]: new_obj
```

```
Out[96]:
```

```
a    0  
b    1  
d    3  
e    4
```

```
In [97]: obj.drop(['d', 'c'])
```

```
Out[97]:
```

```
a    0  
b    1  
e    4
```

对于DataFrame，可以删除任意轴上的索引值：

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),  
...:                         index=['Ohio', 'Colorado', 'Utah', 'New York'],  
...:                         columns=['one', 'two', 'three', 'four'])
```

```
In [99]: data.drop(['Colorado', 'Ohio'])
```

```
Out[99]:
```

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



```
In [100]: data.drop('two', axis=1)      In [101]: data.drop(['two', 'four'], axis=1)
Out[100]:                                Out[101]:
          one  three  four                  one  three
Ohio      0       2     3                 Ohio      0       2
Colorado  4       6     7                 Colorado  4       6
Utah      8      10    11                Utah      8      10
New York 12      14    15               New York 12      14
```

索引、选取和过滤

Series索引（`obj[...]`）的工作方式类似于NumPy数组的索引，只不过Series的索引值不只是整数。下面是几个例子：

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [103]: obj['b']                      In [104]: obj[1]
Out[103]: 1.0                           Out[104]: 1.0

In [105]: obj[2:4]                      In [106]: obj[['b', 'a', 'd']]
Out[105]:                               Out[106]:
          b   1
          a   0
          d   3

In [107]: obj[[1, 3]]                   In [108]: obj[obj < 2]
Out[107]:                               Out[108]:
          b   1
          d   3                           a   0
                                         b   1
```

利用标签的切片运算与普通的Python切片运算不同，其末端是包含的（inclusive）^{译注1}：

```
In [109]: obj['b':'c']
Out[109]:
          b   1
          c   2
```

设置的方式也很简单：

```
In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
          a   0
          b   5
          c   5
          d   3
```

如你所见，对DataFrame进行索引其实就是获取一个或多个列：

译注1：即封闭区间。



```
In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
...:                         index=['Ohio', 'Colorado', 'Utah', 'New York'],
...:                         columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
   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 [114]: data['two']
Out[114]:
   Ohio      1
   Colorado  5
   Utah      9
   New York 13
Name: two

In [115]: data[['three', 'one']]
Out[115]:
   three  one
   Ohio      2    0
   Colorado  6    4
   Utah      10   8
   New York 14   12
```

这种索引方式有几个特殊的情况。首先通过切片或布尔型数组选取行：

```
In [116]: data[:2]
Out[116]:
   one  two  three  four
   Ohio     0    1      2    3
   Colorado  4    5      6    7

In [117]: data[data['three'] > 5]
Out[117]:
   one  two  three  four
   Colorado  4    5      6    7
   Utah      8    9      10   11
   New York 12   13      14   15
```

有些读者可能会认为这不太合乎逻辑，但这种语法的的确确来源于实践。另一种用法是通过布尔型DataFrame（比如下面这个由标量比较运算得出的）进行索引：

```
In [118]: data < 5
Out[118]:
   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 [119]: data[data < 5] = 0

In [120]: data
Out[120]:
   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
```

这段代码的目的是使DataFrame在语法上更像ndarray。



为了在DataFrame的行上进行标签索引，我引入了专门的索引字段ix。它使你可以通过NumPy式的标记法以及轴标签从DataFrame中选取行和列的子集。之前曾提到过，这也是一种重新索引的简单手段：

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
two    5
three   6
Name: Colorado

In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
      four  one  two
Colorado    7    0    5
Utah       11   8    9

In [123]: data.ix[2]
Out[123]:
one    8
two    9
three  10
four   11
Name: Utah

In [124]: data.ix[:'Utah', 'two']
Out[124]:
Ohio    0
Colorado    5
Utah     9
Name: two

In [125]: data.ix[data.three > 5, :3]
Out[125]:
      one  two  three
Colorado    0    5    6
Utah       8    9   10
New York  12   13   14
```

也就是说，对pandas对象中的数据的选取和重排方式有很多。表5-6简单总结了针对DataFrame数据的大部分选取和重排方式。在使用层次化索引时还能用到一些别的办法（稍后就会讲到）。

注意：在设计pandas时，我觉得必须输入frame[:, col]才能选取列实在有些啰嗦（而且还很容易出错），因为列的选取是一种最常见的操作。于是，我就把所有的标签索引功能都放到ix中了。

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



表5-6：DataFrame的索引选项（续）

类型	说明
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。 <small>译注2</small>

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。我们来看一个简单的例子：

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])  
In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])  
  
In [128]: s1  
Out[128]:  
a    7.3  
c   -2.5  
d    3.4  
e    1.5  
  
In [129]: s2  
Out[129]:  
a   -2.1  
c    3.6  
e   -1.5  
f    4.0  
g    3.1
```

将它们相加就会产生：

```
In [130]: s1 + s2  
Out[130]:  
a    5.2  
c    1.1  
d    NaN  
e    0.0  
f    NaN  
g    NaN
```

自动的数据对齐操作在不重叠的索引处引入了NA值译注3。缺失值会在算术运算过程中传播。

译注2：get_value方法是选取，set-value方法是设置。

译注3：由于本书中多次出现“非重叠”（overlapping）这个词，所以需要简单说明一下。例如，“飞机场”跟“拖拉机”都有个“机”，于是可以认为这两个字符串是“重叠”的；“高富帅”和“矮穷挫”的情况自然就是“非重叠”了。注意，虽然这里没有任何顺序和连续的概念，但有些地方是需要考虑顺序和连续的。



对于DataFrame，对齐操作会同时发生在行和列上：

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),  
...: index=['Ohio', 'Texas', 'Colorado'])
```

```
In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),  
...: index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

	In [131]: df1	In [132]: df2
Out[133]:	b c d	b d e
Ohio	0 1 2	Utah 0 1 2
Texas	3 4 5	Ohio 3 4 5
Colorado	6 7 8	Texas 6 7 8
		Oregon 9 10 11

把它们相加后将会返回一个新的DataFrame，其索引和列为原来那两个DataFrame的并集：

```
In [135]: df1 + df2  
Out[135]:  
          b   c   d   e  
Colorado NaN NaN NaN NaN  
Ohio      3 NaN   6 NaN  
Oregon    NaN NaN NaN NaN  
Texas     9 NaN  12 NaN  
Utah     NaN NaN NaN NaN
```

在算术方法中填充值

在对不同索引的对象进行算术运算时，你可能希望当一个对象中某个轴标签在另一个对象中找不到时填充一个特殊值（比如0）：

```
In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
```

```
In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
```

	In [136]: df1	In [137]: df2
Out[136]:	a b c d	a b c d e
0 0 1 2 3		0 0 1 2 3 4
1 4 5 6 7		1 5 6 7 8 9
2 8 9 10 11		2 10 11 12 13 14
		3 15 16 17 18 19

将它们相加时，没有重叠的位置就会产生NA值：

```
In [140]: df1 + df2  
Out[140]:  
          a   b   c   d   e  
0      0   2   4   6  NaN
```



```
1   9   11  13  15  NaN  
2  18   20  22  24  NaN  
3  NaN  NaN  NaN  NaN  NaN
```

使用df1的add方法，传入df2以及一个fill_value参数：

```
In [141]: df1.add(df2, fill_value=0)  
Out[141]:  
     a    b    c    d    e  
0    0    2    4    6    4  
1    9   11   13   15    9  
2   18   20   22   24   14  
3   15   16   17   18   19
```

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

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)  
Out[142]:  
     a    b    c    d    e  
0    0    1    2    3    0  
1    4    5    6    7    0  
2    8    9   10   11    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

DataFrame和Series之间的运算

跟NumPy数组一样，DataFrame和Series之间算术运算也是有明确规定。先来看一个具有启发性的例子，计算一个二维数组与其某行之间的差：

```
In [143]: arr = np.arange(12.).reshape((3, 4))  
  
In [144]: arr  
Out[144]:  
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.]])  
  
In [145]: arr[0]  
Out[145]: array([ 0.,  1.,  2.,  3.])  
  
In [146]: arr - arr[0]  
Out[146]:
```



```
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

这就叫做广播（broadcasting），第12章将对此进行详细讲解。DataFrame和Series之间的运算差不多也是如此：

```
In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
...:                           index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [148]: series = frame.ix[0]

In [149]: frame
Out[149]:
      b   d   e
Utah  0   1   2
Ohio   3   4   5
Texas  6   7   8
Oregon 9  10  11

In [150]: series
Out[150]:
      b
      0
      d
      1
      e
      2
Name: Utah
```

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

```
In [151]: frame - series
Out[151]:
      b   d   e
Utah  0   0   0
Ohio   3   3   3
Texas  6   6   6
Oregon 9   9   9
```

如果某个索引值在DataFrame的列或Series的索引中找不到，则参与运算的两个对象就会被重新索引以形成并集：

```
In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
      b   d   e   f
Utah  0  NaN  3  NaN
Ohio   3  NaN  6  NaN
Texas  6  NaN  9  NaN
Oregon 9  NaN 12  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。例如：

```
In [154]: series3 = frame['d']

In [155]: frame
Out[155]:
      b   d   e
Utah  0   1   2
Ohio   3   4   5
Texas  6   7   8
Oregon 9  10  11

In [156]: series3
Out[156]:
```



```
b   d   e          Utah      1  
Utah 0   1   2    Ohio       4  
Ohio 3   4   5    Texas      7  
Texas 6   7   8   Oregon     10  
Oregon 9  10  11  Name: d
```

```
In [157]: frame.sub(series3, axis=0)  
Out[157]:  
b   d   e  
Utah -1  0   1  
Ohio  -1  0   1  
Texas -1  0   1  
Oregon -1  0   1
```

传入的轴号就是希望匹配的轴。在本例中，我们的目的是匹配DataFrame的行索引并进行广播。译注4

函数应用和映射

NumPy的ufuncs（元素级数组方法）也可用于操作pandas对象：

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),  
...:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])  
  
In [159]: frame  
Out[159]:  
b           d           e  
Utah -0.204708  0.478943 -0.519439  
Ohio -0.555730  1.965781  1.393406  
Texas  0.092908  0.281746  0.769023  
Oregon 1.246435  1.007189 -1.296221  
  
In [160]: np.abs(frame)  
Out[160]:  
b           d           e  
Utah  0.204708  0.478943  0.519439  
Ohio  0.555730  1.965781  1.393406  
Texas 0.092908  0.281746  0.769023  
Oregon 1.246435  1.007189  1.296221
```

另一个常见的操作是，将函数应用到由各列或行所形成的一维数组上。DataFrame的apply方法即可实现此功能：

```
In [161]: f = lambda x: x.max() - x.min()  
  
In [162]: frame.apply(f)  
Out[162]:  
b    1.802165  
d    1.684034  
e    2.689627  
  
In [163]: frame.apply(f, axis=1)  
Out[163]:  
Utah  0.998382  
Ohio  2.521511  
Texas 0.676115  
Oregon 2.542656
```

译注4：这里需要补充说明一下，作者反复强调“广播”会在第12章介绍，所以如果真看不懂这里就等到12章学完再看不迟。译者已经尽量把原文扩展的描述扩展开，但是文字描述始终没有图形更具体。例如，你可以打开一个Excel，随意找一排单元格并输入一些文字（注意是一排），然后选中这些单元格，将鼠标移至选区右下角，当指针变为加号时，按住向下拉几行，这就是“沿行向下广播”。



许多最为常见的数组统计功能都被实现成DataFrame的方法（如sum和mean），因此无需使用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 [164]: def f(x):
...:     return Series([x.min(), x.max()], index=['min', 'max'])

In [165]: frame.apply(f)
          b      d      e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

此外，元素级的Python函数也是可以用的。假如你想得到frame中各个浮点值的格式化字符串，使用applymap即可：

```
In [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
          b      d      e
Utah    -0.20   0.48  -0.52
Ohio    -0.56   1.97   1.39
Texas    0.09   0.28   0.77
Oregon   1.25   1.01  -1.30
```

之所以叫做applymap，是因为Series有一个用于应用元素级函数的map方法：

```
In [168]: frame['e'].map(format)
Out[168]:
Utah    -0.52
Ohio    1.39
Texas   0.77
Oregon  -1.30
Name: e
```

排序和排名

根据条件对数据集排序（sorting）也是一种重要的内置运算。要对行或列索引进行排序（按字典顺序），可使用sort_index方法，它将返回一个已排序的新对象：

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
a    1
b    2
c    3
d    0
```



而对于DataFrame，则可以根据任意一个轴上的索引进行排序：

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
...:                           columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index()
Out[172]:
   d  a  b  c
one  4  5  6  7
three 0  1  2  3

In [173]: frame.sort_index(axis=1)
Out[173]:
   a  b  c  d
three  1  2  3  0
one     5  6  7  4
```

数据默认是按升序排序的，但也可以降序排序：

```
In [174]: frame.sort_index(axis=1, ascending=False)
Out[174]:
   d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

若要按值对Series进行排序，可使用其order方法：

```
In [175]: obj = Series([4, 7, -3, 2])

In [176]: obj.order()
Out[176]:
2   -3
3    2
0    4
1    7
```

在排序时，任何缺失值默认都会被放到Series的末尾：

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [178]: obj.order()
Out[178]:
4   -3
5    2
0    4
2    7
1   NaN
3   NaN
```

在DataFrame上，你可能希望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：

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [180]: frame
Out[180]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [181]: frame.sort_index(by='b')
Out[181]:
   a  b
2  0 -3
0  0  4
1  1  7
3  1  2
```



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

要根据多个列进行排序，传入名称的列表即可：

```
In [182]: frame.sort_index(by=['a', 'b'])  
Out[182]:  
   a   b  
2  0 -3  
0  0  4  
3  1  2  
1  1  7
```

排名（ranking）跟排序关系密切，且它会增设一个排名值（从1开始，一直到数组中有有效数据的数量）。它跟numpy.argsort产生的间接排序索引差不多，只不过它可以根据某种规则破坏平级关系。接下来介绍Series和DataFrame的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 [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])  
  
In [184]: obj.rank()  
Out[184]:  
0    6.5  
1    1.0  
2    6.5  
3    4.5  
4    3.0  
5    2.0  
6    4.5
```

也可以根据值在原数据中出现的顺序给出排名^{译注5}：

```
In [185]: obj.rank(method='first')  
Out[185]:  
0    6  
1    1  
2    7  
3    4  
4    3  
5    2  
6    5
```

当然，你也可以按降序进行排名：

```
In [186]: obj.rank(ascending=False, method='max')  
Out[186]:  
0    2  
1    7
```

译注5：类似于稳定排序。



```
2    2  
3    4  
4    5  
5    6  
6    4
```

表5-8列出了所有用于破坏平级关系的method选项。DataFrame可以在行或列上计算排名：

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],  
...:                 'c': [-2, 5, 8, -2.5]})  
  
In [188]: frame  
Out[188]:  
   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 [189]: frame.rank(axis=1)  
Out[189]:  
   a   b   c  
0  2  3  1  
1  1  3  2  
2  2  1  3  
3  2  3  1
```

表5-8：排名时用于破坏平级关系的method选项

method	说明
'average'	默认：在相等分组中，为各个值分配平均排名
'min'	使用整个分组的最小排名
'max'	使用整个分组的最大排名
'first'	按值在原始数据中的出现顺序分配排名

带有重复值的轴索引

直到目前为止，我所介绍的所有范例都有着唯一的轴标签（索引值）。虽然许多pandas函数（如reindex）都要求标签唯一，但这并不是强制性的。我们来看看下面这个简单的带有重复索引值的Series：

```
In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])  
  
In [191]: obj  
Out[191]:  
a    0  
a    1  
b    2  
b    3  
c    4
```

索引的is_unique属性可以告诉你它的值是否是唯一的：

```
In [192]: obj.index.is_unique  
Out[192]: False
```



对于带有重复值的索引，数据选取的行为将会有些不同。如果某个索引对应多个值，则返回一个Series；而对应单个值的，则返回一个标量值。

```
In [193]: obj['a']          In [194]: obj['c']
Out[193]: 
a    0
a    1
Out[194]: 4
```

对DataFrame的行进行索引时也是如此：

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
In [196]: df
Out[196]:
   0      1      2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [197]: df.ix['b']
Out[197]:
   0      1      2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

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。它们大部分都属于约简和汇总统计，用于从Series中提取单个值（如sum或mean）或从DataFrame的行或列中提取一个Series。跟对应的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。接下来看一个简单的DataFrame：

```
In [198]: 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 [199]: df
Out[199]:
   one  two
a  1.40  NaN
b  7.10 -4.5
c  NaN   NaN
d  0.75 -1.3
```

调用DataFrame的sum方法将会返回一个含有列小计的Series：

```
In [200]: df.sum()
Out[200]:
```



```
one    9.25  
two   -5.80
```

传入axis=1将会按行进行求和运算：

```
In [201]: df.sum(axis=1)  
Out[201]:  
a    1.40  
b    2.60  
c    NaN  
d   -0.55
```

NA值会自动被排除，除非整个切片（这里指的是行或列）都是NA。通过skipna选项可以禁用该功能：

```
In [202]: df.mean(axis=1, skipna=False)  
Out[202]:  
a      NaN  
b    1.300  
c      NaN  
d   -0.275
```

表5-9列出了这些约简方法的常用选项。

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

有些方法（如idxmin和idxmax）返回的是间接统计（比如达到最小值或最大值的索引）：

```
In [203]: df.idxmax()  
Out[203]:  
one    b  
two    d
```

另一些方法则是累计型的：

```
In [204]: df.cumsum()  
Out[204]:  
      one  two  
a  1.40  NaN  
b  8.50 -4.5  
c  NaN   NaN  
d  9.25 -5.8
```



还有一种方法，它既不是约简型也不是累计型。`describe`就是一个例子，它用于一次性产生多个汇总统计：

```
In [205]: df.describe()
Out[205]:
      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 [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)
In [207]: obj.describe()
Out[207]:
count    16
unique     3
top       a
freq      8
```

表5-10列出了所有与描述统计相关的方法。

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

有些汇总统计（如相关系数和协方差）是通过参数对计算出来的。我们来看几个 DataFrame，它们的数据来自 Yahoo! Finance 的股票价格和成交量：

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
                   for tic, data in all_data.iteritems()})
volume = DataFrame({tic: data['Volume']
                     for tic, data in all_data.iteritems()})
```

接下来计算价格的百分数变化：

```
In [209]: returns = price.pct_change()

In [210]: returns.tail()
Out[210]:
          AAPL      GOOG      IBM      MSFT
Date
2009-12-24  0.034339  0.011117  0.004420  0.002747
2009-12-28  0.012294  0.007098  0.013282  0.005479
2009-12-29 -0.011861 -0.005571 -0.003474  0.006812
2009-12-30  0.012147  0.005376  0.005468 -0.013532
2009-12-31 -0.004300 -0.004416 -0.012609 -0.015432
```

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：

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838
```



```
In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

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：

```
In [213]: returns.corr()
Out[213]:
          AAPL      GOOG      IBM      MSFT
AAPL  1.000000  0.470660  0.410648  0.424550
GOOG  0.470660  1.000000  0.390692  0.443334
IBM   0.410648  0.390692  1.000000  0.496093
MSFT  0.424550  0.443334  0.496093  1.000000
```

```
In [214]: returns.cov()
Out[214]:
          AAPL      GOOG      IBM      MSFT
AAPL  0.001028  0.000303  0.000252  0.000309
GOOG  0.000303  0.000580  0.000142  0.000205
IBM   0.000252  0.000142  0.000367  0.000216
MSFT  0.000309  0.000205  0.000216  0.000516
```

利用DataFrame的corrwith方法，你可以计算其列或行跟另一个Series或DataFrame之间的相关系数。传入一个Series将会返回一个相关系数值Series（针对各列进行计算）：

```
In [215]: returns.corrwith(returns.IBM)
Out[215]:
AAPL    0.410648
GOOG    0.390692
IBM     1.000000
MSFT    0.496093
```

传入一个DataFrame则会计算按列名配对的相关系数。这里，我计算百分比变化与成交量的相关系数：

```
In [216]: returns.corrwith(volume)
Out[216]:
AAPL    -0.057461
GOOG    0.062644
IBM     -0.007900
MSFT    -0.014175
```

传入axis=1即可按行进行计算。无论如何，在计算相关系数之前，所有的数据项都会按标签对齐。

唯一值、值计数以及成员资格

还有一类方法可以从一维Series的值中抽取信息。以下面这个Series为例：

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```



第一个函数是unique，它可以得到Series中的唯一值数组：

```
In [218]: uniques = obj.unique()  
In [219]: uniques  
Out[219]: array([c, a, d, b], dtype=object)
```

返回的唯一值是未排序的，如果需要的话，可以对结果再次进行排序（uniques.sort()）。value_counts用于计算一个Series中各值出现的频率：

```
In [220]: obj.value_counts()  
Out[220]:  
c    3  
a    3  
b    2  
d    1
```

为了便于查看，结果Series是按值频率降序排列的。value_counts还是一个顶级pandas方法，可用于任何数组或序列：

```
In [221]: pd.value_counts(obj.values, sort=False)  
Out[221]:  
a    3  
b    2  
c    3  
d    1
```

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

```
In [222]: mask = obj.isin(['b', 'c'])  
  
In [223]: mask      In [224]: obj[mask]  
Out[223]:          Out[224]:  
0   True           0   c  
1   False          5   b  
2   False          6   b  
3   False          7   c  
4   False          8   c  
5   True  
6   True  
7   True  
8   True
```

表5-11给出了这几个方法的一些参考信息。



表5-11：唯一值、值计数、成员资格方法

方法	说明
isin	计算一个表示“Series各值是否包含于传入的值序列中”的布尔型数组
unique	计算Series中的唯一值数组，按发现的顺序返回
value_counts	返回一个Series，其索引为唯一值，其值为频率，按计数值降序排列

有时，你可能希望得到DataFrame中多个相关列的一张柱状图。例如：

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],  
...:                      'Qu2': [2, 3, 1, 2, 3],  
...:                      'Qu3': [1, 5, 2, 4, 4]})  
  
In [226]: data  
Out[226]:  
   Qu1  Qu2  Qu3  
0     1     2     1  
1     3     3     5  
2     4     1     2  
3     3     2     4  
4     4     3     4
```

将pandas.value_counts传给该DataFrame的apply函数，就会出现：

```
In [227]: result = data.apply(pd.value_counts).fillna(0)  
  
In [228]: result  
Out[228]:  
   Qu1  Qu2  Qu3  
1     1     1     1  
2     0     2     1  
3     2     2     0  
4     2     0     2  
5     0     0     1
```

处理缺失数据

缺失数据（missing data）在大部分数据分析应用中都很常见。pandas的设计目标之一就是让缺失数据的处理任务尽量轻松。例如，pandas对象上的所有描述统计都排除了缺失数据，正如我们在本章稍早的地方所看到的那样。

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：

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```



```
In [230]: string_data          In [231]: string_data.isnull()
Out[230]:
0    aardvark
1   artichoke
2      NaN
3   avocado
Out[231]:
0    False
1    False
2     True
3    False
```

Python内置的None值也会被当做NA处理：

```
In [232]: string_data[0] = None
In [233]: string_data.isnull()
Out[233]:
0    True
1   False
2    True
3   False
```

我不敢说pandas的NA表现形式是最优的，但它确实很简单也很可靠。由于NumPy的数据类型体系中缺乏真正的NA数据类型或位模式，所以它是能想到的最佳解决方案（一套简单的API以及足够全面的性能特征）。随着NumPy的不断发展，这个问题今后可能会发生变化。

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

滤除缺失数据

过滤掉缺失数据的办法有很多种。纯手工操作永远都是一个办法，但dropna可能会更实用一些。对于一个Series，dropna返回一个仅含非空数据和索引值的Series：

```
In [234]: from numpy import nan as NA
In [235]: data = Series([1, NA, 3.5, NA, 7])
In [236]: data.dropna()
Out[236]:
0    1.0
2    3.5
4    7.0
```



当然，也可以通过布尔型索引达到这个目的：

```
In [237]: data[data.notnull()]
Out[237]:
0    1.0
2    3.5
4    7.0
```

而对于DataFrame对象，事情就有点复杂了。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：

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
...:                      [NA, NA, NA], [NA, 6.5, 3.]])
In [239]: cleaned = data.dropna()

In [240]: data           In [241]: cleaned
Out[240]:              Out[241]:
   0   1   2           0   1   2
0   1   6.5  3         0   1   6.5  3
1   1   NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3
```

传入how='all'将只丢弃全为NA的那些行：

```
In [242]: data.dropna(how='all')
Out[242]:
   0   1   2
0   1   6.5  3
1   1   NaN  NaN
3  NaN  6.5  3
```

要用这种方式丢弃列，只需传入axis=1即可：

```
In [243]: data[4] = NA

In [244]: data           In [245]: data.dropna(axis=1, how='all')
Out[244]:              Out[245]:
   0   1   2   4           0   1   2
0   1   6.5  3  NaN       0   1   6.5  3
1   1   NaN  NaN  NaN     1   1   NaN  NaN
2  NaN  NaN  NaN  NaN     2  NaN  NaN  NaN
3  NaN  6.5  3  NaN       3  NaN  6.5  3
```

另一个滤除DataFrame行的问题涉及时间序列数据。假设你只想留下一部分观测数据，可以用thresh参数实现此目的：

```
In [246]: df = DataFrame(np.random.randn(7, 3))
In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```



```
In [248]: df
Out[248]:
      0      1      2
0 -0.577087    NaN    NaN
1  0.523772    NaN    NaN
2 -0.713544    NaN    NaN
3 -1.860761    NaN  0.560145
4 -1.265934    NaN -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

In [249]: df.dropna(thresh=3)
Out[249]:
      0      1      2
5  0.332883 -2.359419 -0.199543
6-1.541996 -0.970736 -1.307030
```

填充缺失数据

你可能不想滤除缺失数据（有可能会丢弃跟它有关的其他数据），而是希望通过其他方式填补那些“空洞”。对于大多数情况而言，`fillna`方法是最主要的函数。通过一个常数调用`fillna`就会将缺失值替换为那个常数值：

```
In [250]: df.fillna(0)
Out[250]:
      0      1      2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

若是通过一个字典调用`fillna`，就可以实现对不同的列填充不同的值：

```
In [251]: df.fillna({1: 0.5, 3: -1})
Out[251]:
      0      1      2
0 -0.577087  0.500000    NaN
1  0.523772  0.500000    NaN
2 -0.713544  0.500000    NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

`fillna`默认会返回新对象，但也可以对现有对象进行就地修改：

```
# 总是返回被填充对象的引用
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
Out[253]:
      0      1      2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
```



```
3 -1.860761 0.000000 0.560145  
4 -1.265934 0.000000 -1.063512  
5 0.332883 -2.359419 -0.199543  
6 -1.541996 -0.970736 -1.307030
```

对reindex有效的那些插值方法也可用于fillna:

```
In [254]: df = DataFrame(np.random.randn(6, 3))
```

```
In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA
```

```
In [256]: df
```

```
Out[256]:
```

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	NaN	1.004812
3	1.327195	NaN	-1.549106
4	0.022185	NaN	NaN
5	0.862580	NaN	NaN

```
In [257]: df.fillna(method='ffill')
```

```
Out[257]:
```

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	1.349742	-1.549106
5	0.862580	1.349742	-1.549106

```
In [258]: df.fillna(method='ffill', limit=2)
```

```
Out[258]:
```

	0	1	2
0	0.286350	0.377984	-0.753887
1	0.331286	1.349742	0.069877
2	0.246674	1.349742	1.004812
3	1.327195	1.349742	-1.549106
4	0.022185	NaN	-1.549106
5	0.862580	NaN	-1.549106

只要稍微动动脑子，你就可以利用fillna实现许多别的功能。比如说，你可以传入Series的平均值或中位数：

```
In [259]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [260]: data.fillna(data.mean())
```

```
Out[260]:
```

0	1.000000
1	3.833333
2	3.500000
3	3.833333
4	7.000000

表5-13列出了fillna的参数参考。

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（hierarchical indexing）是pandas的一项重要功能，它使你能在一轴上拥有多个（两个以上）索引级别。抽象点说，它使你能以低维度形式处理高维度数据。我们先来看一个简单的例子：创建一个Series，并用一个由列表或数组组成的列表作为索引。

```
In [261]: 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 [262]: data
Out[262]:
```

a	1	0.670216
	2	0.852965
	3	-0.955869
b	1	-0.023493
	2	-2.304234
	3	-0.652469
c	1	-1.218302
	2	-1.332610
d	2	1.074623
	3	0.723642

这就是带有MultiIndex索引的Series的格式化输出形式。索引之间的“间隔”表示“直接使用上面的标签”：

```
In [263]: data.index
Out[263]:
```

```
MultiIndex
[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1) ('c', 2) ('d', 2)
 ('d', 3)]
```

对于一个层次化索引的对象，选取数据子集的操作很简单：

```
In [264]: data['b']
Out[264]:
```

1	-0.023493
2	-2.304234
3	-0.652469



```
In [265]: data['b':'c']
Out[265]:
b 1 -0.023493
2 -2.304234
3 -0.652469
c 1 -1.218302
2 -1.332610

In [266]: data.ix[['b', 'd']]
Out[266]:
b 1 -0.023493
2 -2.304234
3 -0.652469
d 2 1.074623
3 0.723642
```

有时甚至还可以在“内层”中进行选取：

```
In [267]: data[:, 2]
Out[267]:
a 0.852965
b -2.304234
c -1.332610
d 1.074623
```

层次化索引在数据重塑和基于分组的操作（如透视表生成）中扮演着重要的角色。比如说，这段数据可以通过其`unstack`方法被重新安排到一个DataFrame中：

```
In [268]: data.unstack()
Out[268]:
      1      2      3
a  0.670216  0.852965 -0.955869
b -0.023493 -2.304234 -0.652469
c -1.218302 -1.332610      NaN
d      NaN   1.074623   0.723642
```

`unstack`的逆运算是`stack`：

```
In [269]: data.unstack().stack()
Out[269]:
a 1 0.670216
2 0.852965
3 -0.955869
b 1 -0.023493
2 -2.304234
3 -0.652469
c 1 -1.218302
2 -1.332610
d 2 1.074623
3 0.723642
```

`stack`和`unstack`将在第7章中详细讲解。

对于一个DataFrame，每条轴都可以有分层索引：

```
In [270]: 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 [271]: frame
Out[271]:
      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）。如果指定了名称，它们就会显示在控制台输出中（不要将索引名称跟轴标签混为一谈！）：

```
In [272]: frame.index.names = ['key1', 'key2']

In [273]: frame.columns.names = ['state', 'color']

In [274]: frame
Out[274]:
      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
```

由于有了分部的列索引，因此可以轻松选取列分组：

```
In [275]: frame['Ohio']
Out[275]:
      color    Green    Red
key1 key2
a 1      0      1
2      3      4
b 1      6      7
2      9     10
```

可以单独创建MultiIndex然后复用。上面那个DataFrame中的（分级的）列可以这样创建：

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                       names=['state', 'color'])
```

重排分级顺序

有时，你需要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，并返回一个互换了级别的新对象（但数据不会发生变化）：



```
In [276]: frame.swaplevel('key1', 'key2')
Out[276]:
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
```

而sortlevel则根据单个级别中的值对数据进行排序（稳定的）。交换级别时，常常也会用到sortlevel，这样最终结果就是有序的了：

```
In [277]: frame.sortlevel(1)
Out[277]:
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 [278]: frame.swaplevel(0, 1).sortlevel(0)
Out[278]:
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
```

注意：在层次化索引的对象上，如果索引是按字典方式从外到内排序（即调用sortlevel(0)或sort_index() 的结果），数据选取操作的性能要好很多。

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。再以上面那个DataFrame为例，我们可以根据行或列上的级别来进行求和，如下所示：

```
In [279]: frame.sum(level='key2')
Out[279]:
state      Ohio      Colorado
color     Green   Red     Green
key2
1        6       8      10
2       12      14      16
```

```
In [280]: frame.sum(level='color', axis=1)
Out[280]:
color      Green   Red
key1 key2
a    1        2       1
      2        8       4
b    1       14       7
      2       20      10
```

这其实是利用了pandas的groupby功能，本书稍后将对其进行详细讲解。



使用DataFrame的列

人们经常想要将DataFrame的一个或多个列当做行索引来用，或者可能希望将行索引变成DataFrame的列。以下面这个DataFrame为例：

```
In [281]: 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 [282]: frame
Out[282]:
   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
```

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

```
In [283]: frame2 = frame.set_index(['c', 'd'])

In [284]: frame2
Out[284]:
      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
```

默认情况下，那些列会从DataFrame中移除，但也可以将其保留下来：

```
In [285]: frame.set_index(['c', 'd'], drop=False)
Out[285]:
      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`的功能跟`set_index`刚好相反，层次化索引的级别会被转移到列里面：



```
In [286]: frame2.reset_index()
Out[286]:
   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
```

其他有关pandas的话题

这里是另外一些可能在你的数据旅程中用得着的有关pandas的话题。

整数索引

操作由整数索引的pandas对象常常会让新手抓狂，因为它们跟内置的Python数据结构（如列表和元组）在索引语义上有些不同。例如，你可能认为下面这段代码不会产生一个错误：

```
ser = Series(np.arange(3.))
ser[-1]
```

在这种情况下，虽然pandas会“求助于”整数索引，但没有哪种方法（至少我就不知道）能够既不引入任何bug又安全有效地解决该问题。这里，我们有一个含有0、1、2的索引，但是很难推断出用户想要什么（基于标签或位置的索引）：

```
In [288]: ser
Out[288]:
0    0
1    1
2    2
```

相反，对于一个非整数索引，就没有这样的歧义：

```
In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])
In [290]: ser2[-1]
Out[290]: 2.0
```

为了保持良好的一致性，如果你的轴索引含有索引器，那么根据整数进行数据选取的操作将总是面向标签的。这也包括用ix进行切片：

```
In [291]: ser.ix[:1]
Out[291]:
0    0
1    1
```



如果你需要可靠的、不考虑索引类型的、基于位置的索引，可以使用Series的`iget_value`方法和DataFrame的`irow`和`icol`方法：

```
In [292]: ser3 = Series(range(3), index=[-5, 1, 3])  
  
In [293]: ser3.iget_value(2)  
Out[293]: 2  
  
In [294]: frame = DataFrame(np.arange(6).reshape(3, 2), index=[2, 0, 1])  
In [295]: frame.irow(0)  
Out[295]:  
0    0  
1    1  
Name: 2
```

面板数据

pandas有一个Panel数据结构（不是本书的主要内容），你可以将其看做一个三维版的DataFrame。pandas的大部分开发工作都集中在表格型数据的操作上，因为这些数据更常见，而且层次化索引也使得多数情况下没必要使用真正的N维数组。

你可以用一个由DataFrame对象组成的字典或一个三维ndarray来创建Panel对象：

```
import pandas.io.data as web  
  
pdata = pd.Panel(dict((stk, web.get_data_yahoo(stk, '1/1/2009', '6/1/2012'))  
                     for stk in ['AAPL', 'GOOG', 'MSFT', 'DELL']))
```

Panel中的每一项（类似于DataFrame的列）都是一个DataFrame：

```
In [297]: pdata  
Out[297]:  
<class 'pandas.core.panel.Panel'>  
Dimensions: 4 (items) x 861 (major) x 6 (minor)  
Items: AAPL to MSFT  
Major axis: 2009-01-02 00:00:00 to 2012-06-01 00:00:00  
Minor axis: Open to Adj Close  
  
In [298]: pdata = pdata.swapaxes('items', 'minor')  
In [299]: pdata['Adj Close']  
Out[299]:  
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 861 entries, 2009-01-02 00:00:00 to 2012-06-01 00:00:00  
Data columns:  
AAPL    861 non-null values  
DELL    861 non-null values  
GOOG    861 non-null values  
MSFT    861 non-null values  
dtypes: float64(4)
```



基于ix的标签索引被推广到了三个维度，因此我们可以选取指定日期或日期范围的所有数据，如下所示：

```
In [300]: pdata.ix[:, '6/1/2012', :]  
Out[300]:  
      Open   High    Low  Close  Volume  Adj Close  
AAPL  569.16  572.65  560.52  560.99  18606700  560.99  
DELL   12.15   12.30   12.05   12.07  19396700   12.07  
GOOG  571.79  572.65  568.35  570.98  3057900  570.98  
MSFT   28.76   28.96   28.44   28.45  56634300   28.45
```

```
In [301]: pdata.ix['Adj Close', '5/22/2012':, :]  
Out[301]:  
          AAPL    DELL    GOOG    MSFT  
Date  
2012-05-22  556.97  15.08  600.80  29.76  
2012-05-23  570.56  12.49  609.46  29.11  
2012-05-24  565.32  12.45  603.66  29.07  
2012-05-25  562.29  12.46  591.53  29.06  
2012-05-29  572.27  12.66  594.34  29.56  
2012-05-30  579.17  12.56  588.23  29.34  
2012-05-31  577.73  12.33  580.86  29.19  
2012-06-01  560.99  12.07  570.98  28.45
```

另一个用于呈现面板数据（尤其是对拟合统计模型）的办法是“堆积式的” DataFrame 形式：

```
In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()  
  
In [303]: stacked  
Out[303]:  
      Open   High    Low  Close  Volume  Adj Close  
major   minor  
2012-05-30 AAPL  569.20  579.99  566.56  579.17  18908200  579.17  
          DELL   12.59   12.70   12.46   12.56  19787800   12.56  
          GOOG  588.16  591.90  583.53  588.23  1906700  588.23  
          MSFT   29.35   29.48   29.12   29.34  41585500   29.34  
2012-05-31 AAPL  580.74  581.50  571.46  577.73  17559800  577.73  
          DELL   12.53   12.54   12.33   12.33  19955500   12.33  
          GOOG  588.72  590.00  579.00  580.86  2968300  580.86  
          MSFT   29.30   29.42   28.94   29.19  39134000   29.19  
2012-06-01 AAPL  569.16  572.65  560.52  560.99  18606700  560.99  
          DELL   12.15   12.30   12.05   12.07  19396700   12.07  
          GOOG  571.79  572.65  568.35  570.98  3057900  570.98  
          MSFT   28.76   28.96   28.44   28.45  56634300   28.45
```

DataFrame有一个相应的to_panel方法，它是to_frame的逆运算：

```
In [304]: stacked.to_panel()  
Out[304]:  
<class 'pandas.core.panel.Panel'>  
Dimensions: 6 (items) x 3 (major) x 4 (minor)
```



Items: Open to Adj Close

Major axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00

Minor axis: AAPL to MSFT

