Hierarchical Indexing

层次化索引

> Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas `Series` and `DataFrame` objects, respectively. Often it is useful to go beyond this and store higher-dimensional data–that is, data indexed by more than one or two keys. While Pandas does provide `Panel` and `Panel4D` objects that natively handle three-dimensional and four-dimensional data (see Aside: Panel Data), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional `Series` and two-dimensional `DataFrame` objects.

直到目前为止，我们主要集中在一维和二维数据上，它们被存储在Pandas的`Series`和`DataFrame`对象当中。很多时候，我们需要超越二维来存储更高维度的数据，即用来检索的关键字会超过1个或2个。虽然Pandas提供了`Panel`和`Panel4D`对象（参见额外内容: Panel数据），但是我们在实践中更常用的方式是使用*层次化索引*（也被成为*多重索引*）来将多个索引*层次*在一个索引中结合起来。使用这种方法，高维数据也可以用紧凑的方式表示成我们熟悉的一维`Series`和二维`DataFrame`对象。

> In this section, we'll explore the direct creation of `MultiIndex` objects, considerations when indexing, slicing, and computing statistics across multiply indexed data, and useful routines for converting between simple and hierarchically indexed representations of your data.

在本节中，我们会讨论`多重索引`对象的直接创建方式，当我们在多重索引数据中进行索引、切片和统计的方式，还会介绍在简单索引和多重索引之间进行转换的方法。

> We begin with the standard imports:

首先还是先进行标准载入：

In [1]:
```python
import pandas as pd
import numpy as np
```

A Multiply Indexed Series

多重索引Series

> Let's start by considering how we might represent two-dimensional data within a one-dimensional `Series`. For concreteness, we will consider a series of data where each point has a character and numerical key.

我们从在一维`Series`中表示二维数据开始。我们考虑一个序列的数据，每个数据点都有一个字符串和数字关键字。

The bad way

不好的做法

> Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

设想你想追踪州人口两个不同年份的数据。使用我们已经学过的Pandas工具，你可能会想简单的使用Python元组来作为key：

In [2]:
```python
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

Out[2]:
```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

> With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

使用这种索引策略，你可以直接在series中对多个索引进行检索或切片：

In [3]:
```python
pop[('California', 2010):('Texas', 2000)]
```

Out[3]:
```
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
dtype: int64
```

> But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

但是这种便利性也就到此为止了。例如，如果你需要2010年的全部数据，就需要写一些没那么直观（且可能低性能的）的代码来实现了：

In [4]:
```python
pop[[i for i in pop.index if i[1] == 2010]]
```

Out[4]:
```
(California, 2010)    37253956
(New York, 2010)      19378102
(Texas, 2010)         25145561
dtype: int64
```

> This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

结果是正确的，但是对比起我们已经开始喜爱的Pandas切片语法来说，代码并没那么易读（或者在处理大数据集时低效）。

The Better Way: Pandas MultiIndex

更好的方法：Pandas多重索引

> Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

幸运的是，Pandas提供了一个更好的方法。刚才的那个元组索引的方式是一个初级的多重索引，Pandas `MultiIndex`类型提供了我们需要的真正的多重索引功能。我们可以按照下面的方式从元组创建一个多重索引：

In [5]:
```python
index = pd.MultiIndex.from_tuples(index)
index
```

Out[5]:
```
MultiIndex([('California', 2000),
            ('California', 2010),
            (  'New York', 2000),
            (  'New York', 2010),
            (     'Texas', 2000),
            (     'Texas', 2010)],
           )
```

> Notice that the `MultiIndex` contains multiple *levels* of indexing–in this case, the state names and the years, as well as multiple *labels* for each data point which encode these levels.

注意上面的`MultiIndex`对象包含多重*层级*的索引，本例中为州名和年份，同时也有多个编码*标签*对应着每个数据点。

> If we re-index our series with this `MultiIndex`, we see the hierarchical representation of the data:

如果我们使用这个`MultiIndex`对我们的series进行重新索引，我们可以看到这个数据集的层级展示：

In [6]:
```python
pop = pop.reindex(index)
pop
```

Out[6]:
```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

> Here the first two columns of the `Series` representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

上表中`Series`的头两列代表着多重索引的值，第三列代表数据值。第一列中有些行的数据缺失了，在多重索引展示中，缺失的索引值数据表示它与上一行具有相同的值。

> Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

现在想要获取第二个索引值为2010年的数据，我们只需要简单的使用Pandas的切片语法即可：

In [7]:
```python
pop[:, 2010]
```

Out[7]:
```
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

> The result is a singly indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We'll now further discuss this sort of indexing operation on hieararchically indexed data.

结果变成了一个单一索引的数组，且仅带有我们感兴趣的索引。这个语法显然比起我们前面使用元组作为多重索引的方案方便多了（当然性能也优异很多）。我们会深入讨论在层次化索引数据上进行操作的方法。

MultiIndex as extra dimension

多重索引作为额外维度

> You might notice something else here: we could easily have stored the same data using a simple `DataFrame` with index and column labels. In fact, Pandas is built with this equivalence in mind. The `unstack()` method will quickly convert a multiply indexed `Series` into a conventionally indexed `DataFrame`:

你可能已经注意到上例中，我们可以很简单的将数据存储在一个简单的`DataFrame`里面，州名作为行索引，年份作为列索引。实际上，Pandas已经内建了这种等同的机制。`unstack()`方法可以很快地将多重索引的`Series`转换成普通索引的`DataFrame`：

In [8]:
```python
pop_df = pop.unstack()
pop_df
```

Out[8]:

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

> Naturally, the `stack()` method provides the opposite operation:

自然而然的，`stack()`方法提供了相反的操作：

In [9]:
```python
pop_df.stack()
```

Out[9]:
```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

> Seeing this, you might wonder why would we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to represent two-dimensional data within a one-dimensional `Series`, we can also use it to represent data of three or more dimensions in a `Series` or `DataFrame`. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18) ; with a `MultiIndex` this is as easy as adding another column to the `DataFrame`:

看到这里，你可能会疑惑为什么我们需要使用层次化索引。原因很简单：就像我们可以使用多重索引来将一维`Series`表示成二维数据一样，我们也可以使用`Series`或`DataFrame`来表示三维或多维的数据。每个多重索引中的额外层次都代表着数据中额外的维度；利用这点我们可以灵活地详细地展示我们的数据，例如我们希望在上面各州各年人口数据的基础上增加一列（比方说18岁以下人口数）；使用`MultiIndex`能很简单的为`DataFrame`增加一列：

In [10]:
```python
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

Out[10]:

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

> In addition, all the ufuncs and other functionality discussed in Operating on Data in Pandas work with hierarchical indices as well. Here we compute the fraction of people under 18 by year, given the above data:

除此之外，所有在在Pandas中操作数据中介绍过的ufuncs和其他功能也可以应用到层次化索引数据上。下面我们计算18岁一下人口的比例：

In [11]:
```python
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

Out[11]:

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

> This allows us to easily and quickly manipulate and explore even high-dimensional data.

这允许我们能简单和迅速的操作数据，甚至是高维度的数据。

Methods of MultiIndex Creation

多重索引创建的方法

> The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

最直接的构建多重索引`Series`或`DataFrame`的方式是向index参数传递一个多重列表。例如：

In [12]:
```python
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
df
```

Out[12]:

		data1	data2
a	1	0.024362	0.784210
	2	0.383360	0.278085
b	1	0.679827	0.063426
	2	0.704108	0.689651

> The work of creating the `MultiIndex` is done in the background.

创建`MultiIndex`的工作会自动完成。

> Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

类似的，如果你使用元组作为关键字的字典数据传给Series，Pandas也会自动识别并默认使用`MultiIndex`：

In [13]:
```python
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

Out[13]:
```
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
New York    2000    18976457
            2010    19378102
dtype: int64
```

> Nevertheless, it is sometimes useful to explicitly create a `MultiIndex`; we'll see a couple of these methods here.

然而，有时候显式地创建`MultiIndex`对象也是很有用的；我们下面会看到一些这些方法。

Explicit MultiIndex constructors

显式 MultiIndex 构造器

> For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, as we did before, you can construct the `MultiIndex` from a simple list of arrays giving the index values within each level:

当你需要更灵活地构建多重索引时，你可以使用`pd.MultiIndex`的构造器。例如，你可以使用多重列表来构造一个和前面一样的`MultiIndex`对象：

In [14]:
```python
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
```

Out[14]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

> You can construct it from a list of tuples giving the multiple index values of each point:

你也可以使用一个元组的列表来构建一个多重索引：

In [15]:
```python
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

Out[15]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

> You can even construct it from a Cartesian product of single indices:

你还可以用两个单一索引的笛卡尔乘积来构造：

In [16]:
```python
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

Out[16]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

> Similarly, you can construct the `MultiIndex` directly using its internal encoding by passing `levels` (a list of lists containing available index values for each level) and `labels` (a list of lists that reference these labels):

同样，你可以用`MultiIndex`构造器来构造多重索引，你需要传递`levels`（多重列表包括每个层次的索引值）和`labels`（多重列表包括数据点的标签值）参数：

译者注：Pandas的`MultiIndex`构造器参数中labels后续可能被弃用，需要使用`codes`参数，下面代码进行了相应修改。

In [17]:
```python
pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
              codes=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Out[17]:
```
MultiIndex([('a', 1),
            ('a', 2),
            ('b', 1),
            ('b', 2)],
           )
```

> Any of these objects can be passed as the `index` argument when creating a `Series` or `Dataframe`, or be passed to the `reindex` method of an existing `Series` or `DataFrame`.

上面创建的这些对象都能作为`index`参数传递给`Series`或`DataFrame`构造器使用，或者作为`reindex`方法的参数提供给`Series`或`DataFrame`对象进行重新索引。

MultiIndex level names

MultiIndex 层次名称

> Sometimes it is convenient to name the levels of the `MultiIndex`. This can be accomplished by passing the `names` argument to any of the above `MultiIndex` constructors, or by setting the `names` attribute of the index after the fact:

为了方便有时需要给`MultiIndex`的不同层次进行命名。这可以通过在上面的`MultiIndex`构造方法中传递`names`参数，或者创建了之后通过设置`names`属性来实现：

In [18]:
```python
pop.index.names = ['state', 'year']
pop
```

Out[18]:
```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

> With more involved datasets, this can be a useful way to keep track of the meaning of various index values.

在复杂的数据集中，这种命名方式让不同的索引值保持它们原本的意义。

MultiIndex for columns

列的 MultiIndex

> In a `DataFrame`, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

在一个`DataFrame`中，行和列是完全对称的，就像前面看到的行可以有多层次的索引，列也可以有多层次的索引。看下面的例子，用来模拟真实的医疗数据：

In [19]:
```python
# 行和列的多重索引
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# 模拟一些真实数据
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# 创建DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

Out[19]:

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	35.0	35.9	21.0	37.5	37.0	38.2
	2	28.0	37.3	43.0	38.3	35.0	36.2
2014	1	51.0	36.3	33.0	39.0	29.0	35.8
	2	43.0	35.7	19.0	36.2	43.0	36.1

> Here we see where the multi-indexing for both rows and columns can come in *very* handy. This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full `DataFrame` containing just that person's information:

我们看到多重索引对于行和列来说都是非常方便的。上面的数据集实际上是一个四维的数据，四个维度分别是受试者、测试类型、年份和测试编号。创建了这个`DataFrame`之后，我们可以使用受试者的姓名来很方便的获取到此人的所有测试数据：

In [20]:
```python
health_data['Guido']
```

Out[20]:

	type	HR	Temp
year	visit		
2013	1	21.0	37.5
	2	43.0	38.3
2014	1	33.0	39.0
	2	19.0	36.2

> For complicated records containing multiple labeled measurements across multiple times for many subjects (people, countries, cities, etc.) use of hierarchical rows and columns can be extremely convenient!

对于这种包含着多重标签的多种维度（人、国家、城市等）数据。使用这种层次化的行和列的结构会非常方便。

Indexing and Slicing a MultiIndex

在 MultiIndex 上检索和切片

> Indexing and slicing on a `MultiIndex` is designed to be intuitive, and it helps if you think about the indices as added dimensions.

在`MultiIndex`上进行检索和切片设计的非常直观，你可以将其想象为在新增的维度上进行检索能帮助你理解。

> We'll first look at indexing multiply indexed `Series`, and then multiply-indexed `DataFrame`s.

我们先来看一下多重索引`Series`的方法，然后再看多重索引的`DataFrame`。

Multiply indexed Series

多重索引 Series

> Consider the multiply indexed `Series` of state populations we saw earlier:

回头再看看前面的那个人口的多重序列`Series`：

In [21]:
```python
pop
```

Out[21]:
```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

> We can access single elements by indexing with multiple terms:

我们可以使用多重索引值获取单个元素：

In [22]:
```python
pop['California', 2000]
```

Out[22]:
```
33871648
```

> The `MultiIndex` also supports *partial indexing*, or indexing just one of the levels in the index. The result is another `Series`, with the lower-level indices maintained:

`MultiIndex`同样支持*部分检索*，即仅在索引中检索其中的一个层次。得到的结果是另一个`Series`但是具有更少的层次结构：

In [23]:
```python
pop['California']
```

Out[23]:
```
year
2000    33871648
2010    37253956
dtype: int64
```

> Partial slicing is available as well, as long as the `MultiIndex` is sorted (see discussion in Sorted and Unsorted Indices):

部分切片同样也是支持的，只要`MultiIndex`是排序的（参见有序和无序的索引）：

In [24]:
```python
pop.loc['California':'New York']
```

Out[24]:
```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
dtype: int64
```

> With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

在有序索引的情况下，部分检索也可以用到低层次的索引上，只需要在第一个索引位置传递一个空的切片即可：

In [25]:
```python
pop[:, 2000]
```

Out[25]:
```
state
California    33871648
New York      18976457
Texas         20851820
dtype: int64
```

> Other types of indexing and selection (discussed in Data Indexing and Selection) work as well; for example, selection based on Boolean masks:

其他类型的索引和选择（参见数据索引和选择）也是允许的；例如，使用布尔遮盖进行选择：

In [26]:
```python
pop[pop > 22000000]
```

Out[26]:
```
state       year
California  2000    33871648
            2010    37253956
Texas       2010    25145561
dtype: int64
```

> Selection based on fancy indexing also works:

使用高级索引进行选择：

In [27]:
```python
pop[['California', 'Texas']]
```

Out[27]:
```
state       year
California  2000    33871648
            2010    37253956
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Multiply indexed DataFrames

多重索引 DataFrame

> A multiply indexed `DataFrame` behaves in a similar manner. Consider our toy medical `DataFrame` from before:

对`DataFrame`进行多重索引也是同样的。再看前面我们的医疗`DataFrame`数据：

In [28]:
```python
health_data
```

Out[28]:

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	35.0	35.9	21.0	37.5	37.0	38.2
	2	28.0	37.3	43.0	38.3	35.0	36.2
2014	1	51.0	36.3	33.0	39.0	29.0	35.8
	2	43.0	35.7	19.0	36.2	43.0	36.1

> Remember that columns are primary in a `DataFrame`, and the syntax used for multiply indexed `Series` applies to the columns. For example, we can recover Guido's heart rate data with a simple operation:

请注意`DataFrame`中主要的索引是列，你可以将上面的多重索引`Series`的方法应用到`DataFrame`的列上。例如，通过一个简单的操作就能获得Guido的心率数据：

In [29]:
```python
health_data['Guido', 'HR']
```

Out[29]:
```
year  visit
2013  1        21.0
      2        43.0
2014  1        33.0
      2        19.0
Name: (Guido, HR), dtype: float64
```

> Also, as with the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in Data Indexing and Selection. For example:

同样，就像单一索引的情况那样，我们可以使用在（数据索引和选择）中介绍的`loc`、`iloc`和`ix`索引符。例如：

In [30]:
```python
health_data.iloc[:2, :2]
```

Out[30]:

	subject	Bob	
	type	HR	Temp
year	visit		
2013	1	35.0	35.9
	2	28.0	37.3

> These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

这些索引符提供了一个底层二维数据的数组视图，并且`loc`或`iloc`中每个独立的索引都可以传递一个多重索引的元组。例如：

In [31]:
```python
health_data.loc[:, ('Bob', 'HR')]
```

Out[31]:
```
year  visit
2013  1        35.0
      2        28.0
2014  1        51.0
      2        43.0
Name: (Bob, HR), dtype: float64
```

> Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

使用这种索引元组并不是特别的方便；例如试图在元组中使用切片会产生一个语法错误：

In [32]:
```python
health_data.loc[(:, 1), (:, 'HR')]
```

```
  File "<ipython-input-32-fb34fa30ac09>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                     ^
SyntaxError: invalid syntax
```

> You could get around this by building the desired slice explicitly using Python's built-in `slice()` function, but a better way in this context is to use an `IndexSlice` object, which Pandas provides for precisely this situation. For example:

解决上述问题的方法可以是显式调用Python內建的`slice()`函数，还有一个更好的方式是使用`IndexSlice`对象，该对象是Pandas专门为这种情况准备的。例如：

In [33]:
```python
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

Out[33]:

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	1	35.0	21.0	37.0
2014	1	51.0	33.0	29.0

> There are so many ways to interact with data in multiply indexed `Series` and `DataFrame`s, and as with many tools in this book the best way to become familiar with them is to try them out!

访问多重索引的`Series`和`DataFrame`对象中的数据有很多种方法，除了阅读本书中介绍的这些工具外，熟悉它们的最好方式就是在实践中使用它们。

Rearranging Multi-Indices

重新排列多重索引

> One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

使用多重索引数据的一个关键技能是掌握如何有效地转换数据形式。Pandas提供了一些操作能保留数据集的信息，并根据不同目的的计算需要对数据进行重新排列。前面我们已经看到了`stack()`和`unstack()`方法的简单介绍，实际上还有更多操作可以用来精细控制数据集的层次化的行和列索引，下面我们来介绍它们。

Sorted and unsorted indices

有序和无序的索引

> Earlier, we briefly mentioned a caveat, but we should emphasize it more here. *Many of the `MultiIndex` slicing operations will fail if the index is not sorted.* Let's take a look at this here.

前面我们稍微提到了有序和无序索引的概念，这里我们要强调一下。*如果索引是无序的话，很多`MultiIndex`的切片操作都会失败。*

> We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

我们来创建一些简单的多重索引数据，它们的索引不是*具有自然顺序*的：

In [34]:
```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
```

Out[34]:
```
char  int
a     1      0.923424
      2      0.785119
c     1      0.878949
      2      0.473416
b     1      0.505453
      2      0.064504
dtype: float64
```

> If we try to take a partial slice of this index, it will result in an error:

如果我们视图对这个`Series`对象进行切片，结果会发生错误：

In [35]:
```
try:
    data['a':'b']
except KeyError as e:
    print(type(e))
    print(e)
```

```
<class 'pandas.errors.UnsortedIndexError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

> Although it is not entirely clear from the error message, this is the result of the MultiIndex not being sorted. For various reasons, partial slices and other similar operations require the levels in the `MultiIndex` to be in sorted (i.e., lexographical) order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the `DataFrame`. We'll use the simplest, `sort_index()`, here:

虽然错误的信息并不是那么清晰易懂，实际上这是MultiIndex没有排序的结果。许多因素决定了，当对`MultiIndex`进行部分的切片和其他相似的操作时，都需要索引是有序（或者说具有自然顺序）的。Pandas提供了方法来对索引进行排序；例如`DataFrame`对象的`sort_index()`和`sortlevel()`方法。我们在这里使用最简单的`sort_index()`方法：

In [36]:
```
data = data.sort_index()
data
```

Out[36]:
```
char  int
a     1      0.923424
      2      0.785119
b     1      0.505453
      2      0.064504
c     1      0.878949
      2      0.473416
dtype: float64
```

> With the index sorted in this way, partial slicing will work as expected:

当索引排好序后，索引的切片就可以正常工作了：

In [37]:
```
data['a':'b']
```

Out[37]:
```
char  int
a     1      0.923424
      2      0.785119
b     1      0.505453
      2      0.064504
dtype: float64
```

Stacking and unstacking indices

索引的堆叠和拆分

> As we saw briefly before, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use:

我们前面已经看到，我们可以将一个堆叠的多重索引的数据集拆分成一个简单的二维形式，还可以指定使用哪个层次进行拆分：

In [38]:
```
pop.unstack(level=0)
```

Out[38]:

state	California	New York	Texas
year			
2000	33871648	18976457	20851820
2010	37253956	19378102	25145561

In [39]:
```
pop.unstack(level=1)
```

Out[39]:

year	2000	2010
state		
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

> The opposite of `unstack()` is `stack()`, which here can be used to recover the original series:

`unstack()`的逆操作是`stack()`，我们可以使用它来重新堆叠数据集：

In [40]:
```
pop.unstack().stack()
```

Out[40]:
```
state       year
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

Index setting and resetting

设置及重新设置索引

> Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a `DataFrame` with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

还有一种重新排列层次化数据的方式是将行索引标签转为列索引标签；这可以使用`reset_index`方法来实现。在人口数据集上调用这个方法能让结果`DataFrame`的列有层次化的州和年份标签，它们是从原来的行标签转换过来的。为了清晰起见，我们可以设置列的标签：

In [41]:
```
pop_flat = pop.reset_index(name='population')
pop_flat
```

Out[41]:

	state	year	population
0	California	2000	33871648
1	California	2010	37253956
2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

> Often when working with data in the real world, the raw input data looks like this and it's useful to build a `MultiIndex` from the column values. This can be done with the `set_index` method of the `DataFrame`, which returns a multiply indexed `DataFrame`:

通常当我们处理真实世界的数据的时候，我们看到的就会是如上的数据集的形式，因此从列当中构建一个`MultiIndex`会很有用。这可以通过在`DataFrame`上使用`set_index`方法来实现，这样会返回一个多重索引的`DataFrame`：

In [42]:
```
pop_flat.set_index(['state', 'year'])
```

Out[42]:

		population
state	**year**	
California	**2000**	33871648
	2010	37253956
New York	**2000**	18976457
	2010	19378102
Texas	**2000**	20851820
	2010	25145561

> In practice, I find this type of reindexing to be one of the more useful patterns when encountering real-world datasets.

在实践中，作者发现当处理真实世界数据集时，这种重新索引的方法会经常被用到。

Data Aggregations on Multi-Indices

多重索引的数据聚合

> We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, and `max()`. For hierarchically indexed data, these can be passed a `level` parameter that controls which subset of the data the aggregate is computed on.

前面我们已经了解到Pandas有內建的数据聚合方法，例如`mean()`、`sum()`和`max()`。对于层次化索引的数据而言，这可以通过传递`level`参数来控制数据沿着那个层次的索引来进行计算。

> For example, let's return to our health data:

例如，再看我们的那个健康数据集：

In [43]:
```
health_data
```

Out[43]:

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	**visit**						
2013	**1**	35.0	35.9	21.0	37.5	37.0	38.2
	2	28.0	37.3	43.0	38.3	35.0	36.2
2014	**1**	51.0	36.3	33.0	39.0	29.0	35.8
	2	43.0	35.7	19.0	36.2	43.0	36.1

> Perhaps we'd like to average-out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

可能我们希望能将每年测量值进行平均。我们可以用level参数指定我们需要进行聚合的标签，这里是年份：

In [44]:
```
data_mean = health_data.mean(level='year')
data_mean
```

Out[44]:

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	31.5	36.6	32.0	37.9	36.0	37.20
2014	47.0	36.0	26.0	37.6	36.0	35.95

> By further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

通过额外指定`axis`关键字，我们可以在列上沿着某个层次`level`进行聚合：

In [45]:
```
data_mean.mean(axis=1, level='type')
```

Out[45]:

type	HR	Temp
year		
2013	33.166667	37.233333
2014	36.333333	36.516667

> Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. This syntax is actually a short cut to the `GroupBy` functionality, which we will discuss in Aggregation and Grouping. While this is a toy example, many real-world datasets have similar hierarchical structure.

虽然只有两行代码，我们已经能够计算得到所有受试者每年多次测试取样的平均的心率和提问。这个语法实际上是`GroupBy`函数的一种简略写法，我们会在聚合和分组一节中详细介绍。虽然这只是一个模拟的数据集，但是很多真实世界的数据集也有相似的层次化结构。

Aside: Panel Data

额外知识：Panel数据

> Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) `Series` and (two-dimensional) `DataFrame` structures. Once you are familiar with indexing and manipulation of data in a `Series` and `DataFrame`, `Panel` and `Panel4D` are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in Data Indexing and Selection extend readily to these higher-dimensional structures.

Pandas还有一些其他的基础数据结构我们没有介绍到，名称为`pd.Panel`和`pd.Panel4D`的对象。这两个对象被认为是对应于一维的`Series`和二维的`DataFrame`相应的三维和四维的通用数据结构。一旦你熟悉了`Series`和`DataFrame`的使用方法，`Panel`和`Panel4D`的使用相对来说也是很直观的。特别的，我们在数据索引和选择中介绍过的`ix`、`loc`和`iloc`索引符在高维结构中也是直接可用的。

> We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simpler representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multi-indexing is fundamentally a sparse data representation. As the number of dimensions increases, the dense representation can become very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the `Panel` and `Panel4D` structures, see the references listed in Further Resources.

我们不会在本书中继续介绍Panel结构，因为作者认为在大多数情况下多重索引会更加有用，在表现高维数据时概念也会显得更加简单。而且更加重要的是，面板数据从基本上来说是密集数据，而多重索引从基本上来说是稀疏数据。随着维度数量的增加，使用密集数据方式表示真实世界的数据是非常的低效的。但是对于一些特殊的应用来说，这些结构是很有用的。如果你希望获取更多有关`Panel`和`Panel4D`结构的内容，请查阅更多资源。

< 处理空缺数据 | 目录 | 组合数据集：Concat 和 Append >

Open in Colab